HUMAN MACHINE COOPERATIVE TELEROBOTICS

FINAL TECHICAL/TOPICAL REPORT

Work Performed under contract:

DE-AR26-97FT34315

September 9, 1997 through June 30, 2003

Prepared for:

U.S. Department of Energy

National Energy Technology Laboratory

Morgantown, West Virginia 26507-0880

COR: V. P. Kothari < vijendra.kothari@netl.doe.gov>

Prepared by:

William R. Hamel, Principal Investigator <whamel@utk.edu>

Spivey Douglass

Sewoong Kim

Pamela Murray

Yang Shou

Sriram Sridharan

Ge Zhang

University of Tennessee

414 Dougherty Engineering Building

Knoxville, TN 37996-2210

Scott Thayer

Carnegie Mellon University

Field Robotics Center

Pittsburgh, PA

Rajiv V. Dubey

University of South Florida

Tampa, FL

July 23, 2001

DISCLAIMER

TABLE OF CONTENTS

LIST OF FIGURES

1. BACKGROUND

The remediation and deactivation and decommissioning (D&D) of nuclear waste storage tanks using telerobotics is one of the most challenging tasks faced in environmental cleanup. Since a number of tanks have reached the end of their design life and some of them have leaks, the unstructured, uncertain and radioactive environment makes the work inefficient and expensive. However, the execution time of teleoperation consumes ten to hundred times that of direct contact with an associated loss in quality. Thus, a considerable effort has been expended to improve the quality and efficiency of telerobotics by incorporating into teleoperation and robotic control functions such as planning, trajectory generation, vision, and 3-D modeling. One example is the Robot Task Space Analyzer (RTSA), which has been developed at the Robotics and Electromechanical Systems Laboratory (REMSL) at the University of Tennessee in support of the D&D robotic work at the Oak Ridge National Laboratory and the National Energy Technology Laboratory. This system builds 3-D models of the area of interest in task space through automatic image processing and/or human interactive manual modeling. The RTSA generates a task plan file, which describes the execution of a task including manipulator and tooling motions. The high level controller of the manipulator interprets the task plan file and executes the task automatically. Thus, if the environment is not highly unstructured, a tooling task, which interacts with environment, will be executed in the autonomous mode. Therefore, the RTSA not only increases the system efficiency, but also improves the system reliability because the operator will act as backstop for safe operation after the 3-D models and task plan files are generated. However, unstructured conditions of environment and tasks necessitate that the telerobot operates in the teleoperation mode for successful execution of task. The inefficiency in the teleoperation mode led to the research described as Human Machine Cooperative Telerobotics (HMCTR). The HMCTR combines the telerobot with robotic control techniques to improve the system efficiency and reliability in teleoperation mode.

In this topical report, the control strategy, configuration and experimental results of Human Machines Cooperative Telerobotics (HMCTR), which modifies and limits the commands of human operator to follow the predefined constraints in the teleoperation mode, is described. The current implementation is a laboratory-scale system that will be incorporated into an engineering-scale system at the Oak Ridge National Laboratory in the future.

2. HUMAN MACHINE COOPERATIVE TELEROBOTICS

2.1 The HMCTR Concept

When the task environment of telerobotics is not highly complex, the task can be performed in the autonomous mode and there is no need for human intervention. If the environment is highly unstructured and the task requires interaction between the tool and the environment, the task cannot be executed in the autonomous mode due to the increased complexities of tasks and errors in the model of the environment. In the latter situation, the task must be executed in the teleoperation mode, and the human abilities of cognition, creativity, and innovation can counteract the certain occurrence of unexpected events. However, in the teleoperation mode tasks contacting with the environment using tools, such as bandsaws and drills, make the maneuvering of the manipulator difficult. This difficulty makes the human operator fatigue easily. Fatigue then decreases the efficiency and performance of the human operator, and task execution time consequently increases. There is therefore a need to improve the system maneuverability and efficiency in teleoperation by exploiting the robotic control techniques. These robotic motion techniques, which are called assist functions in the HMCTR, enhance the human command and improve the system efficiency by modifying the human command.

In the HMCTR, three assist functions – position assist, velocity assist, and force assist – have been developed based on the requirements of a nuclear tank clean-up. Each of these functions can be selected and used on demand by the operator. In the position assist function, the constraints are either linear (linear assist function) or planar (planar assist function). The position assist functions constrain the end-effector to a line or plane during the execution of the task by changing the scaling factor of the velocity components. For example, the linear assist function is used during drilling to make the tool move along the constrained line (the axis of the tool bit). The planar assist function is used during pipe cutting to ensure that the bandsaw moves only in the cutting plane. These assist functions reduce the level of operator effort required by decreasing the degrees of freedom that must be controlled by the human operator. The velocity assist function is used to decrease the movement time of the end-effector by increasing its velocity when it is moving in free space or is far away from the task location and by decreasing its velocity when it approaches the task location or an obstacle. This function allows the operator to control a faster moving manipulator without fear of impacting it or of overshooting the task location. The force assist function keeps the individual contact forces and moments within pre-defined bounds when the task requires the manipulator to

make contact with the environment. For example, in a drilling task, the force assist function maintains a constant force component in the drilling direction and zeroes the other force and moment components. Therefore, the force assist function prevents damage to the environment and the tool, improves the operator efficiency, and improves work quality. The HMCTR is designed to work with the Robot Task Space Analyzer (RTSA), which has also been developed at the Robotics and Electromechanical Systems Laboratory at the University of Tennessee. The RTSA first builds a 3-D model of the task space. Using the RTSA, the human operator can then define the tasks and select the proper assist, or autonomous, function for each task. Finally, the RTSA creates the task plan file. The high level controller in the HMCTR interprets the task plan file and activates the proper assist functions during the teleoperation, or executes the task plan autonomously.

2.2 System Configuration

The developed telerobotic system, shown in Fig. 1, consists of two main components; the Robot Task Space Analyzer (RTSA) and the Human Machine Cooperative Telerobot (HMCTR). In the RTSA, the human operator selects a region of interest (ROI) in the task space of the robot and captures the image using the stereo camera pair (not shown in the Figure 1). The captured image can be sent to the automated image reasoning process to build a 3-D model of the ROI if the operator believes that route is feasible. Otherwise, the 3-D model of the environment can be generated by a human operator using the interactive manual modeling Graphical User Interface (GUI) of the RTSA. Then, the operator builds the task plan file, which describes the execution of task, including the manipulator and tooling motions, using the task planner. The task planner is an interactive GUI system, and the operator is only required to select the part and tooling position on the 3-D model of the ROI. Detailed data, such as the required position and rotation of the end-effector or the manipulating procedure, are automatically generated by the task planner of RTSA. These autonomous functions of RTSA increase the efficiency of the task execution and improve the system reliability by allowing the operator to check for safe operation by examining the 3-D models and the task plan file generated.

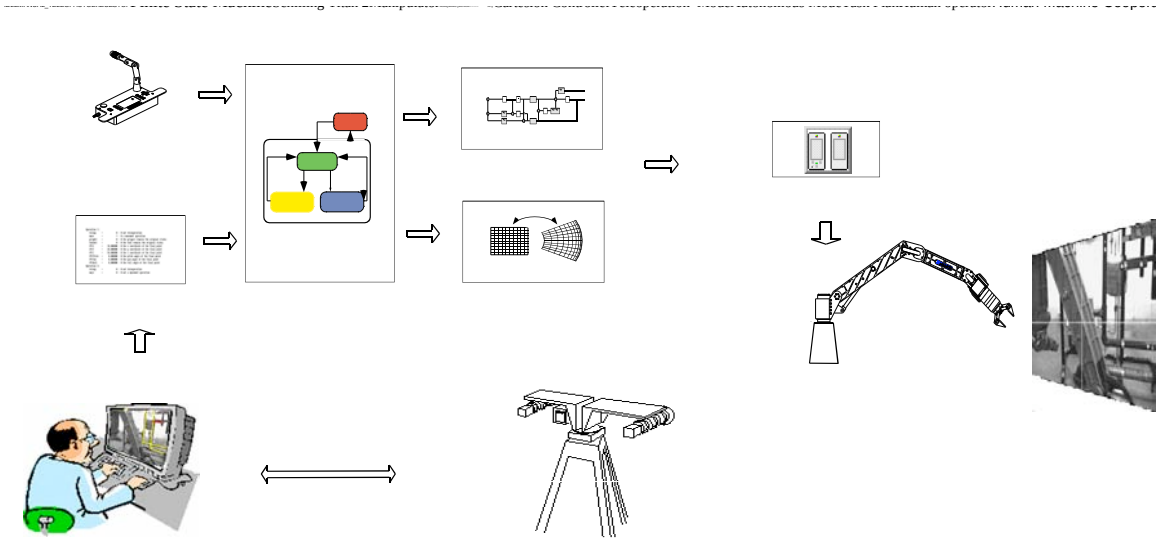


Figure 1 Configuration of the HMCTR.

The control scheme of HMCTR, which introduces the assist functions into the teleoperation mode, is shown in Figure 2. The HMCTR consists of a finite state machine (FSM), interpreter, and assist functions. The FSM selects the operating mode, and the interpreter activates the functional blocks that correspond to the operating mode. In the autonomous mode, HMCTR transforms the commands, which are listed in the task plan file, from Cartesian space to manipulator joint space and sends them to the low-level controller. In the teleoperation with assist function mode, the human operator generates the commands, which are increments of joint angles, using the mini-master. The assist functions modify the commands to limit the action of the end-effector and thereby satisfy the predefined constraints on the end-effector. The assist functions consist of a linear assist function, which keeps the end-effector in the line, a planar assist function, which keeps the end-effector on the plane, and a force assist function, which limits the contact forces to prescribed values.

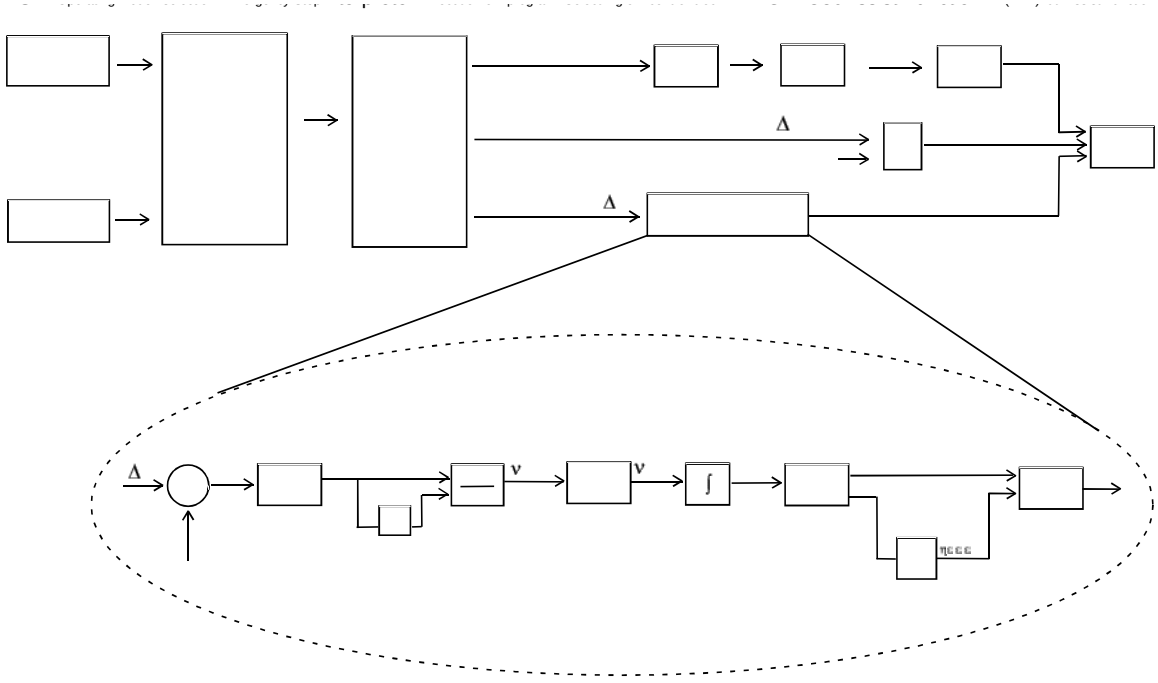


Figure 2 Human machine cooperative telerobotics control scheme.

2.3 Task Planning

Graphical user interface

The graphical user interface of the task planner allows the operator to plan a set of tasks for the system to execute. The interface seeks to maximize user efficiency by minimizing input time and knowledge needed to run the system accurately. Based on these requirements, a basic task planner design was constructed, which prompts the operator through the required inputs with a minimum number of windows. The output of the task planner is a list of actions, which are downloaded as a text file to the real-time control computer. A sample file is shown in Figure 3. Each action in the task plan file gives Boolean information about the gripper state, time duration, and position. The format of the file is such that it can be parsed by a state transition component in ControlShell called 'fetchandparse.' A trajectory generator there creates a smooth motion along a path from the manipulator's current position to the given final position.

```
1  move  x 1420.8  y 395.5  z 1577.1  roll  9.0  pitch  3.7  yaw -1.0  Gripper 70  TimeDuration 60
2  move  x 1519.4  y 411.2  z 1570.6  roll  9.0  pitch  3.7  yaw -1.0  Gripper 70  TimeDuration 60
3  move  x 1617.9  y 426.8  z 1564.2  roll  9.0  pitch  3.7  yaw -1.0  Gripper 70  TimeDuration 60
4  move  x 1519.4  y 411.2  z 1570.6  roll  9.0  pitch  3.7  yaw -1.0  Gripper 70  TimeDuration 60
5  move  x 1420.8  y 395.5  z 1577.1  roll  9.0  pitch  3.7  yaw -1.0  Gripper 70  TimeDuration 60
6  move  x 1420.8  y 395.5  z 1577.1  roll  9.0  pitch  3.7  yaw -1.0  Gripper 70  TimeDuration 60
7  move  x 1519.4  y 411.2  z 1570.6  roll  9.0  pitch  3.7  yaw -1.0  Gripper 70  TimeDuration 60
8  move  x 1617.9  y 426.8  z 1564.2  roll  9.0  pitch  3.7  yaw -1.0  Gripper 70  TimeDuration 60
9  move  x 1519.4  y 411.2  z 1570.6  roll  9.0  pitch  3.7  yaw -1.0  Gripper 70  TimeDuration 60
10 move  x 1420.8  y 395.5  z 1577.1  roll  9.0  pitch  3.7  yaw -1.0  Gripper 70  TimeDuration 60
```

Figure 3 Sample task plan.

The basic structure of the task planner is shown in Figure 4. As shown, the task planner allows all possible tooling scenarios to be planned from twelve simple self-explanatory windows. Prior to entering the task planner a manual model of the cut area must be constructed.

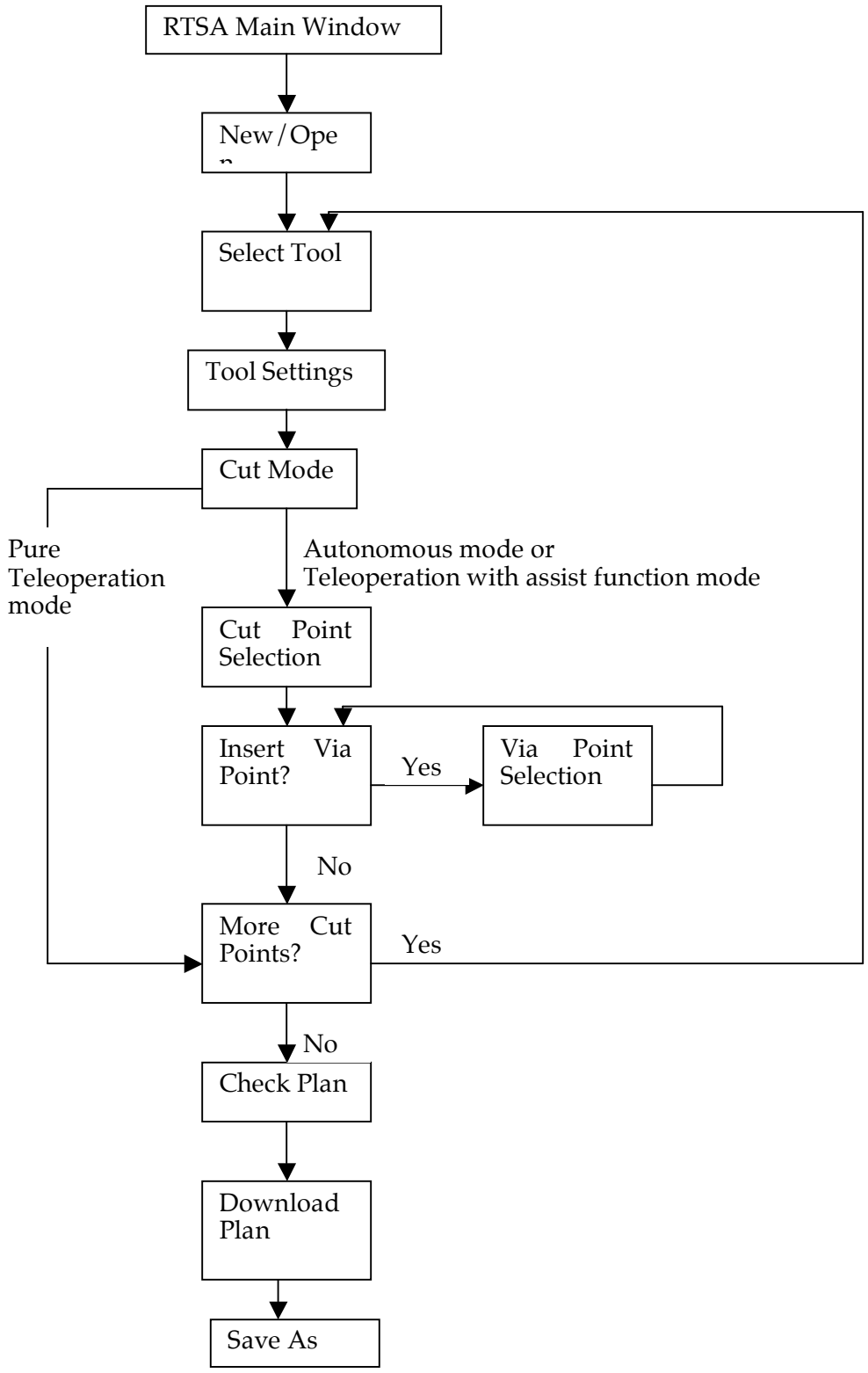


Figure 4 Task Planner Scheme

The task planner is entered through the Main RTSA window (Figure 5).

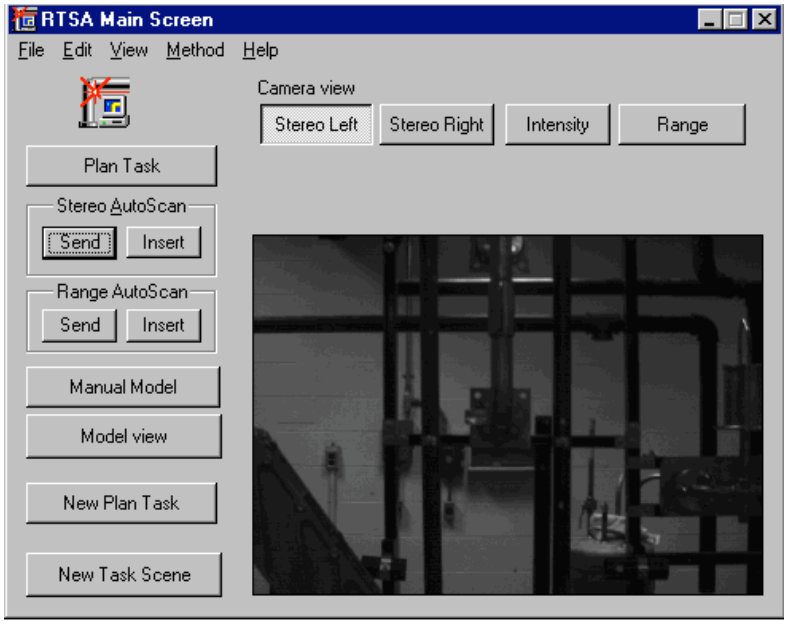


Figure 5 Main RTSA Window

New/Open window

Then the user begins a new task plan file using the New/Open window (Figure 6). Eventually, this window will be expanded to allow the user to edit previously created files.

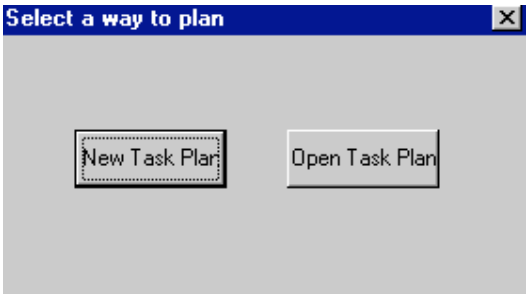


Figure 6 New/Open Window

Select tool window

The Select tool window (Figure 7) allows the tool to be used in the tooling operation to be selected.

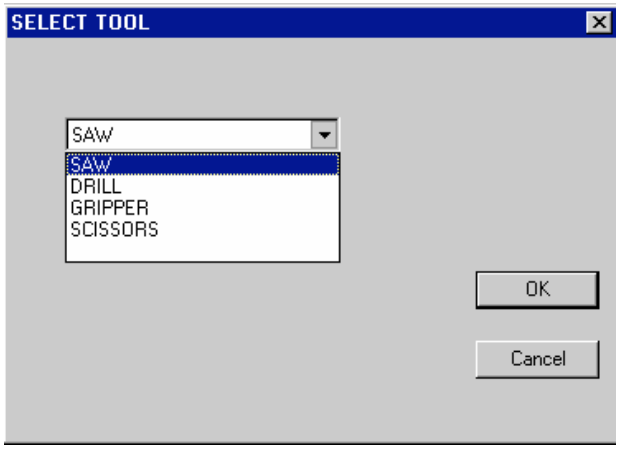


Figure 7 Select Tool Window

Tool settings window

Then, in the Tool settings window (Figure 8), the tool specific parameters for that operation are selected.

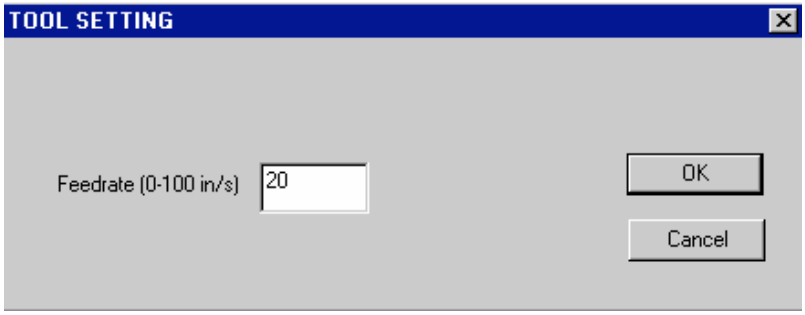


Figure 8 Tool Setting Window

Mode selection window

The Mode selection window (Figure 9) is used to select the mode of operation. Choices are autonomous mode, teleoperation using an assist function, or pure teleoperation. Linear, planar, and velocity assist functions are available. The assist functions interact with the task planner and allow

motion constraints to be implemented during teleoperation. These functions are useful during tooling tasks such as maintaining alignment with bolts during removal. They are defined in the task plan and implemented in the real-time controller.

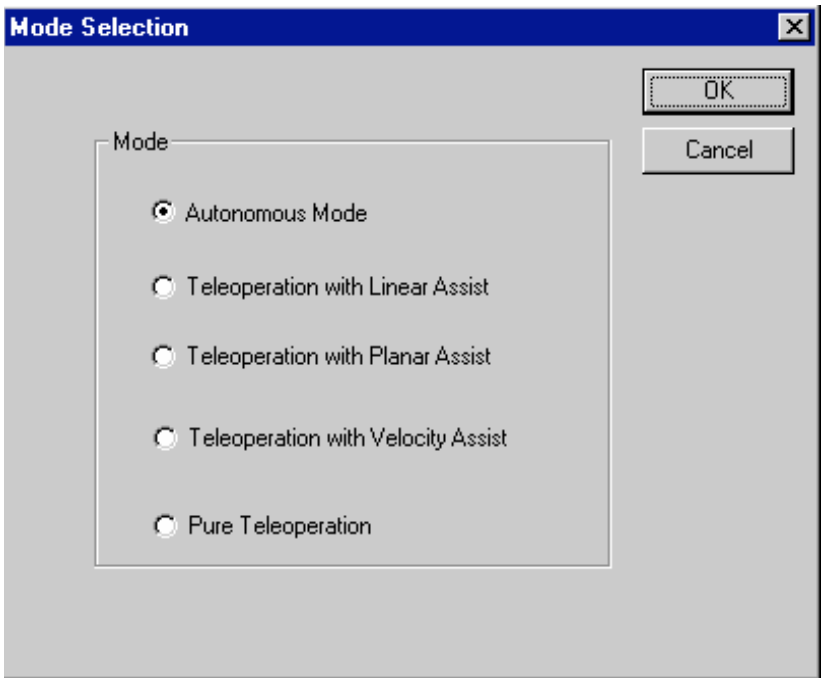


Figure 9 Mode Selection Window

Cut point selection window

Next, in the Cut point selection window (Figure 10), the cut plane is selected from the model previously created. At the top of this window is a drop list used to select the part that is to be cut. When a part is selected, it is highlighted in red in the OpenGL window and a cutting plane in the shape of a flat arrow appears along the axis of the part. Buttons on the window allow the operator to translate the cutting plane along the z-axis of the part, and to rotate the direction of the approach around the z-axis. When the selections have been made, the operator stores this point. The cut point is calculated from the selection of a cutting plane and direction. Based on the cut point, approach points and final points are calculated. The approach point is an intermediate point at which velocity scaling can begin. The final point is where the cut will end.

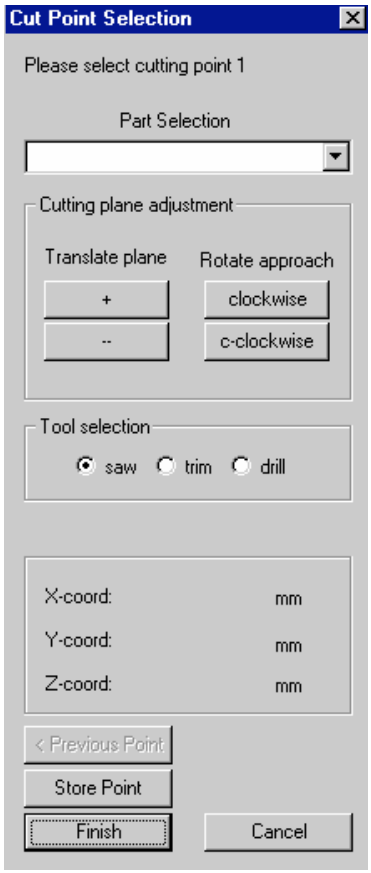


Figure 10 Cut Point Selection Windows

Via point windows

Once the cutting point has been selected, any needed via points can be added through the Via point windows (Figures. 11 and 12). A via point is an extra point added by the operator into the motion trajectory of the manipulator. Via points may be included to ensure that the manipulator moves around expected obstacles between cuts or during tool retrieval and replacement. After cut points have been selected from the previously described window, the drop list in this window will contain a list of those cutting operations. The operator may then choose to insert via points after any of the operations in this list. They will be included in the final motion trajectory created for download to the controller. The via point coordinates are entered manually.

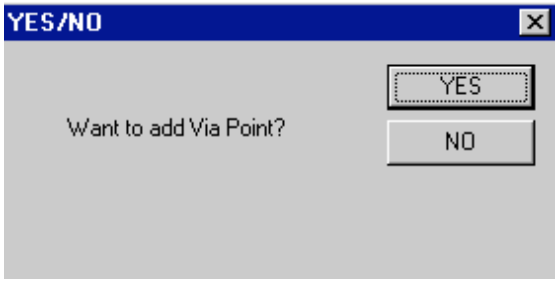


Figure 11 Via Points Window

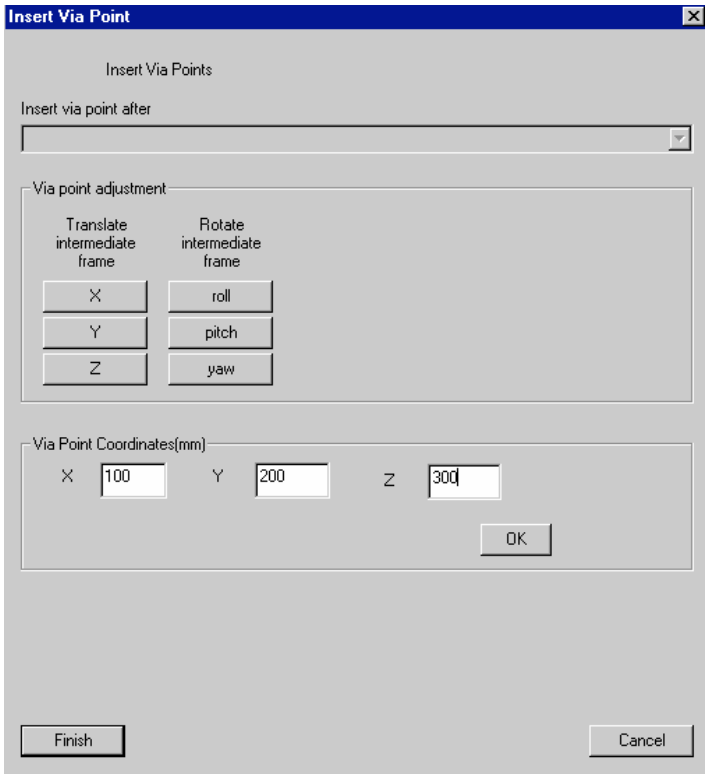


Figure 12 Insert Via Point Window

More cuts window

Finally, the operator is asked whether more cuts should be added to the plan in the More cuts window (Figure 13). If more cuts are needed in the task plan file, the operator is prompted to begin

the cut plane selection process again with Figure 7. If not, the plan is completed by checking and downloading it.

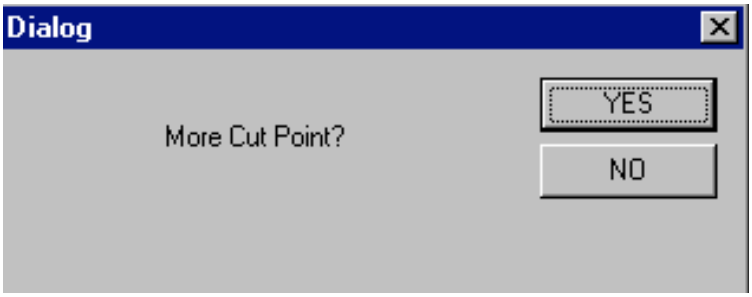


Figure 13 More Cuts Window

Check and download plan window

The Check and download plan window (Figure 14) allows the basic structure of the file to be checked. It also allows the operator to choose to download the task plan file to the location of her choice. When opened, the check plan window displays a simple alphanumeric list representing the results of the high-level task planning operations.

Figure 14 Check and Download Plan Window

Save as window

The task plan file can be saved (for those cases when execution may be implemented at a later time) into any file location through the Save as window (Figure 15).

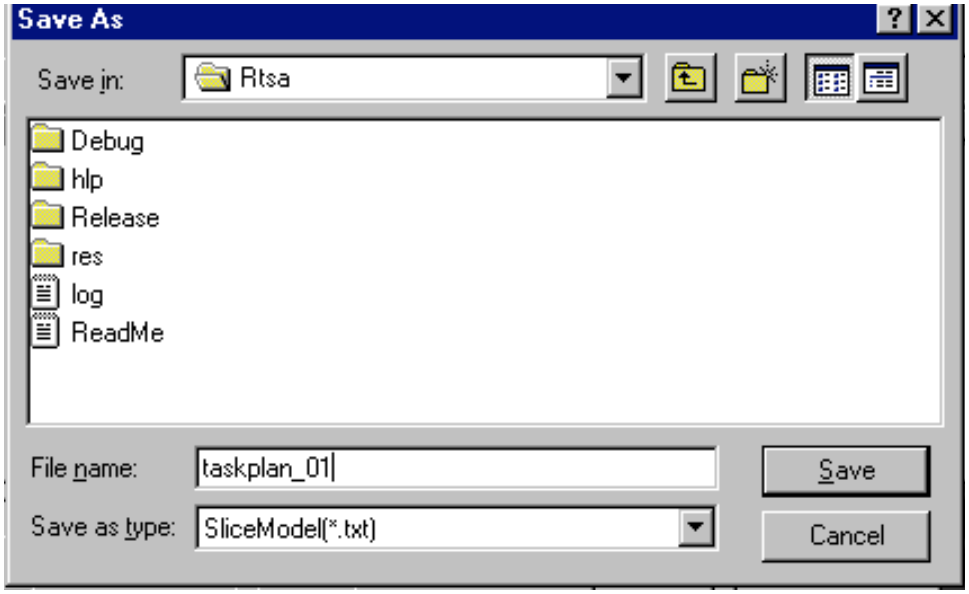


Figure 15 Save As ... Window

2.4 Computer Assistance Functions

Assist functions were developed to enhance teleoperator control over position, velocity, and force during specific tedious and challenging tasks. The system operator retains control over which assist function is used to perform a given task. The motivation to use assist functions is to increase speed and/or reduce fatigue.

2.4.1 Position Assist Functions

The position assist function has three modes that can be selected based on the manipulator task: planar constraint, linear constraint, and velocity assistance. For example, when cutting a pipe, the user must keep the manipulator in a certain orientation, typically one that moves the cutter normal to the pipe. This motion is difficult because of the many undesired inputs to the end-effector from the operator. The planar assist function can constrain the motion to the desired cutting plane. Similarly, in the drilling task, there are also many undesired inputs from the teleoperator. The linear assist function constrains the motion of the end-effector along the drilling line (axis of the drill bit).

The operator chooses the task for the manipulator to perform. For example, if the task is cutting a pipe, “P”, for planar assistance, is chosen. To drill a hole, ”L”, for linear assistance, is chosen. The “P”, ”L” and “V” selections represent, respectively, planar constraint, linear constraint and velocity scaling.

The following is a list of variables to be used in the assist function.

- GP, PP1, and PP2 – three points to determine a constraint plane. And GP is the goal point of the movement, from which the manipulator begin doing tasking, cutting or drilling etc.
- LP1, LP2 – two points to determine a constraint line
- CP – current point of the end-effector. At the beginning, CP is the same as the MIP (Manipulator initial point).
- Projection – the projection of the CP in a plane or on a line.
- VNS – nonscaled master velocity command
- Vcut – the transformed velocity of VNS in the new frame during cutting work
- Vline – the transformed velocity of VNS in the new frame during linear work
- Vscaled – the scaled velocity in the constraint frame
- Vmodified – the modified master velocity command

2.4.1.1 Planar Constraint

Use of the planar constraint position assist function will result in:

- The end-effector moving only in the constraint plane
- The final end-effector position coinciding with the goal point

These objectives are accomplished by scaling velocity components on the constraint plane. First, the end-effector is moved to the constraint plane, and then the orientation is fixed to that needed for the task. The end-effector then moves along the constraint plane to the task location. The velocity is adjusted based on the distance between the end-effector and the task location. Once at the task location, the operation can begin. These steps will be described in more detail in the following paragraphs.

Before doing a certain task in mode "P", such as cutting, the position and the orientation of the end-effector should be adjusted to successfully and efficiently complete the desired task. First of all, the position of the end-effector must coincide with the goal point. The orientation is adjusted according to the following parameters. To successfully obtain these parameters, assist functions are called to aid in the teleoperation of the manipulator. In the planar assist function, used in the cutting operation, the requirements are:

- The current position of the tip of the end effector must be in the constraint plane. So the velocity in Z-axis direction (the normal to the constraint plane) should be scaled down close to zero
- The Roll axis is kept on the constraint plane. So the Yaw velocity will assist the orientation to follow the angle of the constraint plane. Any deviation from the plane will be scaled down close to zero. Since the orientation is set to the desired position initially, the yaw velocity gain will be scaled close to zero.
- During cutting operations, the end effecter does not roll, so the Roll velocity must scaled close to zero.
- Once the teleoperator chooses "P," the function uses GP1, PP1 and PP2 to build a constraint plane. Then it checks if the CP (at the beginning, it is the same as the MIP) is on this constraint plane. If not, it calculates the projection of the CP to the constraint plane. If so, the projection is the same as CP.
- Based on projection, MIP and GP, it builds a constraint frame.

  X axis--- from Projection to GP;

Y axis—determined by X and Z;

Z axis--- the normal of the plane

If the CP is not on the constraint plane and the orientation is not appropriate for the task, the linemove( ) function and rotationAdjust() will be called in a while loop. The linemove () function helps the teleoperator to move the manipulator along a line from CP to projection. During this movement, its absolute velocity is scaled depending on the distance between the projection and the CP. The rotationAdjust() function adjusts the rotation of the end effector. Only when the end effector has the right position and orientation does the program exit from this while loop.

After exiting from the while loop, the end effector is on the constraint plane and its orientation is appropriate for the task. Then from this position and orientation, the manipulator moves toward to the GP. During this movement, the commanded absolute velocity is scaled depending on the distance between GP and the CP. Then the teleoperator begins the task. During task operation, the function does the following:

VNS→ Vcut (Transform master velocity command from master space into constraint frame.)

Vcut→ Vscled (In constraint frame, the Z-axis velocity is scaled. The scale factor is 0.1. This allows minimal movement without completely constraining the end-effector.)

Vscaled→ Vmodified (Transform the scaled velocity back to master space. This modified master velocity is the new master velocity command that is sent to the low-level controller.)

The above is done continuously until the teleoperator exits the assist function after completing the desired task.

2.4.1.2 Theory of the planar assistance function

The theory of the planar assist function is based upon scaling velocities relevant to a constraint frame. This constraint frame is defined as the coordinate frame in which the task is being performed. The constraint frame is constructed, and then the initialization is performed to ensure the end-effector is on the constraint frame with the proper orientation.

The following steps are performed:

1. Calculate the plane equation

The equation of a plane through three points is

(1)

where (x0, y0, z0), (x1, y1, z1), (x2, y2, z2) are GP, PP1, PP2 respectively.

The plane equation can be expressed by normal form

$$aX+bY+cZ+d=0 \tag{2}$$

where a, b, c, d are coefficients of the plane equation, they have been calculated from Eq.(1).

2. Calculate the projection of the MIP on this plane

The equation of the line from MIP to the unknown projection is:

$$\frac{X - MIP[0]}{a} = \frac{Y - MIP[1]}{b} = \frac{Z - MIP[2]}{c} = K \tag{3}$$

where a, b, and c are coefficients of plane equation. From Eq.(2) and Eq.(3), we can get

$$K = -\frac{a \cdot MIP(0) + b \cdot MIP(1) + c \cdot MIP(2) + d}{a^2 + b^2 + c^2} \tag{4}$$

$$\begin{aligned} Projection(0) &= ka + MIP(0) \\ Projection(1) &= kb + MIP(1) \\ Projection(2) &= kc + MIP(2) \end{aligned} \tag{5}$$

3. Construct the constraint plane

The origin of the constraint frame is the projection of MIP on the constraint plane.

