1997

The ToySim Project

Anthony Rodriguez

*Edith Cowan University*

Follow this and additional works at: [https://ro.ecu.edu.au/theses_hons](https://ro.ecu.edu.au/theses_hons)

Part of the [Artificial Intelligence and Robotics Commons](https://ro.ecu.edu.au/theses_hons) and the [Software Engineering Commons](https://ro.ecu.edu.au/theses_hons)

**Recommended Citation**


This Thesis is posted at Research Online. [https://ro.ecu.edu.au/theses_hons/468](https://ro.ecu.edu.au/theses_hons/468)
Edith Cowan University

Copyright Warning

You may print or download ONE copy of this document for the purpose of your own research or study.

The University does not authorize you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site.

You are reminded of the following:

• Copyright owners are entitled to take legal action against persons who infringe their copyright.

• A reproduction of material that is protected by copyright may be a copyright infringement. Where the reproduction of such material is done without attribution of authorship, with false attribution of authorship or the authorship is treated in a derogatory manner, this may be a breach of the author’s moral rights contained in Part IX of the Copyright Act 1968 (Cth).

• Courts have the power to impose a wide range of civil and criminal sanctions for infringement of copyright, infringement of moral rights and other offences under the Copyright Act 1968 (Cth). Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
The ToySim project.

By

Anthony Rodriguez

A thesis submitted in partial fulfilment of the requirements for the degree of

Bachelor of Engineering (Electronic Systems)

School of Engineering

1597
USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.
I certify that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any institution of higher education; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text.

Signature:

Date: 6-12-98
# Contents

## 1.0 Introduction
*An in-depth introduction to the project, and this thesis.*

### 1.1 What is ToySim?

### 1.2 Overview of objectives,

1.2.1 Objective 1 - A graphical display.
1.2.2 Objective 2 - Immediate motion.
1.2.3 Objective 3 - Indirect motion.
1.2.4 Objective 4 - Robot specifications.
1.2.5 Objective 5 - Robot control.

### 1.3 The RTX robot arm,

1.3.1 Introducing the RTX.
1.3.2 RTX components.
1.3.3 RTX coding.
1.3.4 RTX program level.

## 2.0 Resource Literature
*Researching for ideas, methods and alternatives.*

2.1 Current simulator packages.
2.2 Development packages.
2.3 Pros and cons.

## 3.0 Compendium
*Describing the project in abstract terms.*

3.1 Theory in 3D.

3.1.1 The Cartesian coordinate system.
3.1.2 Translation in 3D.
3.1.3 Rotation in 3D.
3.1.4 Scaling in 3D.
3.1.5 Transformations in 3D.
3.1.6 Inverting transforms.

3.2 The graphical display.

3.2.1 Screen.
3.2.2 Load.
3.2.3 Scene.
3.2.4 Object.

3.3 Immediate motion.

3.4 Indirect motion.

3.4.1 Examining START.COM.
Contents

3.5 Robot specifications. 38
3.6 Robot control 39

4.0 Application Technology 42
Descriptive analysis of existing technologies and their applications.

4.1 What is DirectX™? 44
4.2 Introducing Direct3D™ 45

4.3 Useful Direct3D™ data types. 45
   4.3.1 General Direct3D™ data types. 46
   4.3.2 DirectDraw™ data types. 46
   4.3.3 Direct3D™ device data types. 47
   4.3.4 Retained mode display data types. 48
   4.3.5 Retained mode data types. 49

4.4 Useful Direct3D™ functions. 50
   4.4.1 IDirect3DRM interface functions. 51
   4.4.2 IDirectDraw interface functions. 53
   4.4.3 IDirectDrawClipper interface functions. 53
   4.4.4 IDirect3DRMFrame interface functions. 54
   4.4.5 IDirect3DRMWinDevice interface functions. 57
   4.4.6 IDirect3DRMViewport interface functions. 57
   4.4.7 IDirect3DRMDevice interface functions. 57
   4.4.8 IDirect3DRMMeshBuilder interface functions. 58
   4.4.9 IDirect3D interface functions. 60

5.0 Implementation 61
Analysis of solution in the 'real world'.

5.1 A graphical display. 61
   5.1.1 FindDevice. 62
   5.1.2 CreateDevice. 64
   5.1.3 CreateScene. 66
   5.1.4 LoadTextures. 71
   5.1.5 AlterRenderQuality. 72
   5.1.6 Render. 76
   5.1.7 Routines found in WindowProc. 76

5.2 Immediate motion. 79

5.3 Indirect motion. 81
   5.3.1 TSDOSDRV.ASM. 84
   5.3.2 ProcessCOMEvent. 87
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.3 WaitForCOMEvent.</td>
<td>88</td>
</tr>
<tr>
<td>5.3.4 WriteBytesToCOM.</td>
<td>89</td>
</tr>
<tr>
<td>5.3.5 ReadBytesFromCOM.</td>
<td>90</td>
</tr>
<tr>
<td>5.3.6 UpdateCOMButtons.</td>
<td>91</td>
</tr>
<tr>
<td>5.3.7 UpdateMotorsAfterMove.</td>
<td>93</td>
</tr>
<tr>
<td>5.3.8 AdjustMotorMode.</td>
<td>94</td>
</tr>
<tr>
<td>5.3.9 ForceClipACK.</td>
<td>95</td>
</tr>
<tr>
<td>5.3.10 RTX_ARM_GO.</td>
<td>96</td>
</tr>
<tr>
<td>5.3.11 UpdateClipboard.</td>
<td>99</td>
</tr>
<tr>
<td>5.3.12 Routines found in WindowProc.</td>
<td>103</td>
</tr>
</tbody>
</table>

5.4 Robot specifications. 105

5.5 Robot control. 107

5.6 Robot extras.
- 5.6.1 The SNAP function. 113
- 5.6.2 The Dynamic Trails module. 114
- 5.6.3 About and Display Envelope. 116

6.0 Verification 119

**Proving the scheme, by quantitative analysis.**

6.1 Verification expectations. 119

6.2 The companion programs.
- 6.2.1 START.COM. 119
- 6.2.2 INIT.COM. 120
- 6.2.3 TEST.COM. 120
- 6.2.4 DRIVE.COM. 121
- 6.2.5 DEMO.COM. 124

6.3 Custom programs.
- 6.3.1 Interpolate program. 125
- 6.3.2 RawCommunications program. 128

7.0 Conclusion 133

**Wrap-up and inferences.**

8.0 References 136

**List of significant references.**
1.0 Introduction

This section introduces us to the ToySim project and its related objectives, all of which will be itemised and explained in general terms.

In the coming sections we will once again examine these objectives, in more technical terms, and attempt to satisfy them within the scheme of the project.

1.1 What is ToySim?

The ToySim project was so-named so as to induce a sense of irony. That is, this project represents the culmination of seven months of intense programming and testing, resulting in a package that is not a mere ‘toy’, but a real-world useable software suite.

Put simply, the ToySim project consists of two main components:

- A PC-compatible computer, and
- A programmable robot arm. (specifically, the RTX robot arm)

The objective of the ToySim project is to create an environment capable of simulating the movement, and behaviour, of a robotic arm and, moreover, make it available to students for the purpose of program development and debugging.

The project’s ultimate aim is to be able to allow a student to visualise a robotic environment in three dimensions and then to interact in the said ‘3D’ environment, dispensing with the need for an actual hardware robot, that being replaced by a software robot.

1.2 Overview of objectives

Following is a list of objectives first introduced with the original project proposal, extended to cover the completed project.

It should be noted that the objectives list is targeted to an audience with a modicum amount of computer literacy, and so introduces terms which will not be covered until later in this document. For completeness-sake these terms will be tentatively explained along with the objectives.
1.2.1 Objective 1 - A graphical display

The user interface should be easy to interpret and, ideally, be graphically oriented.

