1995

A graphical robotic simulation

Adam James Bell
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Bell, Adam James, "A graphical robotic simulation" (1995). *Master's Theses*. 973.
DOI: <https://doi.org/10.31979/etd.7ccd-7jmg>
https://scholarworks.sjsu.edu/etd_theses/973

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

A GRAPHICAL ROBOTIC SIMULATION

A Thesis

Presented to

The Faculty of the Department of Mathematics

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Arts

by

Adam James Bell

May 1995

UMI Number: 1374564

UMI Microform 1374564

Copyright 1995, by UMI Company. All rights reserved.

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI

**300 North Zeeb Road
Ann Arbor, MI 48103**

APPROVED FOR THE DEPARTMENT OF MATHEMATICS

Dr. Max Agoston

San Jose State University

Dr. Rudolf Rucker

San Jose State University

Dr. Jon H. Pearce

San Jose State University

APPROVED FOR THE UNIVERSITY

© 1995

Adam James Bell

ALL RIGHTS RESERVED

ABSTRACT

A GRAPHICAL ROBOTIC SIMULATION

by

Adam James Bell

This thesis work is the study and implementation, in the form a computer program, of the mathematics required for the kinematics and inverse kinematics solutions to the Puma 560 robot manufactured by Unimation Incorporated. The kinematics and inverse kinematics equations for the Puma 560 are developed. A motion control system that drives the robot's inverse kinematics system is described for Cartesian and rotational motions.

The computer program graphically simulates the robot performing various actions controlled by the user via the graphical interface. The program is written in C++ for an IBM PC compatible computer running DOS.

User documentation describes all the features within the simulation. There is also technical documentation which describes the C++ class hierarchy and the more important aspects of the code.

Dedication and Acknowledgment

To my wife, Cindy, for her endless love and support.

This work would not have been possible without it.

To my children, Kayleigh, Anthony, and Kathryn.

Their enthusiasm inspires me.

To Dr. Max K. Agoston whose advise helped greatly in this thesis work.

Table of Contents

Introduction	1
Frames and Notation	6
The Standard Frames	9
Kinematics	10
Wrist Frame Generation	20
Inverse Kinematics	21
Path Generation and Motion Control	38
The User Interface Documentation	45
The Main Menu	46
The Watch Menu	48
The Display Menu	51
The Robot Menu	53
The Operation Menu	55
The Help Menu	57
The Program Technical Documentation	58
The Event Loop	58
The Display Classes	60
The Shape Class	65
The Dsurf Class	70
Derived Dsurf Classes	74
Complex Shapes	74
The Tool Class	75

The Robot Class	76
Frame Operations	87
Future Development	89
Conclusions	90
References	91
Appendix A	92
Appendix B	94

List of Figures

Figures

1. The simulation display	3
2. A picture of the Puma 560 robot	4
3. The standard frames	11
4. A generic robotic link	12
5. The Puma 560 links and joints	14
6. Multiple configurations for a single solution to the Puma 560	36
7. A sample velocity profile	40
8. A sample help window	46
9. The Main menu	47
10. The Watch menu	48
11. The linear position watch window	49
12. The angular position watch window	50
13. The Display menu	51
14. The Robot menu	53
15. The Operation menu	56
16. The Help menu	57
17. The display classes hierarchy	60
18. The shape tree representation	64
19. The shape classes hierarchy	65

List of Tables

Tables

1. The link parameters for the Puma 560	13
2. The joint angles for all eight solutions to the inverse kinematic equations	37

Introduction

The development costs of products are of critical concern for most companies. When hardware systems and software systems intimately interact, there can be many wasted hours of work due to the serial nature of development. The need exists to decouple hardware and software development during the initial stages of system design. Development time can be shortened if these two groups can proceed in parallel. This yields a faster time to market and higher profits.

The low level software that implements motion control typically is written and tested without the benefit of system time on a physical robot. Systems engineers may have critical design flaws in their software that is very difficult to detect until the software and hardware integration stage of development is reached. Debugging this type of code can be very difficult without actually seeing the robot execute a programmed motion. The presence of hardware can also impede the debugging process due to real time constraints in the system.

The systems engineer who can develop the system software in a simulation environment has the benefit of writing code, executing code, and debugging code within a single environment without the use of the final target system.

The main objective of this thesis was to develop a computer simulation of the Puma 560 multi-joint robot manufactured by the Unimation Corporation. The written part of the thesis describes the mathematics

needed to manipulate the robot, documents the user interface of the program, and describes the technical aspects of the program.

Upon starting the robot program, the user is presented a display as shown in Figure 1. The robot is displayed in a three dimensional work cell consisting of a single tool held in a tool rack and a table for a work surface. The main menu appears in the upper right hand corner of the display. A photograph of a real Puma 560 robot is shown in Figure 2. The menu system allows the user to interact with the environment via the keyboard or the mouse.

The motion control algorithms and data structures are as they would appear in an operational robot with one exception. A joint angle defines the relative orientation between the components in the robot and is stored in radians. In a real application, the joint angles are stored in direct actuator units. Actuator units for stepper motors are typically electrical angles or microsteps.

The written part of this thesis is organized into the following sections.

The Frames and Notation section describes a fundamental unit, the frame, that is used in the graphical system as well as the kinematics and inverse kinematics equations for the robot. Standard frames for robotic systems are also described.

The Kinematics section describes how the frames for robotic joints are assigned. The assigned frames are used to solve the kinematics for the Puma 560 robot. Any given solution for the kinematics of a robot will yield the position and orientation of each joint within the robot. Each joint frame

Figure 1. The simulation environment as seen when the program begins.

is given as a function of the joint angle and closed forms are described.

Servo motors create torque until the commanded position is achieved and thus require positional feedback. Since no positional feedback is available, the program written applies only to applications where stepper motors are used. The representation of the joint angles does not affect the mathematics involved for the robotic kinematics and inverse kinematics.

The Inverse Kinematics section describes the method used to solve

Figure 2. A picture of a real Puma 560 robot.

the inverse kinematics for the Puma 560. The inverse kinematics equations are not linear and the Puma 560 equations have eight solutions for any given configuration. A solution is defined as a set of joint angles that place the robot in the required configuration. The method for selecting the final solution is described.

The Motion Control section describes the mathematics used to generate a list of frames. This list describes a trajectory that the robot follows in simulated real time. The trajectory accounts for velocity and acceleration control. Linear motion and rotational motion schemes are also described.

The User Interface Documentation section describes the functions that may be selected within the menu system. The menu system is presented in a top down fashion. The menus give the user control over the simulation environment. The major features are

- 1) Specification of new positions and orientations for the tool or the manipulator
- 2) Direct manipulation of the robotic joints
- 3) Execution of a predefined manufacturing routine to simulate a drilling operation
- 4) Dynamic examination of Cartesian and angular joint positions
- 5) Graphical display manipulations
- 6) On-line help

The on-line help system is also described.

The Program Technical Documentation section provides a top down description of the program structure and function for this thesis work. The program is written in C++ and uses the Borland C++ compiler. The details of the important C++ classes are given. The target system is a 486 or a 386 with a math coprocessor. The simulation requires an EGA or VGA color monitor.

The Future Work section lists enhancements that can be made to this simulation. An interpreter for a command language and a learn option are described.

Appendix A defines some notation used in this paper. Appendix B is a list of the source code modules written for this simulation.

Frames and Notation

The objects modeled in the simulation represent rigid objects in the real world. With each object, such as a block, a cylinder, or the system camera (the user's point of view), there is associated a frame. A frame basically corresponds to a coordinate system with respect to which the object is defined. Formally a frame is a 4-tuple (u_1, u_2, u_3, p) where (u_1, u_2, u_3) is an orthonormal basis for $\mathcal{R}^3$ and p is a vector thought of as the origin of the frame or the coordinate system it defines. Although a frame is being defined in terms of vectors it can be interpreted in several different ways. A frame can be identified with the transformation which sends a point q to a point q' where

$$q' = q \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} + p \quad (1)$$

Note that this transformation maps frame coordinates into world coordinates and therefore frames can also be thought of as transforming from one coordinate system 'a' to another coordinate system 'b'. Throughout this paper no distinction is made between a frame as a tuple of vectors and as the transformation described above. Furthermore, it will often be convenient to use homogeneous coordinates. Which type of coordinates are being used will be clear from the context.

The following notation is useful. The matrix

$$\begin{bmatrix} u_1^T & u_2^T & u_3^T \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \quad (2)$$

is called the rotational matrix associated with the frame and is denoted by ${}_aR_b$ or just R if the coordinate systems in question are clear. The matrix, in homogeneous coordinates, for the transformation in Equation 1 is denoted by ${}_aT_b$ and has the form

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3)$$

It is frequently convenient to denote the positional part of the transformation ${}_aT_b$ as ${}_aP_b$. The notation P_b will be used to indicate the coordinates of a point P with respect to coordinate system b .

Frame operations correspond to matrix multiplications. There are three important frame operations needed to manipulate objects efficiently. These are the move operation, the compose operation, and the invert operation.

The move operation is applied to a point P to move it to a new position. This is equivalent to Equation 1. Using the transformation ${}_aT_b$ to move point P_b to point P_a

$$P_a = {}_aT_b P_b \quad (4)$$

A point P can be moved several times to reach a destination point. For example, the equation

$$P_a = {}_aT_b {}_bT_c P_c \quad (5)$$

corresponds to moving P_c by ${}_bT_c$ and then by ${}_aT_b$. There are two matrix multiplications required for this move operation. This is inefficient if the operation is to be carried out many times. A better solution is to form the composition of the two transformations and then apply the single transformation to P_c to obtain P_a . Given two transformations, ${}_aT_b$ and ${}_bT_c$, the composition of the transformations is defined by

$${}^aT_c = \begin{bmatrix} & {}^aR_b {}^bR_c & {}^aR_b ({}^bP_c) + {}^aP_b \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (6)$$

By composing two frames, the number of multiplications and additions in the move operation are reduced. This is often used when calculating the joint positions of the Puma 560 and is explained in the kinematics section.

The invert operation calculates the inverse of a frame. Given any frame aT_b , ${}^aT_b^{-1} = {}^bT_a$ and

$${}^bT_a = \begin{bmatrix} & {}^aR_b^T & -{}^aR_b^T ({}^aP_b) \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7)$$

where ${}^aR_b^T$ denotes the transposition of the rotational matrix. Note that ${}^aR_b^{-1} = {}^aR_b^T$.

The Standard Frames

Typically, a frame is defined relative to a known frame. Many times the known frame is defined by the standard XYZ Cartesian coordinate unit vectors, (e_1, e_2, e_3) , and the position $(0, 0, 0)$. The e_i vectors are defined in Appendix A. However, the Puma 560 joint frames are defined relative to each other. The robot has a special frame called the base frame or the zero

frame. This frame is the origin for the robot's world. The robot has a station frame, ${}_B T_S$, that is defined relative to the robot base frame. Any motion to be performed is specified by a goal frame, ${}_S T_G$, that is defined relative to the station frame. Any tool that can be attached to the robot has an associated tool frame, ${}_W T_T$, that is defined relative to the wrist frame of the manipulator. The wrist frame, ${}_B T_W$, is defined to be the frame for the last component in the robot. An example of these frame definitions is shown in Figure 3.

Kinematics

Kinematics is defined as the study of motion. However, kinematics does not account for the forces that cause the motion. Manipulator dynamics deal with the forces required to induce motion. This includes the torque required to move the robot and how the robot reacts to external forces. These considerations are not contained in this thesis work. The kinematics of the Puma 560 describe the position, velocity, and acceleration of the robotic joints. Only the positional kinematics are implemented in this thesis work.