Z-axis: the normal of the constraint plane (a, b, c)
X-axis: the direction from projection to GP.
Y-axis: to complete a right-hand coordinate system.

The transformation is:

$$R = \begin{bmatrix} a_{11} & b_{11} & c_{11} \\ a_{21} & b_{21} & c_{21} \\ a_{31} & b_{31} & c_{31} \end{bmatrix} \tag{6}$$

Where (a_{11}, b_{11}, c_{11}), (a_{21}, b_{21}, c_{21}), (a_{31}, b_{31}, c_{31}) are the direction of X, Y and Z-axis of constraint frame.

4. Initialization

Before doing scaling operation, the algorithm checks if the CP (current point of the end effector) is on the constraint plane and if the orientation is appropriate for the task. At the beginning, the CP is the same as MIP. An iterative procedure was previously described for correction if the CP is not on the constraint plane and the orientation is not appropriate. Only when the end effector has right position and rotation does the program exit this while loop. The appropriate roll, pitch, and yaw angles are calculated using the transformation matrix from the base frame to the constraint frame. These angles are calculated from the following equations:

$$\beta(pitch) = a\tan 2\left[-a_{31}, \sqrt{a_{11}^{\ 2} + a_{21}^{\ 2}}\right] \tag{7}$$

$$\alpha(roll) = a\tan 2\left[\frac{a_{21}}{\cos(\beta)}, \frac{a_{11}}{\cos(\beta)}\right] \tag{8}$$

$$\gamma(yaw) = a\tan 2\left[\frac{b_{31}}{\cos(\beta)}, \frac{c_{31}}{\cos(\beta)}\right] \tag{9}$$

5. Linear Velocity Scaling

The following is performed in order to scale the velocity in Z-axis (normal to the constraint plane):

a). Define the input velocity from the master as the velocity command. It is non-scaled velocity, represented using $V_{noscaled}$.

b) Now transform this velocity with respect to constraint plane, this new velocity is called V_{cut}.

$$\begin{aligned} &\text{tranR}=R^{T} \\ &V_{cut} = tranR \cdot Vnoscaled \end{aligned} \tag{10}$$

c) The next step is designing a scale matrix, represented by scale

$$scale = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & ScaleZ \end{bmatrix} \tag{11}$$

d). This scale matrix is multiplied by the non-scaled velocity to obtain the scaled velocity, V_{scaled} .

$$V_{scaled} = scale \cdot V_{cut} \tag{12}$$

The scaled velocity in Z-axis depends on the value of the ScaleZ. If ScaleZ is zero, the Z-velocity will be zero. In order to get the scaled velocity represented in the base frame to send to the controller as the master command, the transformation matrix must be transformed. The modified velocity in the base frame is the new master velocity command, represented by $V_{modified}$.

$$V_{\mathrm{mod}\,ified} = R \cdot V_{scaled} \tag{13}$$

In the control program, this new master velocity command is sent to the low-level controller. Thus the Z-axis of the constraint frame is scaled.

6. Angular Velocity Scaling

In order to scale Roll, Pitch and Yaw velocities, the following is performed:

- Similar to linear scaling, the input velocity from the master is the velocity command. It is a non-scaled angular velocity, represented by $\omega_{noscled}$. Using the same transformation matrix of the constraint frame with respect to base frame R (and its transpose tranR), this angular velocity is transformed into $\omega_{cut,}$ the angular velocity with respect to constraint plane.

$$\omega_{cut} = tranR * \omega_{noxcaled} \tag{14}$$

- The next step is to construct a scale matrix, represented by scaleRPY

$$scaleRPY = \begin{bmatrix} sRoll & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & sYaw \end{bmatrix} \tag{15}$$

The scale calculation is made using this scale matrix. The scaled velocity is expressed using ω_{scaled}.

$$\omega_{scaled} = scaleRPY \cdot \omega_{cut} \tag{16}$$

- The Pitch velocity is free. The scaled Roll and Yaw velocity depends on the values of the scale factors. All the scale factors in the assist function are 0.1.

- The scaled velocity is transformed from the constraint frame to the base frame, producing the modified velocity, which is the new master velocity command, represented by $w_{modified}$.

$$\omega_{modified} = R \cdot \omega_{scaled} \tag{17}$$

In the control program, this new master angular velocity command is sent to the low-level controller.

2.4.1.3 Linear Constraint

The linear assist function is similar to the planar assist function. The difference is that additional scaling is needed to keep the end-effector in a linear constraint frame. This assist function performs the following objectives:

- The velocities in the X and Y directions are scaled by 0.1.
- The Roll axis must be kept on the constraint line.
- The Pitch and Yaw velocities must be very small, so the scale factor is set to 0.1.

The linear constraint follows the same steps as the planar constraint. The calculations however, are different to achieve these objectives. The calculations are described below:

1. Obtain the line equation of the constraint line through two points: LP1, LP2

$$\frac{X - X_1}{X_2 - X_1} = \frac{Y - Y_1}{Y_2 - Y_1} = \frac{Z - Z_1}{Z_2 - Z_1} = K \tag{18}$$

where (X1, Y1, Z1),(X2,Y2,Z2) are two constraint points LP1, LP2 respectively.

2. Calculate the projection of the MIP on this constraint line. The projection is represented using P_0

From Eq. (14), we can get the following relationship:

$$\text{Line L1:} \quad \begin{aligned} X &= K(X_2 - X_1) + X_1 \\ Y &= K(Y_2 - Y_1) + Y_1 \\ Z &= K(Z_2 - Z_1) + Z_1 \end{aligned} \tag{19}$$

(Assuming the three coordinates of P_0 is (X, Y, Z).

The line perpendicular to L1 is L2. Its direction vector is:

$$\{(X_0 - X),(Y_0 - Y),(Z_0 - Z)\}$$

where (X_0, Y_0, Z_0) is the MIP, and the direction vector of L1 is:

$$\{(X_2 - X_1),(Y_2 - Y_1),(Z_2 - Z_1)\}$$

Because L1•L2=0, we have

$$(X_0 - X)(X_2 - X_1) + (Y_0 - Y)(Y_2 - Y_1) + (Z_0 - Z)(Z_2 - Z_1) = 0 \tag{20}$$

Substitute Eq.(16) into Eq.(17), we can solve K and then (X,Y,Z) , which is the value of the projection of MIP on the constraint line .

3. Construction of the Constraint Frame.

   Z-axis: the direction of the constraint line

   X-axis: from P_0 to MIP

   Y-axis: to complete a right-hand coordinate system.

   The origin is P_0.

   The transformation from base frame to constraint frame can be obtained, which is similar to the transformation in planar case.

4. Initialization

   As in the planar case, the algorithm checks if the CP (current point of the end effector) is on the constraint line and the orientation of the end effector is appropriate for the task. If not, the linemove( ) function and rotationCheck() will be called in a while loop. Only when the end effector has right position and rotation, the program exits the while loop. In order to calculate the appropriate roll pitch and yaw angles, the constraint frame rotation matrix is used. This is a 3 x 3 rotation matrix that relates the constraint frame to the appropriate roll, pitch, and yaw angles. These are the equations:

$$\beta(pitch) = a\tan 2\left[-r_{31}, \sqrt{r_{11}^{\ 2} + r_{21}^{\ 2}}\right] \tag{21}$$

$$\alpha(roll) = a\tan 2\left[\frac{r_{21}}{\cos(\beta)}, \frac{r_{11}}{\cos(\beta)}\right] \tag{22}$$

$$\gamma(yaw) = a\tan 2\left[\frac{r_{32}}{\cos(\beta)}, \frac{r_{33}}{\cos(\beta)}\right] \tag{23}$$

5. Linear Velocity Scaling

   The following is performed in order to scale the velocity in X-axis and Y-axis close to zero:

   a) We define the input velocity from the master as master velocity command. It is non-scaled velocity, represented using $\mathbf{V_{noscled}}$.
   b) Now transform this velocity with respect to constraint plane, we named this new velocity using $\mathbf{V_{line}}$.

$$tranRl = Rline^{T}$$
$$V_{line} = tranRl \cdot V_{noscaled} \quad (24)$$

c) The next step is designing a scale matrices, represented by **scale**

$$scale = \begin{bmatrix} sX & 0 & 0 \\ 0 & sY & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (25)$$

d) Use this scale matrix to do scale calculation. The scaled velocity is denoted using $\mathbf{V_{scaled}}$.

$$V_{scaled} = scale \cdot V_{line} \quad (26)$$

The scaled velocities in X-axis and Y-axis depend on the value of the scale factors.

e) Transform the scaled velocity of constraint frame back to modified velocity with respect to base frame, which is new master velocity command, $\mathbf{V_{modified}}$.

$$V_{\mathrm{mod}\,ified} = R \cdot V_{scaled} \quad (27)$$

In the control program, the new master velocity command is sent to the low-level controller.

6. Angular Velocity Scaling

These calculations are similar to the linear velocity scaling and can be calculated in the following steps:

a) Define the input master angular velocity as the angular velocity command. It is the non-scaled velocity, represented by $\boldsymbol{\omega_{noscled}}$.
b) Use the same transformation matrix, that of the constraint frame with respect to base frame R_{line} and its transpose tranRl.
c) Transform this angular velocity into the angular velocity with respect to the constraint plane, $\boldsymbol{\omega_{line}}$.

$$\omega_{line} = tranRl \cdot \omega_{noscaled} \quad (28)$$

d) The next step is designing a scale matrices, represented by **scaleRPY**

$$scaleRPY = \begin{bmatrix} sX & 0 & 0 \\ 0 & sPitch & 0 \\ 0 & 0 & sYaw \end{bmatrix} \tag{29}$$

e) Using this scale matrix to do scale calculation. The scaled velocity is expressed using ω_{scaled}.

$$\omega_{scaled} = scaleRPY \cdot \omega_{line} \tag{30}$$

The scaled Roll, Pitch and Yaw velocities depend on the values of the scale factors.

f) Now transform the scaled angular velocity from constraint frame to base frame, getting modified velocity, which is the new master angular velocity command, represented by $\omega_{modified}$.

$$\omega_{modified} = R \cdot \omega_{scaled} \tag{31}$$

In the control program, the modified angular velocity is sent to the low-level controller.

2.4.2 Velocity Assist Functions

The velocity assistance function works to speed up or slow down the absolute velocity of the end-effector depending on certain obstacles and goal points. For example, if the teleoperator wants to do a certain task, but the goal point is far away from the current end-effector point, the absolute velocity is increased. In the same way, as the current point of the end-effector approaches the goal point, or an obstacle, the absolute velocity will be decreased. This assist function is necessary to allow the teleoperator to perform tasks more quickly by cutting down travel time. Also, as the end-effector approaches the goal point or an obstacle, the velocity is decreased to avoid collisions resulting from errant master inputs. The velocity is increased or decreased depending on the distance. This scale calculation is changed gradually depending on the distance. The following equations relate the scaling factor to the safe distance and actual distance.

If the distance is greater than the Safedistance:

$$scalefactor = 2 - \frac{safeDis}{disActual} \tag{32}$$

If the distance is less than the Safedistance:

$$scalefactor = \frac{disActual}{safeDis} \tag{33}$$

The commanded velocity is scaled to accomplish this objective. Since the absolute velocity will be scaled, no transformation is required. The following equation is used.

$$V_{\mathrm{mod}\,ified} = Scale \cdot V_{noscale} \tag{34}$$

This results in increased performance for travel time, as well as assisting in obstacle avoidance.

2.4.3 Force Assist Functions

The force assistance function uses the individual contact force and moment components (3 force components and 3 moment components) when the manipulator contacts with the environment to assist the teleoperator. The controlled force and moment components are defined in the constraint frame shown in Figure 16.

Figure 16 Constraint frame and sensor frame.

The force assistance function (FAF) is similar the position assist function and the velocity assist function. It helps the operator by modifying the velocity commands based on the sensory data. Since the FAF operates during the task operation, it must precede the other assist functions to insure proper assistance. The FAF uses the force/torque sensor and the operator's reference force to calculate a scaling matrix to apply to the commanded velocity from the mini-master or task plan file. This scaling matrix is based on the contact forces with the robots surroundings. During task operation, the robot will encounter forces on the end-effector. To prevent damage to the end-effector, the robot itself, the tool being used, and the environment, the velocities are scaled down upon encountering forces.

Since the FAF works with the other assist functions during task operation, the constraint frame already exists. The constraint frame is the frame in which the operator sets a reference force. The force/torque data from the sensor frame is also transformed to the constraint frame. Moreover, the FAF modifies the commanded velocity before the other assist functions, so the FAF mainly scales

down velocities that are not scaled in the other assist function. For example, if the task being performed is cutting a pipe, the planar assist function will already be turned on. Once the FAF is activated, the constraint frame is obtained from the planar assist function. When force is measured from the sensor, a velocity scaling matrix is calculated. The scaling matrix will scale the velocity down according to the magnitude of the measured force. The output is the modified velocity. This modified velocity provides the input to the planar assist function. The planar assist function simply scales down the velocities that point away from the constraint frame. The output of this scaling is the modified velocity, which is sent to the low level controller. The purpose of the FAF is to modify the commanded velocities within the allotted movement of the constraint frame. To accomplish this, the FAF should be calculated before the other assist functions. The FAF calculations are explained in the following section.

2.4.3.1 Description of Force Assist Function Calculations

The force assist function calculates a velocity and angular velocity scaling matrix based on the force/torque sensor data and the reference force. First of all, the reference force and the sensor force are transformed into the constraint frame. The reference force should be set by the teleoperator to be in the constraint frame. Once the two forces are in the constraint frame, the scaling matrix can be calculated using the following equations. Since the reference force, $f_{reference}$, is relevant to the task, a scale matrix is created according to the following equation:

$$S(i,i) = \frac{f_{reference}(i) - f_{sensor}(i)}{f_{reference}(i)} \; for (i = 1..6) \tag{35}$$

where S(i,i) denotes the diagonals of the scaling matrix, S.

The six values correspond to the force in x, y, and z, and the moments in x, y, and z. Since the desire of an assist function is to scale the velocities, this equation decreases the value of the scale upon an increase in force/moment from the sensor. Also, in addition to equation (3), there exists the following three conditions:

if $\frac{f_{reference}(i) - f_{sensor}(i)}{f_{reference}(i)} < 0.1$ then S(i) =0.1

or

if $f_{sensor}(i) > f_{reference}(i)$ then S(i) is set to 0.1

or

if $f_{reference}(i)$ =0, then S(i) is set to 0.1

The first condition means that the scaling will never be less than 0.1 of the commanded velocity. This is because the manipulator should never be scaled down too far in order to prevent from getting stuck in that position. The second condition means that if the sensor force exceeds the reference force, the lowest value for *S*(i) will remain at 0.1. Similarly, the third condition states that if the reference force is input as zero from the teleoperator, and the velocity is immediately scaled down to 0.1 under the assumption there will always be some f_{sensor} value for that given direction. Therefore, if there is any force at all for a zero reference force value, the scale will be 0.1.

After the scaling matrix is calculated using equation [3] and the three conditions, it is multiplied by the input velocity command from the mini-master. The scaling matrices are given by the following equations:

$$S_F = \begin{bmatrix} S_{Fx} & 0 & 0 \\ 0 & S_{Fy} & 0 \\ 0 & 0 & S_{Fz} \end{bmatrix} \quad \text{and} \quad S_M = \begin{bmatrix} S_{Mx} & 0 & 0 \\ 0 & S_{My} & 0 \\ 0 & 0 & S_{Mz} \end{bmatrix} \tag{36}$$

S_F denotes the force scaling matrix to be multiplied by the velocity matrix. S_M denotes the moment scaling matrix to be multiplied by the angular velocity matrix. Before the S_F matrix and the S_M matrix can be multiplied to the commanded velocity, they must be transformed to the constraint frame. This is shown in the following equation:

$$V_{commanded-constra\,\mathrm{int}} = T^{base}_{constra\,\mathrm{int}} * V_{commanded-base} \tag{37}$$

Now the commanded velocity is in the constraint frame, and the scaling matrices can be applied:

$$V_{\mathrm{mod}\,ified} = S_F * V_{commanded-constra\,\mathrm{int}} \tag{38}$$

and

$$\Omega_{\mathrm{mod}\,ified} = S_M * \Omega_{commanded-constra\,\mathrm{int}} \tag{39}$$

These modified V and Ω are transformed back to the base frame as the output to the force assist function.

$$V_{\mathrm{mod}\,ified-base} = T^{constra\,\mathrm{int}}_{base} * V_{\mathrm{mod}\,ified} \tag{40}$$

and

$$\Omega_{\mathrm{mod}\,ified-base} = T_{base}^{constra\,\mathrm{int}} * \Omega_{\mathrm{mod}\,ified} \tag{41}$$

The modified velocity and the modified angular velocity are in the form of one 6-element vector, $V_{modified}$. Therefore the FAF assists the operator by scaling the velocity relative to the force on the environment, resulting in constraining the motion within the constraint frame, in order to further assist the teleoperator.

2.4.3.2 Definition of the constraint frame and the sensor frame

The force sensor frame is different from constraint frame, so $f_{contact}$ and $f_{reference}$ must be transformed from the sensor frame to the constraint frame. The transformation is obtained from the following:

- According to a certain task, define a constraint frame that is relative to the base frame. So ${}^{B}_{C}R$ will be obtained.
- Define sensor frame with respect to the base frame and obtain ${}^{B}_{S}R$.
- From

$${}^{C}_{S}R = {}^{C}_{B}R \times {}^{B}_{S}R = ({}^{B}_{C}R)^{-1} \times {}^{B}_{S}R \tag{42}$$

  the transformation from sensor frame to constraint frame is obtained.
- Get the force-moment transformation T_f, then transform the force feedback data from the sensor frame into the constraint frame.

$$\begin{bmatrix} {}^{C}F_{C} \\ {}^{C}N_{C} \end{bmatrix} = \begin{bmatrix} {}^{C}_{S}R & 0 \\ {}^{C}P_{SORG} \times {}^{C}_{S}R & {}^{C}_{S}R \end{bmatrix} \begin{bmatrix} {}^{S}F_{S} \\ {}^{S}N_{S} \end{bmatrix} \tag{43}$$

$${}^{C}F_{s} \qquad T_{\mathrm{f}} \qquad {}^{S}F_{s}$$

After the force vector has been transformed into constraint frame, all the calculations necessary for the assist function can be made.

3. EXPERIMENTAL STUDIES

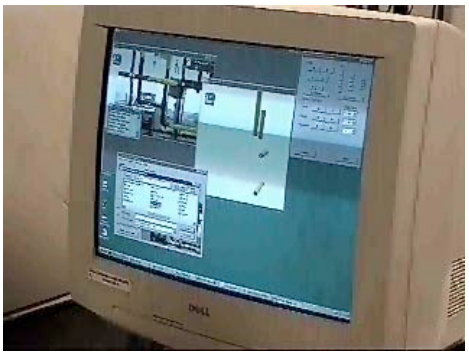
Over the course of the project, various types of experimental tests have been performed. During this phase, the experimental focus was on tests involving automated task execution and human intervention to overcome unexpected operational faults. Key results are summarized in the following section.

3.1 Autonomous Operations

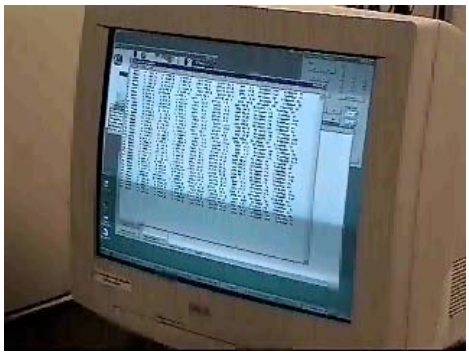
In this test, HMCTR operates in the autonomous mode, and then the commands for manipulating are described in the task plan file. Figure 17 shows the process to build the task plan file in the RTSA. Figure 17 (a) shows the 3-D modeling of task space, and the (b) shows the selection of cutting plane on the 3-D model. Figure 18 shows the preliminary pipe cutting test without the use of the blade trailing guide on the band saw. It shows that the manipulator is trapped after the pipe cut is completed when the blade extends beyond the outer edge of the pipe.. A very large disturbance, generated either during or after pipe cutting, causes the control system to overshoot moving the end-effector from the correct position after the pipe is cut through. The results of this transient event make it impossible to withdraw the saw blade through the pipe cut opening. As shown in Figure 16, the human operator tried to move the end-effector back to the approach point in the teleoperation mode, but it proved to be virtually impossible because of the complexity of the situation and the control limitations of the teleoperation mode. Figure 19 shows a successful D&D operation of the HMCTR on the mockup built in the Robotics and Electromechanical Systems Laboratory.

(a) modeling

(b) selection of cutting plane

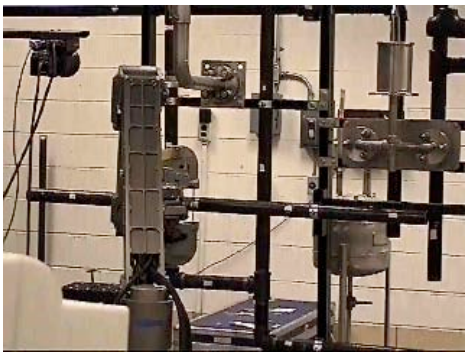


(c) saving of task plan file



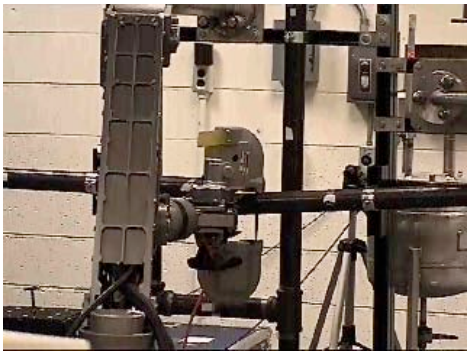
(d) review of the task plan file

Figure 17 Building of task plan file in RTSA.

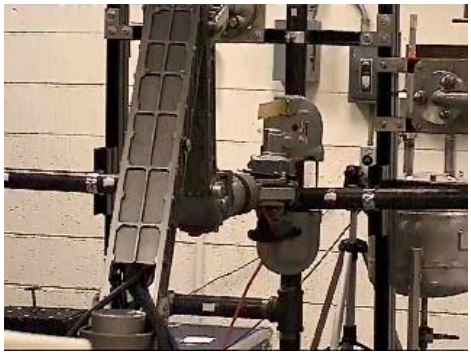


(a)

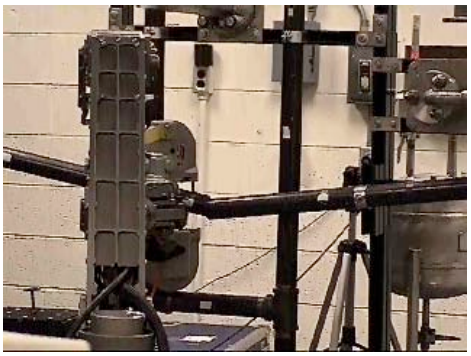
(b)



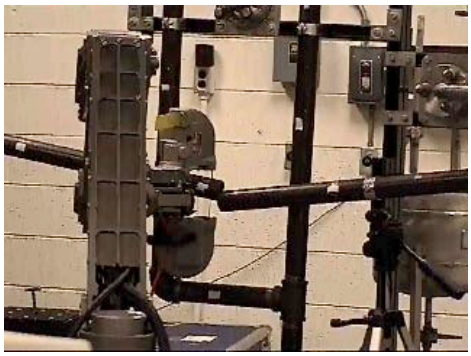
(c)



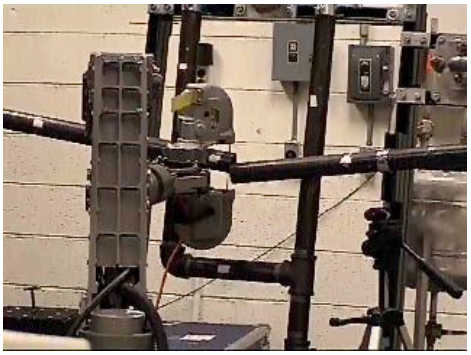
(d)



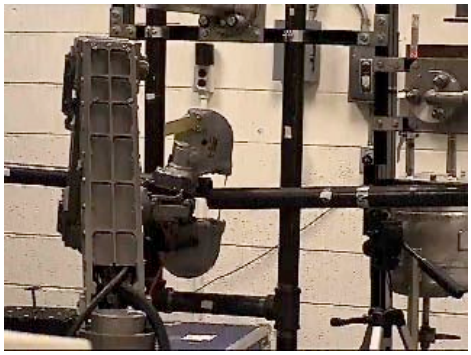
(e)



(f)

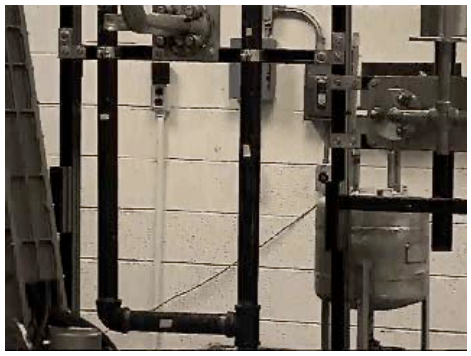


(g)

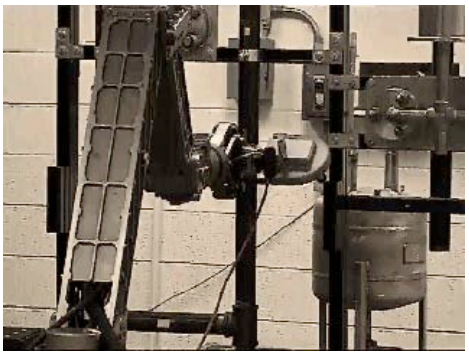


(h)

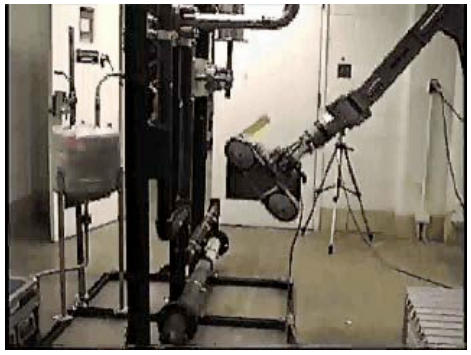
Figure 18 Manipulator trapped after cutting.



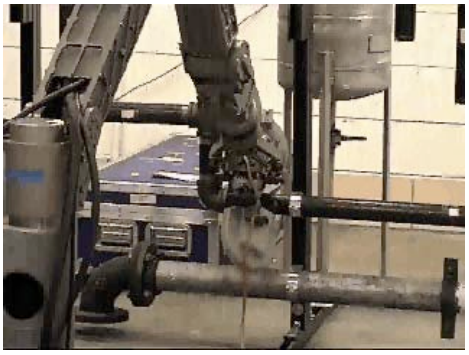
(a)



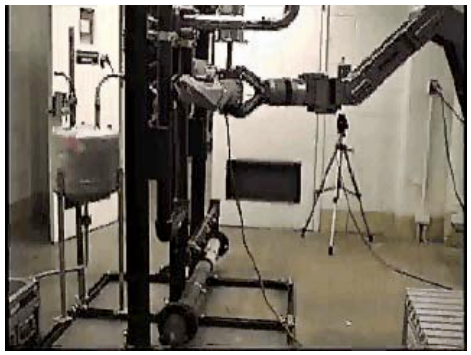
(b)



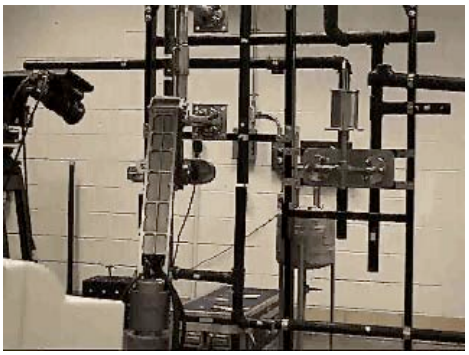
(c)



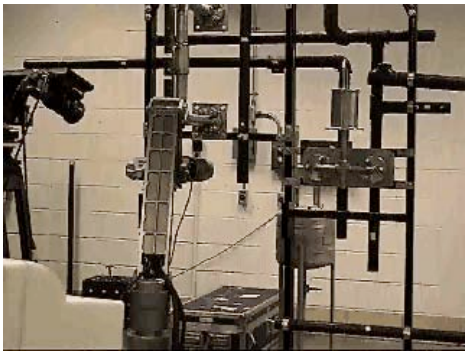
(d)



(e)



(f)



(g) (h)

Figure 19 Successful D&D operation on the mock-up in autonomous mode.

These test results show clearly that the HMCTR overall system including the RTSA capability are capable of automated subtask execution for complex scenarios such as band saw pipe cutting. The robustness of automating a particular subtask depends strongly on the nature of the tooling and task environment details.

4. RTSA ENHANCEMENTS

The original HMCT contract was modified to incorporate an additional technical objective associated with the Robot Task Space Analyzer, an integral part of a total HMCTR. The original RTSA project was performed using a laboratory type sensor head that was obtained through an equipment loan agreement with the Oak Ridge National Laboratory. The contract modification was implemented so that a prototype sensor head compatible with actual field deployment requirements could be developed. The modification also funded additional R&D to enhance the sensor head performance and to eliminate earlier dependence on an expensive commercial software package used as a 3D graphics display engine. The following discussions described the enhanced sensor head and its performance.

4.1 Sensor Head

4.1.1 Overall System Description

Definition of Sensor Head Coordinate System

Figure 4.1 shows the coordinates frames fixed on the sensor head system. The coordinate systems are right-handed Cartesian coordinate frames, and the orientations of all coordinate frames are defined to be the same as that of the OpenGL coordinate system. The pan and tilt angles are both zero at the orientation and the position shown. Therefore, the axes of coordinate frames point in the same directions: x-axes point rightward, y-axes point upward, and z-axes point backward of sensor head. The direction of the z-axes is opposite to the gaze direction of the camera. The positions of each coordinate frames are shown in Table 4.2 in Section 4.2.4.

Sensor Head Coordinate Frame

The sensor head coordinate frame is fixed on the underside of the sensor head assembly with its y-axis along the pan axis. The x-axis is attached to point in the rightward of the sensor head assembly. The z-axis is defined by the right-hand rule to point to the rear of the sensor head.

Pan Coordinate Frame

The pan coordinate frame is fixed on the top of the pan unit, and the y-axis is defined along the pan axis. Its orientation is identical to the sensor head frame when the pan angle is zero.

Tilt Coordinate Frame

The tilt coordinate frame is placed inside the post, which links the pan unit and tilt unit, with its x-axis along the tilt axis. The y- and z-axes are defined to have zero tilt angles when the orientation is identical to the sensor head frame.

Camera Coordinate Frame

The camera coordinate frame is placed on the surface of the right lens of the camera. The z-axis is defined to point in the opposite direction of the camera gazing direction, and the x-axis is not only parallel to the x-axis of the sensor head frame, but also passes through the left lens of the camera.

LRF Coordinate Frame

The LRF coordinate frame is placed on the surface of the laser range finder. The z-axis is defined to point in the opposite direction of the laser beam, and the y-axis is defined to be parallel with not the incline of the LRF, but the y-axis of the sensor head frame.

Figure 20. Coordinate Frames of the Sensor

4.1.2 Mechanical Design

The sensor head is comprised of the three main hardware components, viz. the Pan and Tilt unit, the CCD cameras and the Laser Range Pointer (LRP). Performance criteria were specified for each of these units, and hence the selection of these individual units was critical to the performance of the sensor head. A survey of available hardware that closely meets the specifications was made. A brief description of the selected hardware units with their features can be found in the following sections.

A picture of the pan and tilt unit is shown below.

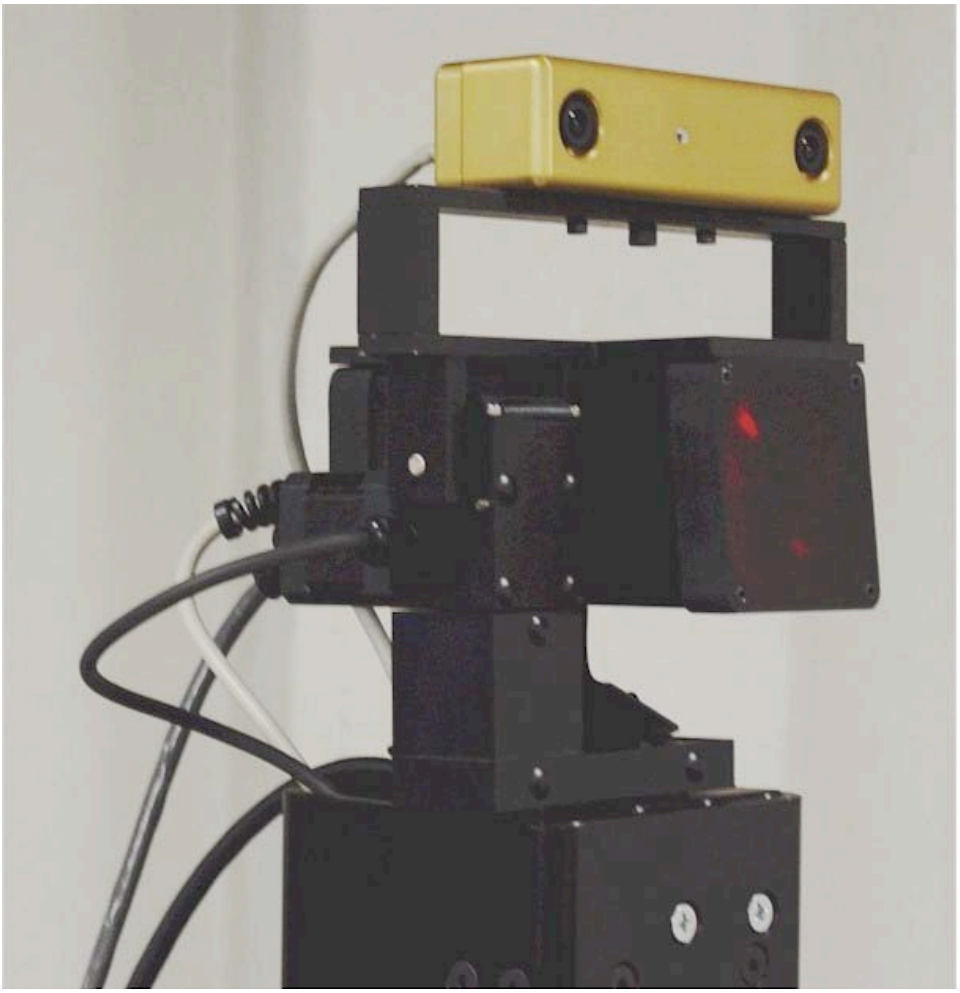


Figure 21. Sensor Head Mounted on Pan/Tilt Unit

The requirement of the pan and tilt unit is to position/orient the cameras and the laser range pointer and to move them through the range of motion of the Schilling Titan II manipulator. A frame was designed to attach the cameras and the LRP to the pan and tilt unit with four ¼-20 mounting holes. This frame is shown with the stereovision cameras mounted above the frame and the LRP below the frame. (AutoDesk Inventor 3D models of the mounting bracket and other mounting hardware are shown in the Attachments to provide more information about these mounts.

A basic calculation on the deflection and natural frequency was made and a ¼ inch aluminum plate was found to provide adequate strength and stiffness to minimize effects of dust and

contamination. The plate was fabricated in the workshop at University of Tennessee, Knoxville. It was black anodized to have better protection against radiation.

Another support was required to fix the pan and tilt unit relative to the manipulator. As shown in Figure AA2 in the Appendix, this structure interfaces with the box beam, which also supports the manipulator arms. A square steel channel 5x5 inches and ¼ inch thick was used. This is supported on a box beam with 4x4 angle plates ¼ inch thick. A ½ inch steel plate supports the pan and tilt unit at the bottom with two ¼ -20 mounting holes. This plate is attached to the steel channel using ¼-20 holes as shown in the figure. All the cablings are routed through the inside of the channel through a notch cut out at the bottom of the tubing to the host computers.

The original design specification had a requirement for an enclosure for the sensor head, but since the different components selected was radiation hardened, the need for the enclosure was eliminated.

4.1.3 Laser Range Pointer (AccuRange 4000 LIR)

The AccuRange 4000LIR is an optical distance measurement sensor with an accuracy of 0.1 inch and a useful range of zero to 50 feet for most diffuse reflective objects. It operates by emitting a collimated laser beam that is reflected from the target surface and collected by the sensor. It is a Class IIIb laser product, available in power levels of 8 mW (Standard) or up to 20 mW High power Laser optionally. AccuRange 4000 LIR uses near infrared light (780 nm wavelength). It is suitable for a wide variety of distance measurement applications that demand high accuracy and fast response times.

Figure 22. Laser Range Pointer

The following are some key features of the AccuRange 4000 LIR:

- Zero to 50 feet operating range for most surfaces.
- 0.1 inch accuracy, 0.02 inch short-term repeatability.
- Optional RS-485/422, 4-20mA current loop, and pulse width outputs. RS-232 serial output standard.
- Reflected signal amplitude output for grayscale images.
- Fast response time: 50KHz maximum sample rate.
- Lightweight, compact, low power design.
- Tightly collimated output beam for small spot size
- Ideally suited to level and position measurement, machine vision, autonomous vehicle navigation, and 3D imaging applications

4.1.4 Pan-Tilt Drive

The Pan/Tilt unit is one of the most important components of the sensor head, in that carries the cameras and the laser range finder through the required range of motion in precise relation to the manipulator. It has two axes about which it can rotate to point the cameras and the laser range finder to a desired location in the three dimensional space. The commands to the pan tilt unit are issued by the operator through the host computer.

The range of motion required of the Pan/Tilt unit is determined by the workspace of the remote manipulator with which it will operated, which in this case is the Schilling Titan II manipulator. The range of motion of the Pan/Tilt unit was thus one the specifications required. Other requirements were the accuracy of the sensor head, which included the precision of the Pan/Tilt unit, the resolution of the cameras and the accuracy of measurements of the laser range finder. Weight of the unit and cost were also significant considerations. The Directed Perception, Inc, model PTU 46-70 N, was selected based on these specifications.

Some of the features of this unit are listed below. As can be seen from this table, all the requirements of the unit are met. Although the tilt range of motion is 78 degrees, it can be altered to meet the requirements by using a controller command, as long as the payload does not strike the pan tilt unit itself. The tilt motion of the manipulator has a 160-degree range, but this is not a direct requirement for the Pan/Tilt unit. Comparing the reachable workspace of the Schilling with the

specifications of this unit showed that with a tilt of 31 degrees up and 70 degrees down, would work fine.