The graphical display should be a 3D environment (ie. 3 Dimensional environment), allowing the display of 3D objects in true 3D space. It should have the capability of translation, rotation, scaling, and animation within the 3D space, and should also show where the user is, within the 3D space, at any time, preferably using the easily recognisable (X,Y,Z) cartesian coordinate format.

Facilitating the ease of perception, solid-modelling should be used for 3D rendering, with the possibility of texture-mapping, should the target computer’s resources, and speed, allow.

The following diagram represents the desired cartesian coordinate 3D system desired for the project ...

![Figure 1.2.1 XYZ Axis.](image)

The coordinate system consists of three axes, each orthogonal to each other, labelled X axis, Y axis and Z axis respectively. Traditionally, any point within this 3D space can be accessed via a unique coordinate based on these three axes, (X,Y,Z), which represent displacements from the origin along the respective axis; the origin being the point at which the three axes intersect.

Movement along any of the three axes is termed translation, and can be in either the positive direction or negative direction. Rotation can occur about any individual axis or about a combination of axes, by rotating a point in the positive or negative direction perpendicular to the axis in question. Altering the size of an object is termed scaling, and can also be performed in any combination of axes, and in the positive and negative directions.

Loosely speaking, an object consists of a collection of points each linked to the other in order to construct a defined shape, or surface. Once a surface has been defined within the constraints of an object, it can be 3D rendered, where the internals (ie. the area enclosed by the
surface) are shaded to a pre-defined colour. Where a picture (bitmap) is used instead of a colour, the process is termed texture-mapping; texture-mapping allows for a more realistic view of the environment in question.

### 1.2.2 Objective 2 - Immediate motion

The user should have the ability to manipulate the simulated robot in an 'immediate' fashion, so as to be able to program it and see the results immediately following. This will allow the user to enter into debug sessions and experiment with 'what-if' scenarios.

Ideally, the interface should appear to the user as an information-rich graphical display, and allow the user to specify where they want the robot to go. Following this theme should be a module whose specific purpose is to translate the information and perform the movements in the graphical space, with appropriate feedback to the user.

### 1.2.3 Objective 3 - Indirect motion

The user should have the ability to manipulate the simulated robot in an 'indirect' fashion, so as to be able to program it and see the results immediately following. This may seem to be identical to objective 2, but its uniqueness is in the subtle difference that to satisfy the previous objective the user need not have a formal program procedure, whereas this is required to satisfy objective 3.

Again, an information-rich graphical user interface should appear and be a transparent interface between a user designed program and the simulator, acting on the commands received, within the graphical space.

This indirect module should be unobtrusive and not affect the user's program in any way. (ie. it would be ideal if the same program that ran with the hardware robot ran with the simulator, without alteration or recompilation).

### 1.2.4 Objective 4 - Robot specifications

Ideally, the simulator should be as device-independent as possible, allowing for the future injection of extra modules, for purposes such as simulating different types of robots, and for more manipulation tools.

To this end the simulator should be configurable, allowing a number of different robots to be implemented by simply providing a new set of design rules and 3D object modules; design rules are a set of rules
that the simulator will use in order to precisely mimic the behaviour of the device in question.

For the purpose of this project this module would be constrained to one robot (the RTX robot), but the framework should still be present for future advancements.

1.2.5 Objective 5 - Robot control

The simulator should be able to use the information entered by the user, either indirectly or immediately, to control the robot arm. For this we require the construction of a robot control system, to interpret the commands and translate them to robot-specific procedures.

The control module should form the most difficult part of the project, with such topics to be covered as path generation, force, acceleration, etc. This module should be tied in quite tightly with the Robot Specifications module, as most of the parameters will be very robot specific.

1.3 The RTX robot arm

Thus far in this introduction we have mentioned the word 'RTX' and the phrase 'robot arm' a number of times, without actually elaborating on the terms.

The RTX robot arm is a stationary robot constructed by Universal Machine Intelligence Limited (UMI), operating out of London, England. Figure 1.3, below, is a diagrammatical representation of the RTX robot arm.
The following sub-sections will help to explain, in simple terms, the background needed to understand the later sections. Most of the following information can be found in some form or another in the manual supplied with the RTX robot, entitled *Introducing RTX*, ©1986 Universal Machine Intelligence Limited.

It should be noted that we have included the set of cartesian axes (X,Y,Z), which we will be referring to throughout this document, and which was introduced earlier in this introduction.

### 1.3.1 Introducing the RTX

The RTX robot, put simply, is a robot arm attached to a stationary base with the ability to be programmed and, henceforth, controlled from an external source, such as a personal computer.
The connection from the RTX to something like a personal computer is achieved through the utilisation of an RS232 serial cable link. RS232 communications require knowledge in the field of serial (or COM) port programming and protocols, and will be covered in later sections. It is sufficient, for now, to just realise the implications of such a link.

As a package, the RTX robot is accompanied with a suite of programming tools to allow users to program its internal registers and, hence, manipulate the robot arm. These tools appear as a set of programming libraries focussing on the PC-compatible languages, PASCAL, FORTH and C. (P'PC-compatibility can be taken to mean any personal computer compatible with IBM®'s original XT model of the 1980's or, more recently, any personal computer capable of supporting Windows™ 95).

3.3.2 RTX components

Referring back to Figure 1.3 we can notice that the RTX consists of seven (7) interdependent parts, or links; the point at which one link connects to another link is termed a joint.

The RTX robot can exist in one of two states: the initialized position and the non-initialized position. The descriptions of the components rely on the robot being in its initialized position. This position is that demonstrated by the robot in Figure 1.3.

The links that go into making up the RTX robot arm are:

**Gripper** - The gripper is constructed from two 'jaws', fitted with free-moving tips on the ends, to allow for subtle changes in shapes of objects to be grabbed. It contains a servo assembly allowing control of position, force and speed; the two jaws are mounted on a threaded rod which is driven by a motor, which rotates in one direction to open the jaws, and in the opposite direction to close the jaws. This is translation along the Y axis.

**Wrist** - The wrist unit contains two motors entitled wrist1 and wrist2, allowing for motions such as pitch and roll; pitch refers to rotation about the Y axis, while roll refers to rotation about the X axis. These two motors are inter-reliant, in that altering the state of one will affect the state of the other. This can be seen from the formula to calculate pitch and roll...
Equation 1.3.2
Wrist Equation.

\[
\text{Pitch} = \frac{\text{wrist1} + \text{wrist2} \times k}{2}
\]

\[
\text{Roll} = \frac{\text{wrist1} - \text{wrist2} \times k}{2}
\]

\(k = \text{converter constant}\)

**Lower arm** - At the end of the lower arm is a motor called the **yaw** motor. The yaw motor allows the wrist and gripper units to be rotated about the Z axis.

**Upper arm** - At the end of the upper arm is a motor called the **elbow** motor. This motor is used to rotate the lower arm, wrist and gripper units about the Z axis.

**Shoulder** - The shoulder unit consists of a motor used to rotate the upper arm, lower arm, wrist and gripper units about the Z axis.

**ZED (or column)** - This consists of an arrangement of pulleys to allow the up and down movement of the entire assembly; that being the shoulder, upper arm, lower arm, wrist and gripper units. This is translation along the Z axis.

**Base** - The base is, by its very nature, the base of all operations. Normally, the base is taken to be cartesian coordinate (0,0,0).

### 1.3.3 RTX coding

In total, the RTX robot arm utilises seven precision motors to perform all its tasks. These motors are fitted with two-phase optical increment encoders, which allow software to reliably know how far each motor has moved, and in what direction.

The value of the encoders at any time is stored in an internal register, which can be viewed and/or changed through software; this forms the basis of all programming operations on the robot. The values associated with the encoders are termed **encoder counts**, and appear in an initialized RTX robot with the following values...
On their own, these values mean very little to the programmer, but by the use of a conversion table, can be converted to real-world measurements, in degrees and millimetres. The conversions are...