The general kinematics problem is to determine the Cartesian position and orientation of each joint given a set of joint angles defined for the robot. Since a set of joint angles completely defines the orientation and position of the robot, this indicates that the kinematics equations must be a function of the joint angles. To solve this general problem, the geometry of the robot,

Figure 3. Standard frame assignments. The frames are shown as the 3 vectors, each intersecting at the position defined within the frame. Sections of the robot have been removed for clarity.

its components, and its joint frames must be discussed.

Each component of the robot is called a link. A link has four properties that describe it. These are the angle of rotation, twist, offset, and length and are denoted by θ_i , α_i , d_i , and a_i , respectively. These values are determined by the geometry of the robot. The Puma 560 has six links and

Figure 4. A generic link.

each link has these four link variables. A generic link is shown in Figure 4.

There are standardized steps to follow when assigning the link parameters to the links in a system. The axes of rotation of each link represent the lines of action that define the parameters. There exists an imaginary common perpendicular line between two adjacent links or where the link axes intersect. At this point, a joint frame is attached for use in the kinematics and inverse kinematics solutions. In Figure 4, the Z_i axis of the frame points in the direction of the i^{th} link axis. This is also known as the

i^{th} joint axis. The X_i axis of the frame either points along the common perpendicular or, if the link axes intersect, the X_i axis is normal to the plane defined by the two link axes. The Y_i axis is the cross product of the vector Z_i and X_i . This method allows for arbitrary assignment of the Z_i direction.

The joint variable a_i represents the distance between Z_i and Z_{i+1} measured along the X_i axis. The joint variable α_i represents the angle between Z_i and Z_{i+1} measured about the X_i axis. The joint variable d_i represents the distance between X_{i-1} and X_i measured along the Z_i axis. The joint variable θ_i represents the angle between X_{i-1} and X_i measured about the Z_i axis. This method is described by Craig (1989). The joint parameters for the Puma 560 are shown in Table 1.

Joint i	α_i (degrees)	a_{i-1} (dist)	d_i (dist)	θ_i (radians)
1	0	0	0	θ_1
2	-90	0	0	θ_2
3	0	a_2	d_3	θ_3
4	-90	a_3	d_4	θ_4
5	90	0	0	θ_5
6	-90	0	0	θ_6

Table 1. The list of the link parameters for the Puma 560 manipulator. For this thesis work, the link variable $a_3 = 0$. Notice that only the θ_i 's are variable.

Even though there are four link parameters, only one is a variable, namely θ_i . The others are fixed by the physical mechanics of the manipulator. The displayed objects that represent the links of the Puma 560 are shown in Figure 5.

Figure 5. This figure shows the objects that represent the links for the Puma 560. Figure 5a shows Link 1, Link 2, and Link 3. Figure 5b shows Link 4, Link 5, and Link 6. The tool is also shown in Figure 5b. The location of each joint is the origin for each joint frame.

In general, Craig (1989) describes the logical transformation ${}_{i-1}T_i$ as the following composition of transformations.

$${}_{i-1}T_i = R_x(\alpha_{i-1}) D_x(a_{i-1}) R_z(\theta_i) D_z(d_i) \quad (8)$$

where $R_x(\alpha_{i-1})$ is a rotation about the x axis of the frame through the angle α_{i-1} , $R_z(\theta_i)$ is a rotation about the z axis of a frame through the angle θ_i , $D_x(a_{i-1})$ is a translation of distance a_{i-1} along the x axis of a frame, and $D_z(d_i)$ is a translation of distance d_i along the z axis of a frame.

This ${}_{i-1}T_i$ transformation defines the i^{th} joint frame and shows the translations and rotations needed to define a relative joint frame from one link to the next. The matrix of the transformation is computed by

$${}_{i-1}T_i = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i c\alpha_{i-1} & c\theta_i c\alpha_{i-1} & -s\alpha_{i-1} & -s\alpha_{i-1}d_i \\ s\theta_i s\alpha_{i-1} & c\theta_i s\alpha_{i-1} & c\alpha_{i-1} & c\alpha_{i-1}d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (9)$$

Here and throughout the rest of the paper $c\theta_i$ denotes the cosine of θ_i and $s\theta_i$ denotes the sine of θ_i . Sometimes this notation is also abbreviated to c_i and s_i , respectively. Each joint has a frame defined by this transformation. Since the only joint variable is θ_i , the transformation becomes more simple. The joint frames for the Puma 560 are defined in Equations 10 - 15.

$${}^0T_1 = \begin{bmatrix} c\theta_1 & -s\theta_1 & 0 & 0 \\ s\theta_1 & c\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (10)$$

$${}^1T_2 = \begin{bmatrix} c\theta_2 & -s\theta_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -s\theta_2 & -c\theta_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11)$$

$${}^2T_3 = \begin{bmatrix} c\theta_3 & -s\theta_3 & 0 & a_2 \\ s\theta_3 & c\theta_3 & 0 & 0 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (12)$$

$${}^3T_4 = \begin{bmatrix} c\theta_4 & -s\theta_4 & 0 & a_3 \\ 0 & 0 & 1 & d_4 \\ -s\theta_4 & -c\theta_4 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (13)$$

$${}^4T_5 = \begin{bmatrix} c\theta_5 & -s\theta_5 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ s\theta_5 & c\theta_5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (14)$$

$${}^5T_6 = \begin{bmatrix} c\theta_6 & -s\theta_6 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -s\theta_6 & -c\theta_6 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (15)$$

0T_1 is defined relative to the base frame. The base frame is a reference frame for the Puma 560 kinematics and inverse kinematics calculations. This frame is located in the same Cartesian location as joint frame one and its axes are parallel to the Cartesian coordinate system. It is a fixed frame that never moves.

Joint frames can be composed and used in other calculations. This is necessary to solve the general kinematics question first posed. One must remember that joint frame i is defined relative to joint frame $i-1$ and the base frame is fixed. Therefore the position and orientation of joint 1 is immediately known from the frame 0T_1 . By performing the composition ${}^0T_1 {}^1T_2$, the frame 0T_2 is obtained. Therefore, the position and orientation of joint 2 is known from frame 0T_2 . In general, iT_j will denote the composition of ${}^iT_{i+1} \dots {}^{j-1}T_j$. Each successive joint frame is composed and each position and orientation is defined with the new frame. This series of

calculations requires 5 compositions since a previous composition can be used as an argument for the next composition.

The most useful transformation is 0T_6 , which was called the wrist frame, BT_W , earlier. This can be calculated by composing each joint frame in series as follows

$${}^0T_6 = {}^0T_1 {}^1T_2 {}^2T_3 {}^3T_4 {}^4T_5 {}^5T_6 \quad (16)$$

Using this transformation and the defined joint angles for each joint will result in the Cartesian orientation and position of joint 6 relative to the base frame. Rather than calculating this composition each time 0T_6 is needed, the wrist frame is found once and is defined in closed form. A closed form equation requires no compositions once it is defined. The closed form will have significantly more complexity than a single joint frame. The closed form for 0T_6 is used as the input for inverse kinematics equations.

A similar method for nested transformations was used by Blinn (1987) for his Blobby Man. Blinn describes the composition of serial transformations as it applies to the simulation of human movement. Blinn assigns frames to the joints of the human body in his simulation. For example when the hip moves, the composition of frames cause the leg to move.

The final result for the Puma 560 closed form transformation 0T_6 is

$${}^0T_6 = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (17)$$

where

$$r_{11} = c_1[c_{23}(c_4c_5c_6 - s_4s_6) - s_{23}s_5c_6] + s_1(s_4c_5c_6 + c_4s_6) \quad (18)$$

$$r_{21} = s_1[c_{23}(c_4c_5c_6 - s_4s_6) - s_{23}s_5c_6] - c_1(s_4c_5c_6 + c_4s_6) \quad (19)$$

$$r_{31} = -s_{23}(c_4c_5c_6 - s_4s_6) - c_{23}s_5c_6 \quad (20)$$

$$r_{12} = c_1[c_{23}(-c_4c_5s_6 - s_4c_6) - s_{23}s_5s_6] + s_1(c_4c_6 - s_4c_5s_6) \quad (21)$$

$$r_{22} = s_1[c_{23}(-c_4c_5s_6 - s_4c_6) - s_{23}s_5s_6] - c_1(c_4c_6 + s_4c_5s_6) \quad (22)$$

$$r_{32} = -s_{23}(-c_4c_5s_6 - s_4c_6) + c_{23}s_5s_6 \quad (23)$$

$$r_{13} = -c_1(c_{23}c_4s_5 + s_{23}c_5) - s_1s_4s_5 \quad (24)$$

$$r_{23} = -s_1(c_{23}c_4s_5 + s_{23}c_5) + c_1s_4s_5 \quad (25)$$

$$r_{33} = s_{23}c_4s_5 - c_{23}c_5 \quad (26)$$

$$p_x = c_1[a_2c_2 + a_3c_{23} - d_4s_{23}] - d_3s_1 \quad (27)$$

$$p_y = s_1[a_2c_2 + a_3c_{23} - d_4s_{23}] + d_3c_1 \quad (28)$$

$$p_z = -a_3s_{23} - a_2s_2 - d_4c_{23} \quad (29)$$

and where c_{23} and s_{23} are defined as

$$c_{23} = c_2 c_3 - s_2 s_3 \quad (30)$$

$$s_{23} = c_2 s_3 + s_2 c_3 \quad (31)$$

This and similar calculations completely define the positional kinematics for the Puma 560. The implementation is described in the technical documentation section. Notice that only the position is dependent on the a_i and d_i parameters.

For specific joint position queries, it is more efficient to use the closed forms for each joint because the number of multiplications and additions are reduced. However, since each joint position is queried dynamically in this simulation, it is more efficient to calculate the position joint by joint and compose the results at each step to obtain all joint positions.

Wrist Frame Generation

When the goal frame and tool frame are identical, the wrist frame, B^T_W , can be calculated. Frame equality occurs when the end effector has reached the goal frame. The equality

$$B^T_W W^T_T = B^T_S S^T_G \quad (32)$$

can be used to solve for B^T_W as follows

$$B^T_W = B^T_S S^T_G (W^T_T)^{-1} \quad (33)$$

The wrist is not coincident with the tool mounting face of the robot. However, the distance from the wrist to the mounting face is a constant offset along the z axis of the wrist frame. Therefore, the distance is added to the tool frame position vector before the frame is inverted and thus it is included in Equation 33. It shows that given a goal frame, a tool frame, and a station frame, the wrist frame can be easily calculated. This is executed each time the inverse kinematics equations are to be solved.

Inverse Kinematics

Inverse kinematics is a much more difficult problem to solve than the direct kinematics. Inverse kinematics calculations are used to generate a set of joint angles that will place the robotic manipulator in the position and orientation specified by the goal frame. Given a specified goal frame, station frame, and tool frame, the wrist frame can be calculated as shown in Equation 33. The wrist frame is used to generate joint angle sets. There is not a unique solution to the inverse kinematics equations because they are nonlinear. Once the solution set has been generated, a specific instance within the solution set is chosen as the final solution. The robot is then moved to the new joint angles.

Issues of solvability have been simplified for the Puma 560 by restricting the α_i variables to 0, -90 degrees, and +90 degrees. Most of the a_i and d_i variables are 0. These restrictions ensure the inverse kinematics for the Puma 560 are solvable.

The Puma 560 solution set contains eight solutions for the specified equations. Problems arise when a solution is attempted for a goal frame that is not physically achievable. This inspired the concept of reachable workspace. The reachable workspace is a set of points in $\mathbb{R}^3$ which the robot's end effector can reach in at least one orientation. This is a superset of the dexterous workspace. The dexterous workspace is set of points in $\mathbb{R}^3$ which the robot's end effector can reach in all orientations. Several general methods for solving inverse kinematics are described by Craig (1989). An algebraic technique he describes is used to solve the Puma 560 inverse kinematics.