Precise position control.
Pan range of 320 degrees and Tilt range of 78degrees.
Position changes can be made on the fly.
Self-calibration upon reset of the unit.
Can be controlled by the host computer through the RS 232 interface.
Load can be placed at the nodal point, which is the point of intersection of the pan and tilt axis. This gives the advantage of reduced calculations for the transformation for that component.
Radiation hardened and weatherized, which eliminates the need for an enclosure for this unit.
Compactness of the unit.
Low cost.

Figure 23. Pan/Tilt Unit Critical Features

4.1.5 BumbleBee™ Stereo Camera System

Bumblebee™ is Point Grey's new two-lens stereo vision camera. It provides a balance between 3D data quality, processing speed, and size. The camera is ideal for applications such as people tracking, gesture recognition, mobile robotics and other computer vision applications. Bumblebee camera is pre-calibrated for lens distortion and camera misalignments. It does not require in-field calibration and is guaranteed to stay calibrated. The left and right images are aligned within 0.05 pixel RMS error. The calibration information is preloaded on the camera, allowing the software to retrieve the image correction information. This allows seamless swapping of the cameras, or retrieving the correct information when multiple cameras are on the bus. Bumblebee is supplied as a full development kit, including the camera head, interface card, 4.5m cable, device driver, image acquisition software, and Triclops library.

Figure 24. Bumblebee™ Stereo Camera

The Camera Specifications are as followings:

Imaging Device	1/3 " progressive scan CCDs Color or B&W 640x480 Option: two Sony® ICX084 Color or BW CCDs 1024x768 Option: two Sony® ICX204 Color or BW CCDs HAD image sensor with square pixels
Supported Frame Rates	640x480 square pixels at 30, 15, 7.5, 3.75 FPS 1024x768 square pixels at 7.5 FPS (enquire about details)
Supported formats	B&W sensor: 8-bit Mono Color sensor: 8-bit Bayer tiled image (color space conversion done on the host computer)
Signal to noise ratio	TBD
Connector	One IEEE-1394 6-pin connector
Power	Through IEEE-1394, consumption at 2.1W
Gain	Auto/Manual (0-34dB, 0.035dB resolution)
Shutter	Auto/Manual (1/16,000 to 1/30 second @ 30 Hz) Please enquire about longer shutter speeds.
Lens focal length	High quality 4mm focal length prefocused micro lenses, approximately 70° HFOV
Baseline	12 cm
Synchronization	Less than 20 μs
Size	16x4x4cm

4.2 Software

The HMCTR software includes: the Robot Task Scene Analyzer, the Range autoscan server, the Stereo autoscan server and the *ControlShell*^{TM} control system.

Software components

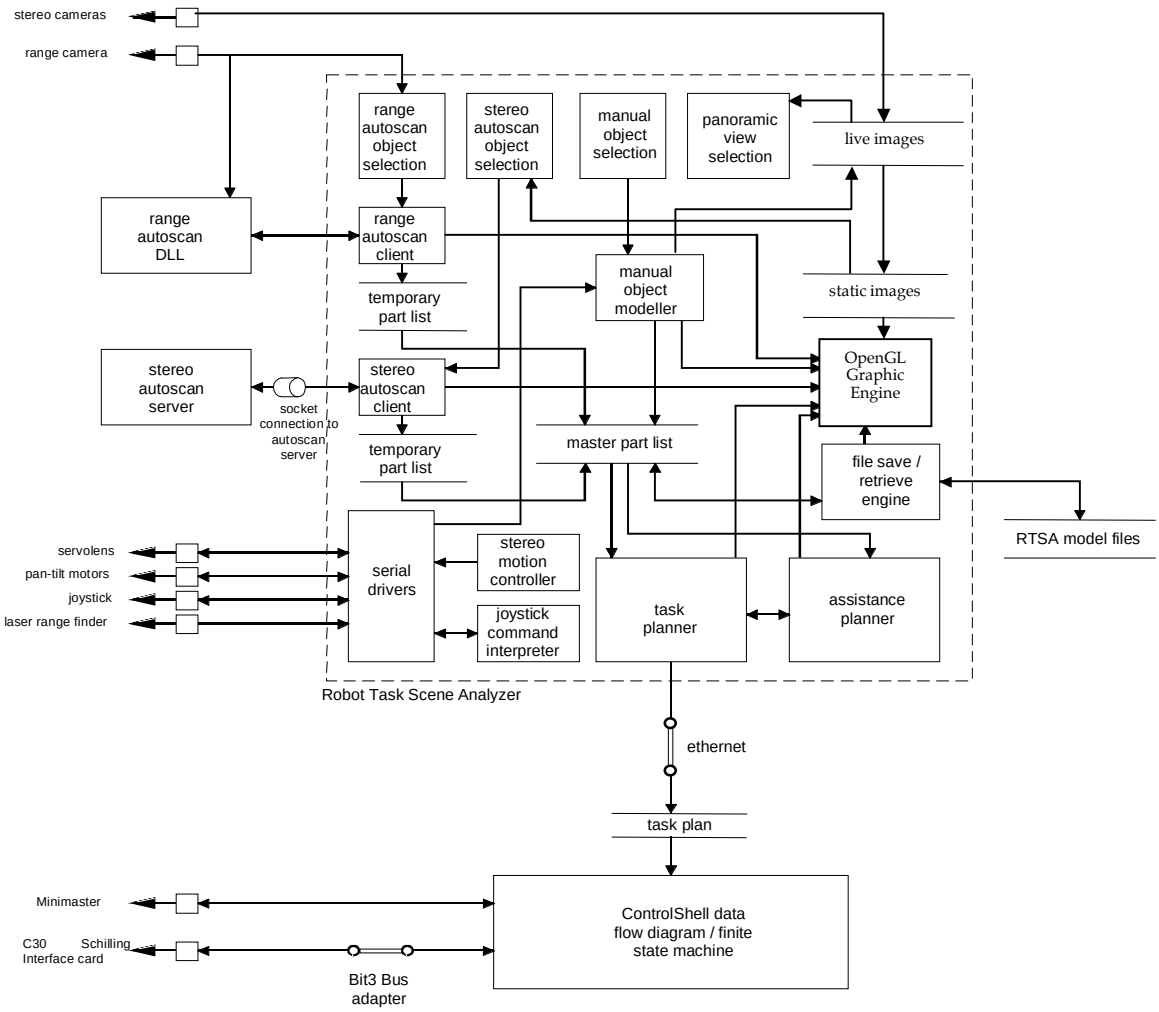


Figure 25. Robot Task Scene Analyzer Software Data Flow Diagram

4.2.1 OpenGL Approach

What is OpenGL? [5][6]

OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. It is a software interface to graphic hardware. This interface includes about 250 commands that can be used to produce interactive 2D/3D applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment. Most OpenGL implementations have a similar order of operations, a series of processing stages called the OpenGL render pipe line (Figure 26). It is not a strict rule of how OpenGL is implemented but provides a reliable prediction of what OpenGL will do [1].

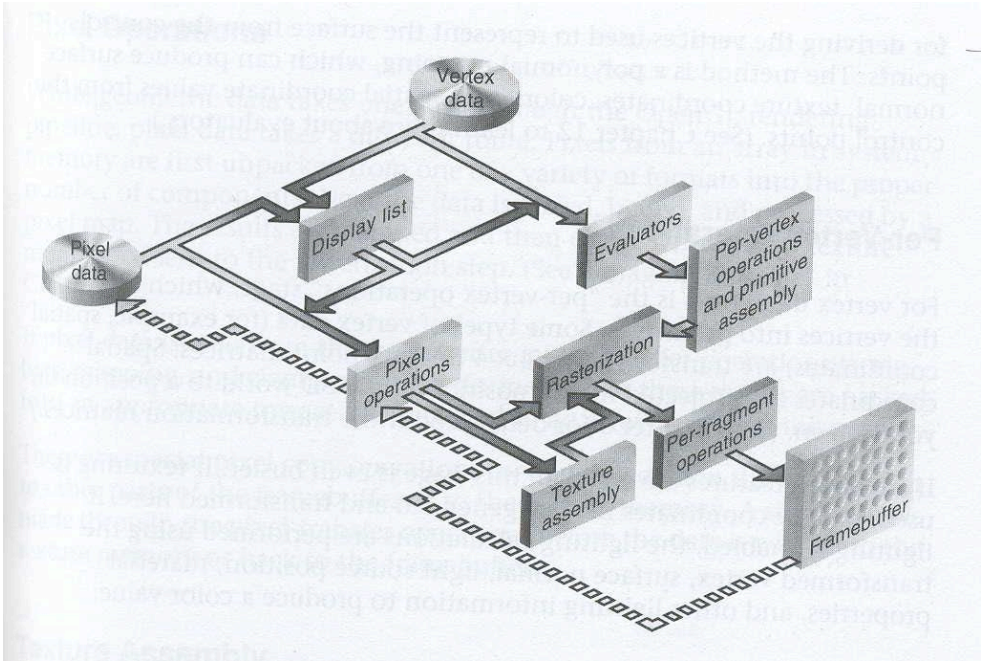


Figure 26. Order of Operations

4.2.2 Revised Software Architecture

Diagram Description

The following section describes the data flow that takes place within the Robot Task Scene Analyzer software. These descriptions refer to the diagram in Figure 6. Small squares in the diagram refer to hardware connections such as serial ports. Horizontal parallel lines with intervening labels represent data storage. Small cylinders represent socket connections, which may take place across an Ethernet network. Labeled blocks represent processes that receive, manipulate, and output data. Each of the subsections below describes the process or data storage unit and how it relates with the others.

RTSA windows tree

The Robot Task Scene Analyzer (RTSA) software is a windows-based interface that has been written in Visual C^{++} and organized in a tree structure. The tree structure is illustrated in Figure 27.

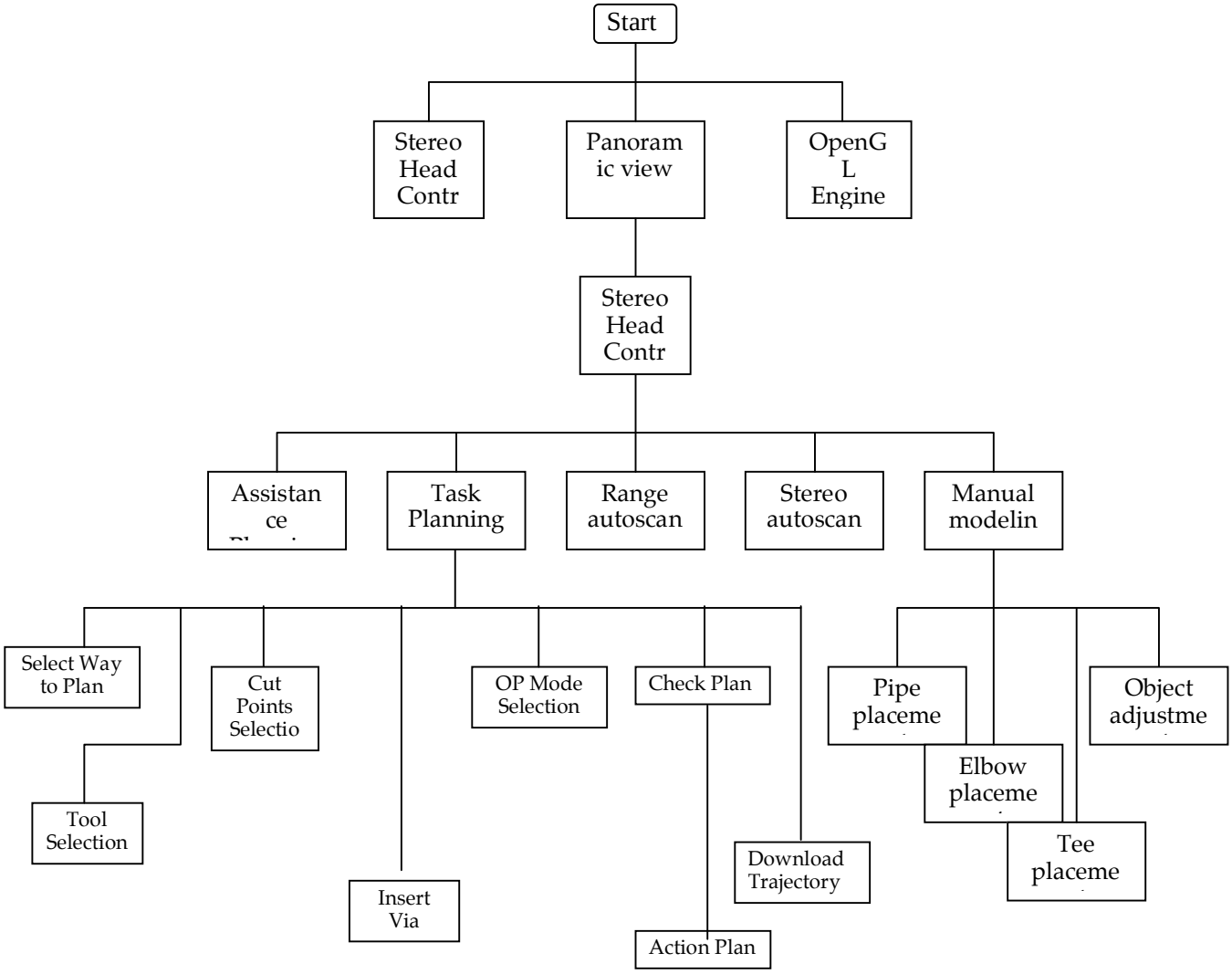


Figure 27. RTSA Windows Tree

Software Modules

Static images

Static images from the Matrox frame grabber are captured and stored in a separate buffer. These static images are necessary for texture mapping to the background block in the OpenGL virtual

environment and for display to the operator and transmission to the stereo autoscan procedure. This is also the scene that is displayed in the main window of RTSA.

OpenGL engine

The OpenGL engine is used to construct and display the virtual models.

Panoramic view selection

This object corresponds to the window that opens at startup of RTSA. It displays live images from the left and right stereo cameras. Since the stereo motion controller (see 0) is operational at this point, the operator may reposition the cameras to select a region to model.

File save/retrieve engine

This set of functions allows a model being built in RTSA to be saved and retrieved. The model is saved by writing the master part list to a file. When the model is retrieved, the current model is erased and the saved information is used to regenerate the original master part list and the previous OpenGL model.

RTSA model files

These files are the model files saved by the file save/retrieve engine functions. They consist of a text file readable by RTSA containing information from the master part list. They appear in the Microsoft file list as RTSA-type files. Double clicking on one of these file icons will open RTSA with that saved model.

Manual object selection

This process corresponds to the manual modeling window. It receives information from the operator about types and attributes of the part to be created. It then relays this information to the manual object modeling process.

Manual object modeler

This process corresponds to several windows that allow the operator to define locations for the part to be created. There is a pipe placement, tee placement, and elbow placement window, each of which has a set of icons for choosing points on the physical mockup. This process receives camera pointing information for calculation of laser spot coordinates and object type information from the manual object selection process. It then uses the information to calculate the part coordinates and creates the part by sending the appropriate commands to the OpenGL CLI command generator as well as making the addition to the master part list.

Serial drivers

The serial drivers consist of the codes that establish connections with the Servolens cameras, pan-tilt motors, joystick, and laser range finder and communicate with these devices through the

serial ports. These drivers run as separate threads of execution so they can continuously monitor the ports without affecting the flow of the rest of the software.

Stereo motion controller

The stereo motion controller corresponds to the stereo head control window. It takes input from the operator to adjust the position of the pan-tilt head in situations such as the initial view selection and laser point selection and as such must communicate these commands through the serial drivers.

Joystick command interpreter

This process is another independently running task, which enables the operator to control the camera head from a joystick by using input commands and converting them into signals to be sent to the pan-tilt mechanism.

Master part list

The master part list is a "C-list" of part information, including part type, size, schedule, location, orientation, etc., as specified by an 'objectInfo' structure in the header file.

Stereo autoscan object selection

This process corresponds to the stereo autoscan window. It displays the static images chosen initially in the panoramic view screen and receives from the operator the part type and attributes to be located by the image processing. When the part information and an image fragment are selected, they are sent to an autoscan client.

Stereo autoscan client

The autoscan client is a separately running thread of execution that gathers the operator defined information and sends it to the autoscan server. The client waits until the server finishes its image processing on the image fragments and sends back the part location information. Then the client places the part information into a temporary C-list and notifies the operator that a list of part locations is ready for insertion into the OpenGL model.

Stereo autoscan server

The stereo autoscan server is an independent executable that runs on a separate computer from RTSA. Once the autoscan server starts, it waits for contact from the stereo autoscan client. When the client sends part information and an image, the server also calculates the part position and orientation in the camera coordinate frame to send back to the client. Data communication is handled over a socket.

Range autoscan object selection

This process corresponds to the range autoscan window. It displays the range images gathered by a range camera and sent to RTSA and receives from the operator the part type and attributes to

be located by the image processing. When the part information and an image fragment are selected, the autoscan window is used to call the range autoscan client.

Range autoscan client

The range autoscan client is a separately running thread of execution that gathers the operator defined information and calls the range autoscan DLL. The client waits until the DLL finishes its image processing on the image fragments and sends back the part location information. Then the client places the part information into a temporary C-list and notifies the operator that a list of part locations is ready for insertion into the OpenGL model.

Range autoscan DLL

The range autoscan DLL is a library of image processing functions that may be called by the range autoscan client. It processes range images and returns part location and orientation information. Since it is a dynamic link library, it may exist on a separate computer to avoid monopolizing CPU time.

Temporary part list

The temporary part lists are C-list of parts returned by the autoscan functions. They exist as member variables and are erased when the operator chooses to insert them into the OpenGL model. This temporary location for autoscan information was created since automatic insertion of parts can interfere with manual insertion of parts at certain stages of manual modeling. In this way, the operator can finish with manual modeling before automatically modeled parts are added to the OpenGL environment.

Task planner

The task planner is a large set of functions responsible for creating a task plan from operator and model information. It provides the operator with a set of windows to select the object to be cut, cutting planes, tools to use, etc. It uses graphical displays of selections of parts and cutting planes in OpenGL and part information from the master part list. The final result is a linked list of actions, which are downloaded as a text file through the Ethernet to the control computer. More information on this module may be found in Section 4.2.

Assistance planner

The assistance planner is a large set of functions that interact with the task planner. It provides the operator several teleoperation assistance methods to be included at various stages of a task plan. This planner uses graphical displays of selections of parts and motion constraints in OpenGL and part information from the master part list. The final result is a set of velocity maps to be included in

the task plan and enabled by the operator during teleoperation episodes of a remote operation. More information on this module may be found in Section 2.4.

Task plan

The task plan is a text file consisting of the set of actions compiled by the task planner. It is downloaded by file transfer protocol (FTP) from the task planner to the control computer and written in a format that the controller can parse.

ControlShell controller

The ControlShell controller is the executable created in the Real-Time Innovations (RTI) product ControlShell. It incorporates a finite state machine and continuous controller, which receive task plans and commands from the master controller and command motion of the Schilling manipulator.

4.2.3 Revised Stereo Autoscan

The original RTSA system was designed to allow task modeling to be done manually and automatically concurrently with intent to minimize the overall model development time. One of the most promising automated scene modeling approaches is based on range images with object recognition occurring between 3D object representations. The earlier work verified the complexity and cost associated with high-resolution laser range cameras. Laser range cameras in the current state of the art are not compatible with remote D&D applications. The contract modification also included an objective of modifying the original range autoscan mode to be based on range images calculated through stereo scene images rather than range maps for laser range cameras. The following sections describe the enhanced stereo autoscan functionality developed around range maps obtained from the BumbleBee camera head and range map software module.

Enhanced Stereo Autoscan Architecture

Stereo Autoscan was constructed to perform automated “As-built” analysis, or inverse CAD using stereo range data. An example of a sub-sampled surface mesh [7] constructed from a stereo data set is displayed in Figure 1. Stereo Autoscan’s design centers on two features that were deemed critical for the application class: (1) the accurate and timely reconstruction of range data sets and (2) the identification and localization of key objects of interest within these datasets.

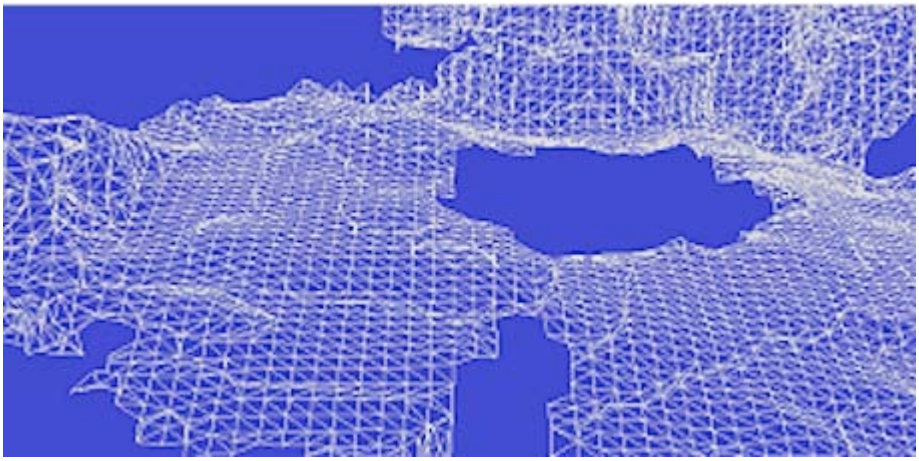


Figure 28 An example of 3-D surface mesh constructed from stereo range data.

The ability to reconstruction model data sets synthesized from individual stereo range images acquired from arbitrary locations within a facility can be decomposed into two key requirements: (1) “matching” and (2) “merging” of co-registered surface meshes into an integrated representation. Together, matching and merging allow a robotic system to co-register overlapping 3-D data sets and to synthesize from that co-registered set a single, integrated uniformly sampled data set suitable for object recognition.

Matching is defined as the process of estimating the relative transformation between overlapping sensor acquisitions. In stereo autoscan, matching is based on Hebert and Johnson’s notion of constructing a “spin-image” for a subset of the oriented points that comprise a surface mesh [8]. Once a characteristic set of spin images are defined for each mesh from the partially overlapping range images, local shape similarity measures, obtained by pair wise cross correlation between individual spin-images, are used to provide a set of candidate points matches. Points with similar spin images are considered to have similar local shape. Once correspondences are found between the data sets, the transformation aligning the datasets may be estimated. Global matching, or registration, can be accomplished through successive pair-wise registration of datasets until a common coordinate frame is realized.

Mesh merging is the construction of seamless, textured surfaces from an arbitrary number of co-registered surfaces meshes. It requires both a robust and efficient mechanism for accumulating support evidence in an incremental fashion, and robust operators that extract uniformly sampled surfaces from them. Stereo Autoscan uses a modified implementation of Johnson and Kang's surface synthesis technology, based on 3-D occupancy grids [9]. Their method represents surfaces using a probabilistic model that encapsulates a sensor error model (stereo or laser) and an artificial point spread function. Each point from a surface mesh, along with the surface normal for that point, is distributed into the occupancy grid using the appropriate models. This evidence, both the surface likelihood estimate as well as the surface normal estimate is accumulated for each point from each input surface mesh. Surface extraction is performed using robust ridge detection on the surface probabilities in the voxel space that forms an implicit surface using the likelihood gradient and the surface normal. Polygonization of the implicit surface is performed using the Marching Cubes algorithm [10]. Figure 2 displays an integrated stereo-based surface mesh from a test area at Carnegie Mellon.

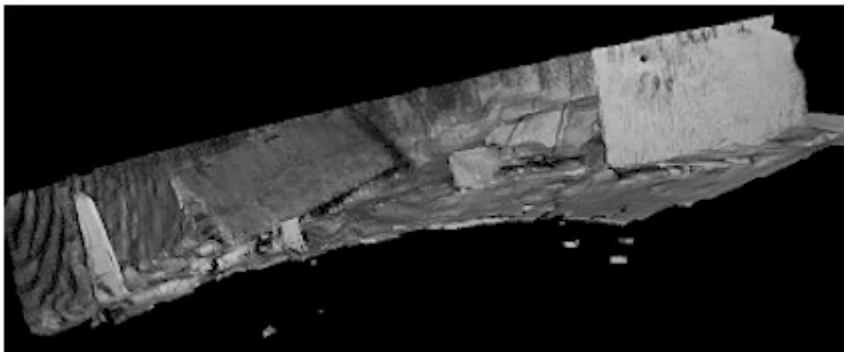


Figure 29 A textured and integrated 3-D surface mesh constructed from 32 individual stereo data sets.

The final requirement for stereo autoscan, recognition and localization of key objects, is a very similar problem to that of mesh matching. As such, a more general form of the spin image-matching algorithm is used [11]. This algorithm can efficiently find multiple known objects within a given scene mesh, storing their position and orientation for future use. The spin image is designed to be resistant to clutter and occlusion--if part of an object in a scene is hidden, recognition is still possible. Using Principal Components Analysis (PCA), the spin images may be compressed, allowing recognition of objects from large libraries in a space- and time-efficient manner [12]. Figure 3 contains a pair of sample object recognitions.

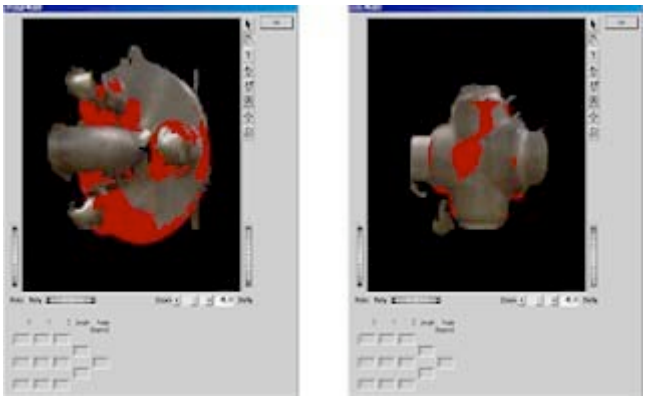


Figure 30 Spin-image recognition of standard U.S. industrial components. (a) Flange (b) tee.

Stereo autoscan is capable of object recognition and the construction of 3-D photo-realistic models from laser or stereo range data. Taken together, these capabilities serve important roles in the arena of semiautomatic teleoperation within structured nuclear facilities. The production of abstract, fully 3-D models via object recognition is essential to the rapid, accurate, and forceful interaction with a nuclear environment that minimizes exposure and automates object recognition tasks.

Integration with RTSA

Historical versions of range Autoscan were artificially grafted onto RTSA using an ad-hoc client/server architecture that required heterogeneous machines to execute the integrated capability – one for RTSA proper and one for range autoscan. This ad-hoc integration was further complicated in that the RTSA software architecture was designed within MS Windows, while Range Autoscan was implemented in Unix. The overall system was inefficient and inelegant. In this effort, the core software modules from Range autoscan were ported to Windows XP using VC++ 6.0 to provide a version of range autoscan that is native to RSTA platforms.

With Native windows software modules, integration with RTSA is implemented using hooks to core range autoscan functions through automated scripts that allow RTSA to provide range autoscan with a bumble range data file and associated parameters, e.g. region of interest or filter coefficients. Once provided, range autoscan converts the 3-D range file into a triangular surface mesh that is matched against autoscan model object libraries for standard industrial components, e.g. three inch schedule 40 tee. Once localized, the instance (multiple recognitions are possible) and pose of the

object are returned such that RTSA modules can update common graphics overlays and databases.

4.2.4 Coordinate Transformation

The transformation matrix from the sensor head frame to the laser range finder frame is calculated by multiplying the transformations from the sensor head frame to pan frame, from pan frame to tilt frame, from tilt frame to camera frame, and from camera frame to laser range finder frame as shown in (1).

(1)

The position of the LRF spot in the sensor head frame is calculated using this transformation matrix, and the transformation matrices for each coordinate frame are provided by the following relationships.

Transformation from the sensor head frame to pan frame:

Transformation from the pan frame to tilt frame:

where φ is the pan angle (Clockwise looking up along positive y axis is positive)

Transformation from the tilt to camera

where θ is the tilt angle (Clockwise looking outward along positive x axis is positive)

Transformation from camera to LRF

, and

Position of the LRF spot in the LRF frame

,

The constants shown in the various coordinate frames were either supplied by the relevant vendor or measured. Table 4.2 provides these values.

x1 = 0.0 mm,	y1 = 62.738 mm	z1 = 0.0 mm
x2 = 75.184 mm	y2 = 87.63 mm	z2 = 0.0 mm
x3 = -15.184 mm	y3 = 103.566 mm	z3 = -1.590 mm

x4 = -60.0 mm	y4 = -110.551 mm	z4 = -39.436 mm

Table 4.2 Values of Transformation Constants

The transformation from the sensor head frame to the LRF frame is shown in (2), and the position of LRF spot in the sensor head frame is given by

(2)

4.3 Performance

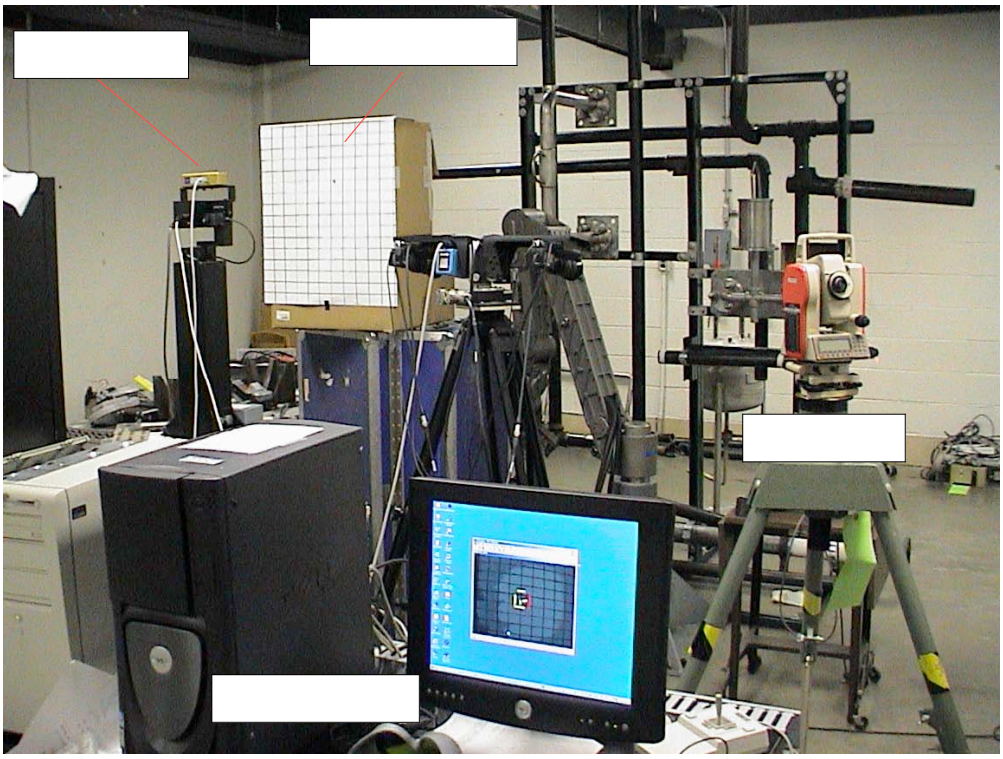
A key objective of the contract modification was to improve the RTSA modeling process accuracy through improvements in the hardware precision and software algorithms. This section describes the analysis and testing performed to assess the accuracy of the enhanced sensor head.

4.3.1 Error Analysis and Calibration

In testing the HMCTR, it was found that the actual position of end-effector was not the same as the position used in the task plan file. This difference was due to the coordinate frames of the sensor head and manipulator not being aligned as designed. Of course, the errors inside the sensor head frame can also lead to errors in the actual position of end effector. Such errors are unavoidable and can be minimized by adjusting parameters using the least-squares method. However, this approach does not reduce errors intrinsic to each physical component.

Each error component in the system propagates through the coordinate transformations, and produces a different contribution and pattern to the total position and rotation errors of the end effector. Error analysis determines the effect of each error source on the final position error and will be used to identify the most significant error source. Thus, the automatic calibration algorithm compensates for the particular error sources instead of minimizing the final error by adjusting parameters using the least-squares method.

After the sensor head system was assembled and the early version of the RTSA software using OpenGL was developed, the system underwent preliminary testing to find errors in the hardware and software. These tests were also used to adjust the gains in the software to properly display the modeled parts in the virtual world. The preliminary tests had two objectives: (1) identify errors in calibration; and (2) determine gain adjustments. The preliminary tests used the Test Grid Panel (cell size 5cm width and 4cm height) shown in Figure 4.3.1. Figure 4.3.2 is the result of the error calibration test. Point 1 is the center point (320×240) of the captured picture, which has 640×480 pixels. Point 1 is located along the z-axis of the camera frame since the picture is captured in the camera frame. Thus, the point is represented by (0,0,A) in the camera frame and (60,253.933,A+1.59) in the sensor head frame. Point 2 is the spot where the laser range finder points the location of (60,253.933,A+1.59) in the sensor head frame. The difference between the Point 1 and Point 2 represents the offset errors in the pan and tilt angles of the laser range finder. The error in the vertical position is caused by the tilt angle offset error, and the error in the horizontal position is caused by the pan angle offset error. The magnitude of the mismatched distance is proportional to the magnitude of the offset angle error.



.

Figure 31. Sensor Head and Calibration Test Equipment

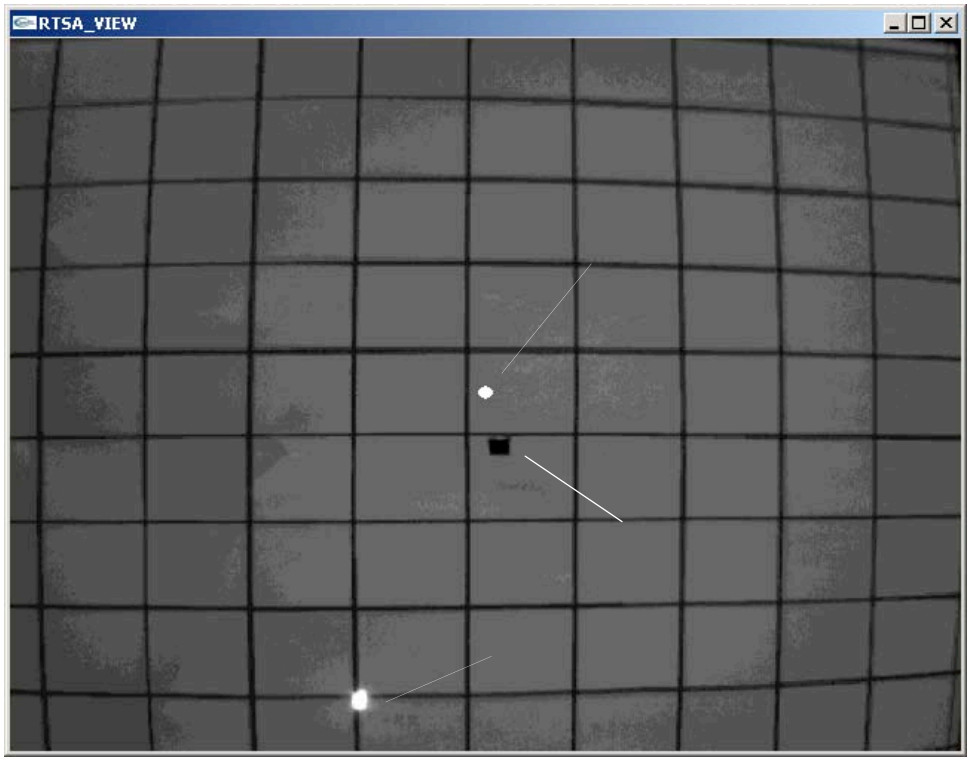
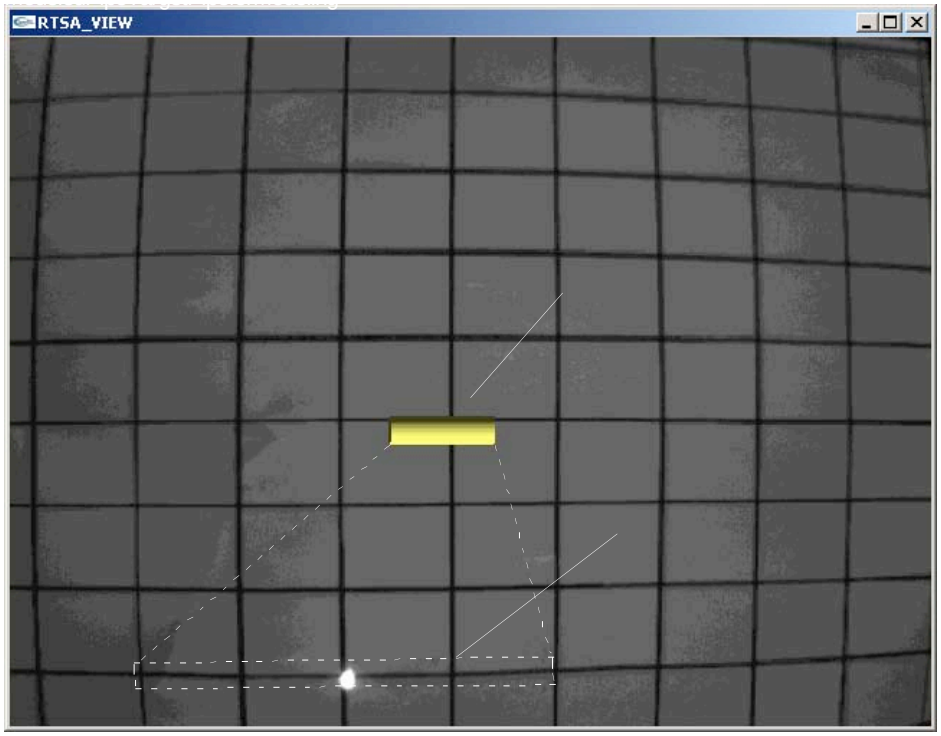
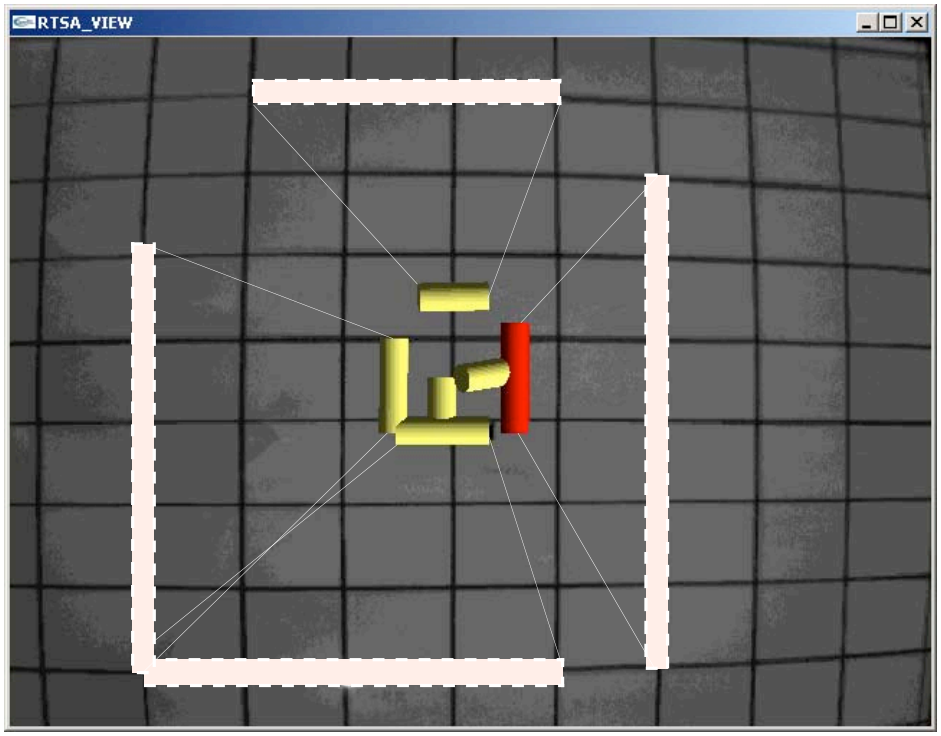


Figure 32. Error Calibration Test Result

Conceptually, the measured lengths and positions of the parts using the LRF can be transformed into the virtual world of OpenGL with the same units (mm or cm). But, the modeled parts in the virtual world must be displayed in the 2-D window on the computer monitor, which has a limited number of pixels. Therefore, the gains, which properly transform the length in mm in the virtual or real world to pixels in the 2-D window, must be determined. In the developed system, the models are viewed using the perspective view, and the gains are determined by the test. Figure 4.3.3 is the test result for the gain adjustment test. It shows that the gains for the part length and the location of the parts are small. The shift of the overall position of the parts may be caused by the offset error. Based on these adjustment tests, the best gains for the display of the parts will be selected.