<table>
<thead>
<tr>
<th>motor</th>
<th>displacement</th>
<th>encoder counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>zed</td>
<td>1 millimetre</td>
<td>3.74953</td>
</tr>
<tr>
<td>shoulder</td>
<td>1 degree</td>
<td>29.2227</td>
</tr>
<tr>
<td>elbow</td>
<td>1 degree</td>
<td>14.6113</td>
</tr>
<tr>
<td>yaw</td>
<td>1 degree</td>
<td>9.73994</td>
</tr>
<tr>
<td>wrist1</td>
<td>1 degree</td>
<td>13.4862</td>
</tr>
<tr>
<td>wrist2</td>
<td>1 degree</td>
<td>13.4862</td>
</tr>
<tr>
<td>gripper</td>
<td>1 millimetre</td>
<td>(0.0584*count)+(10.7*10^{-6}*count^2)</td>
</tr>
</tbody>
</table>

Therefore, from the initialized position, the RTX robot has the following range of movement...

<table>
<thead>
<tr>
<th>axis</th>
<th>encoder count range</th>
<th>'real' range</th>
<th>counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>zed</td>
<td>0&gt;X&gt;-3554</td>
<td>948 mm</td>
<td>3554</td>
</tr>
<tr>
<td>shoulder</td>
<td>2630&gt;X&gt;-2630</td>
<td>180 degrees</td>
<td>6260</td>
</tr>
<tr>
<td>elbow</td>
<td>2206&gt;X&gt;-2630</td>
<td>331 degrees</td>
<td>4836</td>
</tr>
<tr>
<td>yaw</td>
<td>1071&gt;X&gt;-1071</td>
<td>220 degrees</td>
<td>2142</td>
</tr>
<tr>
<td>pitch</td>
<td>w1+w2=108&gt;X&gt;-2642</td>
<td>102 degrees</td>
<td>2750</td>
</tr>
<tr>
<td>roll</td>
<td>w1-w2=4882&gt;X&gt;-3560</td>
<td>313 degrees</td>
<td>8442</td>
</tr>
<tr>
<td>gripper</td>
<td>1200&gt;X&gt;-30</td>
<td>90 mm</td>
<td>1200</td>
</tr>
</tbody>
</table>

As a peculiarity of the RTX robot, we must add the value of the elbow-divided-by-three to either side of the inequality for the yaw encoder count range, to compensate for the fact that the yaw motor will always attempt to keep the back of the wrist pointing to the line (0,0,Z).

Another peculiarity of the robot is in the way it utilizes two (2) Intelligent Peripheral (IP) boards to control the motors; IP0 and IP1. At any time the current IP can be either of the two, so a potential programmer must take that into account when beginning a project. All of the above motors are controlled by IP1, with the exception of 'wrist pitch' and 'wrist roll'.
The range of registers for each motor, which can be read and/or written are:

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR</td>
<td>Difference between CURRENT POSITION and position robot is currently at.</td>
</tr>
<tr>
<td>CURRENT POSITION</td>
<td>The position of the arm at time of request.</td>
</tr>
<tr>
<td>ERROR LIMIT</td>
<td>Defines how far the motor can lag behind the position demanded.</td>
</tr>
<tr>
<td>NEW POSITION</td>
<td>The position to be moved to.</td>
</tr>
<tr>
<td>SPEED</td>
<td>Speed of motor movement</td>
</tr>
<tr>
<td>KP, KI, KD</td>
<td>Constant control parameters.</td>
</tr>
<tr>
<td>DEADBAND</td>
<td>A small range in which the RTX robot will assume it has reached the desired position.</td>
</tr>
<tr>
<td>OFFSET</td>
<td>Bias added to compensate for friction in starting the motor moving.</td>
</tr>
<tr>
<td>MAX FORCE</td>
<td>Maximum force applied by motor.</td>
</tr>
<tr>
<td>CURRENT FORCE</td>
<td>The current force applied by motor.</td>
</tr>
<tr>
<td>ACCEL TIME</td>
<td>Time needed for motor to accelerate to the desired speed.</td>
</tr>
</tbody>
</table>
USER RAM  
(EMPTY). Available for use.

USER IO  
Send signals to spare pins.

ACTUAL POSITION  
The actual position of the robot.

1.3.4 RTX program level

Regardless of the programming language used to program the RTX robot, there are a number of procedures which remain constant. These are identified by the following keywords, together with a short description stating their purpose; the descriptions are kept as short as possible, as they will be further explained in later sections.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM_RESTART</td>
<td>Resets all parameters, and performs an immediate stop.</td>
</tr>
<tr>
<td>ARM_INIT_COMMS</td>
<td>Initialises communications from robot to computer.</td>
</tr>
<tr>
<td>ARM_DEFINE_ORIGIN</td>
<td>Make present position the home position.</td>
</tr>
<tr>
<td>ARM_VERSION</td>
<td>Return library version and revision numbers.</td>
</tr>
<tr>
<td>ARM_SET_MODE</td>
<td>Alter the current motor mode.</td>
</tr>
<tr>
<td>ARM_READ</td>
<td>Read an internal RTX register.</td>
</tr>
<tr>
<td>ARM_WRITE</td>
<td>Write to an internal RTX register.</td>
</tr>
<tr>
<td>ARM_GO</td>
<td>Set one or more motors moving.</td>
</tr>
<tr>
<td>ARM_INTERPOLATE</td>
<td>Move each motor by an increment passed from an array.</td>
</tr>
<tr>
<td>ARM_GENERAL_STATUS</td>
<td>Check current status of robot.</td>
</tr>
<tr>
<td>ARM_MOTOR_STATUS</td>
<td>Check current status of motor.</td>
</tr>
<tr>
<td>ARM_STOP</td>
<td>Emergency stop.</td>
</tr>
<tr>
<td>ARM_RAW_COMMAND</td>
<td>Send a raw command to robot.</td>
</tr>
</tbody>
</table>

Table 1.3.4 RTX Keywords.
Receive a raw response from robot.
Having introduced the objectives we are now left with the task of searching for relevant information for either ideas or, perhaps, find that there already exists an application that meets our requirements.

When searching for resources the internet always makes a good starting point. Many universities listed on the internet are undertaking some kind of robotics course, with many of those undertaking robot simulation projects.

A selection of robot simulator programs, together with brief explanations follow.

2.1 Current simulator packages

• From the University of Illinois comes an industrial robot simulator (no title).

This simulator is described by the author as a **General Purpose Simulator for Robot Manipulator Kinematics and Visualisation**

The simulator is aimed more precisely at addressing the concept of neural network control methods, although it does allow the visualisation of the robot in 3D space. This simulator is designed primarily as a socket device, whereby a number of simulators are running simultaneously and communicating with each other.

• From Luiz Felipe Rudge Encarnacao comes a robot simulator entitled **Simulador de Robo**.

This simulator runs under Windows™ 3.1 and Windows™ 95 and consists of a two window interface, with one window (main window) displaying the different camera views and allowing commands to be sent to the robot and the other window tells the robot how to catch objects. This simulator is really a visual tool for controlling a robot on screen, as it grabs and moves on-screen objects.

• Another simulator is a commercial simulator called **Workspace**.

As the author says, **WORKSPACE is a software package which will create and simulate robot programmes in the native language of the robot.**
This simulator package can be regarded as a high-end package, allowing the full range of 3D video displays and even allows the user to design their own robots, by using the built-in CAD tools.

One problem is the fact that the user's program must be run from inside the simulator and cannot be transferred to something like DOS. Also, the cost is not mentioned, but we can gather a very steep price!

- The next simulator we will be looking at is a robot simulator developed at Melbourne University, Australia, by Andrew Conway and Craig Dillon.