When a robot with revolute and prismatic joints has six degrees of freedom in a single series chain then the system is solvable. D. Pieper states that a manipulator with six revolute joints will have a closed form solution if three neighboring joint axes intersect at a point (Craig 1989). The Puma 560 has joint 4, joint 5, and joint 6 intersecting at one point and all joints are revolute. Therefore, the Puma 560 inverse kinematics equations have a closed form and are solvable.

The solution strategy for the Puma 560 begins with generating the target wrist frame, namely 0T_6 . This frame is known to be the composition of six joint frames, each a function of its joint angle.

$${}^0T_6 = {}^0T_1(\theta_1) {}^1T_2(\theta_2) {}^2T_3(\theta_3) {}^3T_4(\theta_4) {}^4T_5(\theta_5) {}^5T_6(\theta_6) \quad (34)$$

At each step of the solution, a single frame is inverted and both sides of the equation are multiplied by the inverted frame. Specific matrix terms can be

isolated to yield equations where a single unknown joint angle exists. The equations are solved to determine the joint angle. There is typically more than one solution. Each solution is used to further solve for the other joint angles. This is how the set of solutions is generated.

For the Puma 560, joint angle θ_1 is solved for first. Inverting an individual joint frame is simple and multiplying Equation 34 by ${}^0T_1^{-1}$ results in

$${}^1T_6 = {}^0T_1^{-1} {}^0T_6 \quad (35)$$

This is equivalent to

$$= \begin{bmatrix} k_{11} & k_{12} & k_{13} & k_x \\ k_{21} & k_{22} & k_{23} & k_y \\ k_{31} & k_{32} & k_{33} & k_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c_1 & s_1 & 0 & 0 \\ -s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (36)$$

where

$$k_{11} = c_{23}(c_4c_5c_6 - s_4s_6) - s_{23}s_5s_6 \quad (37)$$

$$k_{21} = -s_4c_5c_6 - c_4s_6 \quad (38)$$

$$k_{31} = -s_{23}(c_4c_5c_6 - s_4s_6) - c_{23}s_5c_6 \quad (39)$$

$$k_{12} = -c_{23}(c_4c_5s_6 + s_4c_6) + s_{23}s_5s_6 \quad (40)$$

$$k_{22} = s_4c_5s_6 - c_4c_6 \quad (41)$$

$$k_{32} = s_{23}(c_4c_5s_6 + s_4c_6) + c_{23}s_5s_6 \quad (42)$$

$$k_{13} = -c_{23}c_4s_5 - s_{23}c_5 \quad (43)$$

$$k_{23} = s_4s_5 \quad (44)$$

$$k_{33} = s_{23}c_4s_5 - c_{23}c_5 \quad (45)$$

$$k_x = a_2c_2 + a_3c_{23} - d_4s_{23} \quad (46)$$

$$k_y = d_3 \quad (47)$$

$$k_z = -a_3s_{23} - a_2s_2 - d_4c_{23} \quad (48)$$

The r_{ij} terms are directly accessible from the frame 0T_6 .

Notice that only one term is independent of all joint angles, namely k_y . Therefore, equating terms [2,4] from Equation 36 yields

$$-s_1p_x + c_1p_y = d_3 \quad (49)$$

Applying polar coordinate transformation techniques, define

$$p_x = \rho \cos\phi \quad (50)$$

$$p_y = \rho \sin\phi \quad (51)$$

$$\rho = \sqrt{(p_x^2 + p_y^2)} \quad (52)$$

$$\phi = \text{atan2}(p_y, p_x) \quad (53)$$

The atan2 function finds the arc tangent of the two signed arguments but returns the angle corresponding to the proper quadrant. A complete definition is found in Appendix A. By substituting Equation 50 and Equation 51 into equation 49, Equation 54 is obtained.

$$-s_1 (\rho \cos\phi) + c_1 (\rho \sin\phi) = d_3 \quad (54)$$

There are some needed trigonometric identities.

$$\sin(\phi - \beta) = \sin\phi \cos\beta - \cos\phi \sin\beta \quad (55)$$

$$\cos(\phi - \beta) = \cos\phi \cos\beta + \sin\phi \sin\beta \quad (56)$$

$$\sin^2\beta + \cos^2\beta = 1 \quad (57)$$

Using Equation 55 and dividing by ρ in Equation 54

$$\sin(\phi - \theta_1) = d_3 / \rho \quad (58)$$

From Equation 57 and Equation 58,

$$\cos\beta = \pm\sqrt{(1 - \sin^2\beta)} \quad (59)$$

Let $\beta = \phi - \theta_1$. Substituting this into Equation 59 yields

$$\cos(\phi - \theta_1) = \pm\sqrt{(1 - d_3^2 / \rho^2)} \quad (60)$$

Therefore, by the definition of atan2,

$$\phi - \theta_1 = \text{atan2}(\sin(\phi - \theta_1), \cos(\phi - \theta_1)) \quad (61)$$

Substituting Equation 53, Equation 58, and Equation 60 into Equation 61 yields

$$\theta_1 = \text{atan2}(p_y, p_x) - \text{atan2}(d_3 / \rho, \pm\sqrt{(1 - p_x^2 + p_y^2 - d_3^2)}) \quad (62)$$

Since there is a plus or minus value for the square root in the atan2 term there are two possible solutions to Equation 62. Both values are to be used to generate solutions for the other joint angles.

Joint angle θ_3 is solved for next. Terms [1,3] and [3,4] in Equation 36 are selected so that θ_3 can be solved for independent of the unknown joint angles.

$$c_1 p_x + s_1 p_y = k_{13} = a_3 c_{23} - d_4 s_{23} + a_2 c_2 \quad (63)$$

$$-p_z = -k_z = a_3 s_{23} + d_4 c_{23} + a_2 s_2 \quad (64)$$

Squaring Equation 54, Equation 63, and Equation 64 and adding them results in an expression represented by K.

$$\begin{aligned} K &= a_3 c_3 - d_4 s_3 \\ &= (p_x^2 + p_y^2 + p_z^2 - a_2^2 - a_3^2 - d_3^2 - d_4^2) / a_2 \end{aligned} \quad (65)$$

Equation 65 is only a function of joint variable θ_3 because the only sine and cosine operations are on θ_3 . Equation 65 is of the same form as Equation 49. Following a parallel substitution scheme as for θ_1 , the solution for θ_3 yields

$$\theta_3 = \text{atan2}(a_3, d_4) - \text{atan2}(K, \pm \sqrt{a_3^2 + d_4^2 - K^2}) \quad (66)$$

Once more, since there is a plus or minus value for the square root in the atan2 term in Equation 66, there are two possible solutions to this equation. There are two solutions for θ_1 and two solutions for θ_3 . This represents four partial solutions within the solution set that will be used to generate solutions for the other joint angles.

The next joint angle to solve for is θ_2 . This calculation will use the results from the solutions for θ_1 and θ_3 . The frames ${}^0T_3^{-1}$ and 3T_6 are needed in the equation

$${}^3T_6 = {}^0T_3^{-1} {}^0T_6 \quad (67)$$

This is equivalent to

$$\begin{bmatrix} c_4c_5c_6 - s_4s_6 & -c_4c_5s_6 - s_4c_6 & -c_4s_5 & a_3 \\ s_5c_6 & -s_5s_6 & c_5 & d_4 \\ -s_4c_5c_6 - c_4s_6 & s_4c_5s_6 - c_4c_6 & s_4s_5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c_1c_{23} & s_1c_{23} & -s_{23} & -a_2c_3 \\ -c_1s_{23} & -s_1s_{23} & -c_{23} & a_2s_3 \\ -s_1 & c_1 & 0 & -d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (68)$$

The notation c_{23} denotes $\cos(\theta_2 + \theta_3)$ and is equivalent to $c_2c_3 - s_2s_3$.

The notation s_{23} denotes $\sin(\theta_2 + \theta_3)$ and is equivalent to $c_2s_3 + s_2c_3$.

Terms [1,4] from Equation 68 are equated so that θ_2 can be solved for independent of the unknown joint angles. This yields

$$c_1c_{23}p_x + s_1c_{23}p_y - s_{23}p_z - a_2c_3 = a_3 \quad (69)$$

$$-c_1s_{23}p_x - s_1s_{23}p_y - c_{23}p_z + a_2s_3 = d_4 \quad (70)$$

Solving Equation 69 and Equation 70 for c_{23} and s_{23} one obtains

$$s_{23} = \frac{[(-a_3 - a_2c_3)p_z + (a_2s_3 - d_4)(c_1p_x + s_1p_y)]}{[p_z^2 + (c_1p_x + s_1p_y)^2]} \quad (71)$$

$$c_{23} = \frac{[(a_2s_3 - d_4)p_z - (-a_3 - a_2c_3)(c_1p_x + s_1p_y)]}{[p_z^2 + (c_1p_x + s_1p_y)^2]} \quad (72)$$

These values are used as arguments to atan2 to solve for θ_{23} where

$$\theta_{23} = \theta_2 + \theta_3 \quad (73)$$

The denominators can be excluded because they are identical and positive. Therefore they do not effect the solution of atan2.

$$\theta_{23} = \text{atan2}[(-a_3 - a_2c_3)p_z + (a_2s_3 - d_4)(c_1p_x + s_1p_y), \\ (a_2s_3 - d_4)p_z - (-a_3 - a_2c_3)(c_1p_x + s_1p_y)] \quad (74)$$

Equation 74 is a function of the known joint variables θ_1 and θ_3 . There are four solutions generated for θ_{23} corresponding to the four partial solution sets that have been generated so far. The joint variable θ_2 can be solved for by

$$\theta_2 = \theta_{23} - \theta_3 \quad (75)$$

This finally yields the four solutions for θ_2 .

The next joint variable to solve for is θ_4 . The calculation will use the results from the solutions for θ_1 , θ_2 , and θ_3 . Terms [1,3] from Equation 68 are equated so that θ_4 can be solved for independent of the unknown joint angles, namely θ_5 and θ_6 . Therefore

$$r_{13}c_1c_{23} + r_{23}s_1c_{23} - r_{33}s_{23} = -c_4s_5 \quad (76)$$

$$-r_{13}s_1 + r_{23}c_1 = s_4s_5 \quad (77)$$

Note that the r_{ij} values are known and are contained in the 0T_6 transformation. Dividing by s_5 is allowed if $\theta_5 \neq 0$. The manipulator is said to be in a singular configuration when $\theta_5 = 0$. This means that joint 4 and joint 6 are aligned. In this configuration, the manipulator loses one degree of freedom and the equations break down. At any given time, each joint has a value for its joint angle. If the manipulator is in a singular configuration then the current value of θ_4 is used as the solution. Since θ_5 is not known at this point, the arguments to the atan2 function must be checked to determine if they are both close to zero. If this is the case then θ_4 is assigned as described. Otherwise c_4 and s_4 are solved for by dividing by s_5 .

$$c_4 = -(r_{13}c_1c_{23} + r_{23}s_1c_{23} - r_{33}s_{23}) / s_5 \quad (78)$$

$$s_4 = -r_{13}s_1 + r_{23}c_1 / s_5 \quad (79)$$

The sign of s_5 will effect the atan2 solution but will not effect the final solution set because there are additional solutions due to rotational symmetry. This is described in detail at the end of this section. Therefore

$$\theta_4 = \text{atan2}(-r_{13}s_1 + r_{23}c_1, -r_{13}c_1c_{23} - r_{23}s_1c_{23} + r_{33}s_{23}) \quad (80)$$

There for four solutions for θ_4 due to the four prior solutions for θ_1 , θ_2 , and θ_3 .