(a) Result of horizontal pipe modeling.



(b) Results of vertical and horizontal pipe modeling.

Figure 33 Gain Adjustment Test Result

4.3.2 Tests

Based on the test results described in section 4.3.1, the offset error and the OpenGL gains are compensated and adjusted through calibration tests. First, the dominant offset angle error (tilt offset angle of the laser range finder) is calculated and measured using Pythagoras theory and tests. Compensation tests show that the tilt-offset angle of the laser range finder is -2.0° (it means that the error makes the LRF looks down when the system reads tilt angle as zero). Figure 34 shows the test grid panel after the tilt offset angle is compensated. It shows that the center of the wall paper (test grid panel) nearly matches the center of the view window ("+" in the picture). Also, the quality of the wall paper picture was improved by using the BMP graph file format instead of the JPG file format.

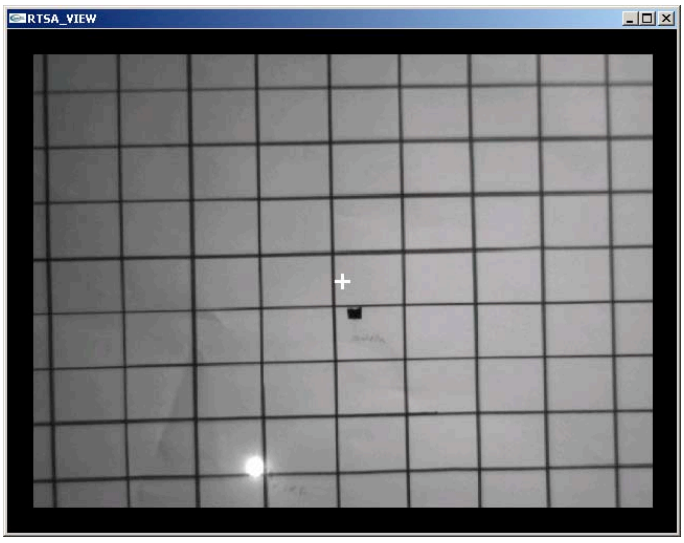


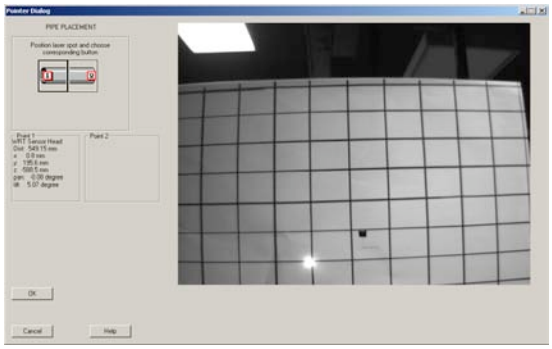
Figure 34 Wallpaper after the LRF tilt offset error is compensated.

Secondly, after the angle offset was compensated, the parameters in the OpenGL functions were adjusted to match the start and end points of the model (pipe) to the start and end points of the wall paper (grid panel) used in the modeling process. Through the calibration tests, it was found that the main error was in the data for the field of view (FOV) and focal length of the Bumblebee camera. According to the specifications, generally the HFOV of the Bumblebee camera is 70~75°, and this parameter was used before the tests. However, camera parameter tests show that the model used for this project has 41.292° HFOV, 31.492° VFOV, and 849.27mm focal length. The manufacturer, Point Grey, Inc, stated that the model used here has 6mm lenses instead of the 4mm lenses normally usually used in the Bumblebee camera, and the HFOV for 6mm lenses is in the range of 40~45°. Figure 35 is the model view window with the wallpaper

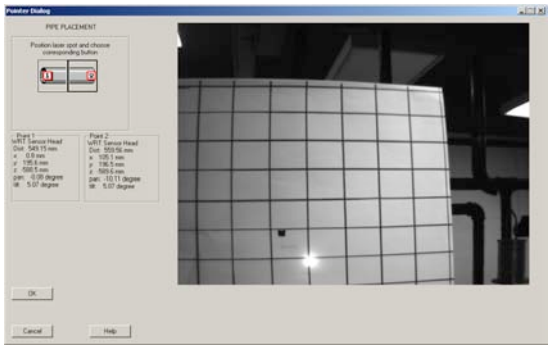
after the OpenGL compensated and the test grid panel is placed in the actual workspace of manipulator.

Figure 35 Model view window with wallpaper after OpenGL parameter adjustment.

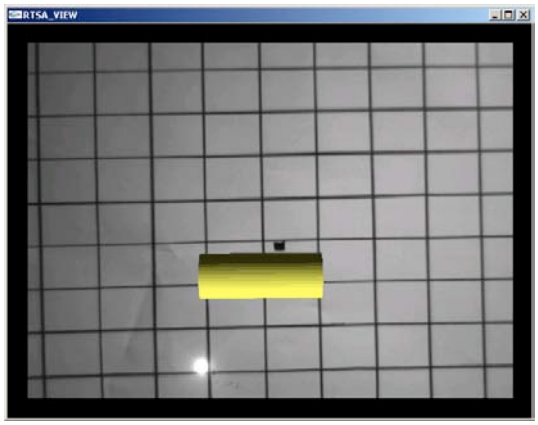
For the system accuracy test, a pipe model was used; the modeling and test procedures are illustrated in Figures 36-37. After the start point and the end points of pipe are defined using the LRF in the modeling window (Figure 36, 37 (a), (b)), the RTSA software generates the pipe model and displays it in the model view window as shown in Figure 36-37 (c). After the model is built, the length of the modeled pipe, which is displayed in the model adjustment windows (red circle of Figure 36, 37 (d)), and the distance between the start point (Figure 36 (a)) and end point (Figure 36 (b)) in the modeling process are compared.



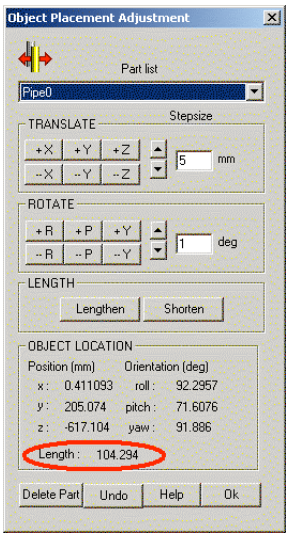
(a) left end of pipe model



(b) right end of pipe model

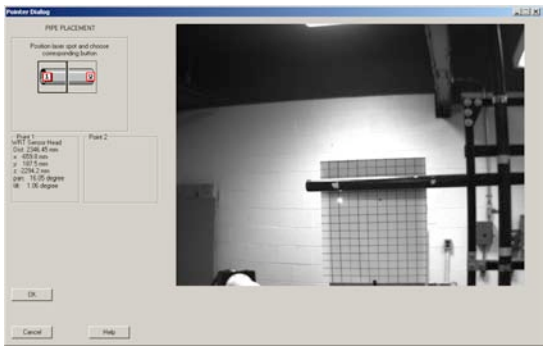


(c) constructed pipe model

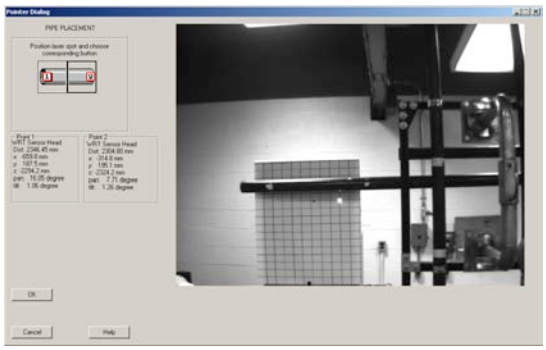


(d) pipe model parameters

Figure 36 System accuracy test procedure (when the wall paper is closed to the sensor head).

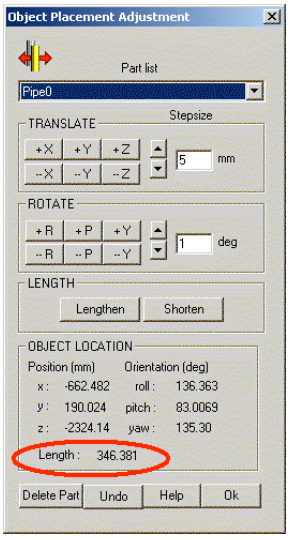


(a) left end of pipe modeling



(b) right end of pipe modeling

(c) built pipe model



(d) parameter of pipe model

Figure 37 System accuracy test procedure (when wallpaper captures the work space).

In these tests, 8 pipes were built using the test grid panel as the wallpaper (Figure 38), and 4 pipes were built with the manipulator workspace as the wallpaper (Figure 39). The accuracy test results are shown in Table 4.3.

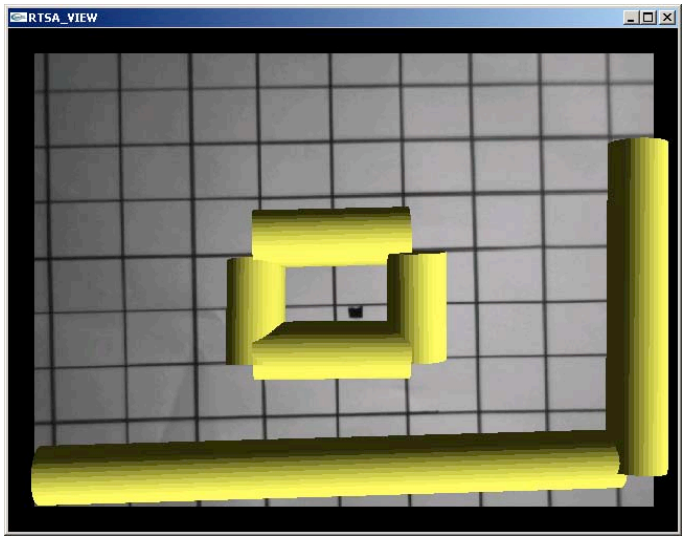


Figure 38 Accuracy test with the grid panel as wallpaper .

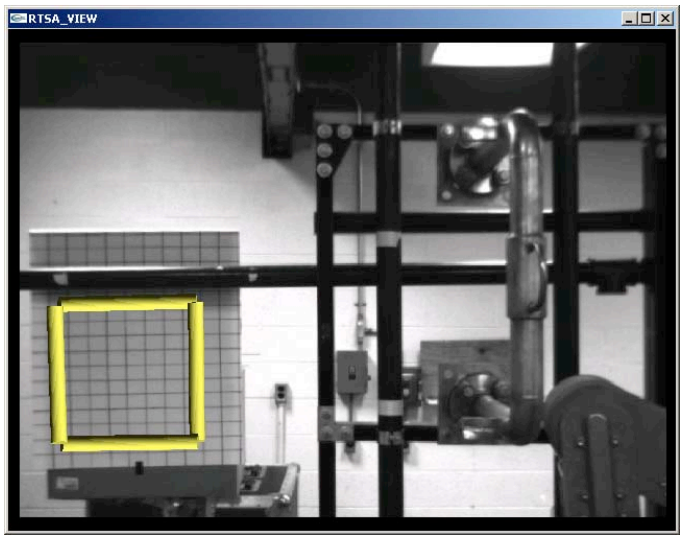


Figure 39 Accuracy test with the workspace as wall paper .

Wallpaper	Actual length*	Modeled length	Absolute error	% error
Grid panel	100	104.129	4.129	4.129
	80	73.7995	6.205	7.756
	100	104.973	4.973	4.973
	80	72.6606	7.34	9.175
	400	389.613	10.3871	2.597
	240	227.31	12.69	5.288
	400	389.746	10.254	2.563
	240	238.889	1.111	0.463
Workspace	350	346.381	3.619	1.034
	360	366.037	6.037	1.677
	350	344.962	5.038	1.439
	360	357.113	2.887	0.802

* Length unit :mm

Table 4.3. Accuracy Test Results

The accuracy test result shows that the maximum absolute error is 10.3871mm (0.41 inch), and the maximum % error is 9.175% (7.34mm, 0.289 inch absolute error). The maximum absolute error was generated in the condition when the long pipe is close to the sensor head. When the wallpaper is close, the main error factor is due to the propagation of the system errors, such as offset angles. When the wallpaper is at a distance, like the workspace wallpaper, the main error factor is the increased diameter of the laser range finder focus as shown in Figure 40 (the diameter increases to almost 1 inch). The absolute errors with the workspace wallpaper (the modeling pipe is at a distance of 2300mm, 7.55 ft from the sensor head) are smaller than the absolute errors with the grid panel wallpaper (the modeling pipe is at a distance; 600mm, 1.97 ft). Therefore, if the workspace is at distance in the range of 1000~3000mm (3.3~10ft) from the sensor head, the developed sensor head system provides ±0.3 inch accuracy, which is smaller than the proposed accuracy of the system.

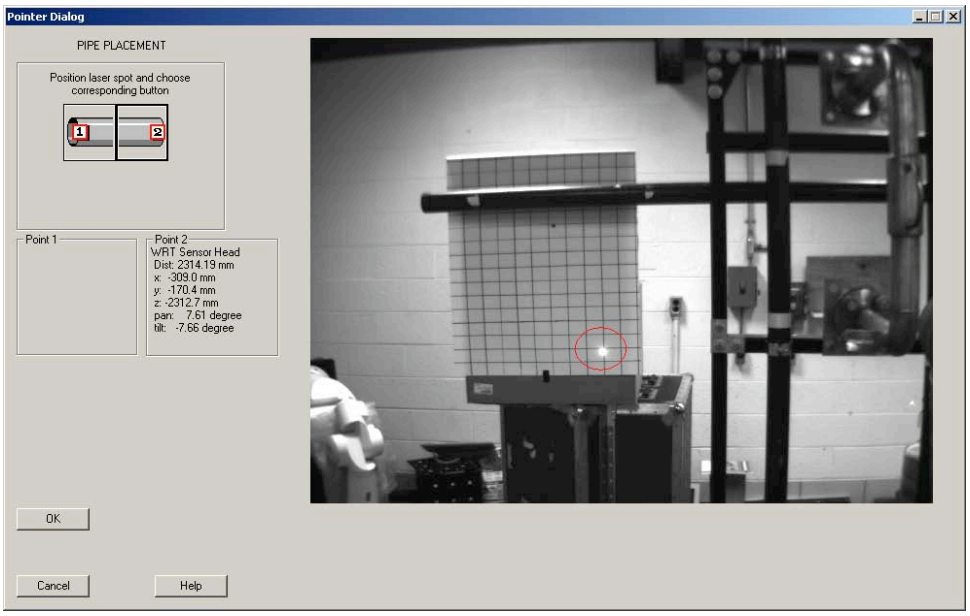


Figure 40 Increased focus of laser range finder.

5. FUTURE CONSIDERATIONS

5.1 Integration with D&D Robotics

Robotics and remote systems technologies are necessary for the efficient D&D of facilities and equipment where the ambient radiation levels complicate, or preclude, contact operations. A focus of the D&D area of the EM-50 Robotics Cross Cutting Program was to pursue R&D that would make D&D operations more cost effective. Subtask automation is one of the best ways to approach this objective and considerable efforts at national laboratories, universities and companies around the country (and world). At the time of this project, the Oak Ridge National Laboratory (ORNL) Robotics and Process Systems Division was the lead national laboratory addressing advanced telerobotic concepts for D&D. It was originally intended that the enhanced sensor head and improved RTSA software would be integrated into the telerobotics test bed at ORNL. The software and hardware features of the system were developed to requirements that would make the ultimate integration as smooth as possible.

Regretfully, at this time the EM-50 Robotics Cross Cutting Program and all of its activities have been shutdown by DOE. A modest amount of continuing funding retains a core individual and his research at ORNL. In addition, UT is partnered with a small business through an SBIR

Phase II contract. These remnants allow the continuation of the telerobotics research albeit at a much smaller scale. Additional hardware components are being loaned to UT such that a complete dual arm telerobotics test bed capability will be completed soon. These modest continuing efforts will allow the continued used and evaluation of the enhanced sensor head under realistic laboratory conditions.

5.2 Technical Issues

There remain several technical areas where continued R&D is needed to further enhance telerobotic subtask automation. These areas will be the topic of future research proposals within DOE and other federal agencies.

5.2.1 Automatic Error Calibration

The overall accuracy of the RTSA modeling process depends upon accurate and current error calibration of subsystems. The frequency and accuracy of calibrations could be increased if they could be performed in situ and on demand. Schemes for pursuing in situ and automated calibrations should be pursued.

5.2.2 Part Libraries

The RTSA modeling process is predicated on the use of a priori knowledge of the size and geometries of objects being modeled. Presently, the RTSA parts libraries encompass standard piping components including straight pipe, elbows and tees. Preliminary studies have shown that it should be possible to modify the RTSA software such that off the shelf CAD libraries could be used to create RTSA parts libraries. In this manner, additional classes of modeling objects such as structural steel, electrical components, etc. could be integrated with RTSA. Additional work is needed to explore the details of integrating commercial CAD parts libraries with RTSA

Task Planning

In the current RTSA/HMCTR environment, the human operator performs all of the task planning process using the 3D model rendering and a collection of “point and click” tools that allow him to designate tooling points and sequences. It is the operator’s responsibility to reason as to which tools should be used when. While this approach is very robust because it is based on powerful human reasoning, it places considerable cognitive burden on the operator. Additional research should explore computational reasoning methods to assist the operator in the task planning phase of subtask automation.

5.2.3 State Transition Management

A telerobotic system is one that can function both as a teleoperator and a robot. The heart of telerobotic control is the state control that must occur as the system mode of operation transitions

between these two states. In the current system, this process is implemented in a finite state machine block within the ControlShell overall controls software. Because the Telerobot is a hybrid system in the sense that it involves simultaneous continuous and discrete control, the details of state transition and continued operation from one state to the next are very complex. The present implementation has addressed the simplest cases. More research is needed to pursue the generalization of an extensible state transition management methodology for this class of systems.

References

[1] William R. Hamel, "Sensor-Based Planning and Control in Telerobotics," in "Control in Robotics and Automation: Sensor Based Integration," edited by B. K. Ghosh, Ning Xi, and T.J. Tam, chapter 10, pp.285-305, Academic Press, 1999.

[2] Steven E. Everett, "Human-Machine Cooperative Telerobotics Using Uncertain Sensor and Model Data," doctor of Philosophy in Mechanical Engineering University of Tennessee, August 1998.

[3] William R. Hamel, Reid L. Kress, "Elements of Telerobotics Necessary for Waste Clean Up Automation," Proceedings of the 2001 IEEE International Conference on Robotics & Automation, pp. 393-400, 2001.

[4] K. A. Manocha, N. Pernalete, and R. V. Dubey, "Variable Position Mapping Based Assistance in Teleoperation for Nuclear Cleanup," Proceedings of the 2001 IEEE International Conference on Robotics & Automation, pp. 374-379, 2001.

[5] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner, OpenGL Programming Guide, Version 1.2, Addison-Wesley, 1999

[6] http://www.opengl.org/developers/about/overview.html

[7] J. R. Shewchuk. “Engineering a 2D Quality Mesh Generator and Delaunay Triangulator”, First Workshop on Applied Computational Geometry, Philadelphia, PA. pp. 124-133, ACM, May 1996.

[8] A. Johnson and M. Hebert. "Surface registration by matching oriented points." Proc. Int'l Conf. on 3-D Digital Imaging and Modeling (3DIM '97), Ottawa, Ontario, May 12-15, 1997.

[9] A. E. Johnson and S. B. Kang, “Registration and Integration of Textured 3-D Data", International Conference on Recent Advances in 3-D Digital Imaging and Modeling, Ottawa, Ontario, May 12-15, 1997.

[10] W. Lorensen and H. Cline. “Marching Cubes: A high-resolution 3-D surface construction algorithm.” Computer Graphics (SIGGRAPH ’87), pp. 163-169, 1987.

[11] A. Johnson and M. Hebert. "Recognizing objects by matching oriented points." Carnegie Mellon Robotics Institute Technical Report CMU-RI-TR-96-04, 1996.

[12] A. Johnson and M. Hebert, “Efficient Multiple Model Recognition in Cluttered 3-D scenes.” Proc. IEEE Conference on Computer Vision and Pattern Recognition. Santa Barbara, June 1998.

Appendices

Appendix A, Hardware Description

A.1 Overall Hardware Configuration

Figure A-1 RTSA Hardware Schematic shows the currently existing hardware and interconnections. The set of hardware in use may be decomposed into a real-time part, including the robot and its peripherals and controller, and a non-real-time part, including the RTSA model builder and task planner and sensor head connections.

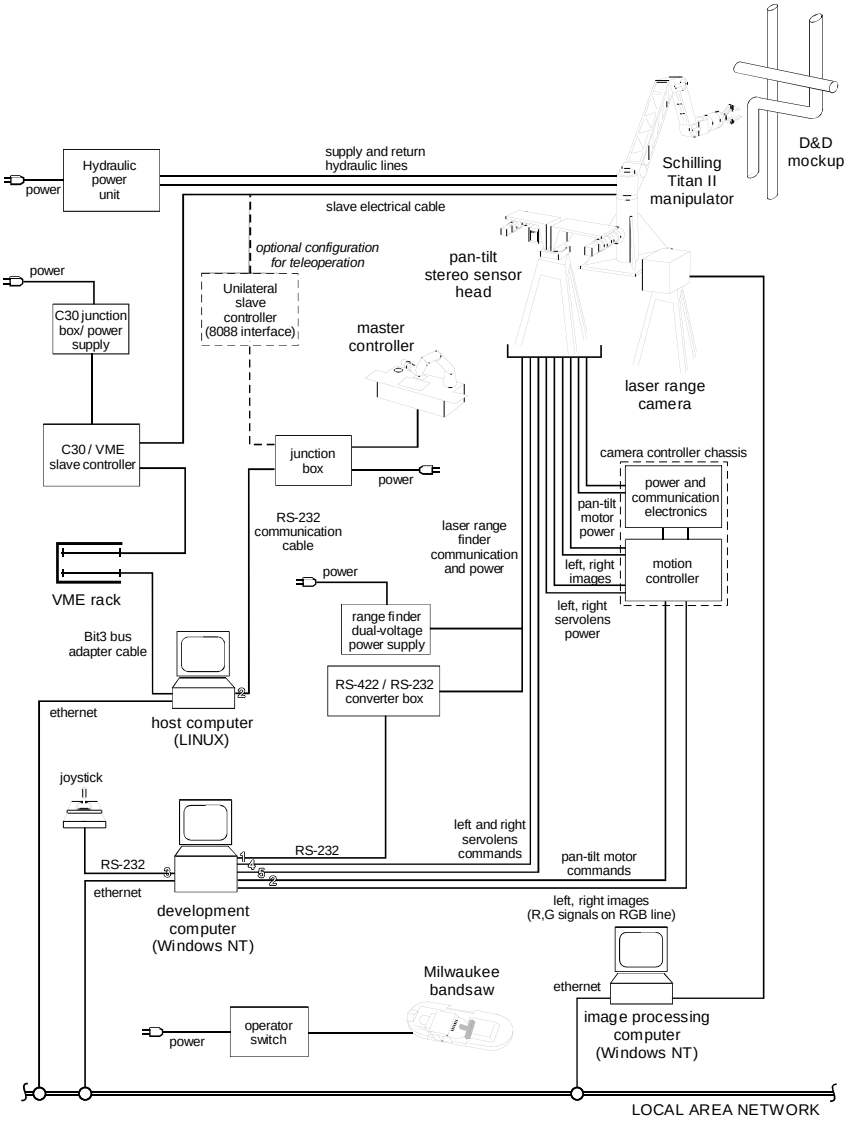


Figure A-1 RTSA Hardware Schematic

A.2 Schilling Manipulator

The Schilling Titan II is a six-degree-of-freedom hydraulic manipulator constructed primarily of titanium and weighing 225 pounds, with a reach of approximately 76 inches and a payload at full extension of 240 pounds. It incorporates a two-finger gripper with a maximum opening of 5 inches. The manipulator in the Robotics and Electromechanical Systems Laboratory is on loan from Oak Ridge national Laboratory (ORNL) and was a part of the Dual Arm Work Platform (DAWP). It is securely mounted on the lab floor and will be used to test the RTSA telerobotic system on a mockup in this lab

A.3 Unilateral Slave Controller

The full name for this controller is the unilateral 8088 backup control. It may be connected directly to the master controller and slave manipulator to provide joint-to-joint control between the two. Its use requires the proper master control PROM. While this configuration for the system controller was used to ensure that the hardware was initially operational, the RTSA system will nominally use the alternative host computer configuration

A.4 Master Controller

The master controller is the input device for manual teleoperation. Its primary component is the master arm itself, a six-degree-of-freedom articulated arm with an approximately 11 inch reach. The box on which it is mounted has a power switch and 12 function keys, an LCD screen, and an RS-232 port with which to communicate with the unilateral slave controller or host computer. These two modes are each controlled by one of two PROM integrated circuits that must be installed on the computer board inside the master enclosure. The master arm has a freeze button on its terminal end, and two textured bands that may be squeezed to open and close the gripper. In the unilateral slave control mode, the master screen has a series of menus to determine the system behavior in various ways. In the other mode, the host computer is responsible for sending and reading commands from the master and displaying information on the screen.

A.5 C30/VME Slave Controller

The C30/VME slave controller is an element of the host computer control system. It is connected directly to the slave manipulator with the same cable used in the unilateral 8088 control configuration. The C30 controller is also connected to a card which provides the interface

to joint, torque, pressure, and other data from the manipulator. This card is meant to be placed in a VME card cage for access by the host computer.

A.6 Milwaukee Band Saw

The Milwaukee Portable Band Saw, Model 6230, will be the primary tool used in RTSA dismantlement demonstrations. It has a ½" wide, 10 tooth per inch blade which can be operated at variable speeds from 0-350 SFPM. The saw has a maximum capacity of 4-3/4" x 4-3/4" for square stock and 4-3/4" diameter round stock.

Figure A-2 Milwaukee Bandsaw with Modified Handle (lerft), grasped by Titan II Manipulator

A.7 Host Computer

The host computer is the real-time computer on which the Schilling controller runs. It is a Dell dual-processor-capable 450 MHz Pentium III PC with 128 M RAM. It is capable of being booted with Windows NT, QNX, or LINUX, although the current system will run ControlShell under the LINUX operating system only. It is interfaced with the master controller through a serial port, with the C30 controller by way of a Bit3 bus-to-bus adapter, and the Ethernet for receiving downloads of task plans.

A.8 AUTODESK INVENTOR MODELS OF THE SENSOR HEAD AND ITS COMPONENTS

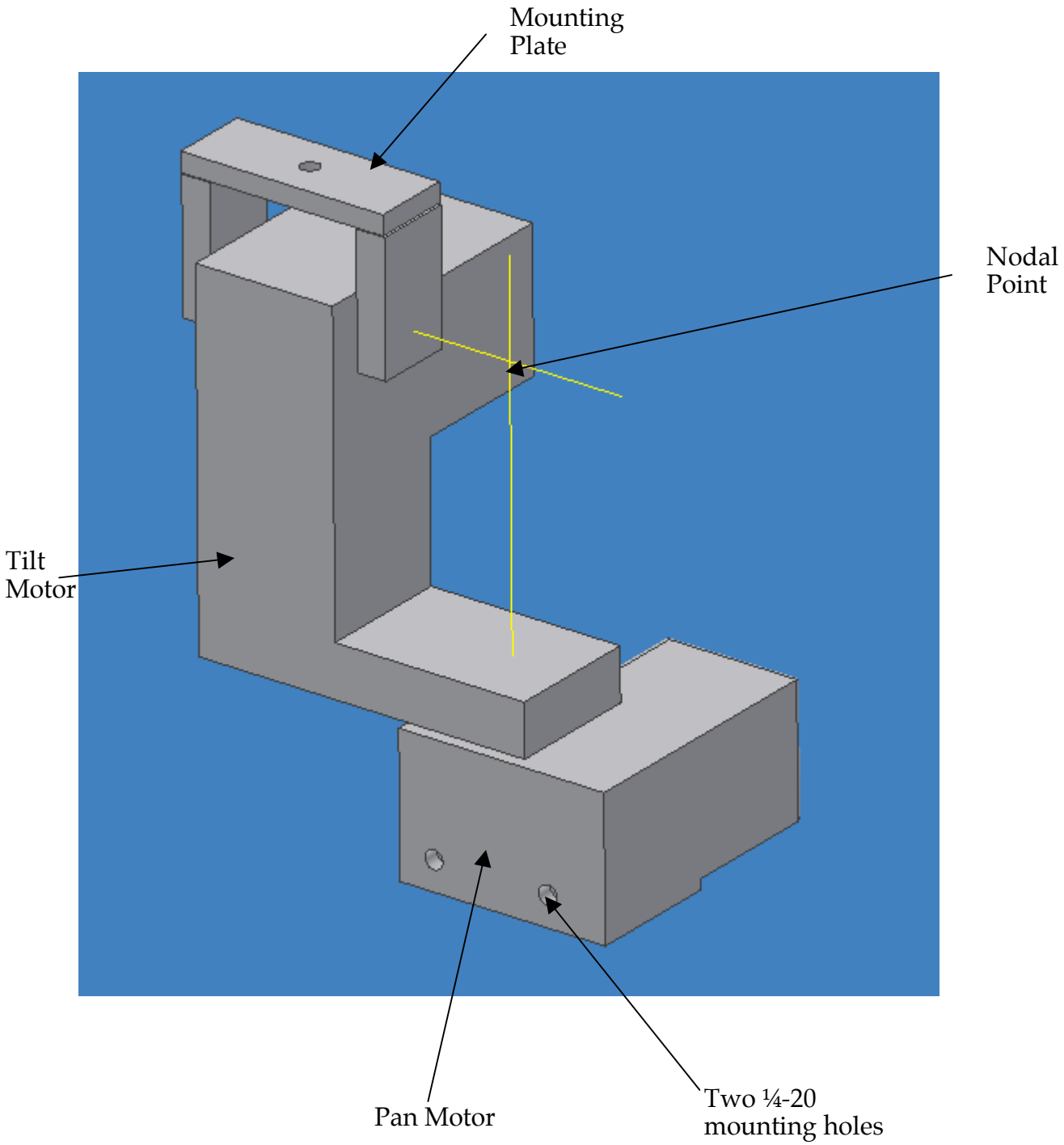


Figure A-3 Autodesk Inventor™ Model of Pan/Tilt Unit

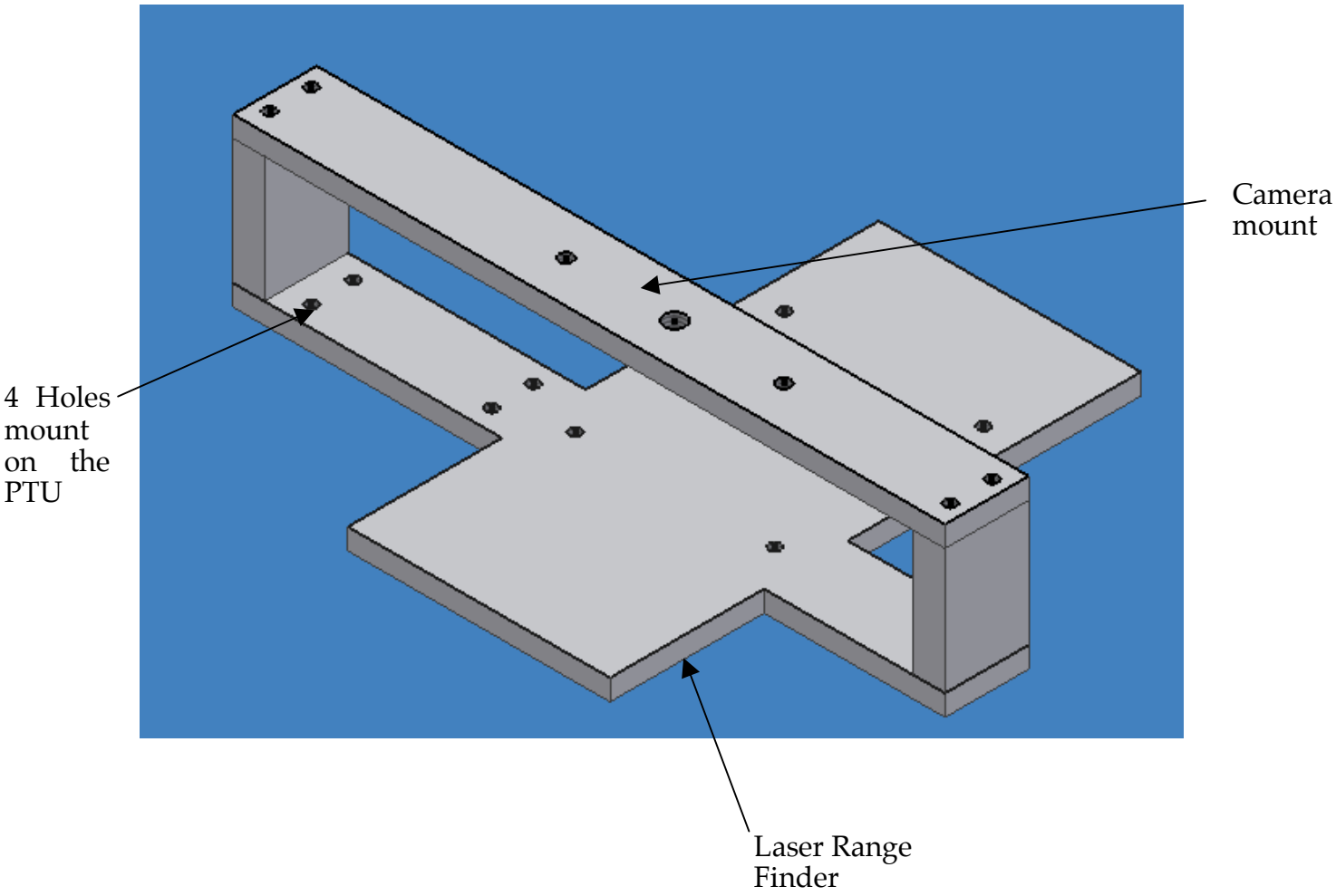


Figure A-4 Autodesk Inventor™ Model of Sensors Mounting Bracket

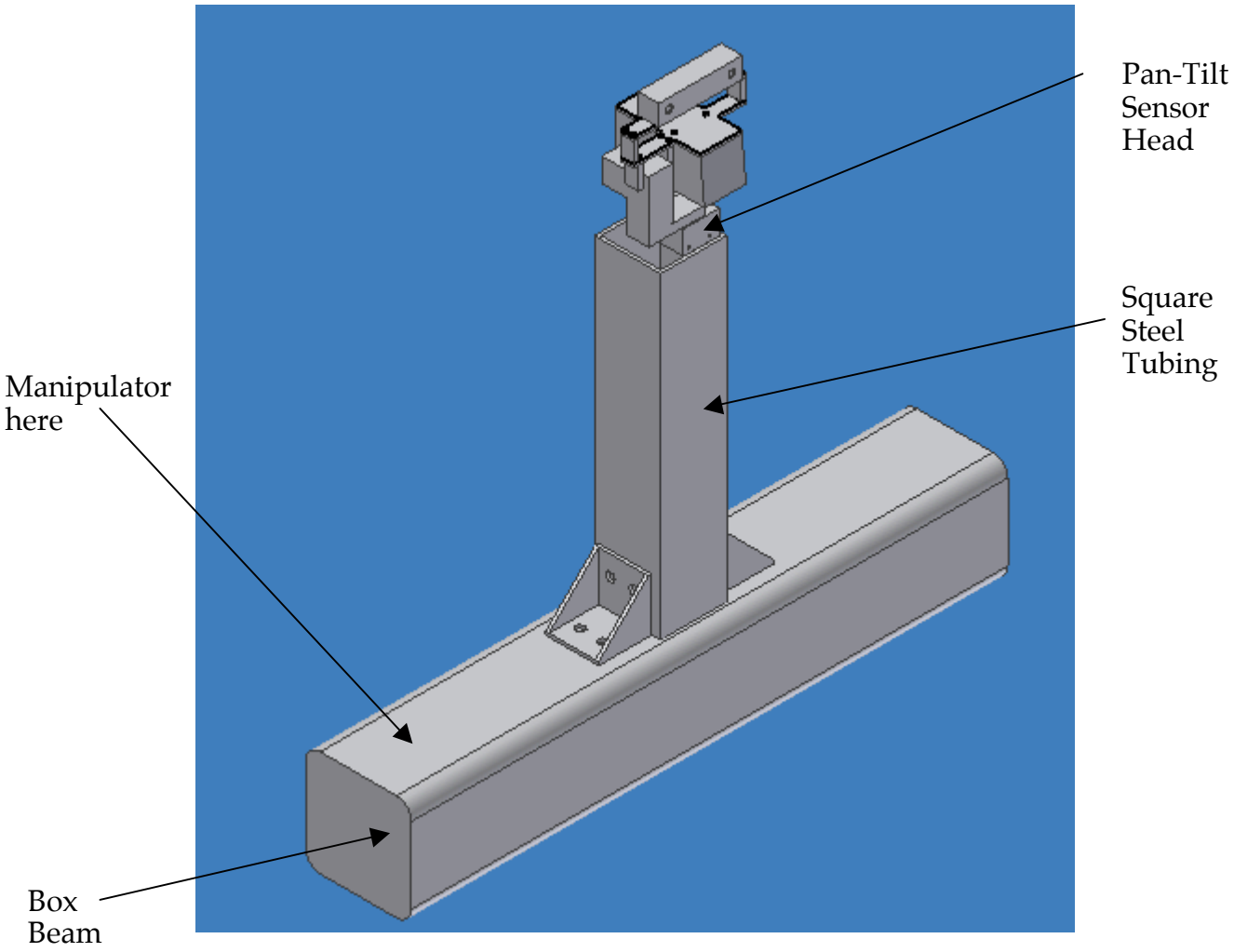


Figure A-5 Autodesk Inventor™ Model of Sensor Head Mounting Structure

APPENDIX B, HMCTR CONTROLSHELL™ IMPLEMENTATION

B.1 Introduction

ControlShell is the development tool created by Real-Time Innovations (RTI) used to implement the Schilling controller. It is a graphical component-based tool that provides some automatic generation of code to build and maintain real-time applications quickly and easily. It allows users to create continuous flow diagrams graphically and link them with finite state machines to dictate the behavior of the system. Programmers may use previously written components that are provided by RTI in a repository or generate special purpose blocks into which user-written code is integrated. The control system is then complied for the desired machine and executes the control strategy in real-time.