This simulator is a graphical robot simulator that attempts discrete time simulation of an arbitrary linked robot arm, with full kinematics and dynamics. There is a discrete-time controller and a standard C interface so that users can create and test different controller algorithms.

As the authors then go on to say, The robot simulator currently only works on SGI machines, and in a no-graphics mode on other machines.

- Yet another robot simulator is the one called EASY-ROB, by Stefan Anton.

This simulator is another commercial package comprising of a robot simulation tool, with 3D graphics and animations.

Like the Workspace simulator, this package allows the user to create simple objects to be manipulated by the robot on-screen. Also, allows world manipulation, such as zooming and translations, etc.

Again, the author has conveniently left out the cost of the package. Through research it was found that the lowest priced robot simulator started at approximately $500US, for the entry level software and a little less for upgrades.

2.2 Development packages

Having had quite a bit of experience in programming in the Windows™ 95 environment and its companion graphics library DirectX™, this will be the only development package we will look into.

It may be said that this is narrow-minded, but in the growing area of games and high-end multimedia development based around the Windows™ 95
architecture, most software houses use DirectX™ to perform their graphics functions. A simple look at the back of computer game boxes will reveal this fact.

Talking about computer games is not as irrelevant as it may sound, as more and more often nowadays, computer game manufacturers are the driving force behind computer hardware advancement. Always trying to get the very best performance, software houses are more often than not at the very edge of extracting performance. Therefore, it can be seen as a sign of great confidence in the DirectX™ package that these software houses continue with it.

Another good point of DirectX™ is that it is actively supported by Microsoft, (current version is 5.0) and is freely available, with any code produced with it, even the example programs supplied with DirectX™, being royalty-free!

DirectX™ requires the use of C++ to program in.

### 2.3 Pros and cons

Rather disappointingly, none of the robot simulator packages that we looked into could perform the objectives we set for ourselves. Most packages were too robot-specific or not powerful enough.

The closest package to our objectives was the Workspace package, but it also suffers from the fact that programs must be developed inside its interface and also executed from there. Grouped together with its inevitable high cost and it becomes a very poor choice for the struggling student.

Another factor is that none of the software simulators have the ability of simulating a user program that can be executed from their program or, the very same program, from the robot. That is, one of our objectives was to have a system whereby the same program that was written for the simulator could run on the real robot without recompilation, thus allowing the users to continue programming in PASCAL.

At this stage it seems the only way to fully realise our objectives, as is often the case, is to design our own simulator.

My background in Windows™ 95, DirectX™ and C++ programming is definitely an advantage and it is this choice that we should be focussing our attentions on.
3.0 Compendium

In this section we begin to discuss the objectives presented earlier, in the introduction. We begin with a brief introduction into 3D theory and follow up with a look into various facets of the project objectives.

Refreshing our memories, the project objectives are:

- A graphical display
- Immediate motion
- Indirect motion
- Robot specifications
- Robot control

We can envisage the required objectives as an hierarchy of modules, which can be arranged in a fashion represented by the following diagram.

As we can see from Figure 3.0 the graphical display can be interpreted as the final result we are attempting to reach. Ultimately, all modules directly or indirectly perform solely for the purpose of creating and translating information, such that it can be displayed to the user.

Adhering to the mission objectives we can see how the following statements hold true for Figure 3.0.

**Immediate motion** - This module allows the user to manipulate the robot in realtime, involving two-way communication with the robot control module and, from there, displaying the updated robot and, from the immediate motion module, displaying the various display parameters, such as panning, banking, etc.

**Indirect motion** - This module also involves two-way communication with the robot control module, allowing the user’s
program to control the motion of the robot, which the robot control module promptly displays, with the indirect motion module displaying the updated robot registers.

**Robot specifications** - This module introduces the design rules for the robot and so is tied in quite tightly with the robot control module, which acts only on these rules.

**Robot control** - This module is the centre of all robot operations, so for the robot to be displayed it must have first passed through the robot control module.

**Graphical display** - The module which represents the end of all operations. It's position at the top of the diagram symbolises how all our efforts ultimately end in a display of some kind.

Figure 3.0 shows how the arrows are only one-sided towards the graphical display module. Therefore we can, in theory, omit the graphical display module altogether. This conflicts with our above arguments in which we have said that the graphical display module is the ultimate result of all operations. We will, therefore, disregard this situation, as it is application specific and can’t be tested in the context of this project.

This section serves to form a basis for the later sections in which we actually begin to code the solution. Descriptions are to be as general as allowable, to permit cross-platform implementations of the methods, but still require some knowledge of matrix math and general computer data-types.

### 3.1 Theory in 3D

Before we can begin to code the solution we must have a strong knowledge base in which to work with. As the project centres almost entirely on the ability to display and manipulate 3-dimensional graphics in 3-dimensional space, it is useful to take a glimpse at what this actually means; we will be introducing and discussing the theory necessary to produce a functional 3 dimensional manipulation library, for robot manipulation, which involves ideas already introduced and new concepts, such as transformations and inverses.

Earlier in this document we introduced the concept of the cartesian axes, (X,Y,Z), for the purpose of defining *translation, rotation* and *scaling*. We will now take a look at what these actually mean.
3.1.1 The cartesian coordinate system

Firstly, we must introduce the concept of the frame. A frame can be defined as a set of four (4) vectors defining the position and orientation of a point, or set of points, in free space.

The concept of the frame arises from the fact that each point in space can be defined entirely by the coordinate pair of position and orientation. We will be discussing the position axis (X, Y, Z) and the rotation axis (H, P, B), which we have not met yet, but can be tentatively defined as (Heading, Pitch, Bank), or alternatively (Y, P, R), which is (Yaw, Pitch, Roll).

Each frame consists of a cartesian axis existing in free space as its own entity, to which the frame's included points are referenced to. This can be better explained by the following diagram:

![Cartesian Axis](image)

In Figure 3.1.1 the cartesian axis belongs to the frame A, denoted by \{A\}. There exists a point in \{A\} which can be referenced by a unique coordinate, (x', y', z'), and defines the position of the point with respect to \{A\}. This, however, does not describe the orientation of the point; that is, we do not know how the point is rotated about any axis. For example, if there was a box positioned at (x', y', z'), how do we know how the box is oriented? The following diagrams illustrates this example:

![Frame {A}](image)
Figure 3.1.1.1 and Figure 3.1.1.2 both represent the same position of the box in question, however, they have different orientation; that is, the box in Figure 3.1.1.2 has been rotated about the point \((x', y', z')\) by -30 degrees.

By introducing the coordinates yaw, pitch and roll we are able to distinguish between the two boxes. As a convention, we can describe yaw (heading) as rotation about the Z axis, pitch as rotation about the Y axis, and roll (banking) as rotation about the X axis. (Throughout this document we will be alternately referring to yaw and heading, and roll and banking. Please note that these terms are interchangeable, and refer to the same property). These axes are described diagrammatically below:

Figure 3.1.1.3 refers to **yaw**, Figure 3.1.1.4 refers to **pitch** and Figure 3.1.1.5 refers to **roll**. These axes, together with the \((X, Y, Z)\) axes allow, what is called, 6 degrees of freedom.
3.1.2 Translation in 3D

As mentioned earlier in this document, translation refers to motion along a single axis, or multiple axes. The following diagram illustrates the process of translation.

![Figure 3.1.1.6](image)

(The symbol denotes a unit vector)

The focus of our attention in Figure 3.1.1.6 is the point P, which can be interpreted as either a 3D translation from frame A, or a 3D translation from frame B. That is, the position of point P can be viewed as the translation from frame B to point P, which is represented by the vector B_p, or the translation from frame A to point P, which is represented by the vector A_p. This is what is regarded as mapping, and allows the position of a point, relative to one frame, to be calculated relative to another frame.