Joint variable θ_5 is solved for next using the equation

$${}^4T_6 = {}^0T_4^{-1} {}^0T_6 \quad (81)$$

This is equivalent to

$$\begin{bmatrix} c_5c_6 & -c_5s_6 & -s_5 & 0 \\ s_6 & c_6 & 0 & 0 \\ s_5c_6 & -s_5s_6 & c_5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} c_1 c_{23} c_4 + s_1 s_4 & s_1 c_{23} c_4 - c_1 s_4 & -s_{23} c_4 & -a_2 c_{34} + d_3 s_4 - a_3 c_4 \\ -c_1 s_{23} s_4 + s_1 c_4 & -s_1 c_{23} s_4 - c_1 c_4 & s_{23} s_4 & a_2 c_{34} + d_3 c_4 + a_3 s_4 \\ -c_1 s_{23} & -s_1 s_{23} & -c_{23} & a_2 s_3 - d_4 \\ 0 & 0 & 0 & 1 \end{bmatrix} *$$

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (82)$$

Terms [1,3] and [3,3] from Equation 82 are selected so that θ_5 can be solved for independent of the last unknown joint angle, namely θ_6 . This yields

$$-s_5 = r_{13}(c_1 c_{23} c_4 + s_1 s_4) + r_{23}(s_1 c_{23} c_4 - c_1 s_4) - r_{33}(s_{23} c_4) \quad (83)$$

$$c_5 = r_{13}(-c_1 s_{23}) + r_{23}(-s_1 s_{23}) + r_{33}(-c_{23}) \quad (84)$$

The joint variable θ_5 can be solved for as

$$\theta_5 = \text{atan2}(s_5, c_5) \quad (85)$$

Equation 85 has four solutions based on the four solutions for θ_1 , θ_2 , θ_3 , and θ_4 .

The final joint variable to solve for is θ_6 using the equation

$${}^5T_6 = {}^0T_5^{-1} {}^0T_6 \quad (86)$$

This is equivalent to

$$\begin{bmatrix} c\theta_6 & -s\theta_6 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -s\theta_6 & -c\theta_6 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} k_{11} & k_{12} & k_{13} & k_x \\ k_{21} & k_{22} & k_{23} & k_y \\ k_{31} & k_{32} & k_{33} & k_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (87)$$

where

$$k_{11} = (c_1c_4c_{23} + s_1s_4)c_5 - c_1s_{23}s_5 \quad (88)$$

$$k_{12} = (s_1c_{23}c_4 - c_1s_4)c_5 - s_1s_{23}s_5 \quad (89)$$

$$k_{13} = -c_4c_5s_{23} - c_{23}s_5 \quad (90)$$

$$k_{21} = -s_5(c_1c_4c_{23} + s_1s_4) - c_1c_5s_{23} \quad (91)$$

$$k_{22} = -s_5(s_1c_4c_{23} - c_1s_4) - c_5s_1s_{23} \quad (92)$$

$$k_{23} = c_4 s_{23} s_5 - c_{23} c_5 \quad (93)$$

$$k_{31} = c_1 s_4 c_{23} - s_1 c_4 \quad (94)$$

$$k_{32} = s_1 s_4 c_{23} + c_1 c_4 \quad (95)$$

$$k_{33} = -s_4 s_{23} \quad (96)$$

$$\begin{aligned} k_x = & k_{11} * (c_1 c_{23} a_3 - c_1 s_{23} d_4 + a_2 c_1 c_2 - s_1 d_3) + \\ & k_{12} * (s_1 c_{23} a_3 - s_1 s_{23} d_4 + a_2 s_1 c_2 + c_1 d_3) + \\ & k_{13} * (-s_{23} a_3 - c_{23} d_4 - a_2 s_2) \end{aligned} \quad (97)$$

$$\begin{aligned} k_y = & k_{21} * (c_1 c_{23} a_3 - c_1 s_{23} d_4 + a_2 c_1 c_2 - s_1 d_3) + \\ & k_{22} * (s_1 c_{23} a_3 - s_1 s_{23} d_4 + a_2 s_1 c_2 + c_1 d_3) + \\ & k_{23} * (-s_{23} a_3 - c_{23} d_4 - a_2 s_2) \end{aligned} \quad (98)$$

$$\begin{aligned} k_z = & k_{31} * (c_1 c_{23} a_3 - c_1 s_{23} d_4 + a_2 c_1 c_2 - s_1 d_3) + \\ & k_{32} * (s_1 c_{23} a_3 - s_1 s_{23} d_4 + a_2 s_1 c_2 + c_1 d_3) + \\ & k_{33} * (-s_{23} a_3 - c_{23} d_4 - a_2 s_2) \end{aligned} \quad (99)$$

Any combination of terms can be selected because θ_6 is the only unknown variable. However, the simplest equation to solve for θ_6 is chosen. Terms [3,1] and [1,1] are selected from Equation 87. The result is

$$\begin{aligned} s_6 = & -r_{11}(c_1 c_{23} s_4 - s_1 c_4) - r_{21}(s_1 c_{23} s_4 + c_1 c_4) + \\ & r_{31}(s_{23} s_4) \end{aligned} \quad (100)$$

$$\begin{aligned}
c_6 = & r_{11}[(c_1c_{23}c_4 + s_1s_4)c_5 - c_1s_{23}s_5] + \\
& r_{21}[(s_1c_{23}c_4 - c_1s_4)c_5 - s_1s_{23}s_5] - \\
& r_{31}(s_{23}c_4c_5 + c_{23}s_5)
\end{aligned} \tag{101}$$

The joint variable θ_6 can be solved for by

$$\theta_6 = \text{atan2}(s_6, c_6) \tag{102}$$

Equation 102 has four solutions based on the four solutions for θ_1 , θ_2 , θ_3 , and θ_4 , θ_5 .

The joint variables θ_4 , θ_5 , and θ_6 have four additional solutions due to rotational symmetry of the manipulator.

$$\theta_4' = \theta_4 + 180^\circ \tag{103}$$

$$\theta_5' = -\theta_5 \tag{104}$$

$$\theta_6' = \theta_6 + 180^\circ \tag{105}$$

Since there were four solutions for θ_4 , θ_5 , and θ_6 there are now eight solutions within the solution set. Care must be taken so the solutions set values are coherent. For example, the values of θ_1 , θ_2 , and θ_3 that solved the θ_4 equation must be used together to solve the θ_5 equation. Each solution corresponds to a single manipulator configuration.

Figure 6 shows the Puma 560 in four different configurations that achieve the same manipulator position and orientation. Four more configurations are possible by altering the joint angles specified in

Figure 6. The Puma 560 manipulator in four configurations where the end effector achieves the same position and orientation. The position is (13", 13", 13") relative to the station frame. The orientation is $\langle 1, 1, 1 \rangle$ relative to the station frame. The specific joint angles are listed in Table 2.

Equations 103 - 105. The eight solution sets for these configurations are listed in Table 2.

The next problem to solve in the general inverse kinematics solution is to select one of the eight existing solutions as the final solution.

At any given time the manipulator is in some existing state with six currently defined joint angles. The final solution is selected by calculating the difference between each current joint angle and a solution angle for

solution	θ_1	θ_3	θ_2	θ_4	θ_5	θ_6
1	37.5	212.6	23.0	101.0	6.2	263.2
2	37.5	327.3	335.3	173.4	68.3	186.7
3	232.4	212.6	156.9	186.5	291.6	353.2
4	232.4	327.6	204.6	258.9	353.7	276.7
5	37.5	212.6	23.0	281.0	353.7	83.2
6	37.5	327.3	335.3	353.4	291.6	6.7
7	232.4	212.6	156.9	6.5	68.3	173.2
8	232.4	327.6	204.6	78.9	6.2	96.7

Table 2. This table lists the joint angle solutions corresponding to Figure 6. Solution 1 corresponds to Figure 6a. Solution 2 corresponds to Figure 6b. Solution 3 corresponds to Figure 6c. Solution 4 corresponds to Figure 6d. The units are degrees.

each angle within a solution set. A weight is assigned to each difference and then all terms are summed. This procedure is applied to each solution set. The solution set with the corresponding minimal sum is chosen as the final solution. The joint angles are stored as non-negative values less than 360° . The difference described is the absolute value of the current joint angle and the solution angle.

The weight assigned to differences for joint 1, joint 2, and joint 3 is 2. The weight assigned to differences for joint 4, joint 5, and joint 6 is 1. This ensures large motions from the primary joints are more significant than the three smaller joints. Some solutions are eliminated due to joint violations. The only joints with defined violations are joint 3 and joint 5. Joint 3 is limited between 50° and 145° . Joint 5 is limited between 85° and 275° .

Path Generation and Motion Control

The motion control system relies heavily on the inverse kinematics mathematics. Robots require a motion control system to generate intermediate frames between the starting frame and the ending frame. The intermediate positions are called way points. Some robots have a fixed XYZ orientation and therefore no rotation matrix is required to accompany the way points. The Puma 560 must have orientation associated with each way point.

A time base is required for motion control. Time is usually represented by an interrupt interval, also denoted by a time slice or a tick. A discrete time was chosen to mock an interrupt time interval in the simulation implemented for this thesis. The time base is one IBM PC BIOS tick. This is equivalent to an interrupt cycle executing at approximately 18.2 Hz. Each time slice in a motion has an intermediate frame associated with it.

The software is easily changed for a higher time cycle if hardware becomes available in the future. Timing must be performed on the inverse kinematics solution pipeline to ensure the time required to determine a solution does not exceed the time slice.

The simulation supports linear and rotational motion. The general approach to the motion control system is to provide a goal frame relative to the station frame. This is passed to the inverse kinematics system in an attempt to move the manipulator to that goal frame. After a wrist frame is generated, as described in the kinematics section, the wrist frame is validated. The validation routine verifies the frame is within the reachable subspace and the frame is not within the body of the robot. This is by no means an absolute validation. It is a simple, quick check and does not validate the entire path from the current manipulator frame to the destination frame.

The linear path is calculated as an acceleration ramp, slew length, and a deceleration ramp. The acceleration ramp and deceleration ramp are the part of the motion in which the end effector is accelerating and decelerating, respectively. The slew is the part of the motion where the end effector is at a constant velocity. These are shown in Figure 7. The acceleration and the

Figure 7. The velocity profile of a linear motion.

slew velocity are predefined. Care must be taken when programming for stepper motors. A linear acceleration or velocity for the end effector could easily force one of the joint axes to exceed the upper acceleration or velocity limit for a stepper motor. This is not as critical for servo motors if the speeds are reasonable. However, if a motor's rotational limits are exceeded, then a deviation in the calculated linear path will occur.