Below the application level, there are objects of several types that may be found in ControlShell. The Composite Object Group (COG) may encapsulate both sample-data system elements and event-driven elements. The Finite State Machine (FSM) is also a composite object that consists of a state transition diagram that represents an event-driven program. The FSM usually appears in its file with associated continuous flow diagrams that are responsible for providing stimuli and running state transition components. There are also three types of primitive components in ControlShell: the state transition component (STC), the data flow component (DFC), and the atomic component (ATC). The state transition component provides actions taken by a FSM in response to a stimulus. The data flow component is the building block for the sampled-data part of the system that executes routines at every sample period. The atomic component provides generic utilities or functions to other components in the same diagram.

Connections between components in ControlShell take one of three forms. Pin connections are inputs and outputs of data from components. Bubble connections provide utility of functions from one method to another. Interfaces allow pins and bubbles to be bundled in a unified connection.

Another important aspect of ControlShell is the ability to define operating modes. A mode is a set of active components which may be enabled and disabled as a unit, allowing the event-driven part of the system to alter the system behavior automatically in response to given stimuli.

B.2 Human-machine Cooperative Telerobotics Controller

This section describes the Schilling telerobotic control system being developed in ControlShell. Most part of the control system has been implemented except the assistance function. The information below describes the current concept.

B.2.1 Top Level controller

At the highest level of the HMCTR controller, there is one Composite Object Group (COG), which contains all other components, shown in Figure B-1. In the main COG, there are two primary COGs: the robot discrete controller and the robot continuous controller, as shown in Figure B-2. The discrete controller is responsible for determining the mode of operation of the controller, whether initiated by the operator or by the downloaded task plan. The continuous controller receives data and commands from the discrete controller to enable predefined sets of components so that the controller behaves in the desired way.

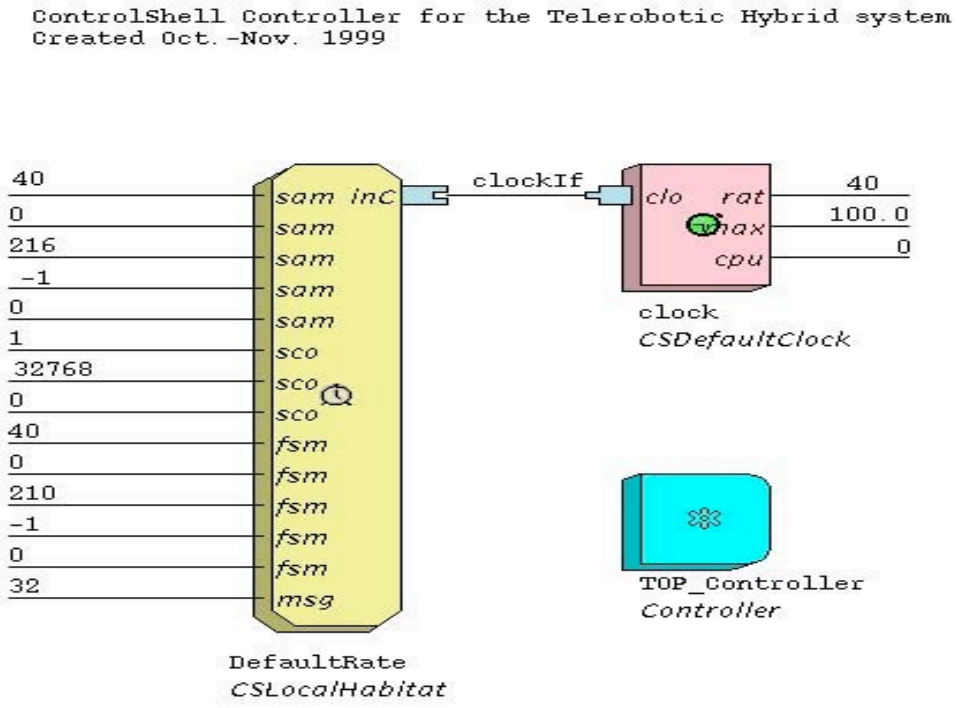


Figure B-1 Top Level Controller

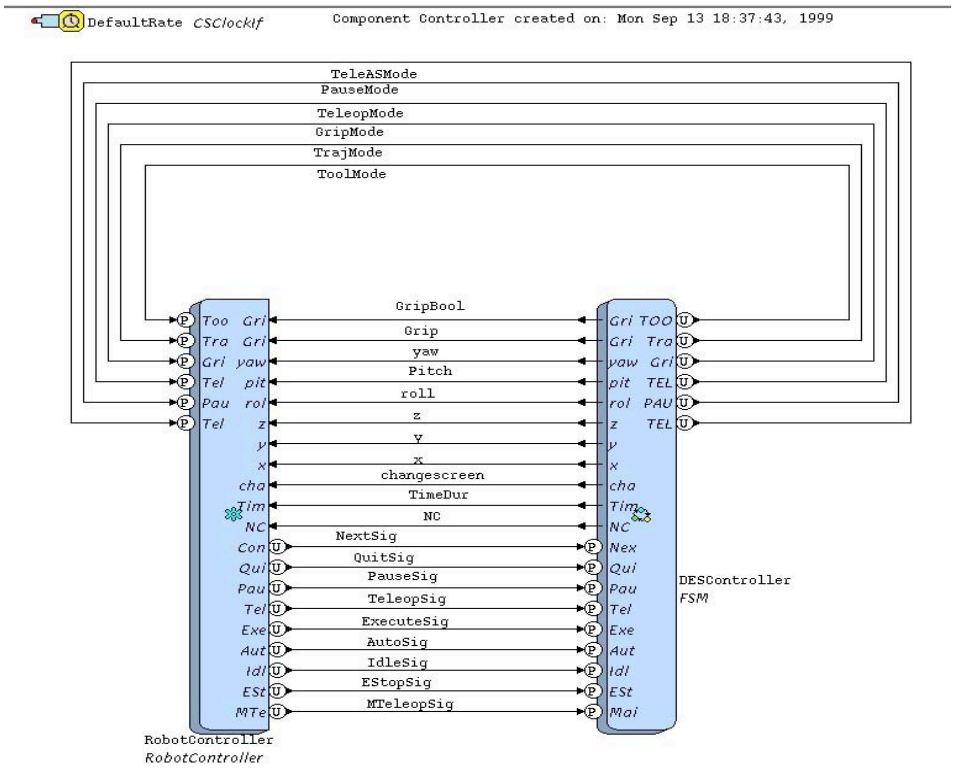


Figure B-2 Robot Continuous and Discrete Control COGs

B.2.2 Robot Controller (Continuous)

The robot continuous controller shown in Figure B-3, is the data flow diagram which encompasses the various capabilities for modes of operation of the manipulator and sends and retrieves information from the lower level control loops in the robot closed loop control COG. The MasterCommunication and Robot components are always activated during operation of the system (The red blocks), but in the other four modes of operation available, a unique set of components is enabled. In the PAUSE mode, TELEOP or TELEOPAS (teleoperation with assistance function) no additional components are activated. In the TRAJ mode, Component Dfc0, Dfc7, Dfc8 and XYZQT_G will be activated.

There are two main sources for robot control signals in this diagram: one the signals from discrete controller, and the other one is the MasterCommunication component. These components are mutually exclusive sources for control, and only one at a time will be enabled in a given mode of operation. Each of the two also has access to the SendStimulus component “NEXT” in the FSM subchain so that when it is finished executing, it can signal for the next action in the plan to be initiated.

There are several ActivateMode components in this diagram which cause the mode to be changed when called from various sources throughout the diagram. The components make it possible to activate the PAUSE, TELEOP, TELEOPAS, or TRAJ mode by providing the proper function to users elsewhere in the system.

The MasterComm component is the communication component with the minimaster. It is always enabled so that button information from the master may be used to transition from state to state regardless of the current state of the system.

The Robot component in this diagram receives high level position commands from either autonomous or manual inputs and outputs its actual position. The methods of closed loop control and available modes are described in Section 3.

Dfc0, Dfc8, Dfc7, and XYZQT_G receive the final point of the trajectory in Cartesian space from the task plan and convert it into joint space and send it to trajectory generator in the low level continuous controller.

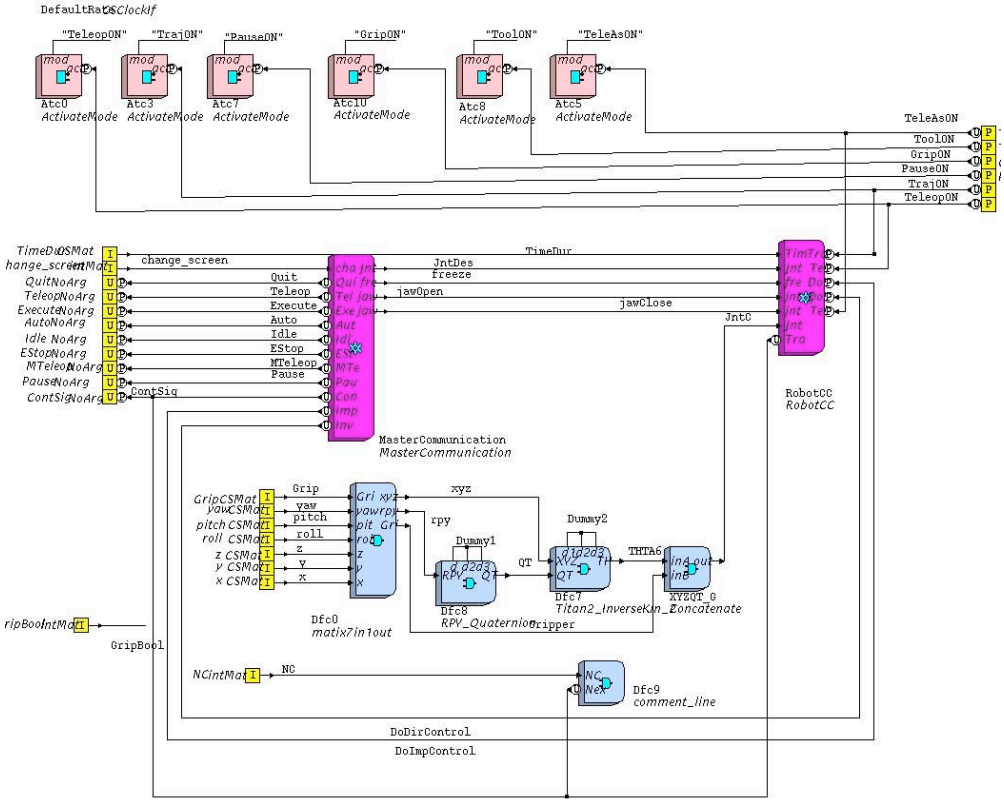


Figure B-3 Continuous Control COG Elements

B.2.2.1 Mastercommunication

The MasterCommunication component is the communication component with the minimaster. It is always enabled so that button information from the master may be used to transition from state to state regardless of the current state of the system. Its 'use' bubbles each correspond to a different button function, rather than to a physical button, so that the same button may call different functions while the master is in different submenus. (The ControlShell text interface and minimaster menu screens are described in Section 4). The MasterComm component also has two continuous outputs, the GripperPos and JntPos signals. Figure B-4 shows the structure of the Mastercommunication component.

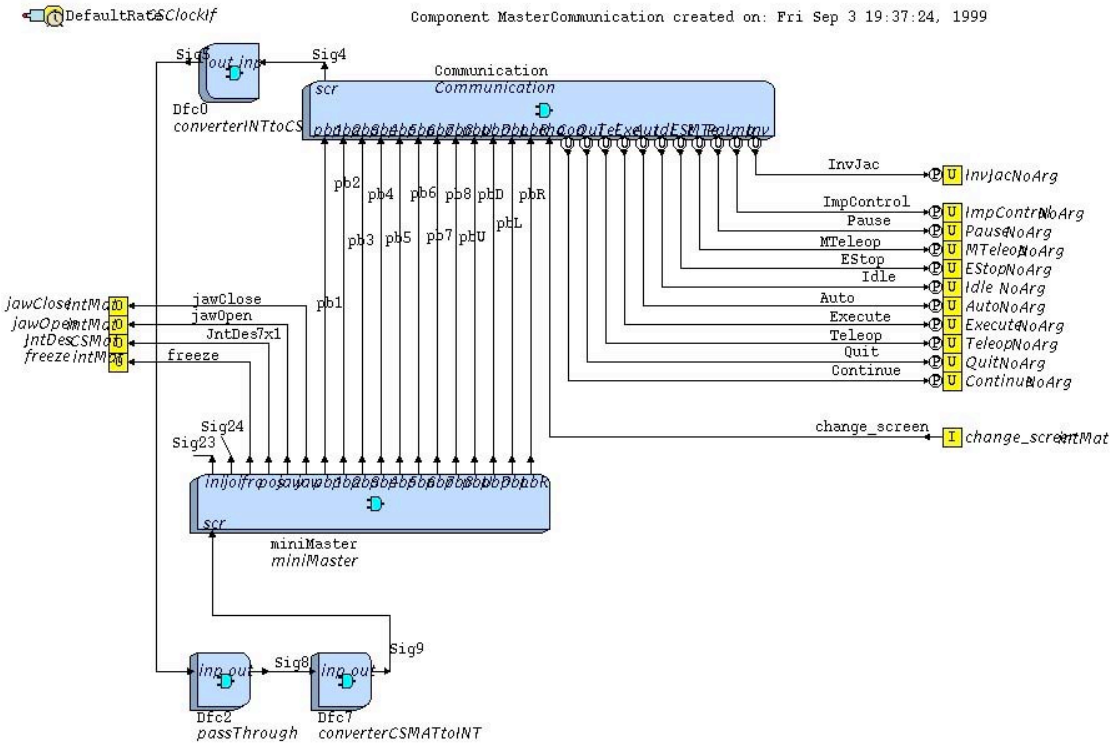


Figure B-4 Communication Structure

B.2.2.2 RobotCC

RobotCC, shown in Figures B-5 through B-9, is the robot low-level closed loop control structure. In addition to the direct joint to joint control (pure teleoperation) between the master and slave, two different continuous closed loop control schemes are considered: teleoperation with assistance function control and autonomous control which get control signal from a joint space trajectory generator. At the beginning of the task execution, the operator selects one of these three control strategies in which to operate with the master console. That button sends the command to activate one of the following modes of the control execution: “Teleop”, “Autonomous”, or read “Teleop_assist” from task plan by using a build-in ActivateMode component in the robot closed loop control diagram. Each of these modes enables a different set of the components. For example, if teleoperation control is selected, the blocks in red are activated (Figure B-5, Figure B-6).

Figure B-7 and Figure B-8 show the mappings for the autonomous control. Analogous to the teleoperation control, Figure B-6 and Figure B-9 show the mapping for the teleoperation with assistance function control.

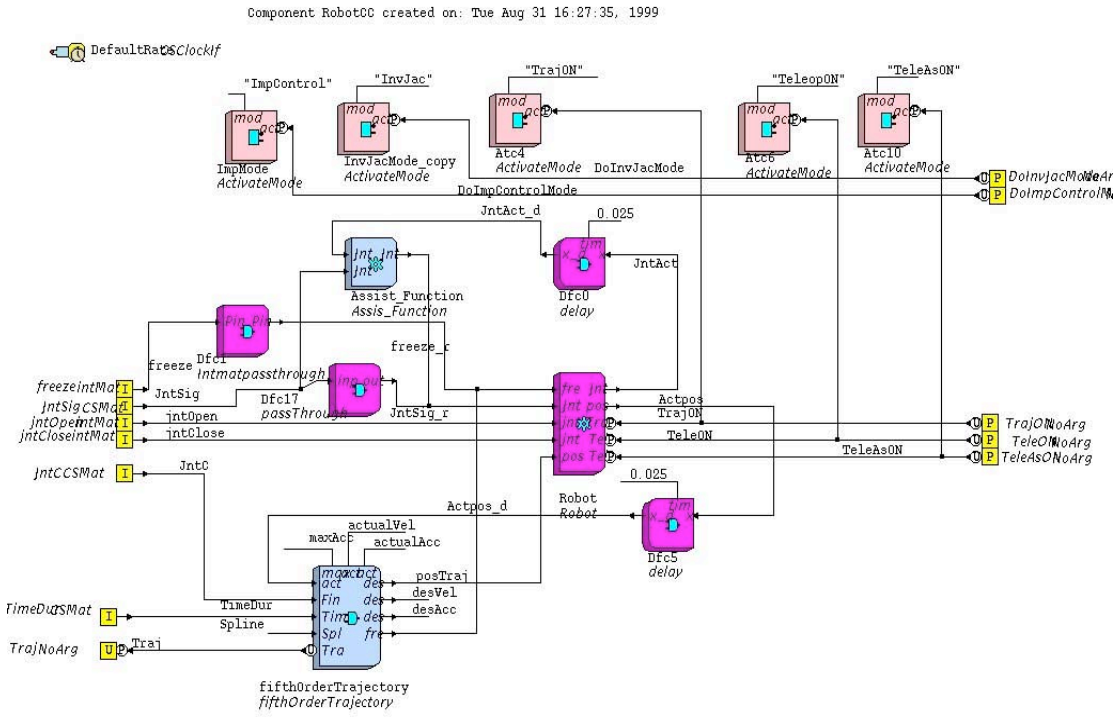


Figure B-5 Closed Loop ControlComponents with Teleoperation Mode Activated

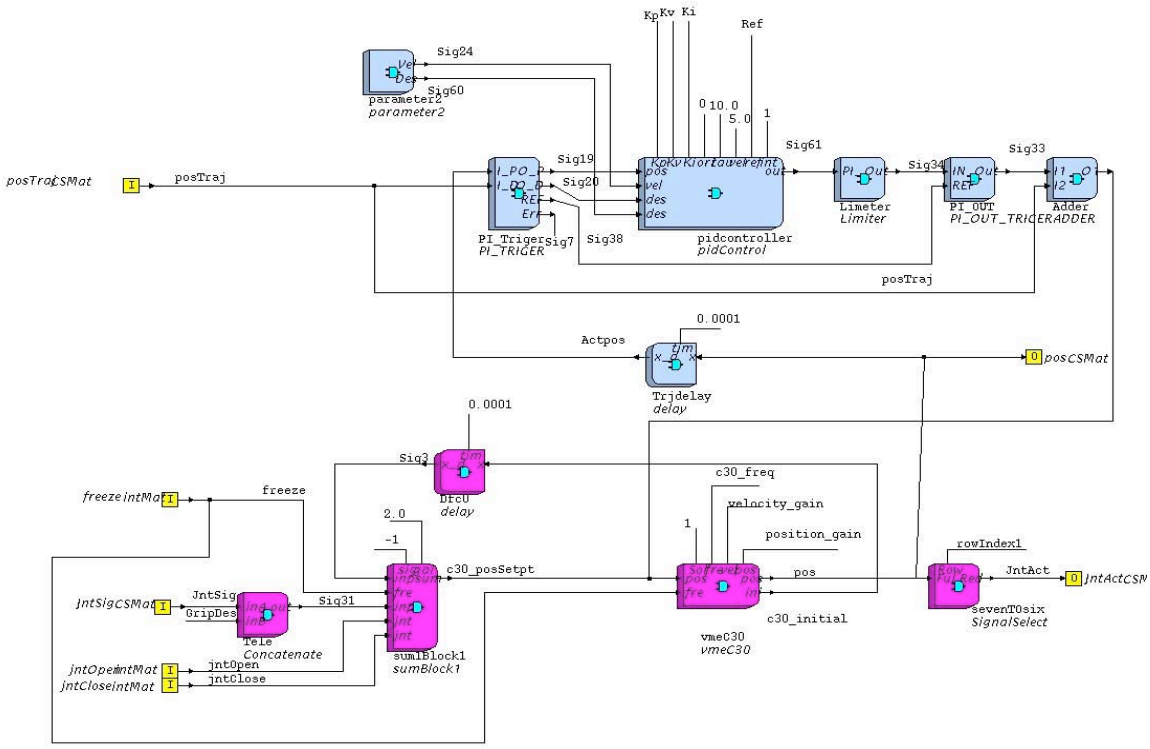


Figure B-6 Lowest Level Control Components With Teleoperation w/o Assistance

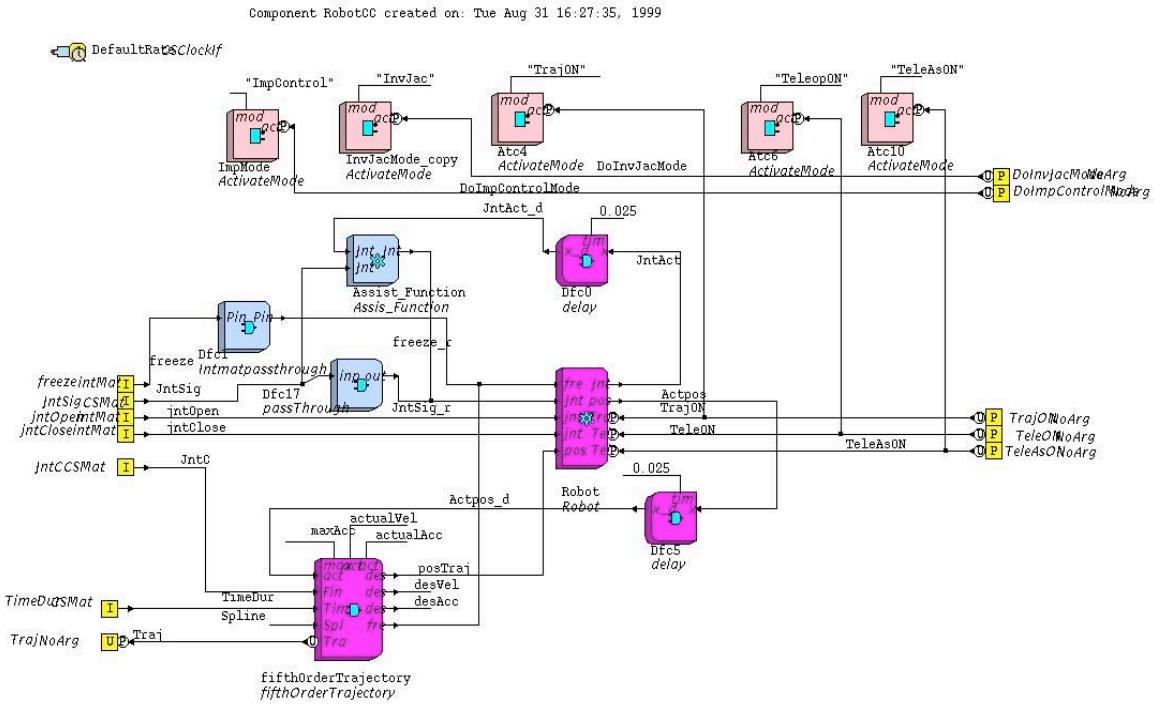


Figure B-7 Closed Loop Control Components with Autonomous Mode Activated

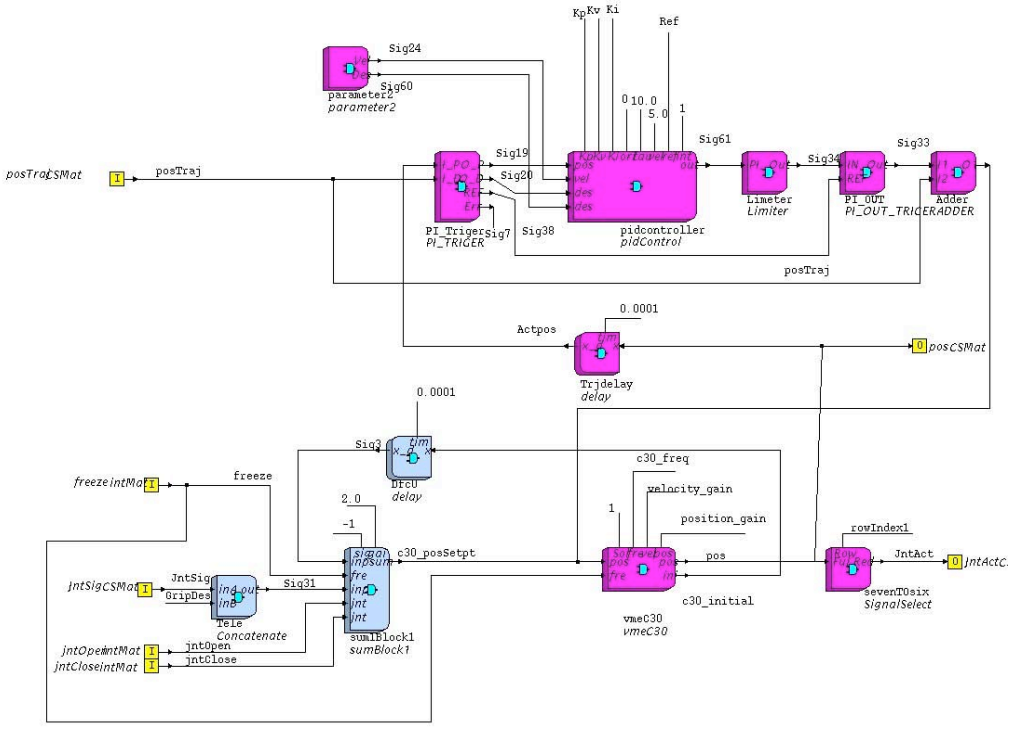


Figure B-8 Lowest Level Control Components with Autonomous Mode Activated

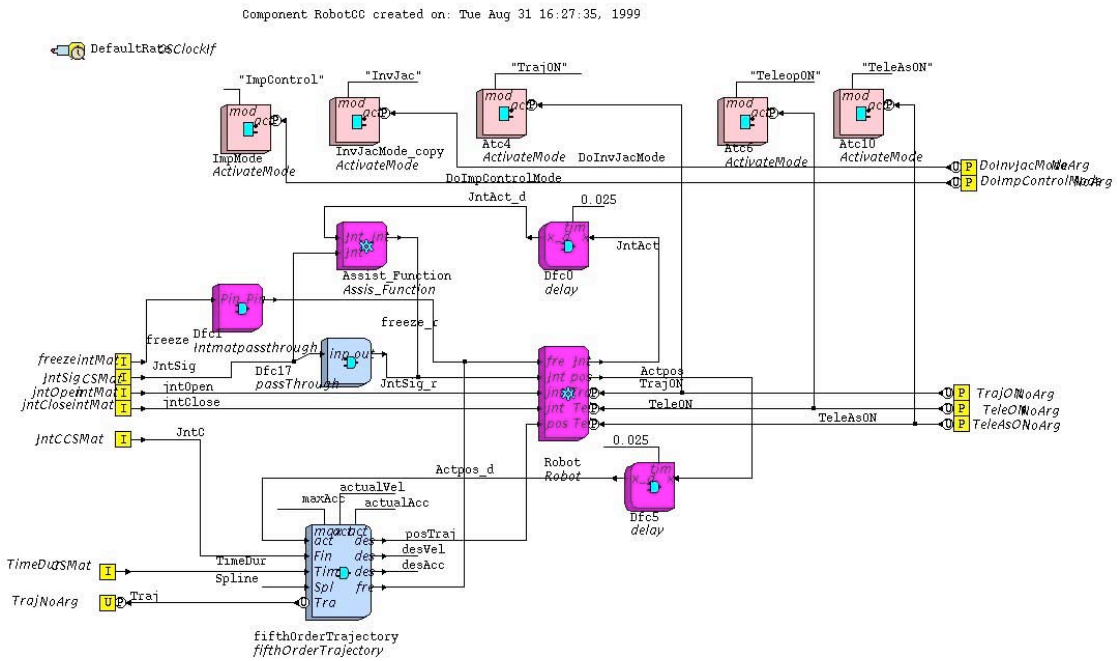


Figure B-9 Closed Loop Control Components with Teleoperation with Assistance Mode Activated

B.2.3 DesController (Finite State Machine)

The components which comprise the DesController COG which decide robot work mode are shown in Figure B-10. At this level, there is a main finite state machine that determines whether the robot is being controlled automatically or manually in the absence of a plan or if it is idle. The system may transition from IDLE to either the autonomous execution state AUTOEXEC or the teleoperation state MANUAL_TELEOP given the stimuli "AUTOEXEC" or "MANUAL_TELEOP", respectively. The AUTOEXEC state is actually composed of a finite state machine subchain described in Section B-2.3.1. Upon transition to one of these states, the state transition component StartTeleop ( ) or SartAuto ( ) runs. When the state is transitioned back to idle, the transition component Backside ( ) runs. The purpose of these transition components is to verify that the proper hardware is connected and operational, and configure the system to run in the desired mode. If there is a problem in the transitions out of the IDLE state, the value ERROR is returned, and the system state returns to IDLE. If everything checks out, the return value is OK and the system transitions to the desired state. There is also an ESTOP state where

the system can transition to from any other state if there has been an emergency stop. From there, it can only transition back to the IDLE state.

Accompanying the main finite state machine in the same file is a set of components which contain the transition and other code associated with this FSM. At this level, there are three state transition components as described above: StartAuto( ), StartTeleop( ), and BackToIdle( ). Each of these is a user of an ActivateMode component found in the robot continuous control diagram (see Figure 3) which enables the set of components corresponding to that state. Also found in this diagram are the components which provide code to the master console buttons (see Figure B-3) to send stimuli such as “ESTOP”, “IDLE”, “AUTOEXEC”, and “TELEOP” which cause the main FSM to transition. The primary component in this diagram accompanying the FSM is component corresponding to the automatic plan execution subchain. It is user and provider of several functions which exist in the robot continuous control diagram, as well as providing some data for that controller originating from the execution plan.

B.2.3.1 Autonomous execution mode control subchain

Since this is an FSM subchain, there are two main parts to the file, the subchain itself and the associated data flow components. THE FSM subchain is responsible for defining the current behavior of the system as determined by operator and automatic stimuli, and the data flow blocks are responsible for retrieving planned execution steps and enabling the proper set of components for the given action.

The FSM subchain consists of seven states, including START, PAUSE, EXEC, UNPLAN TELEOP, END PLAN, END, and ERROR. The START state is a beginning condition when the subchain is entered from the main FSM, and automatically transitions to the PAUSE state, where the system waits for the operator to signal for execution of a predefined plan or other desired action. From PAUSE, the operator may send stimuli associated with master console buttons corresponding to “QUIT”, “EXECUTE”, or “TELEOP”. The “QUIT” stimulus transitions the diagram to an END state in which the subchain exits to the main FSM and to the IDLE state. This signal is used if a plan is to be abandoned and the associated GotoQuit( ) transition component disables the appropriate components. The “TELEOP” stimulus is sent if the operator wishes to initiate an episode of teleoperation, which has not been incorporated into the downloaded plan. It is used in the case of an unforeseen obstacle or change in approach to a task. The StartTeleop( ) transition component runs at this point to activate the teleoperation mode by using an ActivateMode component in the robot continuous control diagram (see Figure B-3). Transition

back to the PAUSE state from the UNPLAN TELEOP state is accomplished by sending the "PAUSE" stimulus and the appropriate ActivateMode component is called by the GotoPause( ) transition component.

The "EXECUTE" stimulus initiates or restarts the execution of the downloaded task plan. The FetchandParse( ) state transition component accompanies it and the "NEXT" stimulus in the transition out of the EXECUTE state. It is responsible for opening the task file on its first call, retrieving the first unexecuted action from this file, calling the appropriate ActivateMode components in the robot continuous control diagram, and setting values for the gripper state for a toggle gripper action or final coordinate of the manipulator for a move action. Once modes are changed and values are set, it returns one of three values: CONT, DONE, or ERROR. The CONT return value results in a branch, which takes the diagram back to the EXEC state, where the next action can be retrieved upon completion of the previous action. The DONE return value signals that the action retrieved was the last in the file and the diagram should transition to an end state to wait for this last action to be completed before it transitions out of the subchain. The ERROR return value indicates that the file has been corrupted and the system should display an error message and transition out of the autonomous execution subchain. The "NEXT" stimulus may be sent by the SendStimulus component in this diagram, which is a provider to several other components in the continuous control diagram. It may be sent by a component executing an automatic action when that action has finished, or by the operator signaling that a planned episode of teleoperation has been completed.

The END PLAN state was included so that a final action could be completed before transitioning out of the AUTOEXEC subchain. When in this state, the "NEXT" signal sent to indicate a previous action is complete will cause the transition to the END state where the diagram automatically transitions back to the main FSM.

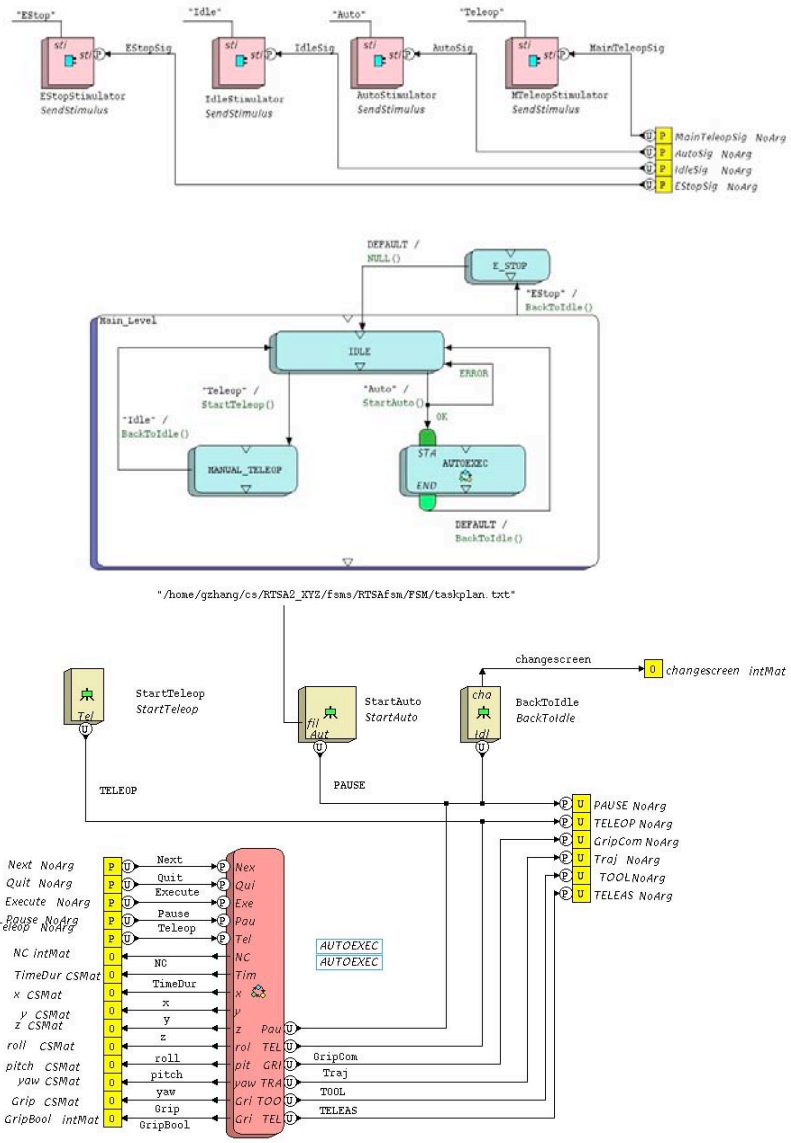


Figure B-10 DESController COG Components

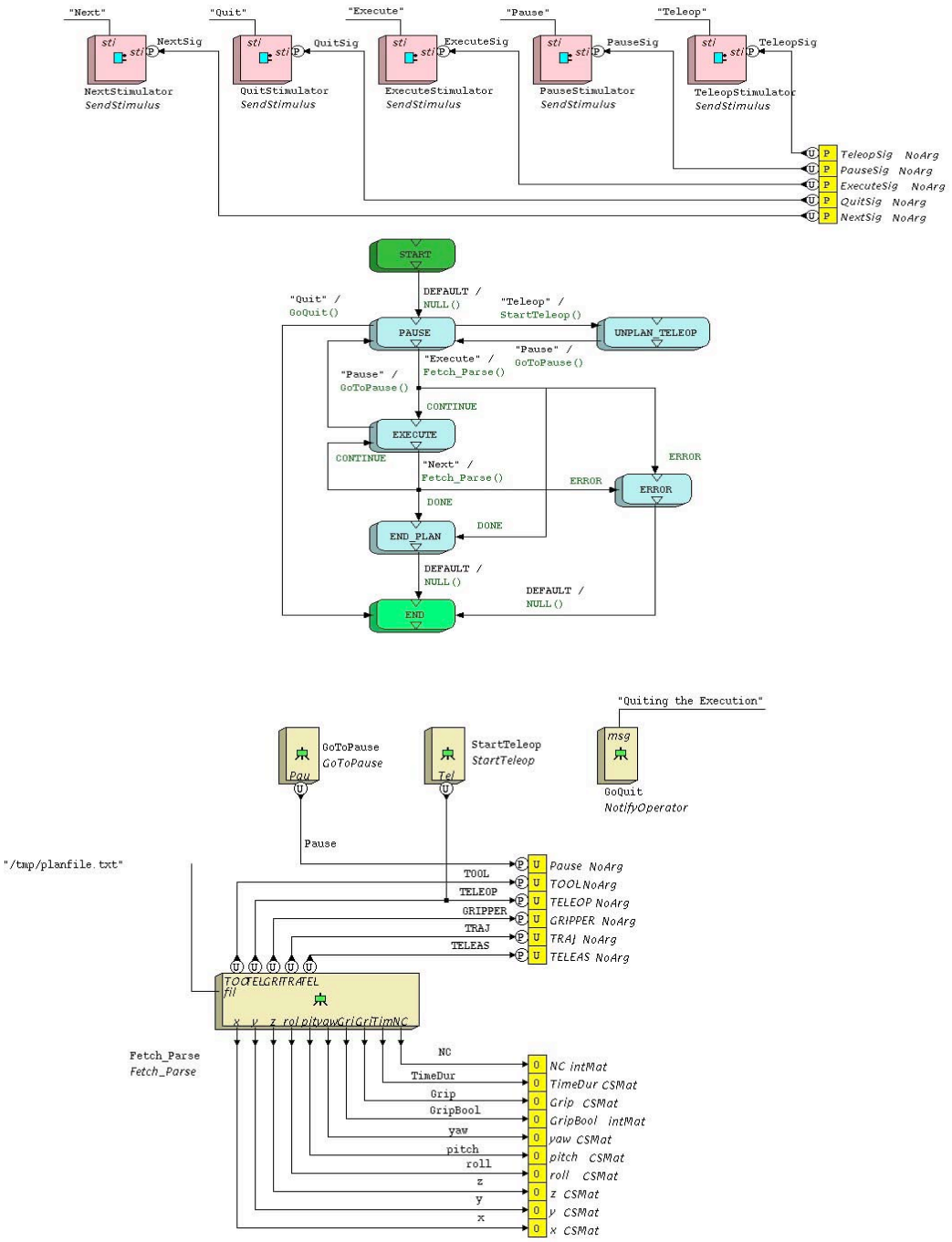


Figure B-11 Autonomous Execution Subchain Components

B.3 Control Interface

B.3.1 ControlShell text interface

ControlShell provides access to a run-time menu, which can be used to alter operating modes or variable values during execution. Additional custom menu items in the Schilling telerobotic system will allow the operator to inspect, edit, exit, and re-enter the task plan which has been downloaded when desired. When the operator wants to see or to edit the task plan, this menu option may be invoked by typing the command string ‘task plan’. The method of integration of this menu option and associated submenus into the standard ControlShell menu has not yet been established.