For example, Figure 3.1.1.6 shows a point P, with position vector B_p, relative to frame B, and we wish to know P’s position relative to frame A. We therefore construct another vector A_{pB0}, which is the position of frame B relative to frame A. Consequently, the resultant vector A_p represents the vector sum of B_p and A_{pB0}.

That is,

\[ A_p = B_p + A_{pB0} \]

3.1.3 Rotation in 3D

The following diagram represents the rotation of a frame B, relative to a frame A by some arbitrary angle (a_1, a_2, a_3), where a_1 is the yaw, a_2 is the pitch, and a_3 is the roll.
If we take the point P, then its orientation is governed by the vector $\mathbf{A}_p$, relative to frame A, and the vector $\mathbf{B}_p$, relative to frame B. The two frames are again constructed here for the purpose of mapping, and so given frame B, and the vector $\mathbf{B}_p$, we can, by mapping, find the vector $\mathbf{A}_p$. It should be noted that both frame A and frame B share a common origin, with $\mathbf{A}_p$ and $\mathbf{B}_p$ occurring collinearly.

Using vector methods we can construct the vector $\mathbf{A}_p$ by first finding its component vectors, which follow the X, Y and Z axes. These vectors are labelled $\mathbf{A}_{px}$, $\mathbf{A}_{py}$ and $\mathbf{A}_{pz}$, and are calculated by performing a vector dot product between the unit vectors of A, in frame B, and the resultant vector $\mathbf{B}_p$.

That is,

$\mathbf{A}_{px} = {^B}X'_A \cdot \mathbf{B}_p$

$\mathbf{A}_{py} = {^B}Y'_A \cdot \mathbf{B}_p$

$\mathbf{A}_{pz} = {^B}Z'_A \cdot \mathbf{B}_p$

$(X', Y', Z')$ are unit vectors.

Therefore, we can extract the rotation matrix $^{A}_B\mathbf{R}$.

That is,

$^{A}_B\mathbf{R} = 
\begin{bmatrix}
{^B}X'_A^T \\
{^B}Y'_A^T \\
{^B}Z'_A^T
\end{bmatrix}$

... where $\mathbf{M}^T$ is the transpose of $\mathbf{M}$.

Finally, we end up with the result for $\mathbf{A}_p$ being,

$\mathbf{A}_p = ^{A}_B\mathbf{R} \mathbf{B}_p$. 
3.1.4 Scaling in 3D

Scaling in 3 dimensions is accomplished by simply multiplying the position vector by some constant.

That is, if we have a point \( P \) at \( (X, Y, Z) \) and we introduce a scaling factor \( k \), then the resultant vector is \( k(X, Y, Z) \).

\[
\text{... or,} \quad (X', Y', Z') = (kX, kY, kZ)
\]

3.1.5 Transformations in 3D

When it comes to plotting a 3 dimensional movement (change in position or orientation) we require a method to efficiently and concisely perform the task. For this we employ the method of 3D transformations.

A 3D movement can be viewed as a string of single movements multiplied in sequence, to result in the total movement; any movement in 3D space can be seen to be made up of individual translations and rotations, and are represented by the transform matrix.

The transform matrix takes the general form of:

\[
\begin{pmatrix}
  r_{11} & r_{12} & r_{13} & q_x \\
  r_{21} & r_{22} & r_{23} & q_y \\
  r_{31} & r_{32} & r_{33} & q_z \\
  0 & 0 & 0 & 1
\end{pmatrix}
\]

, and can be customised for either translation, rotation, or both. The 'sub-vector' bounded by \( r_{11} \) to \( r_{33} \) is known as the rotation matrix, \( R \), while the \( q \) vector is the translation.

When we are only interested in translation the transform matrix takes the form,

\[
\begin{pmatrix}
  1 & 0 & 0 & q_x \\
  0 & 1 & 0 & q_y \\
  0 & 0 & 1 & q_z \\
  0 & 0 & 0 & 1
\end{pmatrix}
\]

, where \( (q_x, q_y, q_z) \) represents the translation in 3 dimensions.

The transform matrices for rotations about \( X, Y \) and \( Z \) are:
Equation 3.1.5.2
Rotation About X.

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Rotation about X

Equation 3.1.5.3
Rotation About Y.

\[
\begin{bmatrix}
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Rotation about Y

Equation 3.1.5.4
Rotation About Z.

\[
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 & 0 \\
\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Rotation about Z,

where the angle \( \theta \) represents the amount to rotate about the axis.

A complete transformation involves the matrix multiplication of one, or a number, of matrices, in reverse order. For example, to rotate about the Z axis some angle and then translate along the X axis some distance, we would first multiply our position vector by the translation matrix and then by the rotation matrix, to get the desired end matrix.

### 3.1.6 Inverting transforms

Sometimes we may wish to go backwards and find a point’s position and orientation after a transformation has taken place. That is, if we have transformed a point from frame A to frame B, we end up with a transformation of B with respect to A, \(^B^A T\). To find the matrix of A with respect to B, \(^A^B T\), we just invert the transform.

To find the inverse of the transform it is simply a matter of constructing the inverse transform matrix and multiplying it with the original transform matrix.

Therefore, the inverse transform matrix becomes,
\[ \mathbf{T} = \mathbf{T}^{-1} = \begin{bmatrix} \mathbf{R}^T & -\mathbf{R}^T \mathbf{A}_{\text{world}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

where \( \mathbf{R} \) is the rotation matrix, defined earlier.

Now, this is as far as we need to go in our look into 3D theory, for as we will see in a later section there are a number of graphic programming libraries to do most of the tedious work and are available for a number of different programming languages.

### 3.2 The graphical display

When we refer to a graphical display we are, of course, referring to a computer screen, which is made up of a large array of dots, or pixels. We have to keep this in mind when thinking about 3D objects, because in mathematics any 3D object is constructed of points, lines and polygons, which can be expressed as mathematical expressions, while in computer terms the mathematical expression must first be converted into an array of dots and then plotted to screen.

Globally, we can look at the graphical display module as its own entity encompassing the following functions:

1. Initialising the display
2. Setting the screen resolution and colour mode
3. Setting the render quality
4. Loading in the world objects
5. Loading in the world textures
6. Loading in the composite effects
7. Creating the scene
8. Rendering the scene
9. Translating an object
10. Rotating an object
11. Scaling an object

This list of functions can also be viewed as a function hierarchy, diagrammatically represented as follows...
Notice that there are a number of entry and exit points; each entry point has at least one corresponding exit point, designated by a unique number. The logic behind this structure is as follows...

**Entry/Exit 1**: This is the initialisation function, which requires that all steps be taken in order to ensure a completely initialised display.

**Entry/Exit 2**: This allows the user to alter, at a later date, the screen resolution, colour mode and render quality without having to completely initialise the screen, and thus being able to keep the current screen contents intact.

**Entry/Exit 3**: This exists so that any object in the scene can be rendered to the screen at some constant frame rate. It is required that the rest of the objects remain unchanged, to maintain consistency and so it would be ridiculous to initialise the scene after every frame.

**Entry/Exit 4**: Allows the updating of the various coordinates for any object, without having to initialise, or render the scene. This permits the updating of the object coordinates in a more analog fashion and only plotting to the screen at some predefined time interval.

We will now take a look at the functions, above, by hierarchy.
3.2.1 Screen

Resolution - The resolution refers to the number of pixels (dots) that can be displayed on the screen. This becomes an important factor in determining how accurate the representation of the 3D object is to become. That is, the higher the resolution the more accurate the display of the object. However, the higher the resolution, the greater the number of pixels and, hence, the greater the processing power needed to plot the object onto screen; resulting in a decrease in system performance (slower screen updates).

Colour mode - Creating a 3D scene requires not only the setting of the screen resolution, but also the number of colours involved; that is, the colour mode(!). Like resolution, the greater the number of colours, the more realistic the object will look on screen. Unlike resolution, however, altering the number of colours will not make the object more accurate, only appear more accurate.