Since the time slice is discrete, all calculations that involve time must manipulate time as an integer value. Let the cycle time be denoted by `ISR_HZ` in units of ticks per second. The discrete motion formulas are derived from the basic motion formulas

$$t = v / a \quad (106)$$

$$d = \frac{1}{2} a t^2 \quad (107)$$

Setting

$$\text{ramp_time} = t \quad (\text{sec})$$

$$\text{ramp_length} = d \quad (\text{inches})$$

$$\text{slew_velocity} = v \quad (\text{inches/sec})$$

$$\text{index_acceleration} = a \quad (\text{inches/sec}^2)$$

yields

$$\text{ramp_time} = \text{slew_velocity} / \text{index_acceleration} \quad (108)$$

$$\text{ramp_length} = \frac{1}{2} * \text{index_acceleration} * \text{ramp_time}^2 \quad (109)$$

and

$$\text{inches_per_tick} = \text{slew_velocity} / \text{ISR_HZ} \quad (110)$$

$$\text{slew_ticks} = (\text{move_dist} - 2 * \text{ramp_length}) / \text{inches_per_tick} \quad (111)$$

Equation 108 changes the time units from inches per second to inches per tick. If `slew_ticks` is less than zero, then no time is spent at the slew velocity and the acceleration and deceleration ramp will dominate the motion. Otherwise, the distance traveled at the slew velocity is subtracted

from the complete distance to yield the remaining distance. The discrete time spent accelerating must be calculated as shown in equation 112.

$$\text{accel_ticks} = \text{ISR_HZ} * \sqrt{(\text{remaining_dist} / \text{index_acceleration})} \quad (112)$$

Accel_ticks must be greater than zero. Since the time is discrete, the required acceleration may not be exactly equal to the index_acceleration that was defined. The new acceleration is calculated as

$$\text{new_accel} = (\text{remaining_dist} * \text{ISR_HZ}^2) / \text{accel_ticks}^2 \quad (113)$$

$$\text{ramp_time} = \text{accel_ticks} / \text{ISR_HZ} \quad (114)$$

$$\text{ramp_length} = \text{new_accel} * \text{ramp_time}^2 / 2 \quad (115)$$

The slew velocity may not be exactly equal to the defined slew velocity. It can be calculated as

$$\text{new_slew_velocity} = \text{new_accel} * \text{accel_ticks} \quad (116)$$

These calculations are performed for the Cartesian axis with the greatest distance to move. This axis motion requires more time to complete than the other two axes require. Therefore, new accelerations are calculated for the other two Cartesian axes. The time parameters are identical for all three Cartesian axes.

$$\text{axis_ramp_length} = \text{long_ramp_length} * \text{axis_distance} / \text{long_axis_distance} \quad (117)$$

$$\text{axis_slew_distance} = \text{axis_distance} - 2 * \text{axis_ramp_length} \quad (118)$$

$$\text{axis_accel} = 2 * \text{axis_ramp_length} / \text{accel_ticks}^2 \quad (119)$$

The `axis_slew_distance` is forced to be non-negative.

A complete motion is defined by a list of primitive motion items. A primitive motion item is an acceleration and a corresponding time of duration. These primitive motion items are integrated to produce a position list, or point list, on the specified time slice. The points in the point list lie on the linear path between the start position and the end position defined by their respective frames. These points become the positional portion of the frames used to describe the linear motion. If the point list is followed in real-time, then the requested acceleration and velocity will be closely approximated.

The Puma 560 must also interpolate between the current orientation and the goal orientation. The motion is initiated at time 0 and the total motion time is known. The orientation is introduced at a time slice for way point i by

$$\text{scale} = \text{current_time} / \text{total_motion_time} \quad (120)$$

$$v_1 + (v_2 - v_1) * \text{scale} \quad (121)$$

where v_1 denotes U_i within the starting frame and v_2 denotes U_i within the goal frame. From these calculations, a new frame is created for each way

point in the point list. Therefore, the point list is converted to a frame list on the defined time slice.

The linear motion control software first generates the wrist frame as described in the kinematics section. The point list is created and then converted to a frame list. Each frame is accessed and a request to the inverse kinematics is made for the frame. The inverse kinematics code solves and selects the appropriate solution as described in the inverse kinematics section. The simulation simply draws the robot in the new configuration. A real hardware system would command the motors to move to the new configuration.

The rotational motion system is required to allow an alternative way to move the robot. Some linear paths can cause joint violations to occur. Other linear paths may attempt to move the manipulator through the body of the robot. Therefore, rotational motion is required.

The rotational motion calculations are very similar to the linear motion calculations in terms of generating a wrist frame and a goal frame. However, the intermediate data creation is slightly different. Frames are not created for this motion type because the motion control directly manipulates the joint angles. The inverse kinematics solutions are not needed after the wrist frame has been generated from the goal frame. The axis that requires the most motion is found first as in the linear motion calculations. In this simulation, the change in angle per time slice is a fixed angle. The rotational motion calculations can be extended to use the equations described in the linear motion section as they apply to rotational motion. For the remaining

five joint angles, the change in angular position per time slice is calculated so that motion begins and end at the same time for each joint.

There are cases, such as homing the joint axes, where the user may specify the goal angles directly. In this case, all inverse kinematics calculations are avoided.

The rotational motion system first generates or is given the final joint angles to move to. The angular motion list is created. Each angle is read from the motion list and the proper joint angles are updated. The simulation simply draws the robot in the new configuration. A real hardware system would command the motors to move to the new configuration.

The User Interface Documentation

This section describes how the user interacts with the robot program and its menus. The menus are presented in a top down fashion. A menu item can be activated moving the mouse over the menu item and clicking the left mouse button. The user may also use the cursor keys to move the highlight bar onto the desired menu item and press the enter key. Finally, the user may press the key corresponding to the yellow hot key in the menu item name. The hot keys appear as underlined characters in the menu figures. The right mouse button and the escape key have the same effect. Each will remove the current menu and activate its parent menu or terminate the current action. No action is taken if the main menu is active. If a menu item activates a submenu, then the submenu appears just below

Figure 8. A sample help screen.

that menu item.

Some actions have other keys available that affect the display and are described where appropriate. There is an on line help system in the simulation. The F1 key is always active and will display a help screen with information about the currently selected menu item. A sample help screen is shown in Figure 8.

The Main Menu

This menu, shown in Figure 9, allows the user to activate submenus that contain groups of functions to manipulate the robot or to manipulate the graphical display.

The Program menu item is not functional. This will eventually allow a

Figure 9. The main menu.

user to execute a user written program to control the robot's motions for a specific application. This function requires an interpreter within the source code of the simulation program. Most, if not all, commercial systems have an interface for this function.

The Watch menu item activates a submenu that allows the user to activate a watch window. The watch window displays the dynamic position, velocity, or acceleration of the joints of the robot. The display units can be either a Cartesian representation in inches or an angular representation in degrees. Since the robot is not displayed in real time, the watch display will slow the drawing throughput of the graphical interface.

The Display menu item activates a submenu that allows the user to select from several items that will manipulate the graphical interface.

The Robot menu item activates a submenu that allows the user to select from several items that will manipulate the robot. There are also items that will command the robot to execute predefined motion sequences.

The Operation menu item activates a submenu that allows the user to initiate a predefined motion sequence that simulates useful work.

The Help menu item activates a submenu that allows the user to read information about on line help and the simulation program version.

The Quit menu item allows the user to exit the simulation.

The Watch Menu

This menu, shown in Figure 10, contains six menu items that allow the user to display or hide six different watch windows. The term watch window is used because the data displayed can be dynamically 'watched'. These menu items act as a toggle. A check mark to the left of the menu

Figure 10. The watch menu is shown here below the Watch selection in the main menu. The check mark indicates the linear position watch window is currently displayed.

item indicates a watch window is active. Selecting the menu item again will remove the watch window.

Selecting the Linear Pos item will display a window that shows the instantaneous absolute Cartesian position for selected joints of the robot and for the tool. The frames for the base, joint 1, and joint 2 have the same positional component. The frames for joint 4, joint 5, and joint 6 also have the same positional component. Therefore, only the position of the base frame, the position of joint 3, the position of joint 6, and the position of the tool are displayed in this watch window. This linear position watch window is shown in Figure 11.

Bx:	34.000
By:	-48.000
Bz:	37.000
X1:	34.000
Y1:	-48.000
Z1:	37.000
X2:	34.000
Y2:	-48.000
Z2:	37.000
X3:	45.250
Y3:	-45.750
Z3:	37.000
X6:	45.250
Y6:	-45.750
Z6:	28.750
Tx:	45.250
Ty:	-45.750
Tz:	28.750

Figure 11. The linear position watch window. The units are inches and are relative to the absolute origin of the simulated world.

θ1:	10.000
θ2:	340.000
θ3:	300.000
θ4:	45.000
θ5:	30.000
θ6:	10.000

Figure 12. The angular position watch window. The units are degrees.

The Linear Vel selection is not yet implemented but would display the instantaneous absolute Cartesian velocity for each joint of the robot.

The Linear Acc selection is not yet implemented but would display the instantaneous absolute Cartesian acceleration for each joint of the robot

The Angular Pos selection displays a window that shows the instantaneous angular position for each joint of the robot. These are unique to each joint, unlike the Cartesian position. The units displayed are degrees. This angular position watch window is shown in Figure 12.

The Angular Vel selection is not yet implemented but would display the instantaneous angular velocity for each joint of the robot.

The Angular Acc selection is not yet implemented but would display the instantaneous angular acceleration for each joint of the robot.

The Display Menu

This menu, shown in Figure 13, contains menu items that allow the user to manipulate the graphical display.

Selecting the Redraw item causes the display to be redrawn. The viewport is cleared and every object in the world is drawn.

Selecting the Zoom in item causes the display to be redrawn with the displayed objects appearing closer than before.

Selecting the Zoom out item causes the display to be redrawn with the displayed objects appearing farther away than before.

Figure 13. The display menu is shown here below the Display selection in the main menu. The check mark indicates that the objects displayed have their hidden faces, or back faces, removed from the display.

Selecting the **Face Norm** item causes the display to be redrawn with the displayed objects showing their face normals in the form of small vectors emanating from the center of the face. This menu item is not a toggle. Any subsequent display manipulation will erase the face normals.

Selecting the **Vrtx Norm** item causes the display to be redrawn with the displayed objects showing their vertex normals in the form of small vectors emanating from each vertex. This menu item is not a toggle. Any subsequent display manipulation will erase the vertex normals.

Selecting the **BF Elim** item causes the display to be redrawn with the displayed shapes in a new display mode. This selection toggles between wireframe mode and back face elimination mode. Wire frame mode displays every edge of an object. Back face elimination mode displays only the faces of the objects that are facing the system camera (the user's point of view).

Selecting the **Move** item allows the user to pan the world left, right, up, or down. The panning is done by pressing the cursor keys. When this menu item is selected a message is put in the system message box that instructs the user on what to do. This action is terminated by pressing the escape key.

Selecting the **Rotate** item allows the user to rotate the view of the world. The rotation is done by pressing the cursor keys. The left and right cursor keys rotate the camera about a vertical axis. The up and down cursor keys rotate the camera about a horizontal axis. When this menu item is selected, a message is put in the system message box that instruct the user on what to do. This action is terminated by pressing the escape key.

The Robot Menu

This menu, shown in Figure 14, contains menu items that allow the user to directly manipulate the robot. Several menu items allow predefined motion sequences to be initiated.

Selecting one of the Rotate items causes the corresponding joint of the robot to begin rotating. The robot rotates in small motions that are defined by direction and arc length. By default, the direction is in the positive angular direction for the specified joint and the arc length is five degrees. The user can reverse the direction by pressing the num_lock key.

Figure 14. The robot menu is shown here below the Robot selection in the main menu.

The user can reduce the arc length to one degree by pressing the scroll_lock key. The original parameters are restored by pressing the keys again. The robot motion may appear to be jerky. However, each motion moves the joint exactly one arc length and time is required between the calculations for each motion. The robot is simply making many small motions and not a large jerky motion. No joint violation checking is done for this type of motion.

Selecting the Home item causes the robot to return to the home position. The home configuration is achieved when every joint angle is zero.

Selecting the Frames item causes the display to be redrawn with the tool frame and each joint frame shown. Any subsequent display manipulation will erase these frames.