B.3.2 ControlShell Messages

Other option of the ControlShell text interface with the operator is C++ print messages. If the C++ code for the corresponding ControlShell function includes any print message commands this message will be printed on the display each time function is executed. For example, if ControlShell is unable to open the TaskPlan.txt file during the FetchandParse function execution, a message such as “Task plan file cannot be opened. Returning to ‘Idle’ State” will be displayed. The message communication between ControlShell and the operator allows the operator to always be aware of what is going on during the program execution.

B.3.3 Minimaster Menu Screens

The easiest way for the operator to communicate with the controller is through the minimaster buttons. They are accessed in the software by using the MasterCommunication interface. The master has twelve different buttons which can be used to activate the different stimuli in ControlShell. By defining different submenu screens on the minimaster, the same buttons can be used to call several different functions. The MasterCommunication COG is shown in Figure B-4.

Each signal in the miniMaster block corresponds to a different button on the Minimaster. When a button is pressed, a Boolean signal is sent to the Communication block. Depending on the submenu screen number in the Communication block, each signal is used to call the corresponding function through a “bubble” provided by code outside this file. In addition, when the submenu must be changed, the Communication block sends the “screen” Boolean signal to the miniMaster.

Two different submenu screens have been created. Figure B-12 shows screen number 1 corresponding to the beginning of the program or each time program has to be restarted.

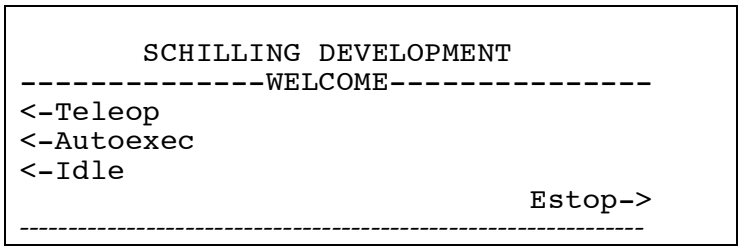


Figure B-12 Welcome Screen in which Control Scheme is Selected

After the operator makes a selection and pushes the corresponding button on the Minimaster, the screen will be changed to the next one using the Boolean "screen" signal from the Communication block. The submenu screen shown in Figure B-13 corresponds to the Autoexec state of the system.

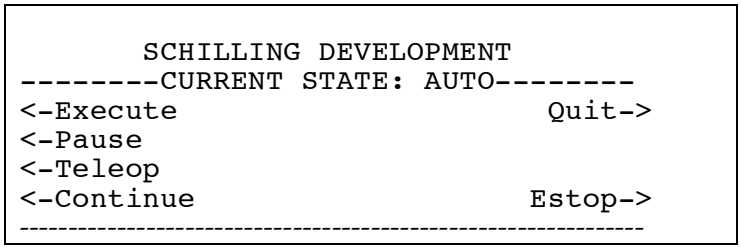


Figure B-13 Minimaster Autonomous Execution Menu

B.4 Task Plan File

The task plan file is a text document containing the sequence of atomic actions that are to be performed by the telerobotic system. Each action is fully described by a C structure teleoperation state, and final position information for move commands. It is downloaded over the Ethernet by the task planner to the real-time control computer when the operator is satisfied it is complete. The format of the file is such that it can be parsed by a state transition component in ControlShell

called ‘fetchandparse’, as described in Section 2.3.1. An example of an actual plan file created by the planner and downloaded to the controller is shown as follows:

```
1  move    x -265.0    y 1400.0    z 960.0    roll 90.0    pitch 43.0    yaw  0.0   Gripper 100   TimeDur 15
2  move    x -265.0    y 1580.0    z 895.0    roll 90.0    pitch 43.0    yaw  0.0   Gripper 100   TimeDur 15
3  move    x -265.0    y 1572.0    z 895.0    roll 90.0    pitch 43.0    yaw  0.0   Gripper 70   TimeDur 15
4  move    x -265.0    y 1572.0    z 1100.0    roll 90.0    pitch 45.0    yaw  0.0   Gripper 70   TimeDur 15
5  move    x 100.0    y 1200.0    z 1300.0    roll 90.0    pitch 45.0    yaw  0.0   Gripper 70   TimeDur 15
6  move    x 1274.1    y 187.0    z 1181.5    roll 28.5    pitch -0.6    yaw 178.0   Gripper 70   TimeDur 15
7 teleop_manual
9  move    x 1296.1    y 198.9    z 1181.7    roll 28.5    pitch -0.6    yaw 178.0   Gripper 70   TimeDur 15
10  move    x 1318.1    y 210.9    z 1182.0    roll 28.5    pitch -0.6    yaw 178.0   Gripper 70   TimeDur 90
11  move    x 1296.1    y 198.9    z 1181.7    roll 28.5    pitch -0.6    yaw 178.0   Gripper 70   TimeDur 90
12  move    x 1274.1    y 187.0    z 1181.5    roll 28.5    pitch -0.6    yaw 178.0   Gripper 70   TimeDur 15
13  move    x 1238.3    y -31.8    z 1080.7    roll -0.4    pitch -1.9    yaw 179.7   Gripper 70   TimeDur 15
14  move    x 1263.3    y -32.0    z 1081.5    roll -0.4    pitch -1.9    yaw 179.7   Gripper 70   TimeDur 15
15  move    x 1288.3    y -32.1    z 1082.3    roll -0.4    pitch -1.9    yaw 179.7   Gripper 70   TimeDur 90
16  move    x 1263.3    y -32.0    z 1081.5    roll -0.4    pitch -1.9    yaw 179.7   Gripper 70   TimeDur 90
17  move    x 1238.3    y -31.8    z 1080.7    roll -0.4    pitch -1.9    yaw 179.7   Gripper 70   TimeDur 15
18  teleoop_assist
19  move    x 600.0    y 1000.0    z 1400.0    roll  0.0    pitch  0.0    yaw 90.0   Gripper 70   TimeDur 15
20  move    x -265.0    y 1400.0    z 1100.0    roll 90.0    pitch 43.0    yaw  0.0   Gripper 70   TimeDur 15
21  move    x -265.0    y 1572.0    z 895.0    roll 90.0    pitch 43.0    yaw  0.0   Gripper 70   TimeDur 15
22  move    x -265.0    y 1572.0    z 895.0    roll 90.0    pitch 43.0    yaw  0.0   Gripper 100   TimeDur 15
23  move    x -265.0    y 1400.0    z 1100.0    roll 90.0    pitch 43.0    yaw  0.0   Gripper 100   TimeDur 15
24  move    x -265.0    y 1000.0    z 1100.0    roll 90.0    pitch 43.0    yaw  0.0   Gripper 100   TimeDur 15
```

Figure B-14 Example Plan File

Appendix C, Simulation Results and Code Assist Functions

C.1 Simulation Results of Planar Assistance function

C.1.1 Planar Assist Function Simulation Results

A test of the code was performed to measure the planar assistance function. The program was executed at the USF test bed, and the scaled and non-scaled velocities were recorded. The following graphs represent the results obtained from the planar assistance function showing the actual velocity command sent to the robot controller.

Figure C-1 represents the linear velocity scaling. The graph contains the Z-axis velocity of the constraint plane. Motion in the Z-axis causes the manipulator to go away from the constraint plane. The graph shows that the motion away from the constraint plane is scaled down by the scale value of 0.1.

Furthermore, Figure C-2 shows the roll velocity command and the scaled roll velocity. During the cutting operation the roll axis should not change. From the graph, the angular velocity in the roll direction is scaled down by a value of 0.1.

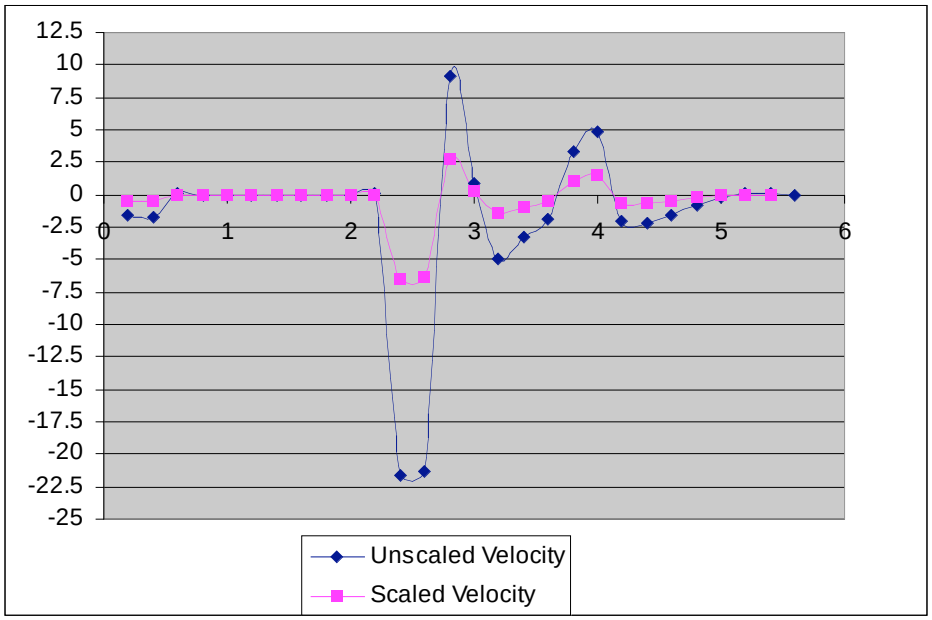


Figure C-1 Scaled and Non-Scaled Z-Axis Velocity

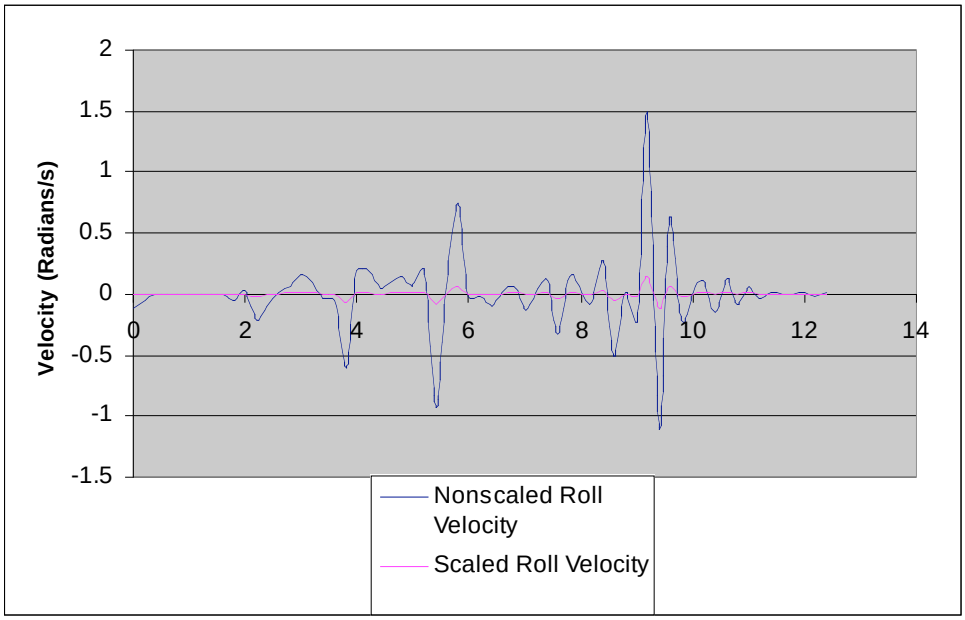


Figure C-2 Scale and Non-Scaled Roll Velocity

The planar assist function also requires the roll axis to be pointing in the direction of the constrain plane. This means that the yaw direction should line up with the plane initially, and then stay in that direction during the task operation. This means that the yaw velocity is scaled down by a value of 0.1. Figure C-3 shows the yaw angular velocity command and the scaled velocity.

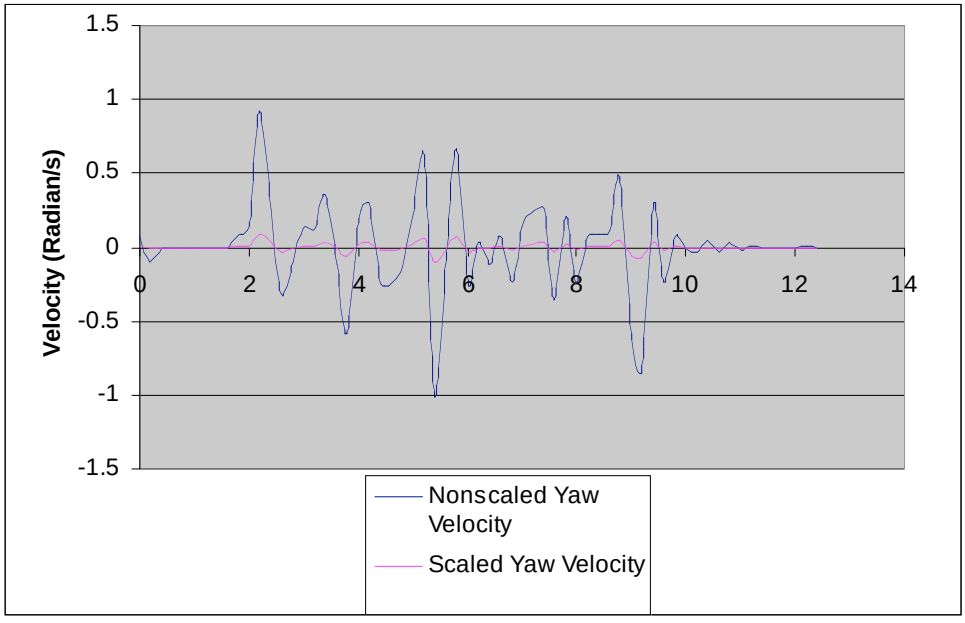


Figure C-3 Scaled and Non-Scaled Yaw Velocity

C.1.2 Planar Assist Function Programming Code

Header File (plane.h)

```
/*****************************************************************************
  Plane.h:  This is the header of plane class.
  Date: August 29,2001
  Revised on: Oct. 25,2001
  Author: Wentao Yu (University of South Florida)
  Contact: wyu@eng.usf.edu

*****************************************************************************/

/***************  Changes made On October 25 2001 by Wentao Yu  *************

  1. RotationAdjust() function has been added a argument. Now it is
     RotationAdjust(int k).
  2. Added a constant toPlaneAccuracy which is used to check if the CP is
     on the constraint line.

*****************************************************************************/

#define toPlaneAccuracy 0.1

class Planar
{
  private:
    double  gp[3];   // This is the goal point for linear movement
        double  mip[3];
     double  pp1[3],pp2[3];
    double  cp[3],cr[3]; // current position and rotation
    double  safeDis;

    double  vmaster[3],wmaster[3];
        double  vscaled[3],wscaled[3];
    double  vmodified[3],wmodified[3];// Modified velocity command

    double  scale[3][3],scaleRPY[3][3];
     double  Rc[3][3],tranRc[3][3];
     double  projection[3];
    double  a,b,c,d;  // four coefficients of the plane equation
    double  RightPosition[3],RightRotation[3];
  public:
    Planar();
    Planar(double GP[],double PP1[],double PP2[],double MIP[],
         double SafeDis);
```

```
    void Computetoolplane();
    void Getprojection();
    void Gettransform();
    void Transpose();
    void Initialization();

    void Designscale(double x, double y, double z);
    void DesignscaleRPY(double x, double y, double z);

    void LinearMove();
    void RotationAdjust(int k);
    void Scaling();

    double* GetRightP();
    double* GetRightR();


    bool IsCPOnplane();
    void SetCP(double Vmaster[],double Wmaster[],double CP[],double CR[]);

    double* GetV();
    double* GetW();

    void Show();
};
```

Source File (Plane.cpp)

```
/*****************************************************************************
  Plane.cpp:  This file includes all the funstions of plane class.
  Date: August 29,2001
  Revised on: Oct. 25,2001
  Author: Wentao Yu (University of South Florida)
  Contact: wyu@eng.usf.edu

*****************************************************************************/


/***************  Changes made On October 25 2001 by Wentao Yu  *************
```

1. In Gettransform() function, Z axis calculation used a new approach.

```
    The old one is :
 a3=mip[0]-projection[0];
 b3=mip[1]-projection[1];
 c3=mip[2]-projection[2];
    The new one is:
```

a3=a;
b3=b;
c3=c;

This means that the Z-axis of the constraint frame is the normal vector of the constraint plane.

2. In Gettransform() function, y axis calculation has been changed . Now it follows right-hand rule.

The old version is:

```
a2=b1*c3-c1*b3;
b2=c1*a3-c3*a1;
c2=a1*b3-b1*a3;
```

The new version is:

```
a2=b3*c1-c3*b1;
b2=c3*a1-c1*a3;
c2=a3*b1-b3*a1;
```

3.RotationAdjust() function has been added a argument which determine which angle needs to be adjusted. Now it is RotationAdjust(int k).

```
***************************************************************************/

/******************** Some important Comments ***********************

 1. Calculation of the orientation of the Roll, Pitch and Yaw angles.

    beta=Atan2(-r31, sqrt(r11^2+r21^2))
    alpha=Atan2(r21/cos(beta), r11/cos(beta));
    gamma=Atan2(r32/cos(beta), r33/cos(beta));

***********************************************************************/

#include <iostream.h>

#include <math.h>
#include "Plane.h"
```

```
// Constructor .Initialize the gp ,pp1,pp2,mip,etc.and do some necessary calculations.
Planar::Planar(double GP[],double PP1[],double PP2[],double MIP[],
        double SafeDis)
{
  int i;

  for(i=0;i<3;i++)
  {
    gp[i]=GP[i];
    pp1[i]=PP1[i];
    pp2[i]=PP2[i];
    mip[i]=MIP[i];
    projection[i]=0;
    vscaled[i]=0;
    wscaled[i]=0;
  }
  safeDis=SafeDis;

  Computetoolplane();
     // Calculating the coefficients of the constraint plane
  Getprojection();
        // get the the projection
  Gettransform();
      // get the transformation matrix
  Transpose();
      // get the transpose of the matrix above
  Initialization();
      // calculate the right position and orientation for the task
}

// get the coefficients of the constraint plane.
void Planar::Computetoolplane()
{
  double  temp[2][3];

  temp[0][0]=pp1[0]-gp[0];
    temp[0][1]=pp1[1]-gp[1];
  temp[0][2]=pp1[2]-gp[2];
  temp[1][0]=pp2[0]-gp[0];
  temp[1][1]=pp2[1]-gp[1];
  temp[1][2]=pp2[2]-gp[2];

  a=temp[0][1]*temp[1][2]-temp[1][1]*temp[0][2];  // a ,b ,c d are coffecients of
  b=temp[1][0]*temp[0][2]-temp[0][0]*temp[1][2];  // the plane decided by the three
  c=temp[0][0]*temp[1][1]-temp[1][0]*temp[0][1];  // points.
  d=(-1)*gp[0]*a-gp[1]*b-gp[2]*c;
```

```
}

// get the projection of the mip on the constraint plane.
void Planar::Getprojection()
{
  double k;

  k=(-1)*(a*mip[0]+b*mip[1]+c*mip[2]+d)/(a*a+b*b+c*c);

  projection[0]=k*a+mip[0];
  projection[1]=k*b+mip[1];
  projection[2]=k*c+mip[2];

}

// get the thansformation matrix between constraint frame and base frame.
void Planar::Gettransform()
{
  // projection is the original point of the frame
  double  a1,b1,c1,a2,b2,c2,a3,b3,c3;
  double  sum1,sum2,sum3;
  double  a11,b11,c11,a21,b21,c21,a31,b31,c31;

   //The direction of X-axis is :a1,b1,c1 ;
  // For planar assistance function,it is a line perpendicular to the pipe axis.
  // For linear assistance function,it is a line perpendicular to the constraint line.
  a1=gp[0]-projection[0];
  b1=gp[1]-projection[1];
  c1=gp[2]-projection[2];



  // The direction of Z-axis is a3,b3,c3;
    // For planar assistance function,it is a the normal vector of the constraint plane.

  a3=a;
  b3=b;
  c3=c;

  // The direction of Y-axis is a2,b2,c2;
  // Y-axis is the cross product of Z and X. That is , Y=Z*X follows the right-hand rule.

  a2=b3*c1-c3*b1;
  b2=c3*a1-c1*a3;
  c2=a3*b1-b3*a1;

  sum1=sqrt(a1*a1+b1*b1+c1*c1);
```

```
 sum2=sqrt(a2*a2+b2*b2+c2*c2);
  sum3=sqrt(a3*a3+b3*b3+c3*c3);

 a11=a1/sum1;
 b11=b1/sum1;
  c11=c1/sum1;

 a21=a2/sum2;
  b21=b2/sum2;
 c21=c2/sum2;

 a31=a3/sum3;
 b31=b3/sum3;
 c31=c3/sum3;

 Rc[0][0]=a11;
 Rc[0][1]=b11;
 Rc[0][2]=c11;

 Rc[1][0]=a21;
 Rc[1][1]=b21;
 Rc[1][2]=c21;

 Rc[2][0]=a31;
 Rc[2][1]=b31;
 Rc[2][2]=c31;  // This array is the transform matrix
}

// transpose Rc
void Planar::Transpose()
{
 int i,j;

 for(i=0;i<3;i++)
 {
  for(j=0;j<3;j++)
    tranRc[i][j]=Rc[j][i];
 }
}

// Because we have the transformation between constraint frame and base frame. Right before
// doing cutting task, the Roll axis is in the X-axis direction of the constraint frame.Pitch
// axis is in the Z-axis and Yaw axis is in the Y-axis.
void Planar::Initialization()
{
 int i;
```

```
  RightRotation[1]=atan2((-1)*Rc[2][0],sqrt(Rc[0][0]*Rc[0][0]+Rc[1][0]*Rc[1][0]));
    RightRotation[0]=atan2(Rc[1][0]/cos(RightRotation[1]),Rc[0][0]/cos(RightRotation[1]));
  RightRotation[2]=atan2(Rc[2][1]/cos(RightRotation[1]),Rc[2][2]/cos(RightRotation[1]));

  for(i=0;i<3;i++)
    RightPosition[i]=projection[i];
}


double* Planar::GetRightP()
{
  return RightPosition;
}

double* Planar::GetRightR()
{
  return RightRotation;
}

void Planar::Designscale(double x, double y, double z)
{
    scale[0][0]=x;
  scale[0][1]=0;
  scale[0][2]=0;

  scale[1][0]=0;
  scale[1][1]=y;
  scale[1][2]=0;

  scale[2][0]=0;
  scale[2][1]=0;
  scale[2][2]=z;
}

void Planar::DesignscaleRPY(double x,double y, double z)
{
  scaleRPY[0][0]=x;
  scaleRPY[0][1]=0;
  scaleRPY[0][2]=0;

  scaleRPY[1][0]=0;
  scaleRPY[1][1]=y; // Pitch can be free, no scaled.
  scaleRPY[1][2]=0;

  scaleRPY[2][0]=0;
  scaleRPY[2][1]=0;
  scaleRPY[2][2]=z;
```

```
}

void Planar::Scaling()
{
 int i,j;
 double vcut[3],wcut[3];

 for(i=0;i<3;i++)      //Before giving vmodified value, initialize it
 {
  vcut[i]=0;
  wcut[i]=0;
  vscaled[i]=0;
  wscaled[i]=0;
  vmodified[i]=0;
  wmodified[i]=0;

 }

 Designscale(1,1,0.1);
 DesignscaleRPY(0.1,1,0.1);

 for(i=0;i<3;i++)           // making the end-effector back
 {                          // the constraint plane
  for(j=0;j<3;j++)
  {
   vcut[i]+=tranRc[i][j]* vmaster[j];
   wcut[i]+=tranRc[i][j]* wmaster[j];
  }
 }

 for(i=0;i<3;i++)
 {
  for(j=0;j<3;j++)
  {
   vscaled[i] +=scale[i][j]* vcut[j];
   wscaled[i] +=scaleRPY[i][j]* wcut[j];
  }
 }

 for(i=0;i<3;i++)
 {
  for(j=0;j<3;j++)
  {
   vmodified[i] +=Rc[i][j]* vscaled[j];
   wmodified[i] +=Rc[i][j]* wscaled[j];
  }
 }
```

```
}

bool Planar::IsCPOnplane()
{
  double disCPtoPlane;
    disCPtoPlane=fabs(a*cp[0]+b*cp[1]+c*cp[2]+d)/sqrt(a*a+b*b+c*c);
  if(disCPtoPlane< toPlaneAccuracy)
    return true;
  else
    return false;
}

void Planar::RotationAdjust(int k)
{
  double wslave[3];
  double RotateScale[3][3];
  int i,j;

  if(k==0)
  {
    RotateScale[0][0]=1;
    RotateScale[1][1]=0.1;
    RotateScale[2][2]=0.1;
  }
  else if(k==1)
  {
    RotateScale[1][1]=1;
    RotateScale[0][0]=0.1;
    RotateScale[2][2]=0.1;
  }
  else
  {
    RotateScale[2][2]=1;
    RotateScale[1][1]=0.1;
    RotateScale[0][0]=0.1;
  }

  RotateScale[0][1]=0;
  RotateScale[0][2]=0;

  RotateScale[1][0]=0;
  RotateScale[1][2]=0;

  RotateScale[2][0]=0;
  RotateScale[2][1]=0;

  for(i=0;i<3;i++)      //Before giving vmodified value, initialize it
```

```
 {
  wslave[i]=0;
  wscaled[i]=0;
  vmodified[i]=0;
 }

 for(i=0;i<3;i++)          // making the end-effector back
                           // the constraint plane
  for(j=0;j<3;j++)
    wslave[i]+=tranRc[i][j]* wmaster[j];

 for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    wscaled[i] +=RotateScale[i][j]* wslave[j];

 for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    wmodified[i] +=Rc[i][j]* wscaled[j];

}

void Planar::SetCP(double Vmaster[],double Wmaster[],double CP[],double CR[])
{
 int i;
 for(i=0;i<3;i++)
 {
  vmaster[i]=Vmaster[i];
  wmaster[i]=Wmaster[i];
  cp[i]=CP[i];
  cr[i]=CR[i];
 }
}

void Planar::LinearMove()
{
 double LineMovescale[3][3];
 double dis,ScaleFactor;
 double vLineMove[3];
 int i,j;

 dis=sqrt((projection[0]-cp[0])*(projection[0]-cp[0])+(projection[1]-cp[1])*(projection[1]-cp[1])
       +(projection[2]-cp[2])*(projection[2]-cp[2]));
 if(dis>safeDis)
  ScaleFactor=2;
 else
  ScaleFactor=0.5;
```

```
 LineMovescale[0][0]=0.1;
 LineMovescale[0][1]=0;
 LineMovescale[0][2]=0;

 LineMovescale[1][0]=0;
 LineMovescale[1][1]=0.1;
 LineMovescale[1][2]=0;

 LineMovescale[2][0]=0;
 LineMovescale[2][1]=0;
 LineMovescale[2][2]=ScaleFactor;

 for(i=0;i<3;i++)      //Before giving vmodified value, initialize it
 {
  vLineMove[i]=0;
  vscaled[i]=0;
 }

 for(i=0;i<3;i++)           // making the end-effector back
                        // the constraint plane
  for(j=0;j<3;j++)
    vLineMove[i]+=tranRc[i][j]* vmaster[j];

 for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    vscaled[i] +=LineMovescale[i][j]* vLineMove[j];

 for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    vmodified[i] +=Rc[i][j]* vscaled[j];

}

double* Planar::GetV()
{
 return vmodified;
}

double* Planar::GetW()
{
 return wmodified;
}

void Planar::Show()
{
 int i;
```

```
  for(i=0;i<3;i++)
    cout<<vmodified[i]<<"\n";

  for(i=0;i<3;i++)
    cout<<wmodified[i]<<"\n";

}
```

C.2 Simulation Results of Linear Assistance function

C.2.1 Linear Assistance function Simulation Results

A test of the code was performed to measure the performance of the linear assistance function. The program was executed at the USF test bed, and the scaled and non-scaled velocities were recorded. The following graphs represent the results obtained from the linear assistance function showing the actual velocity command sent to the robot controller.

Figure C-4 represents the linear velocity scaling. The graph contains the X-axis velocity of the constraint plane. Motion in the X-axis causes the manipulator to go away from the constraint line, according to figure 4. The graph shows that the motion away from the constraint line is scaled down by the scale value of 0.1.

Furthermore, Figure C-5 shows the Y-axis velocity, scaled and non-scaled. Since motion in the Y-axis direction is away from the constraint line, the motion is scaled by a factor of 0.1.

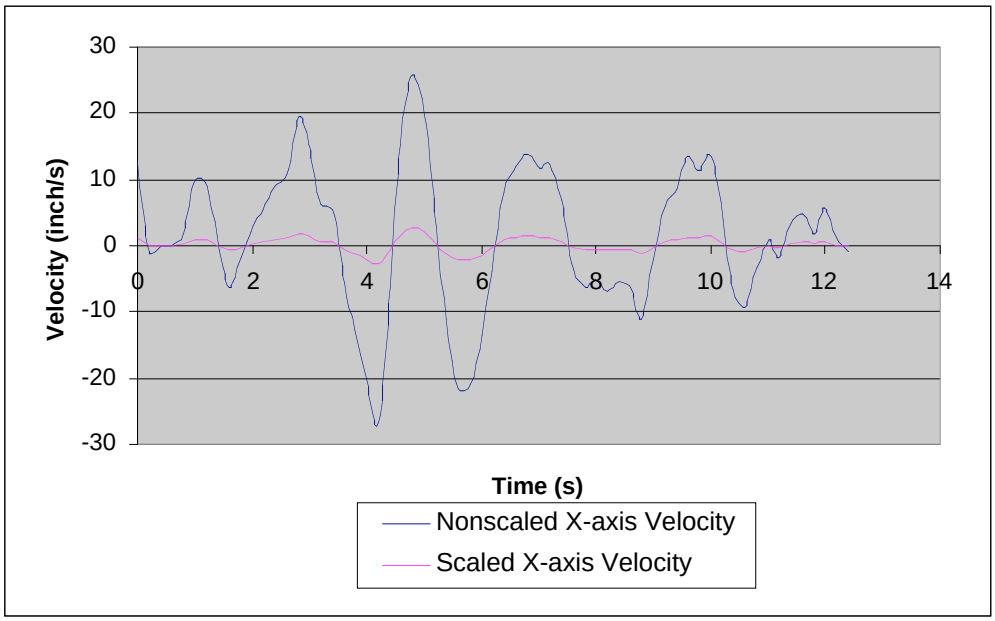


Figure C-4 Scaled and Non-Scaled X-Axis Velocity

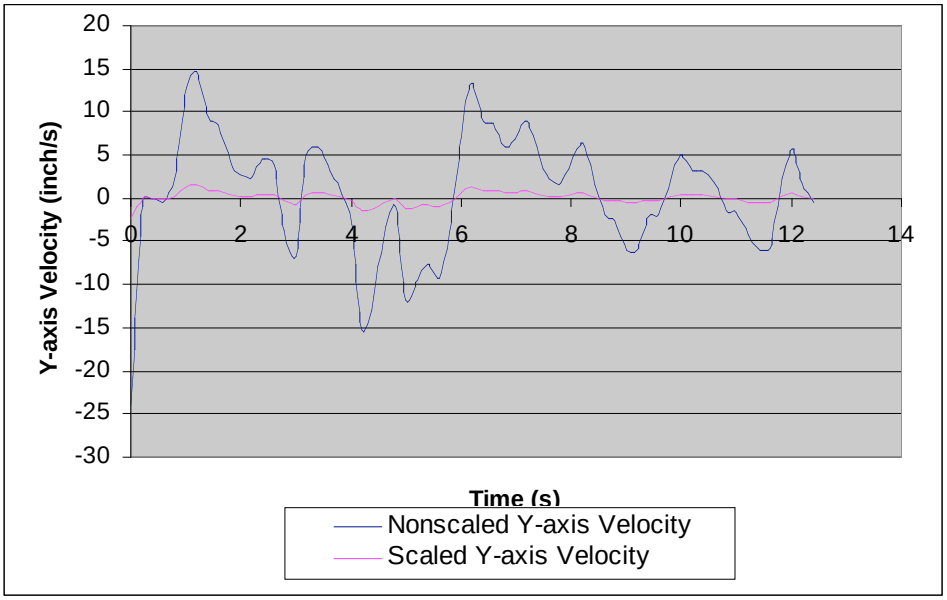


Figure C-5 Scaled and Non-Scaled Y-Axis Velocity

The linear assist function also requires the roll axis to be pointing in the direction of the constrain line. Therefore, during the drilling operation the roll axis should be kept along the line, so the pitch and yaw should not deviate from the initial points, so it is necessary to scale those angular velocities. From the graph in Figure C-6, the pitch velocity command and the scaled pitch velocity is shown, and the angular velocity in the pitch direction is scaled down by a value of 0.1.

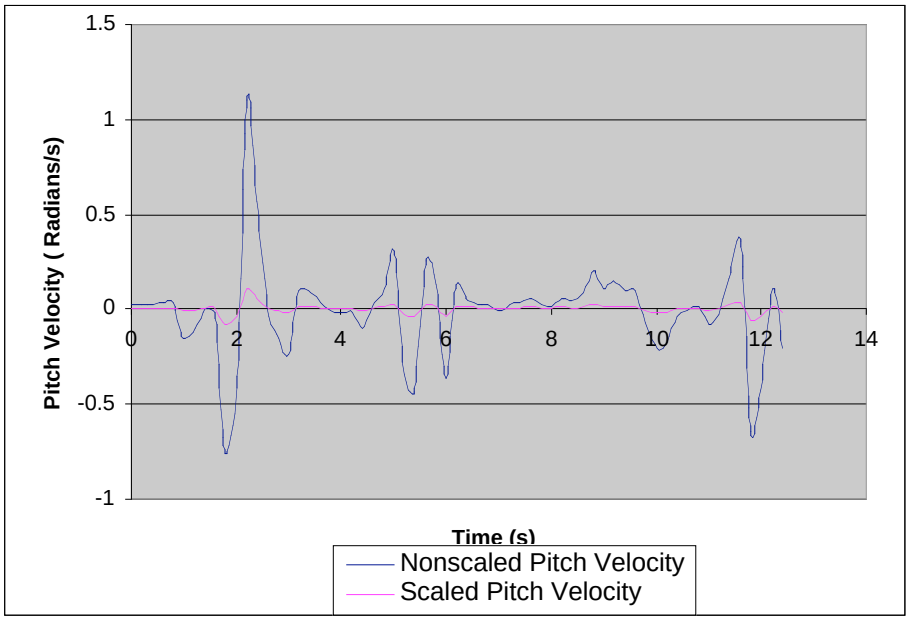


Figure C-6 Scaled and Non-Scaled Pitch Velocity

Similarly, the yaw direction should be kept the same from the initial direction. This means that the yaw velocity is scaled down by a value of 0.1. Figure C-7 shows the yaw angular velocity command and the scaled velocity.

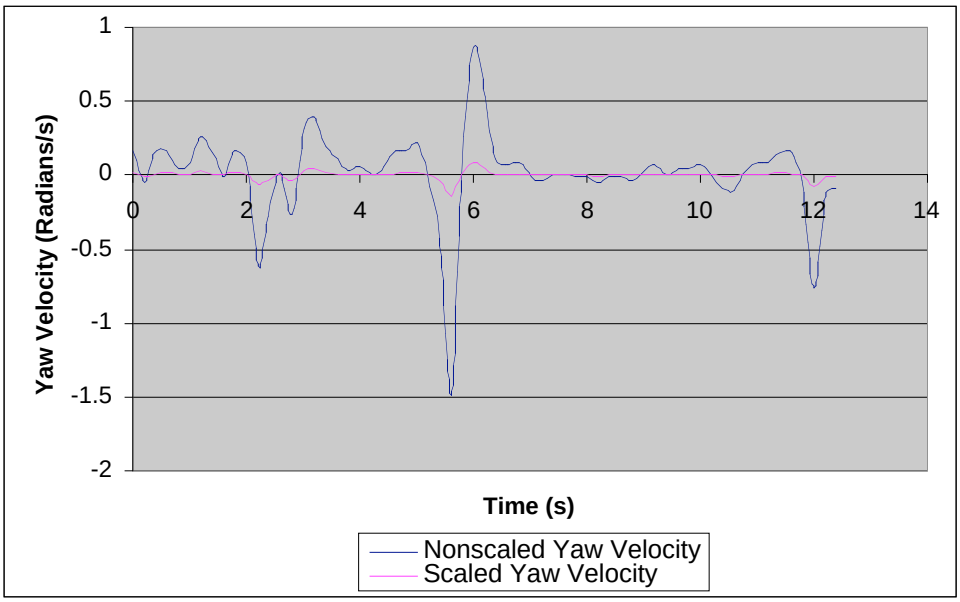


Figure C-7 Scaled and Non-Scaled Yaw Velocity

The simulation results show that the algorithm takes a commanded velocity and scales this velocity command per the requirements of the assistance function, helping the teleoperator perform tedious and tiresome tasks.