For simple displays a small number of colours is needed, but for adding effects, such as lighting and shading effects, more colours are required to construct the necessary palette.

Render quality - The render quality is not directly linked to the previous two, as its property is not dependent on the number of dots or colours on screen, but rather the amount of detail to be displayed. Four common render modes are:

Wireframe - This displays the 3D object as a collection of interlinked lines and represents the simplest form of the 3D object. This is because 3D objects occur naturally as a combination of lines and only appear solid because we have explicitly coloured between the lines.

Solid - Solid objects can be interpreted as wireframe objects which have been filled with a certain colour. In the construction of the object certain bounding lines (lines that join to form a polygon) can be named as surfaces, with each surface having the capability of having a different colour.

Shaded - These are solid 3D objects with a gradient palette. That is, by using lighting effects we can shade the object to...
appear as if it has shadows or to make the perspective appear more realistic.

**Texture mapped** - These are similar to solid objects, except that instead of filling with a colour, a predefined bitmap is plotted over the bounded polygon. Combining texture mapping with shading effects results in the most realistic appearing 3D objects.

### 3.2.2 Load

![Figure 3.2.2 Load Hierarchy.](image)

**Objects** - We require a function to be able to read the representation of a 3D object off disk and convert it to a set of 3D coordinates. As a consequence, we must decide early on on the format of the object file we will be reading. A number of file formats exist and are usually connected very closely to the software company that designed it. For example, `.3DS` and `.MAX` files belong to the Autodesk programs `3DStudio` and `3DStudio Max`.

**Textures** - The textures are usually closely linked to the actual 3D object files, and require loading at the same time the object is being loaded. These can appear as raw bitmaps (no encoding) or as coded bitmaps. They can also occur as a set of lowering-in-resolution bitmaps, which allows faster displaying when scaling the object.

**Composites** - Compositing refers to the procedure of overlaying an object over another object or one or more stationary bitmaps. Here, we are referring to overlaying an object over a stationary bitmap. This serves only to make the appearance of the display more pleasing and serves no functional purpose.
3.2.3 Scene

Create - Creating a scene involves a number of tasks to be performed. These tasks involve setting up the various lights in the scene, initialising the objects, displaying the background bitmaps, initialising all the object parameters, and so forth. Generally, this function serves to prepare everything such that the render function needs only to plot.

Render - This function is the workhorse of the whole module, with its sole purpose of simply plotting all of the objects, bitmaps, textures, lights, etc to screen. The render function requires the most CPU time and so if optimising is to take place this would be the perfect starting point.

3.2.4 Object

Translate - This function should allow the alteration of a specific object's position, preferably in the format (X, Y, Z). No rendering should take place; this follows on for rotate and scale.

Rotate - This function should allow the specified object to be rotated about a specific axis by some given angle.

Scale - This function should allow the specified object to be scaled along one or more axes.

3.3 Immediate motion

Immediate motion, as defined earlier, refers to the situation whereby the user of the final application has the ability to manipulate the simulated robot in realtime, immediately. This leads us to the problem of knowing the position
and orientation of any object at any time. Consequently, we must construct a data structure capable of holding all the possible values each object may have.

For the first time we now introduce the concept of the multi-object robot. What this means is that in the construction of the simulated robot we are left with the predicament of deciding how many objects we need to completely define the robot. As a rule, we could say that we need as many objects as there are links. So, for the RTX robot we would require seven objects. If we focus in on the gripper we notice that it is actually two links at a common joint, so we actually require eight (8) objects.

As we define these links, and subsequent objects, we also need to define the joints. We need to know to which object any object is connected to, for performing tasks such as moving the arm. In other words, if we require to move the shoulder down we first alter the Z value corresponding to the shoulder; what we must keep in mind is that attached to the shoulder is the upperarm, and to that the lower arm, and to that the wrist, etc. So, as we can see, altering one object affects the rest of the objects. Hence, we require something along the lines of a recursive procedure to follow a list of linked objects and update them as necessary.

We can visualise the link objects via the following diagram,

![RTX Link Objects](image)

Taking each joint in turn we can deduce that the joint,

- **column** is stationary,
- **shoulder** is affected by translation in Z,
• **upperarm** is affected by translation in Z and rotation about Z,
• **lowerarm** is affected by translation in Z and rotation about Z,
• **wrist** is affected by translation in Z and rotation about Z,
• **gripper** is affected by translation in Z, rotation about Z, rotation about X and rotation about Y.
• **grip1 and grip2** are affected by translation in X, Y or Z.

We can see that all the possible transformations are used, as a whole; that is, translation along all axes and rotation about all axes. Therefore, we must design a data structure with storage for each object’s (X, Y, Z) coordinate, pitch, bank & yaw values and scaling factors, in X, Y and Z.

Also, we can note that each object is dependent on the previous object. For example, the only reason the upperarm is affected by translation in Z is because it is attached to the shoulder. Therefore, we need to take note of the object that the current object is attached to and so also need some sort of unique identifier to distinguish between objects.

A suitable starting point for such a data structure is:

<table>
<thead>
<tr>
<th>Variable</th>
<th>data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>integer</td>
</tr>
<tr>
<td>IX, IY, IZ</td>
<td>floating point</td>
</tr>
<tr>
<td>Pitch</td>
<td>floating point</td>
</tr>
<tr>
<td>Bank</td>
<td>floating point</td>
</tr>
<tr>
<td>Yaw</td>
<td>floating point</td>
</tr>
<tr>
<td>SX, SY, SZ</td>
<td>floating point</td>
</tr>
<tr>
<td>JX, JY, JZ</td>
<td>floating point</td>
</tr>
<tr>
<td>Link</td>
<td>integer</td>
</tr>
</tbody>
</table>

**Table 3.3**

*Object Data Structure.*

, whereby each defined object has its own structure.

These variables are defined as:

**ID**
- This should be a unique identifier given to the object at time of initialisation, so that the system has some means of distinguishing between objects and so that we have some means of identifying to which object this object should be connected to.

**(IX, IY, IZ)**
- This is the coordinate of the current position of the object in question. Even though the computer display is integer based, we use a floating point number to keep up the accuracy.
Section 3.0 - Page 33

**Pitch** - Holds the current value for the pitch, in degrees or radians.

**Bank** - Holds the current value for the banking, in degrees or radians.

**Yaw** - Holds the current value for the yaw, in degrees or radians.

**\((Sx, Sy, Sz)\)** - These values hold the amount of scaling that is being forced onto the object. This is useful for later calculating distances.

**\((Jx, Jy, Jz)\)** - This is the coordinate of the joint, corresponding to this object. By defining exactly where on this object the joint should be, we are able to link the next object to it.

**Link** - This should be the identifier of the next object this object is connected to. For more advanced designs, this could be an array or dynamic list, to allow linking to more than one object.

We now have some means of addressing our objects, but they are still useless, in this context at least, for the purpose of displaying our virtual robot. What we require now is the construction of another data structure; this time a structure to hold the values associated with the robot.

Originally we stipulated that *immediate motion* allow the user to alter the robot registers immediately and have the virtual robot move on screen. However, with the introduction of *indirect motion* this method will become mostly redundant, with the users being able to alter the registers through programming and with no real advantage to being able to alter the registers on screen.

So, rather than altering the registers, this module exists as a world manipulation module. That is, through this module we allow the user to manipulate the virtual robot, while disregarding the internal registers, which, in fact, remain unchanged throughout. This will allow the user to alter the position and orientation of the robot in 3D space, to examine all angles. For example, the user will be able to examine the back, or underneath, of the robot, while it is actually moving, due to the indirect motion module.

It can be concluded that the necessary structure to perform, and keep a record of, the world manipulations appears as:
**Variable** | **data type**
--- | ---
PanY | floating point
PanZ | floating point
Zoom | floating point
Pitch | floating point
Bank | floating point
Heading | floating point

These variables are defined as:

**PanY**
- This should hold the current Y panning offset, from the centre. That is, the distance to the left or right that the robot has been moved, to result in a pan.