Selecting the Set Angle item allows the user to specify a joint and a target angle for the robot motion system. Joint violations are checked. If the angle specified is valid, then the robot will rotate the specified joint to the specified angle.

Selecting the Set Pos item allows the user to specify a Cartesian coordinate relative to the station frame and an orientation for the end effector. If the specified location and direction are reachable and valid, then the robot will move its end effector to the specified location and orientation. If a tool is attached, then the end effector is defined as the tip of the tool. If a tool is not attached, then the end effector is defined as the face of the robot's tool attachment component.

Selecting the Attach Tool item allows the user to initiate a predefined robotic motion to pick up the tool. The robot moves over the tool, moves

down linearly, and picks up the tool. The robot completes the motion by moving up linearly to a location that ensures a future motion will not crash the tool into the tool rack. If a tool is already attached, then an error message is displayed and no action is taken.

Selecting the **Drop Tool** item allows the user to initiate a predefined robotic motion to place the tool back in the tool rack. This motion is very similar to that of the attach tool motion. The difference is the tool is placed in the tool rack. If no tool is attached, then an error message is displayed and no action is taken.

Selecting the **Release Tool** item allows the user to tell the robot to release of the tool. In the simulation the tool seems to be suspended in mid-air. In reality, the tool would drop to the ground. This is useful in a real application if the user wants to remove the tool from the robot. If no tool is attached then an error message is displayed and no action is taken.

The Operation Menu

This menu, shown in Figure 15, contains menu items that allow the user to simulate a real life application.

Selecting the **Operate** item initiates a sequence of preprogrammed robotic motions that simulate the drilling of a panel. The robot picks up the tool as described for the Attach Tool menu item. The robot then moves over the simulated panel to be drilled. The robot moves linearly up and down simulating the drilling action. When the holes are complete, the robot replaces the tool in the tool rack. These actions make use of all routines

Figure 15. The operation menu is shown here below the Operation selection in the main menu.

within the thesis software created for motion control.

The Learn menu item is not yet implemented but would allow the user to tell the robot to move to intermediate points in a path the user requires. The points would be stored to disk for future use. This would give the user the means to make the robot move in a way not directly defined by the motion control system. It would also remove any ambiguity of motion that arises when using the inverse kinematics system. Recall the final solution is selected from the eight solutions based on the current manipulator configuration. By manually moving the manipulator to a specified configuration, all ambiguity is eliminated.

Figure 16. The help menu is shown here below the Help selection in the main menu.

The Help Menu

This menu, shown in Figure 16, contains menu items that allow the user to access information about the simulation.

Selecting the Help item displays a window that explains how to use the on line help system.

Selecting the About item displays a window with the thesis author's name, the thesis advisor's name, and the thesis title.

The Program Technical Documentation

The software for this simulation is written in C++. The true benefit of the object oriented paradigm is class inheritance and polymorphism. Inheritance allows the general notion of a shape to be derived for more specific shapes. Polymorphism allows the code to handle every derived shape in the same manner.

This section gives a top down description of the simulation software that was written. The event loop that initiates functions is described first, followed by a discussion of the C++ class hierarchy and a description of the more important C++ classes. The section concludes with a description on how frames are implemented.

The Event Loop

After the program initializes itself, the robot and the main menu are displayed as shown in Figure 1. The program enters a loop that waits for events and translates them into the appropriate action codes, namely ENTER_KEY and ESCAPE_KEY. An event is initiated either by the keyboard or the mouse. Each menu has its own encapsulated event loop that is defined as

```

key := NO_KEY;
while (key = NO_KEY) do
  begin
    key := poll_keyboard();    { else key = NO_KEY }
    if (key = NO_KEY) then
      begin
        get_mouse_coords(x, y);
        case (button_hit) of
          LEFT: if (coor_on_menu(x, y)) then
            begin
              highlight_selected_item(x, y);
              pause_for_button_release();
              if (left_button_released()) then
                key = ENTER_KEY;
            end;
          RIGHT:
            pause_for_button_release();
            if (right_button_released()) then
              key = ESCAPE_KEY;
            end; { case }
        end; { if }
      end; { while }
    return (key);    { the event found }
  end;
end;

```


The event loop function continually polls first the keyboard then the mouse. When the `ESCAPE_KEY` or `ENTER_KEY` event has occurred, it is passed back to the caller where the appropriate menu selection is initiated. The `ENTER_KEY` will cause the function corresponding to the currently selected menu item to be activated or cause a predefined submenu to be activated. The `ESCAPE_KEY` will remove the current menu unless the current menu is the main menu and then no action is taken. The `get_event` function can be found in the `menu.cpp` code module.

The Display Classes

The **VIEWPORT** class is responsible for defining the graphics area on the screen. The display class hierarchy is shown in Figure 17.

Figure 17. The display classes hierarchy.

The two important VIEWPORT methods are

```
void VIEWPORT::draw(void);
BOOL VIEWPORT::clip_segment(long *x1, long *y1, long *x2, long *y2);
```

The draw method simply erases the images within the defined viewport by filling the area with a default color. The clip_segment method is an implementation of the Cohen-Sutherland clipping algorithm. The complete implementation is found in viewport.cpp.

The GWINDOW class inherits from viewport class and contains data lists called the shape_cycle and the shape tree. Discussion of shapes and their classes is postponed until the next section. For a shape to appear on the screen it must be explicitly added to the world, which refers to the current list of shapes being displayed.

The SHAPE_CYCLE class corresponds to a circularly linked list of shape pointers that point to shapes in the world. The important methods are

```
void SHAPE_CYCLE::insert_shape(SHAPE *shape);
void SHAPE_CYCLE::remove_shape(SHAPE *shape);
virtual void SHAPE_CYCLE::draw(void);
```

The insert_shape method inserts the shape pointer into the shape pointer cycle. The remove_shape method removes the shape pointer from the shape pointer cycle. The shape cycle's draw method causes the shape

pointer cycle to be traversed and each shape's draw method to be invoked. The shape pointer cycle implementation is found in shape.cpp.

The TREE class corresponds to a tree of nodes that point to the shapes in the world and was created to allow logical grouping of shapes. The tree is organized into levels with each node defined as

```
typedef struct tree_node
{
    tree_node *parent;    // points to parent node
    tree_node *child;     // points to left most child of this
                        // node
    tree_node *next;      // points to next shape on the same
                        // level, this is used to link all the
                        // children of a node together
    SHAPE *shape;         // pointer to shape in the shape cycle
} TREE_NODE;
```

At any given time there is a current shape specified by a private field in the current TREE class. The current shape may be assigned using the method

```
void TREE::set_current_shape(SHAPE *shape);
```

The TREE class has two methods for manipulating the tree structure. The methods are

```
void TREE::insert(SHAPE *shape);  
void TREE::remove(SHAPE *shape);
```

The insert method places the shape into the level just below the current shape. This method sets a pointer within the shape to point back to the tree node where the shape has been inserted. The remove method removes a shape from the tree. The tree implementation is found in tree.cpp. The tree representation is shown in Figure 18.

The GWINDOW class also contains fields for the system camera, the camera_dirty flag, and the basic world draw method. The system camera is

Figure 18. The tree representation. Shape 1 and Shape 2 are on the same level in the shape tree. Shape 3 is one level below Shape 1. The current shape is Shape 3.

represented by an inverted frame and a viewing distance. These two data items are used by the graphics display system to map world coordinates to camera coordinates. The `camera_dirty` flag is true whenever the system camera has been moved and the graphics display has not yet been drawn. This is used for efficiency purposes to reduce the number of calculations needed to draw the world. The `draw` method is defined as

```
void GWINDOW::draw(void)
{
    VIEWPORT::draw();
    shape_cycle->draw();
    camera_dirty = FALSE; // indicates a freshly drawn world
}
```

The `PWINDOW` class inherits from the `GWINDOW` class and is used to create the perspective view of the world. The mathematics used to map the world coordinates to screen coordinates is fairly standard (Watt, 1993) and is not discussed here. All the methods for the `GWINDOW` and `PWINDOW` classes are found in `gwindow.cpp`.

There is a global pointer variable, `pwindow`, which points to the current `PWINDOW` object. It is used to draw the world and to manipulate the world camera. This and other global variable definitions are found in `globals.cpp`. To draw the world, one simply invokes the `draw` method for the `pwindow`'s shape cycle.

Figure 19. The shape class hierarchy structure.

The Shape Class

The **SHAPE** class is the base class to all displayed objects on the screen. The shape class hierarchy is shown in Figure 19. The primary function of the shape class is to provide a virtual interface for derivable classes and to store the data basic to all shapes. The display data for a shape is stored in a record of type DISPLAY and is defined as

```

typedef struct display_data
{
    char  name[LEN];           // name of the shape
    FRAME frame_of;           // frame for the shape

```

```

FRAME last_drawn_frame;    // frame when the shape was last
                             // drawn (needed for animation)

COLOR hue;                 // color of the shape

BOOL frame_dirty;         // This is true when a shape's frame
                             // was altered but the shape has not
                             // yet been drawn.

POINT *world_points;      // current shape points in world
                             // coordinates.

POINT *world_norms;       // current vertex and facet normals
                             // for the shape in world coordinates

BOOL animate;             // true when a shape is in animation
                             // mode.

BOOL first_draw;          // true when a shape is in animation
                             // mode and the world is being drawn
                             // for the first time.

} DISPLAY;

```

A shape has only two protected data fields

```

DISPLAY *display;
TREE_NODE *tree_position;

```

These pointers are inherited by the derived classes so the fields within the records can be updated.

A shape has methods for adding itself to the tree and the shape cycle. They are

```
void SHAPE::put(SHAPE *shape);  
void SHAPE::add(SHAPE *shape);  
void SHAPE::remove(SHAPE *shape);
```

The put method inserts the shape into the current pwindow's tree and shape cycle without invoking the pwindow's draw method. The add method performs the same action as the put method but invokes the pwindow's draw method. The remove method removes the shape from the current pwindow's tree and shape cycle and then invokes the pwindow's draw method.

There is a method within the shape class that allows a shape to change between animation mode and nonanimation mode, namely

```
void SHAPE::set_animation(BOOL value);
```

Typically, moving objects on the screen requires the entire world to be drawn. This is inefficient because the objects that have not been moved are still drawn. System performance relies greatly on the graphics library implementation and it was determined that Borland's implementation to clear a viewport was less than efficient. Therefore, it was necessary to develop this animation code to achieve a smooth motion for the robot.

When a shape is in animation mode, it is drawn in XOR mode. The XOR mode uses the XOR assembly language instruction to combine the line with the current screen. A programmer can erase the line by simply drawing it again. This is different than COPY mode where the MOV assembly language instruction is used to combine the line with the current screen. Animation mode allows one specific object to be drawn at a time. Animating a shape executes the following code sequence:

```
shape->set_animation(TRUE);
pwindow->draw();
operations to move the shape
shape->set_animation(FALSE);
pwindow->draw();
```

If any shape in the cycle is animated, then it is drawn after all other shapes are drawn. This allows the nonanimated shapes to be drawn in COPY mode and the animated shapes to be drawn in XOR mode. When a shape is animated, the shape's first_draw flag is set to true. This informs the shape's draw method to only draw the shape one time in XOR mode. The first_draw flag is then set to false. All subsequent invocations to the shape's draw method will draw the shape twice. The first draw is executed with the last_drawn_frame frame. The second draw is executed with the frame_of frame. This has the effect of erasing the current view of the shape and drawing the new view of the shape.