C.2.2 Linear Assist Function Programming Code

Header File (Line.h)

```
/*****************************************************************************
  Line.h:  This is the header of line class.
  Date: August 29,2001
  Revised on: Oct. 25,2001
  Author: Wentao Yu (University of South Florida)
  Contact: wyu@eng.usf.edu

*****************************************************************************/

/***************  Changes made On October 25 2001 by Wentao Yu  ************

  1. RotationCheck() function has been added a argument. Now it is
     RotationAdjust(int k).
  2. Added a constant toLineAccuracy which is used to check if the CP is
     on the constraint line.

*****************************************************************************/

#define toLineAccuracy 0.1 // the distance to judge if the CP is on the line.

class Linear
{
  private:
       double  safeDis;
     double  lp1[3],lp2[3],mip[3];
         // lp1 is the goal point for linear movement
    double  cp[3],cr[3];

    double vmaster[3],wmaster[3];// Velocity command
     double vscaled[3],wscaled[3]; //Velocity after scaled
    double vmodified[3],wmodified[3];// Modified velocity command

    double scale[3][3],scaleRPY[3][3];
     double Rline[3][3],tranRline[3][3];
    double projection[3]; // MIP on the constraint line
    double RightPosition[3],RightRotation[3];
  public:
    Linear();
    Linear(double LP1[],double LP2[],double mip[],
         double SafeDis);
    void Getprojection();
    void Gettransform();
```

```
    void Transpose();
    bool IsCPOnLine(); // check if CP is on the constraint line
    void SetCP(double Vmaster[],double Wmaster[],double CP[],double CR[]);
    void Initialization(); // calculate the right position and orientation
    double* GetRightP();
    double* GetRightR();

    void Designscale(double x,double y,double z);
    void DesignscaleRPY(double x,double y,double z);

    void LinearMove(); // Move the end effector from MIP to the projection
    void RotationCheck(int k);
    void Scaling(); // do scaling operation

    double* GetV();
    double* GetW();

    void show();
};
```

Source File (Line.cpp)

```
/*****************************************************************************
  Line.cpp:  This file includes all the functions of line class.
  Date: August 29,2001
  Revised on: Oct. 25,2001
  Author: Wentao Yu (University of South Florida)
  Contact: wyu@eng.usf.edu

*****************************************************************************/


/***************  Changes made On October 25 2001 by Wentao Yu  ************

  1.RotationCheck() function has been added a argument which determine which angle
    needs to be adjusted. Now it is RotationAdjust(int k).

  2. In Gettransform() function, the case of MIP is on the constraint line has
     been considered. If MIP is the same as projection, use a arbitrary line which
     is perpendicular with the constraint line as X-axis of the constraint frame.
  3. In LinearMove() function, the scaleFactor value has been changed from constant
     to a v value.

*****************************************************************************/

#include <iostream.h>
#include <math.h>
```

```
#include "Line.h"


Linear::Linear(double LP1[],double LP2[],double MIP[],
        double SafeDis)
{
  int i;
  for(i=0;i<3;i++)
  {
    lp1[i]=LP1[i];
    lp2[i]=LP2[i];
    mip[i]=MIP[i];
    projection[i]=0;
    vscaled[i]=0;
    wscaled[i]=0;
  }
  safeDis=SafeDis;
  Gettransform();
  Transpose();
  Initialization();
}

// get the projection of a point on the constraint line
void Linear::Getprojection()
{

  // This function is getting the projection of the MIP on a line

  double  k1,k2,k;

  k1=(mip[0]-lp1[0])*(lp2[0]-lp1[0])+(mip[1]-lp1[1])*(lp2[1]-lp1[1])+
    (mip[2]-lp1[2])*(lp2[2]-lp1[2]);
    k2=(lp2[0]-lp1[0])*(lp2[0]-lp1[0])+(lp2[1]-lp1[1])*(lp2[1]-lp1[1])+
    (lp2[2]-lp1[2])*(lp2[2]-lp1[2]);
  k=k1/k2;

  projection[0]=k*(lp2[0]-lp1[0])+lp1[0];
  projection[1]=k*(lp2[1]-lp1[1])+lp1[1];
  projection[2]=k*(lp2[2]-lp1[2])+lp1[2];


}

// get the transformation matrix
void Linear::Gettransform()
{
  // projection is the original point of the frame
```

```
double  a1,b1,c1,a2,b2,c2,a3,b3,c3;
double  sum1,sum2,sum3;
double  a11,b11,c11,a21,b21,c21,a31,b31,c31;

Getprojection();

// The direction of Z-axis is a3,b3,c3;
  // For planar assistance function,it is a line parallel with the pipe axis.
// For linear assistance function,it is a line parallel with the constraint line.

a3=lp1[0]-lp2[0];
b3=lp1[1]-lp2[1];
c3=lp1[2]-lp2[2];


//The direction of X-axis is :a1,b1,c1 ;
if(mip[0] == projection[0] && mip[1] == projection[1] && mip[2] == projection[2])
{
  a1=1;
  b1=1;
  c1=(-1)*(a3+c3)/c3;
}
else
{
  a1=mip[0]-projection[0];
  b1=mip[1]-projection[1];
  c1=mip[2]-projection[2];
}

// The direction of Y-axis is a2,b2,c2;
// Y-axis is the cross-product of Z and X.It follows right-hand rule.

a2=b3*c1-c3*b1;
b2=c3*a1-c1*a3;
c2=a3*b1-b3*a1;

sum1=sqrt(a1*a1+b1*b1+c1*c1);
sum2=sqrt(a2*a2+b2*b2+c2*c2);
  sum3=sqrt(a3*a3+b3*b3+c3*c3);

a11=a1/sum1;
b11=b1/sum1;
  c11=c1/sum1;

a21=a2/sum2;
  b21=b2/sum2;
c21=c2/sum2;
```

```
  a31=a3/sum3;
  b31=b3/sum3;
  c31=c3/sum3;

  Rline[0][0]=a11;
  Rline[0][1]=b11;
  Rline[0][2]=c11;

  Rline[1][0]=a21;
  Rline[1][1]=b21;
  Rline[1][2]=c21;

  Rline[2][0]=a31;
  Rline[2][1]=b31;
  Rline[2][2]=c31;  // This array is the transform matrix
}

// get the transpose of the matrix above
void Linear::Transpose()
{
  int i,j;

  for(i=0;i<3;i++)
  {
    for(j=0;j<3;j++)
      tranRline[i][j]=Rline[j][i];
  }
}

void Linear::Designscale(double x,double y,double z)
{
  scale[0][0]=x;
  scale[0][1]=0;
  scale[0][2]=0;

  scale[1][0]=0;
  scale[1][1]=y;
  scale[1][2]=0;

  scale[2][0]=0;
  scale[2][1]=0; //This is the linear scale matrix.We should scale the velocity
  scale[2][2]=z; // other than the the direction of line movement.
}

void Linear::DesignscaleRPY(double roll,double pitch,double yaw)
{
```

```
  scaleRPY[0][0]=roll;    //This is the orientation scaleRPY matrix.
  scaleRPY[0][1]=0;
  scaleRPY[0][2]=0;

  scaleRPY[1][0]=0;
  scaleRPY[1][1]=pitch;
  scaleRPY[1][2]=0;

  scaleRPY[2][0]=0;
  scaleRPY[2][1]=0;
  scaleRPY[2][2]=yaw;
}

bool Linear::IsCPOnLine()
{
  int i,j;
  double Newcp[3];
  double disCPtoLine;

  for(i=0;i<3;i++)
    for(j=0;j<3;j++)
      Newcp[i]+=tranRline[i][j]* cp[j];

    disCPtoLine=sqrt(Newcp[0]*Newcp[0]+Newcp[1]*Newcp[1]);
    // distance from CP to the constraint line
  if(disCPtoLine< toLineAccuracy)
    return true;
  else
    return false;
}

// Initializationization for a task doing to get right position
// and rotation for a task.
void Linear::Initialization()
{
  int i;

  RightRotation[1]=atan2((-1)*Rline[2][0],sqrt(Rline[0][0]*Rline[0][0]+Rline[1][0]*Rline[1][0]));
    RightRotation[0]=atan2(Rline[1][0]/cos(RightRotation[1]),Rline[0][0]/cos(RightRotation[1]));
  RightRotation[2]=atan2(Rline[2][1]/cos(RightRotation[1]),Rline[2][2]/cos(RightRotation[1]));

  for(i=0;i<3;i++)
    RightPosition[i]=projection[i];
}

void Linear::SetCP(double Vmaster[],double Wmaster[],double CP[],double CR[])
{
```

```
 int i;
 for(i=0;i<3;i++)
 {
  vmaster[i]=Vmaster[i];
  wmaster[i]=Wmaster[i];
  cp[i]=CP[i];
  cr[i]=CR[i];
 }
}

//To adjust the Roll, Pitch and Yaw angles
void Linear::RotationCheck(int k)
{
 double wslave[3],wscaled[3];
 double RotateScale[3][3];
 int i,j;

 if(k==0)
 {
  RotateScale[0][0]=1;
  RotateScale[1][1]=0.1;
  RotateScale[2][2]=0.1;
 }
 else if(k==1)
 {
  RotateScale[1][1]=1;
  RotateScale[0][0]=0.1;
  RotateScale[2][2]=0.1;
 }
 else
 {
  RotateScale[2][2]=1;
  RotateScale[1][1]=0.1;
  RotateScale[0][0]=0.1;
 }

 RotateScale[0][1]=0;
 RotateScale[0][2]=0;

 RotateScale[1][0]=0;
 RotateScale[1][2]=0;

 RotateScale[2][0]=0;
 RotateScale[2][1]=0;

 for(i=0;i<3;i++)      //Before giving vmodified value, initialize it
 {
```

```
  wslave[i]=0;
  wscaled[i]=0;
  wmodified[i]=0;
 }

 for(i=0;i<3;i++)           // making the end-effector back
                          // the constraint plane
  for(j=0;j<3;j++)
    wslave[i]+=tranRline[i][j]* wmaster[j];

 for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    wscaled[i] +=RotateScale[i][j]* wslave[j];

 for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    wmodified[i] +=Rline[i][j]* wscaled[j];
}

double* Linear::GetRightP()
{
 return RightPosition;
}

double* Linear::GetRightR()
{
 return RightRotation;
}

// Move the end effector along the line from MIP to projection .
void Linear::LinearMove()
{
 double LineMovescale[3][3];
 double dis,ScaleFactor;
 double vLineMove[3],vscaled[3];
 int i,j;


 dis=sqrt((projection[0]-cp[0])*(projection[0]-cp[0])+(projection[1]-cp[1])*(projection[1]-cp[1])
       +(projection[2]-cp[2])*(projection[2]-cp[2]));
 if(dis>=safeDis)
  ScaleFactor=2-safeDis/dis; // ScaleFactor changes from 2 to 1 when the distance
                          // changes from very large to the safeDis.
 else
  ScaleFactor=1-dis/safeDis;  // ScaleFactor changes from 1 to 0 when the distance
                          // changes from safeDis to 0.
```

```
 LineMovescale[0][0]=ScaleFactor;
 LineMovescale[0][1]=0;
 LineMovescale[0][2]=0;

 LineMovescale[1][0]=0;
 LineMovescale[1][1]=ScaleFactor;
 LineMovescale[1][2]=0;

 LineMovescale[2][0]=0;
 LineMovescale[2][1]=0;
 LineMovescale[2][2]=0.1;

 for(i=0;i<3;i++)      //Before giving vmodified value, Initialize it
 {
  vLineMove[i]=0;
  vscaled[i]=0;
  vmodified[i]=0;
 }

 for(i=0;i<3;i++) // making the end-effector back the constraint plane
  for(j=0;j<3;j++)
    vLineMove[i]+=tranRline[i][j]* vmaster[j];

 for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    vscaled[i] +=LineMovescale[i][j]* vLineMove[j];

 for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    vmodified[i] +=Rline[i][j]* vscaled[j];

}

void Linear::Scaling()
{
 int i,j;
 double vline[3],wline[3];

 Designscale(0.1,0.1,1);
 DesignscaleRPY(1,0.1,0.1);
    // design a scale matrix for linear and angular velocity scaling

 for(i=0;i<3;i++)      //Before giving vmodified value, Initializationize it
 {
  vline[i]=0;
  wline[i]=0;
  vscaled[i]=0;
```

```
  wscaled[i]=0;
  vmodified[i]=0;
  wmodified[i]=0;
 }

 for(i=0;i<3;i++)           // making the end-effector back
 {                          // the constraint plane
  for(j=0;j<3;j++)
  {
   vline[i]+=tranRline[i][j]* vmaster[j];
   wline[i]+=tranRline[i][j]* wmaster[j];
  }
 }


 for(i=0;i<3;i++)
 {
  for(j=0;j<3;j++)
  {
   vscaled[i] +=scale[i][j]* vline[j];
   wscaled[i] +=scaleRPY[i][j]* wline[j];
  }
 }

 for(i=0;i<3;i++)
 {
  for(j=0;j<3;j++)
  {
   vmodified[i] +=Rline[i][j]* vscaled[j];
   wmodified[i] +=Rline[i][j]* wscaled[j];
  }
 }
}

double* Linear::GetV()
{
 return vmodified;
}

double* Linear::GetW()
{
 return wmodified;
}

void Linear::show()
{
 int i;
```

```
 for(i=0;i<3;i++)
   cout<<vmodified[i]<<"\n";

 for(i=0;i<3;i++)
   cout<<wmodified[i]<<"\n";
}
```

C.2.1 Velocity Assistance Function Programming Code

Header File (Velo.h)

```
/*****************************************************************************
 Velo.h:  This is the header of velo class.
 Date: August 29,2001
 Revised on: Oct. 25,2001
 Author: Wentao Yu (University of South Florida)
 Contact: wyu@eng.usf.edu

*****************************************************************************/

/***************  Changes made On October 25 2001 by Wentao Yu  ************



*****************************************************************************/

class Velo
{
 private:
      double  gp[3],cp[3];
   double safeDis;
   double vmaster[3];
   double vscaled[3];
 public:
   Velo();
   Velo(double GP[],double SafeDis);
   void SetCP(double Vmaster[],double CP[]);
   double* Scaling();
};
```

Source File (Velo.cpp)

```
/*****************************************************************************
  Velo.cpp:  This file includes all functions of Velo class.
  Date: August 29,2001
  Revised on: Oct. 25,2001
  Author: Wentao Yu (University of South Florida)
  Contact: wyu@eng.usf.edu

*****************************************************************************/


/***************  Changes made On October 25 2001 by Wentao Yu  ************

  1. The scaling factor calculation has been changed to:
    if(dis>=safeDis)
      ScaleFactor=2-safeDis/dis;
    else
    ScaleFactor=1-dis/safeDis;

    ScaleFactor changes from 2 to 1 when the distance changes from very
    large to the safeDis. ScaleFactor changes from 1 to 0 when the distance
       changes from safeDis to 0.

*****************************************************************************/

#include <math.h>
#include "velo.h"


Velo::Velo(double GP[],double SafeDis)
{
  int i;

  for(i=0;i<3;i++)
  {
    gp[i]=GP[i];
    vscaled[i]=0;
  }
  safeDis=SafeDis;
}

void Velo::SetCP(double Vmaster[],double CP[])
{
  int i;
  for(i=0;i<3;i++)
  {
    vmaster[i]=Vmaster[i];
```

```
   cp[i]=CP[i];
  }
}


double* Velo::Scaling()
{
  int i;
  double dis;

  double ScaleFactor;

  dis=sqrt((gp[0]-cp[0])*(gp[0]-cp[0])+(gp[1]-cp[1])*(gp[1]-cp[1])
        +(gp[2]-cp[2])*(gp[2]-cp[2]));
  if(dis>=safeDis)
    ScaleFactor=2-safeDis/dis; // ScaleFactor changes from 2 to 1 when the distance
                               // changes from very large to the safeDis.
  else
    ScaleFactor=1-dis/safeDis;  // ScaleFactor changes from 1 to 0 when the distance
                                // changes from safeDis to 0.
  for(i=0;i<3;i++)
    vscaled[i]=vmaster[i]*ScaleFactor;
  return vscaled;
}
```

C.3 Simulation Results of Force Assistance function

C.3.1 Force Assist Function Simulation Results

Figure C-8 shows the relationship between the sensor force and the reference force, and the scale value that result.

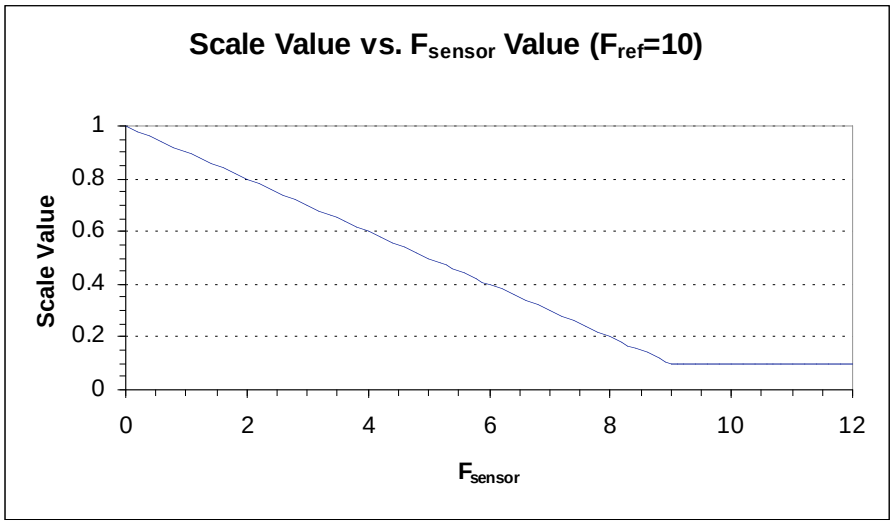


Figure C-8 Scale Values for Corresponding Increasing Force Sensor Data

The scale value uses the reference force and the sensor force, both in the constraint frame, to determine the scale value. The relationship is obviously linear until the scale value reaches 0.1. The scale never descends below 0.1 because these values are not desired since it might cause the manipulator to get stuck in that position without being able to move away from the stranded position.

Furthermore, several different sensor values and base frame commanded velocities were used to calculate the scale value. The following six graphs show, for all six velocity components, the scale matrices that were calculated, and how these scale matrices affected the commanded velocity. These figures illustrate the effect the Force Assist Function has on the commanded velocity for a given force sensor value.

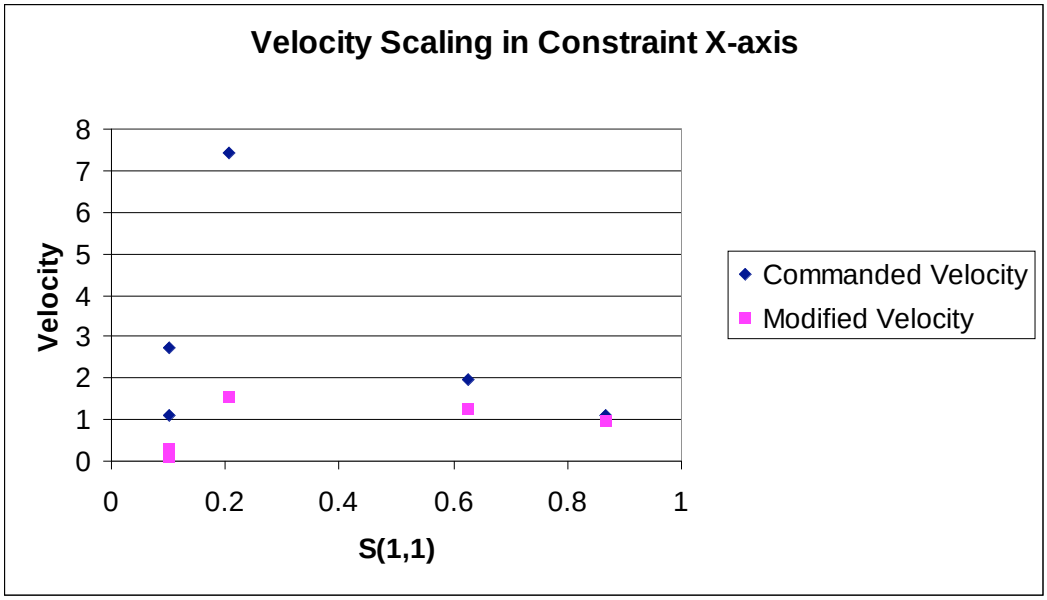


Figure C-9 Comparison of Velocity Scaling for Simulated Force Data in X-Axis

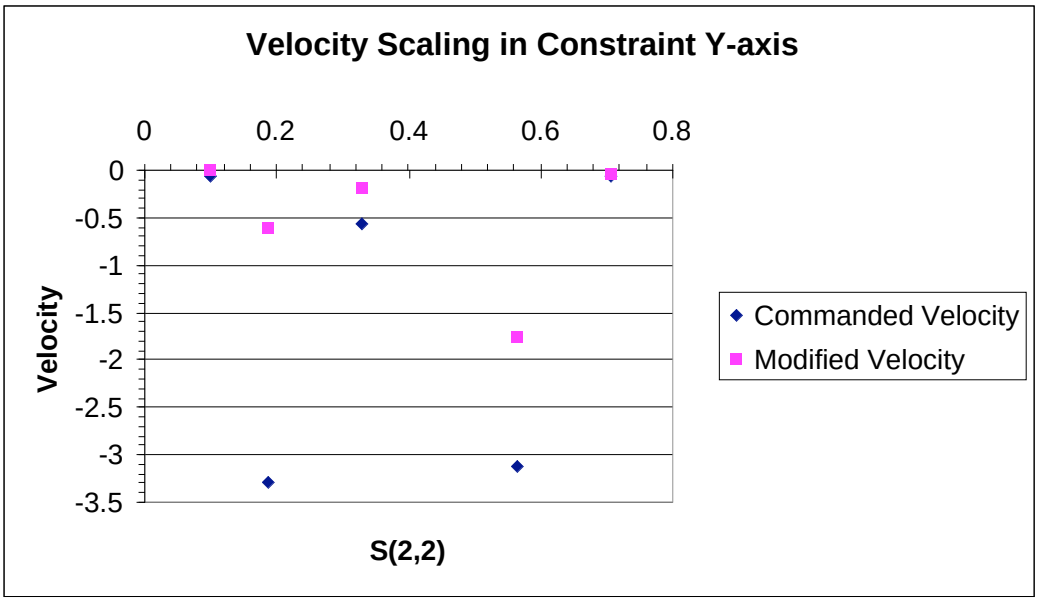


Figure C-10 Comparison of Velocity Scaling for Simulated Force Daa in Y-Axis

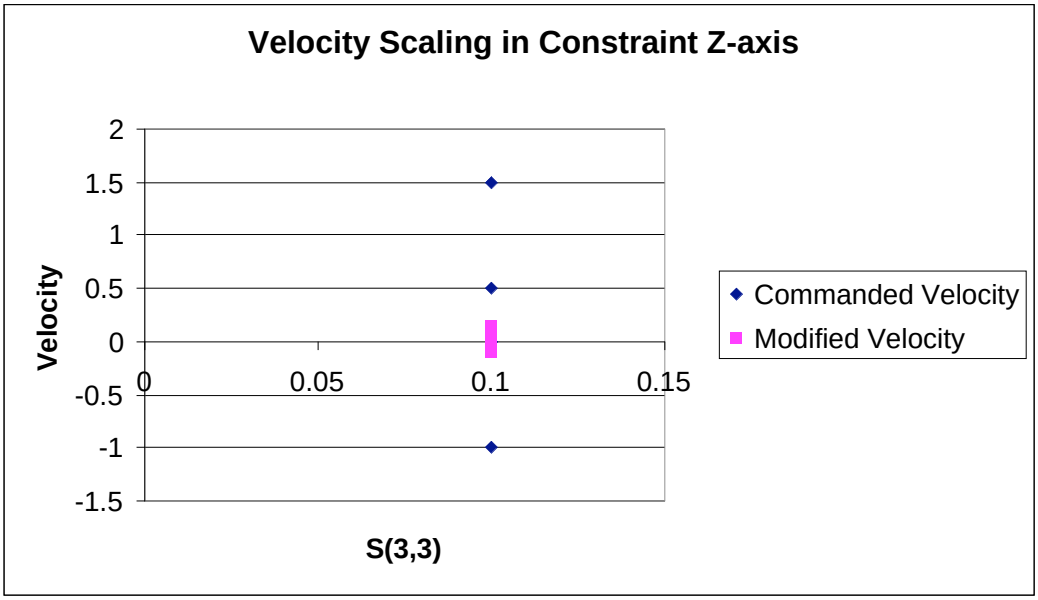


Figure C-11 Comparison of Velocity Scaling for Simulated Force Data in Z-Axis

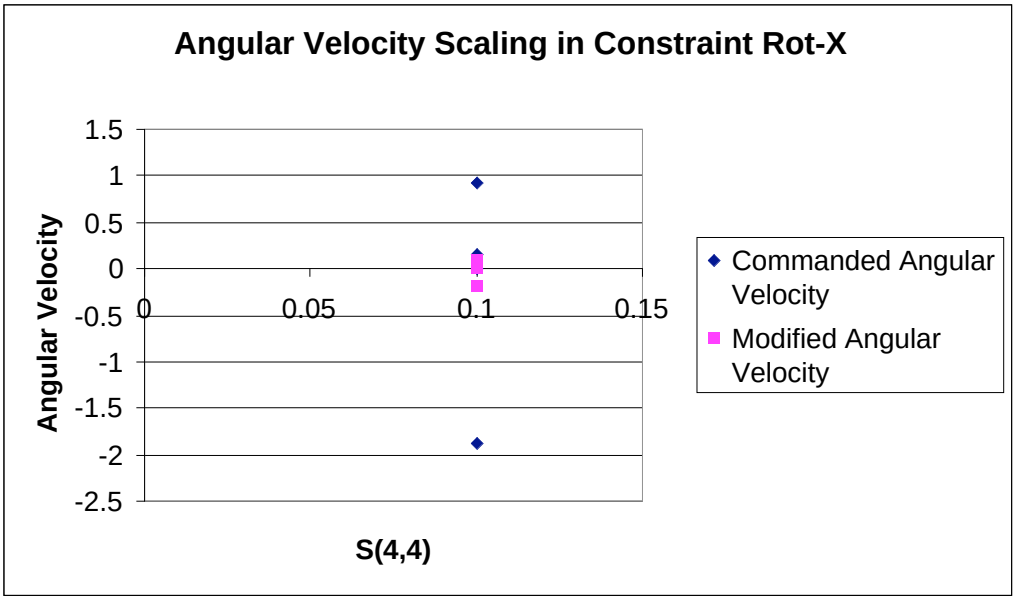


Figure C-12 Comparison of Angular Velocity Scaling for Simulated Force Data in Rot-X

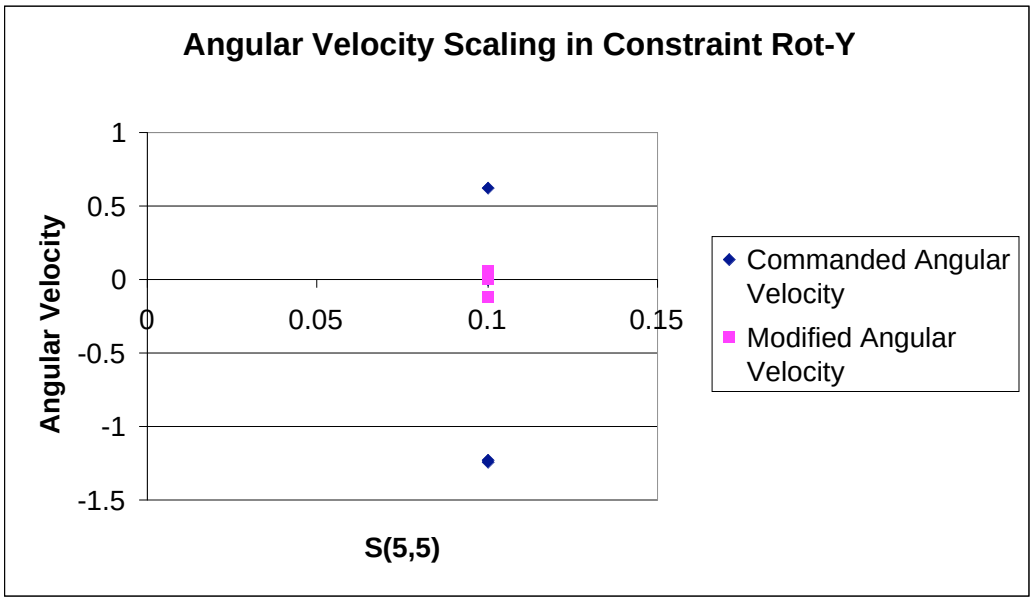


Figure C-13 Comparison of Angular Velocity Scaling for Simulated Force Data in Rot-Y

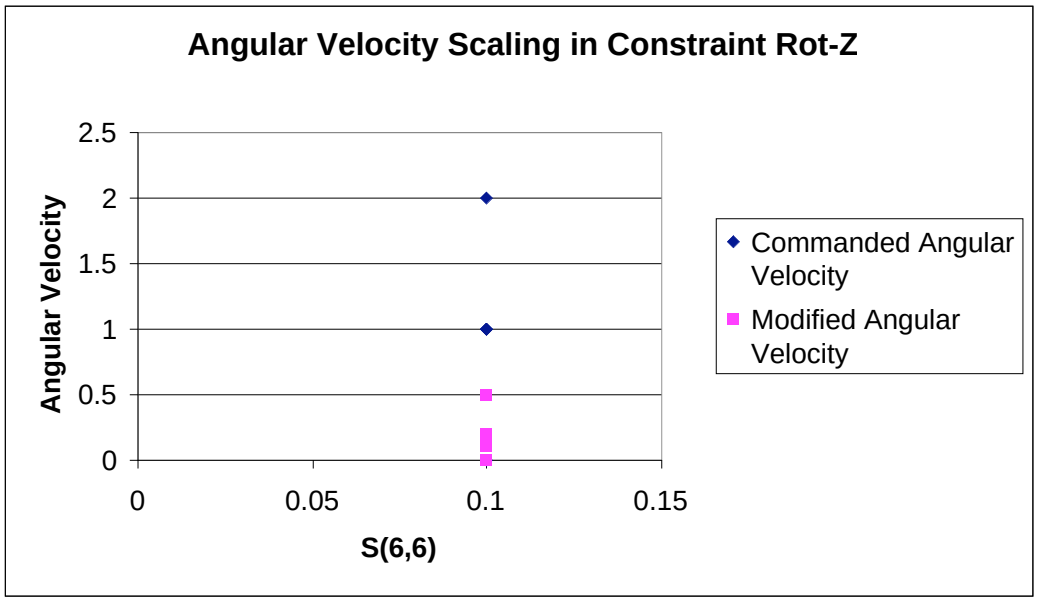


Figure C-14 Comparison of Angular Velocity Scaling for Simulated Force Data in Rot-Z

C.3.2 Force Assist Function Programming Code (C++)

```
#include <iostream.h>
#include "force.h"

int main()
{
  int i;
  static double  rbc[3][3]={{0.866,-0.5,0},{0.5,0.866,0},{0,0,1}};
      // The transformation of constraint frame with respect  to base frmae (world frame).
  static double  rbs[3][3]={{0.933,0.067,0.354},{0.067,0.933,-0.354},{-0.354,0.354,0.866}};
              // The transformation of sensor frame with respect
             //           to base frmae (world frame).
  static double  frc[6]={5,10,0,0,0,0};
              //  Force and moment of reference value in constraint frame.
    static double  fss[6]={1,6,-0.25,3,-0.25,-0.25};
              //  Fss: Sensored force and moment in sensor frame.
    static double  csorg[3]={1,0,0.2};
              // It is a vector which locates the origin of sensor frame
             // with respect to constraint frame.

  static double  velMaster[6]={ 8,1,-1,0.75,-1,2};
              // velocity value in base frame.

  double* VelModified;
              // the modified velocity after force assistance function in base frame.

  Force ForceAssist(rbc,rbs,fss,frc,velMaster,csorg);
         // Initializa a object and set all the vriables needed by the function

  VelModified =ForceAssist.VelocityModify();
         // According to the force difference , use the algorithm introduced in
         // the flow chart to get the position and rotation change.
  for(i=0;i<6;i++)
    cout <<" The modified linear and angular velocity are :"<< *(VelModified+i)<<"\n";
  return 0;
}


class Force
{
  private:
    double CSorg[3]; // It is a vector which locates the origin of sensor frame
                  // with respect to constraint frame.
    double Rbc[3][3],Rcb[3][3],Rbs[3][3],Rcs[3][3];
```

```
                // Rbc: The transformation of constraint frame with respect
                //          to base frmae (world frame).
                // Rbs: between base frame and world frame.
                // Rcs: transformation of constraint frame with respect to
                //       base (world frame).
  double Tf[6][6];   // force-moment transformation.
  double Frc[6],Fss[6],Fsc[6];
                // Frc: Force and moment of reference value in constraint frame.
                // Fss: Sensored force in sensor frame.
                // Fsc: sensored force and moment with respect to constraint frame.

  double velModified[6];
  double ScaleL[3][3],ScaleA[3][3];  //scaling matrix
  double LinearV[3],AngVel[3] ;

 public:
  Force(double rbc[][3],double rbs[][3],double fss[],double frc[6],
     double velMaster[],double csorg[]);
       // Constructor. It initialize all the variables.
  Force();

  void transpose(double r1[][3],double r2[][3]);
       // transpose a matrix.
  void multiply1(double r1[][3],double r2[][3],double r[][3]);
       // multiply two matrix and get a new matrix
  void multiply2(double r[][3],double v1[],double v2[]) ;
  void getRcs();
       // Calculate the transformation of constraint frame relative to sensor frame.
  void buildTf();
       // calculate force-moment transformation and transform force from sensor frame
       // into constraint frame.
  double* VelocityModify();
       // Adjust position and rotation to compensate for the force and moment errors.
};


#include "force.h"
#include <iostream.h>
#include <stdio.h>

Force::Force(double rbc[][3],double rbs[][3],double fss[],double frc[6],
      double velMaster[],double csorg[])
{
 int i,j;

 for(i=0;i<3;i++)
   for(j=0;j<3;j++)
```

```
   {
    Rbc[i][j]=rbc[i][j];
    Rbs[i][j]=rbs[i][j];
    Rcb[i][j]=0;
   }
  for(i=0;i<6;i++)
  {
   Fss[i]=fss[i];
   Frc[i]=frc[i];
   velModified[i]=0;
  }
  for(i=0;i<3;i++)
  {
   LinearV[i]= velMaster[i];
   AngVel[i]= velMaster[i+3];
  }
  for(i=0;i<3;i++)
   CSorg[i]=csorg[i];

   transpose(Rbc,Rcb);
  getRcs();
  buildTf();
}

void Force::transpose(double r1[][3], double r2[][3])
{
  int i,j;

  for(i=0;i<3;i++)
    for(j=0;j<3;j++)
      r2[i][j]=r1[j][i];
}

void Force::getRcs()
{
  multiply1(Rcb,Rbs,Rcs);
}

void Force::multiply1(double r1[][3],double r2[][3],double r[][3])  //The function of matrix multiplying
{
  int i,j;

  for(i=0;i<3;i++)
    for(j=0;j<3;j++)
      r[i][j]=r1[i][0]*r2[0][j]+r1[i][1]*r2[1][j]+r1[i][2]*r2[2][j];
}
```

```
void Force::multiply2(double r[][3],double v1[],double v2[])
{
  int i,j;

  for(i=0;i<3;i++)
    for(j=0;j<3;j++)
      v2[i]+=r[i][j]*v1[j];
}

void Force::buildTf()  // construct a transformation between constraint frame and
                       // sensor frame.The constraint frame is built at the center
     // of end-effector.
{
  int i,j;
  double P[3][3];
  double mid[3][3];

  for(i=0;i<6;i++)
    Fsc[i]=0;

  P[0][0]=0;
  P[0][1]=(-1)*CSorg[2];
  P[0][2]=CSorg[1];
  P[1][0]=CSorg[2];
  P[1][1]=0;
  P[1][2]=(-1)*CSorg[0];
  P[2][0]=(-1)*CSorg[1];
  P[2][1]=CSorg[0];
  P[2][2]=0;

  for(i=0;i<3;i++)
    for(j=0;j<3;j++)
      Tf[i][j]=Rcs[i][j];

  for(i=0;i<3;i++)
    for(j=3;j<6;j++)
      Tf[i][j]=0;

  for(i=3;i<6;i++)
    for(j=3;j<6;j++)
      Tf[i][j]=Rcs[i-3][j-3];

  multiply1(P,Rcs,mid);

  for(i=3;i<6;i++)
    for(j=0;j<3;j++)
      Tf[i][j]=mid[i-3][j];
```

```
   for(i=0;i<6;i++)
   for(j=0;j<6;j++)
     Fsc[i]+=Tf[i][j]*Fss[j];
}

double* Force::VelocityModify()  // This function is to change the position and totation
                        // in order to follow the contact force and moment command
        // which are defined in the constraint frame.
{
  int i,j;
  double scale[6];

  double slaveL[3],slaveA[3];
  double newslaveL[3],newslaveA[3];

  FILE *forcedump;

  for(i=0;i<3;i++)
  {
   slaveL[i]=0;
   slaveA[i]=0;
   newslaveL[i]=0;
   newslaveA[i]=0;
  }

  for(i=0;i<6;i++)
  {
   if(Frc[i]==0)
     scale[i]=0.1;
   else
   {
     if((Frc[i]-Fsc[i])>0)
     {
 scale[i]=(Frc[i]-Fsc[i])/Frc[i];
 if(scale[i]<0.1) scale[i]=0.1;
     }
       else
         scale[i]=0.1;
   }
  }


  for(i=0;i<3;i++)
   for(j=0;j<3;j++)
   {
     if(i==j)
```

```
    {
 ScaleL[i][j]=scale[i];
 ScaleA[i][j]=scale[i+3];
    }
    else
    {
 ScaleL[i][j]=0;
 ScaleA[i][j]=0;
    }

   }


  multiply2(Rcb,LinearV, slaveL);
  multiply2(Rcb,AngVel, slaveA);

  multiply2(ScaleL,slaveL, newslaveL);
  multiply2(ScaleA,slaveA, newslaveA);



  if(!(forcedump = fopen("forcedump.txt", "a+")))
  {
   perror("fopen failed");
  //  exit(-1);
  }

fprintf(forcedump,"%1.2f,%1.2f,%1.2f,%1.2f,%1.2f,%1.2f,%1.2f,%1.2f,%1.2f,%1.2f,%1.2f,%1.2f,%1.2f,%1.2f,%1.2f,%1.2f,%1.2f,%1.2f,
%1.2f\n",
    slaveL[0],slaveL[1],slaveL[2],slaveA[0],slaveA[1],slaveA[2],
    newslaveL[0],newslaveL[1],newslaveL[2],newslaveA[0],newslaveA[1],newslaveA[2],
    Fsc[0],Fsc[1],Fsc[2],Fsc[3],Fsc[4],Fsc[5]);


  multiply2(Rbc,newslaveL, newslaveL);
  multiply2(Rbc,newslaveA, newslaveA);

  for(i=0;i<6;i++)
  {
   if(i<3)
     velModified[i]=newslaveL[i];
   else
     velModified[i]=newslaveA[i-3];
  }

  return velModified;
}
```

C.4 Main Reference Programming Code

C.4.1 The definition of the three classes

Planar class

```
class Planar
{
 private:
   double  gp[3];   // This is the goal point for linear movement
              double  mip[3], pp1[3],pp2[3],cp[3];
   double safeDis;
   double vmaster[3],wmaster[3];
              double vmodified[3],wmodified[3];
   double scale[3][3],scaleRPY[3][3];
          double Rc[3][3],tranRc[3][3];
          double projection[3];
   double a,b,c,d;
   double RightPosition[3],RightRotation[3];
 public:
   Planar();
   Planar(double GP[],double PP1[],double PP2[],double MIP[],
       double SafeDis);
   void Computetoolplane();
   void Getprojection();
   void Gettransform();
   void Transpose();
   void Initial();
   void Designscale();
   void DesignscaleRPY();
   void LinearMove();
   double* GetRightP();
   double* GetRightR();
   void Scaling();
   bool IsCPOnplane();
   void SetCP(double Vmaster[],double Wmaster[],double CP[]);
   double* GetV();
   double* GetW();
   void Show();
};
```

Linear class

```
class Linear
```

```
{
 private:
              double  mip[3];
   double  safeDis;
          double  lp1[3],lp2[3],cp[3]; // lp1equals the goal point for linear movement
   double vmaster[3],wmaster[3];
          double vmodified[3],wmodified[3];
   double scale[3][3],scaleRPY[3][3];
         double Rline[3][3],tranRline[3][3];
   double projection[3];
   double RightPosition[3],RightRotation[3];
 public:
   Linear();
   Linear(double LP1[],double LP2[],double mip[],
       double SafeDis);
   void Getprojection();
   void Gettransform();
   void Transpose();
   bool IsCPOnLine();
   void SetCP(double Vmaster[],double Wmaster[],double CP[]);
   void Initial();
   double* GetRightP();
   double* GetRightR();
   void Designscale();
   void DesignscaleRPY();
   void LinearMove();
   void Scaling();
   double* GetV();
   double* GetW();
};
Velocity class
class Velo
{
 private:
             double  gp[3],cp[3];
   double safeDis;
   double vmaster[3];
   double vmodified[3];
 public:
   Velo();
   Velo(double GP[],double SafeDis);
   void SetCP(double Vmaster[],double CP[]);
   double* Scaling();
};
```

C.4.2 The interpretation of all the variables in the functions

1. assistant.cpp

flag : assistance type(P means planar, L means linear , V means velocity)

double *V,*W: Using these two pointers to obtain the modified velocities

double *RightPosi ,*RightRota: Using a initialization function to get right position and orientation fitting for a cetain task.

double SafeDis: safe distance between the end effetor and the goal point.

double LP1[3] : Linear constraint point 1. It is considered the goal point for linear movement

double LP2[3]: Linear constraint point 2.

double GP[3] : This is the goal point for planar movement.

double PP1[3]: Planar constraint point 1

double PP2[3]: Planar constraint point 2.