**PanZ**
- This should hold the current Z panning offset, from the centre. That is, the distance up or down that the robot has been moved, in order effect a pan.

**Zoom**
- This holds the zoom factor or, more accurately, the scaling factor. Remember our Cartesian axis and you'll notice that the X axis points towards the screen. Therefore, a zoom is in fact just a pan along the X axis!

**Pitch**
- Contains the angle that the robot has been rotated about the Y axis.

**Bank**
- Contains the angle that the robot has been rotated about the X axis.

**Heading**
- Contains the angle that the robot has been rotated about the Z axis.

Diagrammatically, all the available transformations can be expressed as:

**Figure 3.3.1**
All Transforms.

Pan UP | Pan DOWN | Pan LEFT | Pan RIGHT
--- | --- | --- | ---
Zoom IN | Zoom OUT
--- | ---
Rotate UP | Rotate DOWN | Rotate LEFT | Rotate RIGHT
Note that Rotate UP and Rotate DOWN refer to Pitch, Rotate LEFT and Rotate RIGHT refer to Bank, Pan UP and Pan DOWN refer to PanZ, and Pan LEFT and Pan RIGHT refer to PanY.

3.4 Indirect motion

Indirect motion was originally defined, in the introduction, as allowing the ability to manipulate the simulated robot in an indirect fashion, so as to be able to program it and see the results immediately following.

This module forms the basis of all simulation, in that it exists as the midway between the user’s program and the robot simulator. As such, this module should be able to interpret the user’s program and translate the commands into a form that can be applied to the simulated robot. These commands are, in fact, the RTX functions first presented in the introduction.

Before we can accept functions we must have something to work with. Therefore, we require another data structure to hold all of the available RTX data registers, which were explained in the introduction; the RTX registers are all integer values so our data structure should be the same, and should appear something like...

<table>
<thead>
<tr>
<th>Variable</th>
<th>data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR</td>
<td>integer</td>
</tr>
<tr>
<td>CURRENT_POSITION</td>
<td>integer</td>
</tr>
<tr>
<td>ERROR_LIMIT</td>
<td>integer</td>
</tr>
<tr>
<td>NEW_POSITION</td>
<td>integer</td>
</tr>
<tr>
<td>SPEED</td>
<td>integer</td>
</tr>
<tr>
<td>KP</td>
<td>integer</td>
</tr>
<tr>
<td>KI</td>
<td>integer</td>
</tr>
<tr>
<td>KD</td>
<td>integer</td>
</tr>
<tr>
<td>DEAD_BAND</td>
<td>integer</td>
</tr>
<tr>
<td>OFFSET</td>
<td>integer</td>
</tr>
<tr>
<td>MAX_FORCE</td>
<td>integer</td>
</tr>
<tr>
<td>CURRENT_FORCE</td>
<td>integer</td>
</tr>
<tr>
<td>ACCELERATION_TIME</td>
<td>integer</td>
</tr>
<tr>
<td>USER_RAM</td>
<td>integer</td>
</tr>
<tr>
<td>USER_IO</td>
<td>integer</td>
</tr>
<tr>
<td>ACTUAL_POSITION</td>
<td>integer</td>
</tr>
<tr>
<td>PREVIOUS_POSITION</td>
<td>integer</td>
</tr>
<tr>
<td>MODE</td>
<td>integer</td>
</tr>
<tr>
<td>ALL</td>
<td>integer</td>
</tr>
</tbody>
</table>

Table 3.4
Register Data Structure.

, where the only different variables than were first presented in the introduction are PREVIOUS_POSITION, MODE and ALL. Obviously,
there is always an advantage in knowing the previous position, and the \textit{mode} variable just specifies the motor mode applicable to that particular motor. The \textit{all} variable just points to the end of the data structure. This was inserted to allow the data structure to grow and shrink, as necessary, without affecting the rest of the modules.

The available motor modes are defined in the RTX as:

\begin{itemize}
\item \textbf{FORCE MODE} \textit{- This means that the value in the position register should be interpreted as a force measurement.}
\item \textbf{POSITION MODE} \textit{- This means that the value in the position register refers to the position of the motor.}
\item \textbf{ABSOLUTE MODE} \textit{- Refers to the fact that the value in the position register should be interpreted as an absolute position. That is, using \((0,0,0)\) as the reference.}
\item \textbf{RELATIVE MODE} \textit{- Refers to the fact that the value in the position register should be interpreted as an offset from the current position.}
\item \textbf{USER INPUT} \textit{- The register is being used for user storage.}
\item \textbf{USER OUTPUT} \textit{- Similar to user input.}
\end{itemize}

Of course, each motor will require its own data structure.

As stated in the introduction, the values in the registers represent encoder counts, and not decimal values. Therefore, this module should incorporate the capability of decoding the encoder counts back into decimal values. It should also be noted that these registers have not yet been assigned to any entity; it is this module's task to assign a particular motor register to its respective 3D object. How this form of mapping is performed will be covered in the next section, \textit{Robot Specifications}.

Even if we perform all of the above tasks, we are still left with the problem of how to achieve communication between the user's program and our simulator. To examine this problem we can first take a look at how the driver supplied with the RTX robot (\textit{START.COM}) achieves this.
3.4.1 Examining START.COM

Firstly, anyone who has programmed the RTX robot will recognise that to compile their program they must first link it to a programming library, usually named ARMP.INC. This library’s main function is just to convert the function names, presented in the introduction, to integer values and then pass them on to an interrupt routine; this interrupt routine is set up when START.COM is first executed.

In order to satisfy one of our objectives of allowing the user’s original program to be simulated, without the need for re-compiling, we must find some way of emulating this procedure.

When a PC is first booted, an interrupt table is initialised in RAM; this interrupt table is simply an array of 32-bit pointers to various functions in RAM/ROM. When the START.COM program is first executed, it alters one of these pointers (usually interrupt 78HEX) so that it points to the RAM location of the START.COM program. The program then terminates, leaving itself resident in RAM, so that each time the specified interrupt is called, the START.COM program activates and processes the information passed to it through the various machine registers.

If we simply design our own program to reside in memory and place a pointer to our program in place of the START.COM pointer, we can intercept the interrupt call and process the information ourselves.

Hence, we now possess a method of transferring information to and from our simulator, without the need for a re-compiling of the user program, and somewhere to store the data.

The diagram following serves to sum up the above-mentioned process:

As can be seen from the diagram, the user’s program cannot communicate with the simulator directly, but must first pass through either a DOS and interrupt layer, or just an interrupt layer. This serves to keep the user program a separate entity, oblivious to the fact that it is running on a simulator, rather than the real robot.
3.5 Robot specifications

This was originally defined as the implementation of a number of design rules, whereby these were a set of rules that govern the behaviour and appearance of the simulated robot.

If we enter into a situation similar to that here, where we are required to devise a program to simulate some device, and there exists a number of similar devices, it is good practice to design the program to be as independent of a particular device as is allowable. In other words, why limit ourselves to customising the program solely for the RTX robot, when maybe somewhere down the track it is superseded by a different robot. In this case, the program is now unusable.

However, if we can construct the program to run a set of design rules then, by simply changing the design rules, we can simulate a completely different robot with the same program or, at least, a minimally altered program; there will always be factors that cannot be addressed simply by altering the design rules.

For the objective of implementing a set of design rules for the RTX robot, we can take a look at script filing; scripts have been used for many years as the standard for configuring programs. By introducing a number of keywords we can construct a very rudimentary programming language that the user can edit via a simple text editor, and could contain all the information necessary for the construction of the 3D objects that go to make up the virtual robot.