The SHAPE class has methods to directly manipulate the shape's position and orientation. They are

```
void SHAPE::set_frame(const FRAME &f);
void SHAPE::set_position(const POINT &p);
void SHAPE::set_x_rotation(double theta);
void SHAPE::set_y_rotation(double theta);
void SHAPE::set_z_rotation(double theta);
```

Whenever one of these methods is invoked, the frame_dirty variable is set to true. This flag is checked every time before a shape is drawn. If it is true, then the shape's world points and normals in its display record are first recalculated using the frame_of frame. The flag is then reset to false. The set_x/y/z_rotation methods perform frame rotations on the shape's frame. These methods call the corresponding frame procedures described later in the frames section.

One of the most important methods within the shape class is the draw method.

```
virtual void SHAPE::draw(void);
```

This is a virtual function and all shapes are drawn by invoking this method. The code for this procedure is potentially different for each derived shape type. For example, the robot, the tool, and the table each draw themselves in a different way.

The Dsurf Class

The **DSURF** class inherits directly from the shape class. This class is used to create discrete surfaces, to manipulate those surfaces, and to display those surfaces on the graphics screen. The DSURF implementation can be found in dsurf.cpp.

All objects displayed on the screen are descendants of the DSURF class. The simple DSURF objects are blocks, cylinders, cones, arms, and trunks. The trunk is a specialized arm for the robot. This is similar to the arm object except one end is flat. The trunk shape is used for link 3.

The DSURF object saves the point data of any discrete surface in the **cache** field. This is the only data field for this class and has the following structure

```
typedef struct cache_2d
{
    POINT *points;           // raw data points
    int num_points;
    POINT *normals;         // raw normals
    int num_normals;
    VERTEX *vertices;        // array of facet vertex structures
    int num_vertices;
    EDGE *edges;             // array of facet edge structures
    int num_edges;
```



```

typedef struct vertex_data
{
    int px;           // index into the points array
    int begin_facet;   // begin_facet and end_facet define a range
    int end_facet;     // of indices in the findices array that specify
                      // the indices of the facets incident to this
                      // vertex
    int normx;         // index into normals array
} VERTEX;

```

One may get a feel as to how the various fields of a `CACHE_2D` structure are related by reading the `block.cpp` file where the block cache is created. The points are coded either manually or algorithmically, depending on the complexity. The normals are formed by taking the cross product of the two consecutive edges of a facet which are assumed to be noncollinear.

A facet contains indices into the `vindices` array and the normals array. The `vindices` array specifies the vertices incident to the facet. An edge contains indices into the points array. A vertex contains an index into the points array and the normals array. It also has indices into the `findices` array which specify the facets incident to the vertex. This data structure was chosen because certain relationships, such as finding the vertices of a facet or finding all the facets incident to a given vertex, are easily obtained.

Each derived class has its own cache creating method. Once the cache is created, it is never changed. The world coordinates for the shape

are stored in the shape's display record after moving the cache points using the move function with the shape's frame.

When a DSURF object is instructed to draw itself, it first performs back face elimination calculations because only front facing facets are drawn. To determine if a facet is front facing, the facet's normal is checked to determine if it is pointing toward the system camera or away from the system camera. The normal's direction is determined by the dot product of the normal and the system camera's view direction. If the dot product is negative, then the normal is pointing toward the system camera and the facet is marked visible. Otherwise, the facet is marked not visible. Each edge within the shape is marked visible if it belongs to a visible facet.

The DSURF class invokes the pwindow methods necessary to map the world points for its visible edges to the screen. The mapped line segments are then clipped against the viewport. If the line segment is still visible within the viewport, then the Borland routine line is called with the endpoints of the line segment as parameters. The line segment will then be seen on the screen. As indicated earlier, the points only need to be recomputed when the frame_dirty flag is true.

The DSURF class also has methods to display the facet normals and the vertex normals. They are

```
void DSURF::display_facet_normals(void);  
void DSURF::display_vertex_normals(void);
```

When the display_facet_normals method is invoked, each facet normal is checked to determine if it is emanating from a visible facet. If this is the case, then the normal vector is passed to the pwindow's show_vector method and the normal is drawn on the screen. The show_vertex_normals method performs the same operations except it does so for the vertex normals on each vertex within a visible facet.

Derived Dsurf Classes

There are several derived dsurf classes. These are the block, cylinder, cone, arm, and trunc classes. The arm and the trunc are specialized shapes for the Puma 560 robot. Each of these classes have a create_cache method. This method dynamically allocates the memory for the points, normals, edges, and facets in the dsurf cache field. The specifics of each specific shape's create_cache method can be found in its own code module. For example, the block's create_cache method is found in block.cpp.

Complex Shapes

Complex shapes are composed of one or more derived DSURF classes. Examples of these types of shapes are the room, the manipulator that represents joint 4, the tool, and the robot itself. Each component shape within the complex shape has its frame defined relative to the frame for the

complex shape. The best example of this is the robot class. The robot contains arms, trunks, cylinders, and manipulators that model the physical robot. Each of these objects has a frame that is defined relative to a joint frame. When a joint is moved, the frame of the shape that represents the joint is composed with the joint frame. The shape can then be drawn in the correct position and orientation.

The Tool Class

The TOOL class inherits directly from the DSURF class and is a complex shape. The TOOL class has a cone and a cylinder that define the tool's shape. The tool has a frame called the mount_frame. The method

```
void TOOL::get_mount_frame(FRAME &mount_frame)
```

is used to access the mount_frame when the robot is commanded to pick up a tool. The robot uses this frame to generate a goal frame for the inverse kinematics routines. The tool class has another method

```
double TOOL::get_tool_height(void)
```

used to determine the relative distance from the frame defining joint 6 to the tip of the tool. The tool in the simulation has its orientation matching that of joint 6. Therefore, only the tool height is needed. In a more general case

there would be a method to access the frame that is defined relative to joint 6 when attached to the robot.

The Robot Class

The robot is complex shape and is composed of cylinders, an arm, a trunc, and a manipulator. The entirety of the mathematics described in the kinematics, inverse kinematics, and motion control sections are encapsulated within the robot class. Therefore, the details are hidden from the programmer.

The constants and data fields within the robot class are

```
#define NUM_LINKS 8           // 6 + base + tool
LINK_PARAM puma[NUM_LINKS]; // physical parameters
                             // and joint angles
POINT current_pos[NUM_LINKS]; // data for linear watch window
FRAME zero_frame;            // the base frame
FRAME goal_frame;
FRAME station_frame;
INV_KIN_ANGLES inv_solutions; // solution set to inv. kin.
JOINT_VIOLATION joint_violation[NUM_LINKS];
                             // define joint violations
double weighted_err[NUM_LINKS]; // weighted error for each joint
SHAPE *tool;                  // valid when a tool is attached
```

The data types introduced here are explained in the following paragraphs.

The Puma 560 kinematics are defined within the robot class as member functions. The joint angles are dynamic variables. Therefore, the frames are dynamic also. The link parameters are stored in the LINK_PARAM structure. The LINK_PARAM type is defined as

```
typedef struct link
{
    double link_twist;      // in radians
    double link_length;    // in mils
    double link_offset;    // in mils
    double joint_angle;    // in radians
} LINK_PARAM;
```

The first three fields in the LINK_PARAM structure are hard coded numbers and are defined by the physical mechanics of the robot. The array of links is initialized by invoking the init_puma method defined as

```
void ROBOT::init_puma(void);
```

This is part of the robot constructor.

Each joint has a frame function associated with it. By accessing these joint frames and composing the frames sequentially from the base frame to the wrist frame, the robot class can determine the instantaneous

position and orientation of each joint and any attached tool. For example, the function `get_frame_2T3` is defined as

```
void ROBOT::get_frame_2T3 (FRAME &frame)
{
    cosine = cos(puma[LINK3].joint_angle);
    sine   = sin(puma[LINK3].joint_angle);
    a2     = puma[LINK3].link_length;
    d3     = puma[LINK3].link_offset;
    make_3d_frame (cosine,   -sine,   0,   a2,
                  sine, cosine,   0,   0,
                  0,        0,   1,   d3, frame);
}
```

The frame created is of the same form as Equation 12. Each time the robot draws itself, the kinematics calculations are performed to define a frame for each shape that makes up the robot. The `current_pos` array that holds the Cartesian positions is also updated. When a watch window is active, the data stored in this array is displayed in the watch window.

The motion system is composed of the inverse kinematics equations and path generating code. The set of member functions used for a linear move are

```
int ROBOT::move_to(FRAME dest_frame);
void ROBOT::generate_inv_kinematic_frame(FRAME &target_frame);
```

```

int ROBOT::calc_position_list(POINT &start_point, POINT &end_point);
int ROBOT::interp_frame(FRAME &start_frame, FRAME &end_frame,
                        FRAME &temp_frame, int index);
int ROBOT::solve_inverse_kin(FRAME &target_frame, int set_angles,
                             ANGLE_SET angles);

```

The programmer need only request the robot to perform a linear move by invoking the move_to member function with the destination frame defined relative to the station frame. The move_to function invokes the generate_inv_kinematic_frame function. This member function generates the wrist frame, 0T_6 , needed for the inverse kinematics calculations. The position vectors from the destination wrist frame and the current wrist frame are used as parameters to the calc_position_list routine. This routine generates a point list as described in the Path Generation and Motion Control section. A wrist frame is created for each point using the interp_frame routine. Each point within the motion list is defined on the discrete time interval.

The solve_inverse_kin member function has three parameters. The first parameter is the intermediate wrist frame. The second parameter informs the inverse kinematics code either to set the joint angles when found or return the joint angles in the third parameter. The linear motion allows the inverse kinematics equations to set the joint angles. The robot is drawn in the new configuration after all joint angles have been assigned. The robot's end effector will appear to move in a straight line. The specifics

of the solve_inverse_kin routine are postponed until after the motion description is complete.

The programmer may request the robot to perform a rotational move by invoking the rotate_to member function. The member functions used for a rotational move are

```
int ROBOT::rotate_to(FRAME &goal_frame, ANGLE_SET &angles,
                    int force_angles);

int ROBOT::calc_angle_list(ANGLE_SET &curr_angles,
                          ANGLE_SET &goal_angles);
```

along with the member functions discussed for the linear motion. The `force_angles` parameter specifies which of the first two parameters to use during the motion definition. If the goal frame is to be used, then the solve_inverse_kin method is used to determine the goal joint angles. In this case, the angle set is returned from the inverse kinematics equations. If the second parameter is to be used, then the goal angles are known and the solve_inverse_kin method is not executed.

Once the goal angles have been found, the calc_angle_list method is invoked to create an angular motion list using the current set of joint angles and the goal joint angles as parameters. Each entry in the motion list specifies a set of joint angles to be assigned to the robot at the specified discrete time interval. The robot is drawn in the new configuration after each joint angle set has been assigned. The robot appears to rotate to its goal configuration.

The solve_inverse_kin member function invokes the routines that solve the inverse kinematics for a specific target frame. These member functions are

```
int ROBOT::solve_theta_1(const FRAME &target_frame);
int ROBOT::solve_theta_3(const FRAME &target_frame);
int ROBOT::solve_theta_2(const FRAME &target_frame);
int ROBOT::solve_theta_4(const FRAME &target_frame);
int ROBOT::solve_theta_5(const FRAME &target_frame);
int ROBOT::solve_theta_6(const FRAME &target_frame);
int ROBOT::find_best_solution(FRAME &target_frame, int set_angles,
                             ANGLE_SET &angles);
```

The member functions for joint 1 and joint 3 solve their respective equations twice. The other four equations must loop through the partial solution set to generate the required solutions. For example, when solving for joint angle θ_2 , the loop structure is coded as

```
solution_num := SOLUTION1;
for i := SOLUTION1 to SOLUTION2 do
  for j := SOLUTION1 to SOLUTION2 do
    begin
      c1 := cos(inv_solutions[i].theta_1)
      s1 := sin(inv_solutions[i].theta_2)
      c3 := cos(inv_solutions[j].theta_3)
```

```

s3 =: sin(inv_solutions[j].theta_3)
sin( $\theta_{23}$ ) := equation in inverse kinematics section
cos( $\theta_{23}$ ) := equation in inverse kinematics section
 $\theta_{23}$  := atan2 (sin( $\theta_{23}$ ), cos( $\theta_{23}$ ))
inv_solutions[solution_num].theta_2 :=
     $\theta_{23}$  - inv_solutions[j].theta_3
solution_num := solution_num + 1;
end;

```

Similar control loops exist for the inverse kinematics equations for joint angles θ_4 , and θ_5 , and θ_6 .