**** The above three points are used to decide a constraint plane.

double MIP[3]: Initial point of the end effecter.

double CP[3]; This is the current point of the end effecter. It should be obtained online.

**** These points except CP should be from task file .The current point (CP) should be obtained online because it will be used to judge if the end effecter deviates from constraint plane or line. Now I am not sure if you can get it online. So I commented this part of codes in planar and linear functions.

2. Planar.cpp

scale[3][3]: scale matrix for linear velocity scaling

scaleRPY[3][3]: scale matrix for angular velocity scaling .

Rc[3][3]: transformation of constraint frame with respect to the base frame

tranRc[3][3]: the transpose of the Rc[3][3].

vnoscaled[3]: the velocity input for planar for the planar scaling with respect to base frame..

vcut[3]: the velocity input for planar for the planar scaling with respect to constraint frame. vscaled[3]: the velocity after scaled with respect to constraint frame.

vmodified[3]: the velocity after scaled with respect to base frame.

wnoscaled[3],wcut[3],wscaled[3],wmodified[3]: these four variable have similar meaning. The only difference is that they are angular velocity.

projection[3]: The projection of MIP on the constraint plane.

3. Linear.cpp

Rline[3][3]: transformation of constraint frame with respect to the base frame

transRl[3][3]: the transpose of the Rline[3][3].

scale[3][3]: scale matrix for linear velocity scaling

scaleRPY[3][3]: scale matrix for angular velocity scaling .

vnoscaled[3]: similar meaning as in planar case

vscaled[3]: similiar meaning as in planar case

vline[3]: similiar meaning as vcut in planar case

vmodified[3]: similiar meaning as in planar case

wnoscaled[3],wscaled[3],wline[3],wmodified[3]: similiar meaning as in planar case

projection[3]: The projection of MIP on the constraint line.

C.4.3 Main Function Program(Assistance.cpp)

```
/*****************************************************************************
  Assistance.cpp:  This is the main function file.
  Date: August 29,2001
  Revised on: Oct. 25,2001
  Author: Wentao Yu (University of South Florida)
  Contact: wyu@eng.usf.edu

*****************************************************************************/

/******************  Changes made on Oct 25 2001 by Wentao Yu **************

   1.  Units
     The length unit is in.
       The angle unit is radians.
     The linear velocity unit is in/s.
     The angular velocity unit is radians/s.

   2. For planar assistance,Adjusting three rotation angles has been separated
    into three steps. These three steps are in a for loop. RotationAdjust()
    function has a argument which determine which angle needs to be adjusted.

*****************************************************************************/


#include <iostream.h>
#include <math.h>
#include "Plane.h"
#include "Line.h"
#include "Velo.h"

#define RotationAccuracy 0.06 // This is used to judge if the rotation angle is right for a certain task.
#define SafeDis 3             // This is used to judge if the distance from CP to GP is safe.

int main()
{
  char modeflag='L'; // The flag is assistance  type
  int i;

  // Assume we can get all these points and velocities now.

  static double  LP1[3]={1,0,0}; // This is the goal point for linear movement
  static double  LP2[3]={0,0,0};
  static double  GP[3]={0,0,0};   // This is the goal point for planar movement
```

```
  static double  MIP[3]={1,2,2};
 static double  PP1[3]={1,0,0};
 static double  PP2[3]={0,1,0};
 static double  CP[3]={5,4,1};    //current position and orientation of the end-effector.
 static double  CR[3]={0.4,0.5,0.2}; //They should be obtained online.

 static double Vmaster[3]={0.2,0.3,0.4};
 static double Wmaster[3]={0.6,0.1,0.4};


 double *V,*W; //  Using these two pointers to obtain the scaled velocities
 double *RightPosi ,*RightRota;
    // Using a initialization function to get right position
      // and orientation fitting for a certain task.


//  These constraint points will be obtained from task file.But CP
//  (current point) will be obtained online, because it will be used
//  to judge if the end-effeector is on the plane or line

 switch(modeflag)
 {
    case 'P':          // Planar assistance
    {
      // constructor .Initialize all the constraint condition
Planar PAssist(GP,PP1,PP2,MIP,SafeDis);
RightPosi=PAssist.GetRightP();
RightRota=PAssist.GetRightR();
//Firstly, we check if the current position of the manipulator is
//on the constraint plane and if the current rotation fits for the task.If not,
//do linear movement from CP to the projection until the CP is on the plane and
//adjust the rotation.

PAssist.SetCP(Vmaster,Wmaster,CP,CR);
   // update current velocity and position
while(!PAssist.IsCPOnplane())
{
// if CP is not in the constraint plane,call LinearMove() function
// which makes end efector move to the projection.
 // The roll,yaw and roll direction are adjusted to be in right rotation.

  PAssist.LinearMove();
  V=PAssist.GetV();
  W=Wmaster;
  PAssist.SetCP(Vmaster,Wmaster,CP,CR);
        // CP and CR are used
 // to check if the end-effector is in the constraint
```

```
// plane,this is VERY IMPORTANT.Otherwise, the
// program can not get out from the loop.
}

for(i=0;i<3;i++)
{
  while(abs(*(RightRota+i)-CR[i])>RotationAccuracy)
  {
    PAssist.RotationAdjust(i);
    V=PAssist.GetV();
    W=PAssist.GetW();
    PAssist.SetCP(Vmaster,Wmaster,CP,CR);
  }
}

// Now the CP is on the constraint plane and the rotation fits for the certain task.
// Maybe the actual position is different from the right position.But because
// it is on the constraint plane,it is ok.

PAssist.Scaling();  // doing scaling operation.
V=PAssist.GetV();
W=PAssist.GetW(); // get the scaled velocities vector

for(i=0;i<3;i++)
    cout <<" The modified Linear velocities  is :"
        << *(V+i)<<"\n";
            for(i=0;i<3;i++)
    cout <<" The modified angular velocities  is :"
         << *(W+i)<<"\n";
// PAssist.Show();
     }
       break;
   case 'L':            // Linear assistance
     {
        Linear LAssist(LP1,LP2,MIP,SafeDis);
 RightPosi=LAssist.GetRightP();
 RightRota=LAssist.GetRightR();

 LAssist.SetCP(Vmaster,Wmaster,CP,CR);
 while(LAssist.IsCPOnLine())
 {
  LAssist.LinearMove();
  V=LAssist.GetV();
  W=LAssist.GetW();
  LAssist.SetCP(Vmaster,Wmaster,CP,CR);
 // update current velocity and position
 // CP and CR are used
```

```
// to check if the end-effector is on the constraint
// line.This is VERY IMPORTANT.Otherwise, the
// program can not get out from the loop.
}

for(i=0;i<3;i++)
{
  while(abs(*(RightRota+i)-CR[i])>RotationAccuracy)
  {
    LAssist.RotationCheck(i);
    V=LAssist.GetV();
    W=LAssist.GetW();
    LAssist.SetCP(Vmaster,Wmaster,CP,CR);
  }
}

// Now the CP is on the constraint line and the rotation fits for the task.

LAssist.SetCP(Vmaster,Wmaster,CP,CR);
LAssist.Scaling();    // doing scaling operation.
              V=LAssist.GetV();
        W=LAssist.GetW();     // get the scaled velocities vector
for(i=0;i<3;i++)
    cout <<" The modified Linear velocities  is :"
        << *(V+i)<<"\n";
             for(i=0;i<3;i++)
    cout <<" The modified angular velocities  is :"
          << *(W+i)<<"\n";
   //    LAssist.show();

      }
      break;
      case 'V':
      {
Velo Velocity(GP,SafeDis);
   // Velocity assistance
        Velocity.SetCP(Vmaster,CP);
   // update current velocity and position
         V=Velocity.Scaling();
   // increase or decrease the velocity depending on the
   // distance.
for(i=0;i<3;i++)
  cout<<*(V+i)<<"\n";
      }
      break;
//    case 'F':
//      break;
```

```
    default:
    break;
  }
  return 0;
}
```

Appendix D, RTSA OPENGL Software Listing

D1 OpenGL Viewer Head files

D1.1 RTSA_VIEW.h

```
#include <GL/glaux.h>
#include <GL/glut.h>
#include <GL/glu.h>


extern void animation(void);
extern unsigned __stdcall test(void *dummy/* int argc, char** argv */);
extern void addcomp(void);


/* Load image as texture */
extern void LoadGLTextures(char *texfile);

/* Draw a pipe oriented along the Z axis
 * The base of the pipe is placed at z = 0,
 * and the top at z = "height"
 */
extern void pipe(GLdouble r, /* the radius of the pipe */
                 GLdouble height /* height of the cylinder */);


/* pipe menu function */
extern void pipe_choice(int value);

/* Draw a Tee oriented along the Z axis
 * The base of the tee is placed at z = 0,
 * and the top at z = 2*A.
 * Another branch oriented along the X axis,
 * whose length is A
 */

/* ??? need to draw the fladge */
extern void screwedTee(GLdouble A, /* Dimensions of American 150 Lb */
                       GLdouble H, /* Standard Malleable-iron Screwed Tee */
                       GLdouble E);


/* screwedTee menu function */
extern void tee_choice(int value);

/* Draw part of torus,
 * which centered at the modeling coordinates origin
 * whose axis is aligned with the Z aixs
 *
 * this is a PRIVATE FUNCTION TO DRAW ELBOW
 */
extern void partTorus(GLfloat r, /* inner Radius */
                      GLfloat R, /* outer Radius */
                      GLint n, /* 1/nth of torus */
                      GLint nsides, /* number of sides for each radial section */
                      GLint rings /* 4x|8x, number of radial divisions for the torus */) ;

/* Draw a 90 degree Elbow oriented along the Z axis
 * the base of the elbow is placed at z = 0
 */
extern void elbow90(GLdouble r, GLdouble R);
```

```
/* Draw a 45 degree Elbow oriented along the Z axis
 * the base of the elbow is placed at z = 0
 */
extern void elbow45(GLdouble r, GLdouble R);

/* elbow90 menu function */
extern void elbow90_choice(int value);

/* elbow45 menu function */
extern void elbow45_choice(int value);


/* main menu function */
extern void main_menu_select(int value);

/* create menu */
extern void make_menu(void);


extern void init(char *texfile);

extern void param(int c,float s);

extern void Loadwallpaper();

extern void draw(double xPos,double yPos,double zPos,double zAngle,double yAngle,double xAngle,


/* display a pipe or tee */
extern void display(void);


extern void reshape(int w, int h);


/* key 'x'--rotate around x axis
 * key 'y'--rotate around y axis
 * key 'z'--rotate around z axis
 * key 'r'--reset
 * key 'q', escape--exit
 */
extern void keyboard(unsigned char key, int x, int y);
extern void CutPlane(void);
```

D1.2 fixtexture.h

```
#ifndef FIXTEXTURE_H
#define FIXTEXTURE_H

enum {SCALE_FIX, PAD_FIX};

int fixtexture(GLenum format,                   /* input */
               GLint widthin,                   /* input */
               GLint heightin,                  /* input */
               GLenum type,                     /* input */
               GLint *widthout,                 /* output */
               GLint *heightout,                /* output */
               GLvoid **data,                   /* input, output */
               GLenum ScalePad);                /* input */

#endif
```

D1.3 tiffload.h

```
/* Program: tiffload.h
 * Description: header file for loading tiff image for a texture map
 *
 */
#ifndef TIFFLOAD_H
#define TIFFLOAD_H
GLvoid *LoadTIFFtex(char *filename, GLsizei *width, GLsizei *height, GLenum *format, GLenum *type);
GLvoid UnloadTIFFtex(GLvoid *pixels);
#endif
```

D2 OpenGL Viewer Source file (RTSA_View.cpp)

```
#include "StdAfx.h"
#include "rtsa2.h"
#include <GL/glaux.h>
#include <GL/glut.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "tiffload.h"
#include "fixtexture.h"
#include "RTSA_VIEW.h"

#ifndef M_PI
#define M_PI 3.14159
#endif
#ifndef M_METR
#define M_METR 20. // Gain to change 1 inch to the unit in the virtual
#endif
#ifndef M_CAMERA
#define M_CAMERA 1.0
#endif

GLuint ListName;
GLuint flag =1;// flag to draw

//  static int wallpaper = 1; /* wallpaper--1: load image as wallpaper; 0: unload image */

static GLenum drawStyle = GLU_FILL; /* GLU_FILL, GLU_LINE */


static float radius = 1.0;/* radius of pipe */

/* standard data of tee */
static GLdouble a = 1.50;  /* standard data of elbow45, also */
static GLdouble h = 1.77;
static GLdouble e = 0.302;

static GLdouble c = 1.12;   /* standard data of elbow45 */

static float x_offset=1.0;
static float y_offset=0.0;
static float z_offset=-400.0;


struct objectInfo* pglObjList;
struct objectInfo  glObjList;
POSITION SearchPos;

GLuint k_texture[1];
GLvoid *pixels1;
```

```
unsigned int texture;
GLsizei width, height;

/* Load image as texture */
void LoadGLTextures(char *texfile)
{
GLsizei w, h;
                        GLenum format, type;
                        AUX_RGBImageRec *TextureImage[1];
                        memset(TextureImage,0,sizeof(void *)*1);
    TextureImage[0]=auxDIBImageLoad(texfile);
    format=GL_RGB;
    w=TextureImage[0]->sizeX;
    h=TextureImage[0]->sizeY;
    type=GL_UNSIGNED_BYTE;
    pixels1= TextureImage[0]->data;
    fixtexture(format, w, h, type, &w, &h, &pixels1,  SCALE_FIX); //1st~4th is input paramter, Change the image to fit power of 2 ,
pixels is output od image data, &w,&h is new saize of image, SCALE_FIX is the parameter to change the size of picture to power of 2
    width=w;
    height=h;

    glGenTextures(1, &k_texture[0]);
    glBindTexture(GL_TEXTURE_2D, k_texture[0]);
    glTexImage2D(GL_TEXTURE_2D, 0, format, w, h, 0, GL_RGB, GL_UNSIGNED_BYTE, pixels1);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);

            if (TextureImage[0])
                {
                 if (TextureImage[0]->data)                    // If Texture Image Exists
                     {
                      free(TextureImage[0]->data);             // Free The Texture Image Memory
                                                                                                                                  }
                      free(TextureImage[0]);                   // Free The Image Structure
                     }
                 free(pixels1);}


void Loadwallpaper()
{
float p_gain=1.0,p_dist=.51; //p_dist make 5000*p_dist
float wpx=0.0,wpy=0.0,wpz=0.0,wor=0.0,wop=0.0,woy=0.0;

/* load the image as wallpaper */

wpx=((CRtsa2App*)AfxGetApp())->WallPaperOffset.x+x_offset;
wpy=((CRtsa2App*)AfxGetApp())->WallPaperOffset.y+y_offset;
wpz=((CRtsa2App*)AfxGetApp())->WallPaperOffset.z+z_offset;
wor=((CRtsa2App*)AfxGetApp())->WallPaperOffset.roll;
wop=((CRtsa2App*)AfxGetApp())->WallPaperOffset.pitch;
woy=((CRtsa2App*)AfxGetApp())->WallPaperOffset.yaw;



if (wpz>-70.0)
{
   wpz=-70;
}
else if (wpz<-2500)
{
   wpz=-2500;
}

wpz=-900.0;

                        glPushMatrix();
                                glEnable(GL_TEXTURE_2D);
                    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,      GL_DECAL);

                        glTranslatef(wpx,wpy,wpz);
```

```
                        glRotatef(wor, 0, 0, 1);
                        glRotatef(wop, 0, 1, 0);
                        glRotatef(woy, 1, 0, 0);

                        //if(wallpaper)
                        //{
                                glBegin(GL_QUADS);
                                glTexCoord2f(0.0, 0.0); glVertex2f(-320, -240);
                                glTexCoord2f(1.0, 0.0); glVertex2f(320, -240);
                                glTexCoord2f(1.0, 1.0); glVertex2f(320, 240);
                                glTexCoord2f(0.0, 1.0); glVertex2f(-320, 240);
                                glEnd();
                        glFlush();
                        glDisable(GL_TEXTURE_2D);
                        //}
                        glPopMatrix();
}


/* Draw a pipe oriented along the Z axis
 * The base of the pipe is placed at z = 0,
 * and the top at z = "height"
 */

void pipe(GLdouble r, /* the radius of the pipe */
                        GLdouble height /* height of the cylinder */)
{
                        GLint slices = 40; /* the number of subdivisions around the z axis */
                        GLint stacks = 1; /* the number of subdivisions along  the  z axis */
                        GLUquadricObj *obj; /* quadrics object */

                        obj = gluNewQuadric();
                        glPushMatrix();
                        /* cylinder wall */
                        gluQuadricDrawStyle(obj, drawStyle);
                        gluCylinder(obj, r, r, height, slices, stacks);

                        /* bottom */

                        glPushMatrix();
                        glRotatef(180, 1, 0, 0);
                        gluDisk(obj, 0, r, slices, stacks);
                        glPopMatrix();

                        /* top */

                        glTranslatef(0, 0, height);
                        gluDisk(obj, 0, r, slices, stacks);

                        glPopMatrix();
                        gluDeleteQuadric(obj);
}



/* pipe menu function */
void pipe_choice(int value)
{
//                      choice = 1;

        switch(value) {
        case 1:
                        radius = 1.0*M_METR/2.;//1.315 / 2;
                        break;
        case 2:
                        radius = 2.0*M_METR/2.;//2.375 / 2;
                        break;
        case 3:
```

```
                radius = 2.5*M_METR/2.;//2.875 / 2;
                break;
        case 4:
                radius = 3.0*M_METR/2.;//3.5 / 2;
                break;
        case 5:
                radius = 4.0*M_METR/2.;//4.5 / 2;
                break;

        }

        glutPostRedisplay();
}


/* Draw a Tee oriented along the Z axis
 * The base of the tee is placed at z = 0,
 * and the top at z = 2*A.
 * Another branch oriented along the X axis,
 * whose length is A
 */

/* ??? need to draw the fladge */
void screwedTee(GLdouble A, /* Dimensions of American 150 Lb */


{
        glPushMatrix();
        glTranslatef(0, 0, A);
        glRotatef(90, 0, 1, 0);
        pipe(H/2, A);
        glPopMatrix();

        pipe(H/2, 2*A);
}

/* screwedTee menu function */
void tee_choice(int value)
{
//      choice = 2;

        switch(value) {
        case 1:
                a = 1.50;
                h = 1.771;
                e = 0.302;
                break;
        case 2:
                a = 2.25;
                h = 2.963;
                e = 0.422;
                break;
        case 3:
                a = 2.70;
                h = 3.589;
                e = 0.478;
                break;
        case 4:
                a = 3.08;
                h = 4.285;
                e = 0.548;
                break;

        case 5:
                a = 3.79;
                h = 5.401;
                e = 0.661;
                break;

        }
```

```
        glutPostRedisplay();
}

/* Draw part of torus,
 * which centered at the modeling coordinates origin
 * whose axis is aligned with the Z aixs
 *
 * this is a PRIVATE FUNCTION TO DRAW ELBOW
 */
void partTorus(GLfloat r, /* inner Radius */
                          GLfloat R, /* outer Radius */
                          GLint n, /* 1/nth of torus */
                          GLint nsides, /* number of sides for each radial section */
                          GLint rings /* 4x|8x, number of radial divisions for the torus */)
{
  int i, j;
  GLfloat theta, phi, theta1;
  GLfloat cosTheta, sinTheta;
  GLfloat cosTheta1, sinTheta1;
  GLfloat ringDelta, sideDelta;

  ringDelta = 2.0 * M_PI / rings;
  sideDelta = 2.0 * M_PI / nsides;

  theta = 0.0;
  cosTheta = 1.0;
  sinTheta = 0.0;

  for (i = rings / n - 1; i >= 0; i--) {
    theta1 = theta + ringDelta;
    cosTheta1 = cos(theta1);
    sinTheta1 = sin(theta1);
    glBegin(GL_QUAD_STRIP);
    phi = 0.0;
    for (j = nsides; j >= 0; j--) {
      GLfloat cosPhi, sinPhi, dist;

      phi += sideDelta;
      cosPhi = cos(phi);
      sinPhi = sin(phi);
      dist = R + r * cosPhi;

      glNormal3f(cosTheta1 * cosPhi, -sinTheta1 * cosPhi, sinPhi);
      glVertex3f(cosTheta1 * dist, -sinTheta1 * dist, r * sinPhi);
      glNormal3f(cosTheta * cosPhi, -sinTheta * cosPhi, sinPhi);
      glVertex3f(cosTheta * dist, -sinTheta * dist,  r * sinPhi);
    }
    glEnd();
    theta = theta1;
    cosTheta = cosTheta1;
    sinTheta = sinTheta1;
  }
}



/* Draw a 90 degree Elbow oriented along the Z axis
 * the base of the elbow is placed at z = 0
 */
void elbow90(GLdouble r, GLdouble R)
{

  GLint nsides = 40; /* number of sides for each radial section */
  GLint rings = 40; /* 4x, number of radial divisions for the torus */
  GLUquadricObj *obj; /* quadrics object */

  obj = gluNewQuadric();
  gluQuadricDrawStyle(obj, drawStyle);
```

```
  glPushMatrix();
  if(drawStyle == GLU_FILL) {
    /* elbow wall */
    glPushMatrix();
                glRotatef(90, 0, 1, 0);
                glTranslatef(-R, 0, 0);
                glRotatef(90, 0, 0, 1);
    partTorus(r, R, 4, nsides, rings);
                glPopMatrix();

                /* bottom */
                glPushMatrix();
                glRotatef(90, 1, 0, 0);
                gluDisk(obj, 0, r, 20, 1);
                glPopMatrix();

                /* top */
                glPushMatrix();
                glTranslatef(0, R, R);
                gluDisk(obj, 0, r, 20, 1);
                glPopMatrix();

  } else if(drawStyle == GLU_LINE) {
    glPushAttrib(GL_POLYGON_BIT);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
                /* elbow wall */
                glPushMatrix();
                glRotatef(90, 0, 1, 0);
                glTranslatef(-R, 0, 0);
                glRotatef(90, 0, 0, 1);
    partTorus(r, R, 4, nsides, rings);
                glPopMatrix();

                /* bottom */
                glPushMatrix();
                glRotatef(90, 1, 0, 0);
                gluDisk(obj, 0, r, 20, 1);
                glPopMatrix();

                /* top */
                glPushMatrix();
                glTranslatef(0, R, R);
                gluDisk(obj, 0, r, 20, 1);
                glPopMatrix();

    glPopAttrib();
  }
  glPopMatrix();
  gluDeleteQuadric(obj);
}

/* Draw a 45 degree Elbow oriented along the Z axis
 * the base of the elbow is placed at z = 0
 */
void elbow45(GLdouble r, GLdouble R)
{
  GLint nsides = 40; /* number of sides for each radial section */
  GLint rings = 40; /* 8x, number of radial divisions for the torus */
  GLUquadricObj *obj; /* quadrics object */

  obj = gluNewQuadric();
  gluQuadricDrawStyle(obj, drawStyle);

  glPushMatrix();

  if(drawStyle == GLU_FILL) {
    glPushMatrix();
                /* elbow wall */
                glRotatef(90, 0, 1, 0);
                glTranslatef(-R, 0, 0);
```

```
            glRotatef(45, 0, 0, 1);
   partTorus(r, R, 8, nsides, rings);
            glPopMatrix();



            /* bottom */
            glPushMatrix();
            glRotatef(90, 1, 0, 0);
            gluDisk(obj, 0, r, 20, 1);
            glPopMatrix();

            /* top */
            glRotatef(90, 0, 1, 0);
            glTranslatef(-R, 0, 0);
            glRotatef(45, 0, 0, 1);
            glTranslatef(R, 0, 0);
            glRotatef(-90, 1, 0, 0);
            gluDisk(obj, 0, r, 20, 1);

  } else if(drawStyle == GLU_LINE) {
   glPushAttrib(GL_POLYGON_BIT);
   glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
            glPushMatrix();
            /* elbow wall */
            glRotatef(90, 0, 1, 0);
            glTranslatef(-R, 0, 0);
            glRotatef(45, 0, 0, 1);
   partTorus(r, R, 8, nsides, rings);
            glPopMatrix();

            /* bottom */
            glPushMatrix();
            glRotatef(90, 1, 0, 0);
            gluDisk(obj, 0, r, 20, 1);
            glPopMatrix();

            /* top */
            glRotatef(90, 0, 1, 0);
            glTranslatef(-R, 0, 0);
            glRotatef(45, 0, 0, 1);
            glTranslatef(R, 0, 0);
            glRotatef(-90, 1, 0, 0);
            gluDisk(obj, 0, r, 20, 1);

    glPopAttrib();
   }
   glPopMatrix();
   gluDeleteQuadric(obj);
}

/* elbow90 menu function */
void elbow90_choice(int value)
{
//          choice = 3;

            switch(value) {
            case 1:
                        a = 1.50;
                        radius = 1.315 / 2;
                        break;
            case 2:
                        a = 2.25;
                        radius = 2.375 / 2;
                        break;
            case 3:
                        a = 2.70;
                        radius = 2.875 / 2;
                        break;
            case 4:
```

```
                a = 3.08;
                radius = 3.5 / 2;
                break;
        case 5:
                a = 3.79;
                radius = 4.5 / 2;
                break;

        }

        glutPostRedisplay();
}

/* elbow45 menu function */
void elbow45_choice(int value)
{
//      choice = 4;

        switch(value) {
        case 1:
                c = 1.12;
                radius = 1.315 / 2;
                break;
        case 2:
                c = 1.68;
                radius = 2.375 / 2;
                break;
        case 3:
                c = 1.95;
                radius = 2.875 / 2;
                break;
        case 4:
                c = 2.17;
                radius = 3.5 / 2;
                break;
        case 5:
                c = 2.61;
    radius = 4.5 / 2;
                break;

        }

        glutPostRedisplay();
}




void init(char *texfile)
{
        GLfloat mat_ambient_and_diffuse[] = {1.0, 1.0, 1.0, 1.0};
        GLfloat mat_specular[]={1.,1.,1.,1.};
        GLfloat mat_shininess[]={0.};
        GLfloat light_position0[]={100.,-100.,100.,0.};
        GLfloat light_position1[]={1.,1.,1.,0.};
        GLfloat white_light[]={1.,1.,1.,1.};

        glClearColor(0., 0., 0., 0);
        glShadeModel(GL_SMOOTH);
  glShadeModel(GL_FLAT);

//      glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,mat_specular);
//      glMaterialfv(GL_FRONT_AND_BACK,GL_SHININESS,mat_shininess);
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, mat_ambient_and_diffuse);

        glLightfv(GL_LIGHT0,GL_POSITION,light_position0);
        glLightfv(GL_LIGHT0,GL_DIFFUSE,white_light);
//      glLightfv(GL_LIGHT0,GL_SPECULAR,white_light);
```

```
//              glLightfv(GL_LIGHT1,GL_POSITION,light_position1);
//              glLightfv(GL_LIGHT1,GL_DIFFUSE,white_light);
//              glLightfv(GL_LIGHT1,GL_SPECULAR,white_light);

                glEnable(GL_LIGHTING);
                glEnable(GL_LIGHT0);
//              glEnable(GL_LIGHT1);
//              glEnable(GL_DEPTH_TEST);



                glEnable(GL_COLOR_MATERIAL);
                LoadGLTextures(texfile);

 //Added by Kim,for scale
//                                      glScalef(.978,1.0,1.42);
                                        glScalef(1.,1.,1.);


 //   glShadeModel(GL_FLAT);

//              glEnable(GL_TEXTURE_2D);

}



void CutPlane(void)
{


                glEnable(GL_BLEND);
                glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
                glShadeModel(GL_FLAT);
                glClearColor(0.,0.,0.,0.);


                glColor4f(1,0,0,0.55);
                glPushMatrix();
//              glLoadIdentity();
//              glTranslatef(baseX,baseY,baseZ);
//              glRotatef(baseR,0,0,1);
//              glRotatef(0/*baseP*/,0,1,0);
///             glRotatef(baseYaw,1,0,0);

                glPushMatrix();
                glRotatef(180,0,1,0);

                glBegin(GL_POLYGON);
                glVertex3f(0,-40,0);
                glVertex3f(30,-30,0);
                glVertex3f(20,-30,0);
                glVertex3f(20,30,0);
                glVertex3f(-20,30,0);
                glVertex3f(-20,-30,0);
                glVertex3f(-30,-30,0);
                glEnd();
                glPopMatrix();

                glBegin(GL_POLYGON);
                glVertex3f(0,-40,2);
                glVertex3f(30,-30,2);
                glVertex3f(20,-30,2);
                glVertex3f(20,30,2);
                glVertex3f(-20,30,2);
                glVertex3f(-20,-30,2);
                glVertex3f(-30,-30,2);
                glEnd();
```

```
		glBegin(GL_POLYGON);
		glVertex3f(20,30,0);
		glVertex3f(20,30,+2);
		glVertex3f(-20,30,+2);
		glVertex3f(-20,30,0);
		glEnd();

		glBegin(GL_POLYGON);
		glVertex3f(20,30,0);
		glVertex3f(20,30,+2);
		glVertex3f(20,-30,+2);
		glVertex3f(20,-30,0);
		glEnd();

		glBegin(GL_POLYGON);
		glVertex3f(-20,30,0);
		glVertex3f(-20,30,+2);
		glVertex3f(-20,-30,+2);
		glVertex3f(-20,-30,0);
		glEnd();

		glBegin(GL_POLYGON);
		glVertex3f(20,-30,0);
		glVertex3f(20,-30,+2);
		glVertex3f(30,-30,+2);
		glVertex3f(30,-30,0);
		glEnd();

		glBegin(GL_POLYGON);
		glVertex3f(-20,-30,0);
		glVertex3f(-20,-30,+2);
		glVertex3f(-30,-30,+2);
		glVertex3f(-30,-30,0);
		glEnd();

		glBegin(GL_POLYGON);
		glVertex3f(30,-30,0);
		glVertex3f(30,-30,+2);
		glVertex3f(0,-40,+2);
		glVertex3f(0,-40,0);
		glEnd();
		glBegin(GL_POLYGON);
		glVertex3f(-30,-30,0);
		glVertex3f(-30,-30,+2);
		glVertex3f(0,-40,+2);
		glVertex3f(0,-40,0);
		glEnd();

		glPopMatrix();
		glDisable(GL_BLEND);
}

void draw(double xPos,double yPos,double zPos,double zAngle,double yAngle,double xAngle,
			double length,int choice,int color)
{
char *mesg, *title;

		/* draw the origin & coordinate */

			glPushMatrix();
			glColor3f(1, 0, 0);
			glBegin(GL_LINES);
			glVertex3f(0, 0, 0);
			glVertex3f(1, 0, 0);
			glEnd();

			glColor3f(0, 1, 0);
			glBegin(GL_LINES);
			glVertex3f(0, 0, 0);
```

```
                glVertex3f(0, 1, 0);
                glEnd();

                glColor3f(0, 0, 1);
                glBegin(GL_LINES);
                glVertex3f(0, 0, 0);
                glVertex3f(0, 0, 1);
                glEnd();
                glPopMatrix();

                glPushMatrix();
                glColor3f(1, 1, 0.4);
//              glLoadIdentity();
                glTranslatef(xPos, yPos, zPos);
                glRotatef(zAngle, 0, 0, 1);
                glRotatef(yAngle, 0, 1, 0);
                glRotatef(xAngle, 1, 0, 0);
//              printf("choice = %d",choice);

                 if (pglObjList->cutplane==1)
                 {
                        glPushMatrix();
                        double zCutInc=((CRtsa2App*)AfxGetApp())->PlaneBase.bz/M_CAMERA-zPos;
                        double rCutInc=((CRtsa2App*)AfxGetApp())->PlaneBase.broll-zAngle;

//                      glRotatef(rCutInc, 0, 0, 1);
                        glTranslatef(0,0,zCutInc);
                        glRotatef(rCutInc, 0, 0, 1);

                        CutPlane();
                        glPopMatrix();
                 }

                switch(choice) {
                case 1:
        if (color == 1)
                        {
                         glColor3f(1, 0, 0);
                //       color = 0;
                        }

                        pipe(radius, length);
                        glutPostRedisplay();
                        break;
                case 2:

                        if (color == 1) glColor3f(1, 0, 0);
                        screwedTee(a*M_METR/M_CAMERA, h*M_METR/M_CAMERA, e*M_METR/M_CAMERA);
                        glutPostRedisplay();
                        break;
                case 3:
                if (color == 1) glColor3f(1, 0, 0);

                        elbow90(radius*M_METR/M_CAMERA, a*M_METR/M_CAMERA);
                        glutPostRedisplay();
                        break;
                case 4:
                if (color == 1) glColor3f(1, 0, 0);

                        elbow45(radius*M_METR/M_CAMERA, c * tan((90-22.5)/180*M_PI)*M_METR/M_CAMERA);
                        glutPostRedisplay();
                        break;
                default:break;
        }
                glPopMatrix();


}
```

```
/* display a pipe or tee */
void display(void)
{
	double xPos,yPos,zPos,zAngle,yAngle,xAngle,radius,length;
	int choice=0;
	int value;
	int color = 0; //color =1 for red, 0 for yellow
	char *mesg, *title;

	xPos = 0.;
	yPos = 0.;
	zPos = 0.;
	zAngle = 0.;
	yAngle = 0.;
	xAngle = 0.;
	//radius = 0.;
	length = 0.;

	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);


	Loadwallpaper();	//



		pglObjList = &glObjList;
		SearchPos = ((CRtsa2App*)AfxGetApp())
	    ->MasterPartList.GetHeadPosition();

		while (SearchPos!=NULL)
		{
		    glObjList = ((CRtsa2App*)AfxGetApp())
		  ->MasterPartList.GetNext(SearchPos);

				if (pglObjList->sz == oneInch)
					value = 1;
				else if (pglObjList->sz == twoInch)
					value = 2;
				else if (pglObjList->sz == twoAndAHalf)
					value = 3;
				else if (pglObjList->sz == threeInch)
					value = 4;
				else if (pglObjList->sz == fourInch)
					value = 5;

				if (pglObjList->ftng == pip)
				{
					choice = 1;
					length = pglObjList->len;
					pipe_choice(value);
				}

				else if (pglObjList->ftng == tee)
				{
					choice = 2;
					tee_choice(value);
				}
				else if (pglObjList->ftng == elb)
				{
					choice = 3;
					elbow90_choice(value);
				}
				else
					choice =0;

				xPos = pglObjList->pos.x;
				yPos = pglObjList->pos.y;
				zPos = pglObjList->pos.z;
			       zAngle = pglObjList->ornt.r;//roll
```

```
                                        yAngle = pglObjList->ornt.p;//pitch
                                        xAngle = pglObjList->ornt.y;//yaw
                                        if (pglObjList->partselect==1 || pglObjList->cutplane==1)

                                                color = 1;
                                        else
                                                color = 0;


        draw(xPos,yPos,zPos,zAngle,yAngle,xAngle,length,choice, color);



                        }

                        //flag++;
                        glutSwapBuffers();

}


void reshape(int w, int h)
{
//              float theta;
                double wx,wy,wz,cx,cy,cz;
    wx = ((CRtsa2App*)AfxGetApp())->WallPaperOffset.x;
    wy = ((CRtsa2App*)AfxGetApp())->WallPaperOffset.y;
    wz = ((CRtsa2App*)AfxGetApp())->WallPaperOffset.z;

                cx = ((CRtsa2App*)AfxGetApp())->CamPosOrien.x;
                cy = ((CRtsa2App*)AfxGetApp())->CamPosOrien.y;
                cz = ((CRtsa2App*)AfxGetApp())->CamPosOrien.z;



 //   glViewport(-40, -40, w*1.2, h*1.2);
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();


//Kim  fov
   gluPerspective(31.56,  1.3, 0.1, 10000.0);
   gluLookAt(cx,cy,cz,wx,wy,wz,0,1,0);

   glMatrixMode(GL_MODELVIEW);
}


/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
extern unsigned __stdcall test(void *dummy/* int argc, char** argv */)
{

  glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize (640, 480);
  glutCreateWindow ("RTSA_VIEW");
  init("c:\\deneb\\RTSAProj\\TEXTURES\\RightImage.bmp");
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);

  glutMainLoop();
  return 0;
}
```