The inv_solutions data structure is defined as

```

#define MAX_SOLUTIONS 8
typedef struct inv_sol_data
{
    double theta_1[MAX_SOLUTIONS];
    double theta_2[MAX_SOLUTIONS];
    double theta_3[MAX_SOLUTIONS];
    double theta_4[MAX_SOLUTIONS];
    double theta_5[MAX_SOLUTIONS];
    double theta_6[MAX_SOLUTIONS];
} INV_KIN_SOLUTIONS;

```

After the `inv_solutions` field is assigned, as described in the inverse kinematics section, the best solution is selected by invoking the `find_best_solution` method. The target frame is used in the calculations to determine which inverse kinematics solution to select. The following code segments show how a solution is selected.

```

min_err := +100000000;
iter := SOLUTION_1;
for i:= SOLUTION_1 to SOLUTION_2 do
  begin
    for j := SOLUTION_1 to SOLUTION_2 do
      begin
        t1 := inv_solutions.theta_1[i];
        t3 := inv_solutions.theta_3[j];
        t2 := inv_solutions.theta_2[iter];
        t4 := inv_solutions.theta_4[iter];
        t5 := inv_solutions.theta_5[iter];
        t6 := inv_solutions.theta_6[iter];
        iter := iter + 1;
        if not (violations_found()) then
          begin
            err = calc_weighted_error(t1, t2, t3, t4, t5, t6);
            if (err < min_error) then
              begin
                min_err := err;

```



```

        solve_t1 := t1;
        solve_t2 := t2;
        solve_t3 := t3;
        solve_t4 := t4;
        solve_t5 := t5;
        solve_t6 := t6;

    end;

end; { end if no violations found }

end; { end inner for }

end; { end main for }

```

The same code sequence is executed for solutions 5 - 8, then the following code is executed

```
angles.theta[0] := solve_t1;      { the proper solution set }
angles.theta[1] := solve_t2;
angles.theta[2] := solve_t3;
angles.theta[3] := solve_t4;
angles.theta[4] := solve_t5;
angles.theta[5] := solve_t6;
```

```
if (set_angles) then
begin
    err = calc_weighted_error(solve_t1, solve_t2, solve_t3, solve_t4,
                             solve_t5, solve_t6);
```

```

if (err > max_allowable_error) then
    begin
        error("A motion discontinuity has been reached");
        return(ERROR);
    end
else
    begin    // assign the angles directly
        set_joint_angle(LINK_1, solve_t1);
        set_joint_angle(LINK_2, solve_t2);
        set_joint_angle(LINK_3, solve_t3);
        set_joint_angle(LINK_4, solve_t4);
        set_joint_angle(LINK_5, solve_t5);
        set_joint_angle(LINK_6, solve_t6);
    end;
end;

```

After the find_best_solution method is complete, the joint angles have either been assigned directly to the robot or the joint angles have been passed back in the third parameter. The motion discontinuity occurs when a linear move commands the robot to move into a location that causes a joint violation.

The violations_found condition above is an abstraction for the following test

```

if (joint_violation[i].active = TRUE) then
  begin
    if (ti > joint_violation[i].lower and
        ti < joint_violation[i].upper) then
      violation := TRUE;           { so ignore the ith solution }
    end;
  end;

```

where the joint_violation data structure is defined as

```

typedef struct joint_vio_data
{
  double lower;
  double upper;
  int active;
} JOINT_VIOLATION;

```

The calc_weighted_error method is defined as

```

double ROBOT::calc_weighted_error(double &t1, double &t2, double &t3,
                                   double &t4, double &t5, double &t6);

```

and does the following for each joint angle

```

delta := abs ( puma[LINK_i].joint_angle - t_i);
if (delta >  $\pi$ ) then
    delta :=  $2\pi$  - delta;
else if (delta <  $-\pi$ ) then
    delta :=  $2\pi$  + delta;
err[i] := weighted_err[LINK_i] * delta;

```

Whenever the robot is moved on the screen, it is placed into animation mode. If a tool is picked up by the robot, the robot executes the required operations to place the tool into animation mode. The robot will also place the tool into normal draw mode after it replaces the tool in the tool rack.

Frame Operations

The frame operations are located in the frame.cpp code module. Although frames were represented by 4 x 4 homogeneous matrices in the mathematical discussion, the actual data structure used to store a frame is a 3 x 4 matrix. The type definition is

```
typedef struct frm
{
    double r00, r01, r02, xtr;
    double r10, r11, r12, ytr;
    double r20, r21, r22, ztr;
} FRAME;
```

The frame operations are given as

```
void move(POINT p, FRAME f, POINT *new_point);
void compose(FRAME f1, FRAME f2, FRAME *new_frame);
void invert_frame(FRAME f, FRAME, *inv_f);
```

There are several support functions for manipulating individual frames. These functions are

```
void get_frame_position(POINT &p, const FRAME &f);
void set_frame_position(const POINT &p, FRAME &f);
void set_frame_x_rotation(double theta, FRAME &f);
void set_frame_y_rotation(double theta, FRAME &f);
void set_frame_z_rotation(double theta, FRAME &f);
```

The get_frame_position procedure returns the position vector of a frame in parameter p. The set_frame_position procedure will assign a position vector p to the frame parameter. The set_frame_x/y/z_rotation procedures offer a

simple way to assign an orientation to a frame with little programming effort. The programmer can assign multiple rotations to a frame by using any combination of these functions. The result is a frame that contains the rotation about the specified axis. This is used to initialize frames within the displayed objects in the simulation. For example

```
set_frame_x_rotation(45, f);
```

will rotate the frame f by 45 degrees about the x axis.

Future Development

One obvious additional requirement of this simulation, if it were to be seriously used in industry, is an interactive control language and interpreter. A customer buying a robot has a specific application in mind. There must be a way to easily tell the robot what to do. Typically a control program is used. The software system needs a built in interpreter that reads the control program and relays the instructions to the robot. This was the intent of the Program menu item under the main menu.

This simulation assumed the motors used for the robot were stepper motors. Therefore, calculating the torque required for each joint was unnecessary because torque is predefined by the motor size and the voltage driving the motor. However, many industrial robots use servo motors and such calculations may be required. The mathematics for calculating torque,

instantaneous velocity, and instantaneous acceleration are discussed in Craig (1989). This data would be of great use for system engineers during system development and system debugging.

The menu system has a Learn function under the Operation menu item. This is also needed for an industrial application. This would allow the user to manually move the robot to a goal frame or to specify a goal frame. These goal frames could then be saved to disk as a complete user defined motion. The learned motion could then be executed by the Program menu item.

Conclusions

This thesis has presented the simulation program written for the Puma 560 robot designed by Unimation Incorporated. The supporting kinematics and inverse kinematics mathematics were described in detail. Further reading can be found in Craig (1989). The mathematics are encapsulated within the robot class. This allowed a natural way to incorporate the robot into the graphics pipeline for rendering, ensuring the sensitive calculations performed by the robot class are hidden from the programmer. The trajectory calculations were designed to mock a real-time system such that a future real-time implementation could be possible with little recoding effort.

References

1. Blinn, Jim, Nested Transformations and the Blobby Man, *IEEE Computer Graphics and Applications*, October 1987, pp. 59 - 65.
2. Craig, John J., *An Introduction to Robotics, Mechanics and Control*, Addison Wesley Publishing Corp., Massachusetts, 1989.
3. Watt, Alan, *3-D Computer Graphics*, Addison-Wesley Publishing Company, Menlo Park, CA, 1993.

Appendix A

This is a list of notation used in the body of this paper.

e1 : The standard unit vector in Cartesian space. This is the vector $\langle 1, 0, 0 \rangle$.

e2 : The standard unit vector in Cartesian space. This is the vector $\langle 0, 1, 0 \rangle$.

e3 : The standard unit vector in Cartesian space. This is the vector $\langle 0, 0, 1 \rangle$.

atan2 : An extended function for the arc tangent calculation. Atan2 finds the arc tangent of the two signed arguments but returns the angle corresponding to the proper quadrant. This is necessary since joint angles vary from 0 - 360 degrees.

Specifically

$$\begin{aligned} \text{atan2}(y, x) = & \text{undefined, if } x = y = 0, \\ & \pi / 2, \quad \text{if } x = 0 \text{ and } y > 0, \\ & -\pi / 2, \quad \text{if } x = 0 \text{ and } y < 0, \\ & 0, \quad \text{if } y = 0 \text{ and } x > 0, \\ & \pi, \quad \text{if } y = 0 \text{ and } x < 0, \text{ and} \\ & \phi, \quad \text{where } -\pi < \phi < \pi, \\ & \tan \phi = y / x, \text{ and } \phi \text{ lies in} \\ & \text{the same quadrant as } (x, y) \end{aligned}$$

c_i : An abbreviation for $\cos(\theta_i)$.

- s_i : An abbreviation for $\cos(\theta_i)$.
 c_{ij} : An abbreviation for $\cos(\theta_i + \theta_j)$
 s_{ij} : An abbreviation for $\sin(\theta_i + \theta_j)$
 $\mathfrak{R}^3$: Denotes three dimensional Euclidean space.

Appendix B

This is a list of the source code modules (.cpp) and include files (.h) written for the Puma 560 simulation.

<u>.CPP file</u>	<u>.H file</u>	<u>Functions</u>
	Const.h	System constants
	Colors.h	System color definitions
	Gtypes.h	Types used for graphics calculations
	Keydef.h	IBM PC key stroke definitions
	Placement.h	Physical parameters for all shapes in the world
Arm.cpp	Arm.h	Shape class for robot link 2
Block.cpp	Block.h	Shape class for a block
Cone.cpp	Cone.h	Shape class for a cone
Cylinder.cpp	Cylinder.h	Shape class for a cylinder
Dsurf.cpp	Dsurf.h	Dsurf shape class
Frame.cpp	Frame.h	Frame operations
Front.cpp	Front.h	Graphical user interface functions
Gfunc.cpp	Gfunc.h	General vector functions
Global.cpp	Global.h	Global variables
Gwindow.cpp	Gwindow.h	Graphics window class
Help.cpp	Help.h, Helpdoc.h	On-line help system

Menu.cpp	Menu.h	Menu class
Motion.cpp	Motion.h	Path generation functions
Robot.cpp	Robot.h	Shape class for the robot class
Rodent.cpp	Rodent.h	Mouse functions
Room.cpp	Room.h	Shape class for the room
Rotator.cpp	Rotator.h	Shape class for robot link 4
Shape.cpp	Shape.h	Shape class
Sound.cpp	Sound.h	Sound functions
Table.cpp	Table.h	Shape class for the table
Tool1.cpp	Tool1.h	Shape class for the tool
Tree.cpp	Tree.h	Tree class
Trunc.cpp	Trunc.h	Shape class for robot link 3
Userio.cpp	Userio.h	User prompt functions
Viewport.cpp	Viewport.h	Viewport class