By looking at the points and objectives introduced so far we can note the following requirements of each 3D object:

1. It must have a unique identifier to distinguish it from the other objects.
2. It must have a coordinate for its position on the screen.
3. It must have a pitch angle.
4. It must have a bank angle.
5. It must have a yaw angle.
6. It must have a scaling factor.
7. It must have a coordinate that specifies where on the object the joint should be.

In addition, the 3D object may be linked to another 3D object, so it must also have a link number; that is, the identifier of the object it is linked to. Also, since this is a script file we also need the name of the file that holds the data needed for reproducing the object on screen.

With the above information we can construct our tentative command set... (This will be explained fully in later sections).
### Table 3.5 Script Command Set

<table>
<thead>
<tr>
<th>Command</th>
<th>parameter</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>&lt;filename&gt;</td>
<td>The name of the object file.</td>
</tr>
<tr>
<td>POSN</td>
<td>&lt;x&gt; &lt;y&gt; &lt;z&gt;</td>
<td>The position of the object.</td>
</tr>
<tr>
<td>PITCH</td>
<td>&lt;radians&gt;</td>
<td>The pitch of the object.</td>
</tr>
<tr>
<td>BANK</td>
<td>&lt;radians&gt;</td>
<td>The banking of the object.</td>
</tr>
<tr>
<td>YAW</td>
<td>&lt;radians&gt;</td>
<td>The yaw of the object.</td>
</tr>
<tr>
<td>SCALE</td>
<td>&lt;x&gt; &lt;y&gt; &lt;z&gt;</td>
<td>The object's scaling factor.</td>
</tr>
<tr>
<td>JOINT</td>
<td>&lt;x&gt; &lt;y&gt; &lt;z&gt;</td>
<td>Where on the object the joint is.</td>
</tr>
<tr>
<td>LINK</td>
<td>&lt;number&gt;</td>
<td>The object that this object is linked to.</td>
</tr>
</tbody>
</table>

One thing that always seems useful in script files is the ability to insert comments. By introducing a character, such as a # character, or a ; character, we can assume that the text following it is a comment and should be ignored.

#### 3.6 Robot control

Robot control, as defined in the introduction, is optimistic at best, when applied to our virtual computer environment. As an ideal, the abstract concepts of force and acceleration can be very important in the real world, and can be modelled using various dynamic techniques. However, in the virtual world, how do we define force? If one user runs the robot simulator on a 100Mhz computer and another on a 200Mhz computer, what happens to the acceleration?

By examining these problems we can also see a potential problem looming for simulating velocity on different machines. Therefore, we must make a difficult decision here on whether our simulator will support the concepts of force and acceleration.

After much thought on the subject we conclude that force is not a plausible option at this time of the simulator's development. Simulating a concept, such as force, on a computer screen seems far too abstract for the average student to comprehend as they enter a robotics course. A number of questions arise, such as how do you represent force, how do we measure force, etc. The concept of force involves far too much Newtonian mathematics; if we are going to include force we must also include the action of gravity and the various bending and distortions that are inevitably occurring inside each joint.
as it moves ... the friction in the motors ... wind resistance ... it's a veritable Pandora's box.

When it comes to acceleration we must look at just how accurate this simulator is to be. We began in designing a simulator to be a visual tool, allowing a student to write and test their robot programs, with no need for a hardware robot; just how relevant is the display of acceleration on the computer screen? We postulate that if incorporated, this will be invisible to the user and will actually aid in slowing down the overall performance of the simulator, for no rewards. It seems useful to allow the simulation of acceleration, but this can be achieved by delay loops; that is, the frame rate starts off at a rate less than the desired rate, and is then increased to the desired rate in a time interval required to achieve the necessary acceleration profile.

Other facets usually associated with a robot control module can be performed via the user's program. For instance, path generation can be implemented using transforms and inverse transforms, and the like.

As is being demonstrated by the arguments above, the robot control module need not be such a daunting task as first thought. If we can construct the other modules to perform at decent levels, it will be up to the user to design their own method for performing unsupported tasks. As a real-life example, the actual RTX robot doesn’t perform path generation ... it relies on the user to program it themselves.

However, the robot control module remains the most important module. It should be the binder between the immediate and indirect motion modules, and the graphical display; whenever the robot is to be plotted it must have been passed through the robot control module first.

How do we actually control the robot?

Take a look at the figure below:
Figure 3.6 represents a typical user program requesting that the robot be moved. It consists of the user program issuing an ARM_GO command to the interrupt layer which, in turn, retransmits the information to our simulator. It is now the simulator's turn to process the data, and so transfers control to the robot control module.

The robot control module must decide which object is to be moved and then obtains the object's current position from the CURRENT_POSITION register and the new position from the NEW_POSITION register. By subtracting the two we know how much to move the object and, depending on the sign of the result, in which direction.

Remembering that we are dealing in the digital domain, the speed of the move must be calculated as a set of increments over a specified time period. That is, we require an expression such as,

\[
\text{maximum speed} \times \text{speed} = \text{number of frames}
\]

where the maximum speed is defined in the RTX, speed is the value in the SPEED register, and k is a constant multiplier. The result of this expression is the number of frames required to plot the movement.

The resolution, k, can be defined as the minimum number of frames permitted in the best case scenario of a speed of maximum speed. This expression should, however, be limited for the case of speed being small. That is, if the value for speed becomes too low, the number of frames becomes extremely high, and so slows down the simulator. We therefore must perform a check such that the speed will always be scaled to some arbitrary minimum value.

The frame rate defines how many frames to be plotted, per second. This is completely arbitrary, although the actual RTX performs a movement every 16 milliseconds, so forms a good base to start with.

After the move, the object's current position register should be updated to reflect its new position.
4.0 Application technology

In this section we will be undertaking a descriptive analysis of existing technologies and their applications. In other words, using the resources first listed in the section entitled Resource literature, we will be first making our decision on our chosen platform and then providing a knowledge base in that chosen platform.

When making a decision regarding the platform we must realise that our choice will have ramifications throughout, and beyond, our simulator program. To make a complete and unbiased decision we must first be clear on what our requirements for a programming platform really are. The most important factors presented so far are:

- The platform must be cheap
- It must be readily accessible
- It must be transparent
- It must be easy to use
- Ideally have a strong support base
- Be unobtrusive

These factors can be expanded as follows...

The platform must be cheap - In order for the simulator to be accepted it must be affordable to the user. That is, if the cost of the package is too high, user’s will not use it.

It must be readily accessible - The platform should be accessible to the user and not in a format that is unrecognisable; it would be advantageous if the user would have already had experience with the chosen platform.

It must be transparent - The user need not know that the simulator is running and should not be held back through factors such as limited hard drive space or RAM.

It must be easy to use - If the platform is easily recognisable to the user they will find it easier to use and will become accepted quicker.

Ideally have a strong support base - Ideally, the platform should be a well-accepted platform, with good support, in terms of complimentary software and upgrade paths.

Be unobtrusive - That is, the platform should not require the user to radically alter the way they use their computer.

With the above factors in mind, we can almost immediately rule out an all-in-one package, such as those presented in earlier sections. These require the user to learn a
new programming language, eliminating one of our main objectives of allowing the user to execute their non-re-compiled code. In order to effectively use these packages, the user would have to completely rewrite all their code. These factors aside, the cost of these packages far outweighs the limited benefit a typical student user would extract from them.

Therefore, we are left with the task of programming our own simulator; something which we have been leaning towards throughout this document. This is by far the best alternative, as we can customise to our own needs and provide a much better interface to the robot as, unlike other packages, we are concerned purely with the RTX robot, so can better serve its needs.

When choosing the platform we must keep in mind that we are dealing with a very graphic-intensive environment, yet we must also keep in mind the user and their requirements. To this end we propose Windows™ 95, helped mainly by its companion development kit, DirectX™ and component Direct3D™, which we will be discussing through to the end of this section. Note also that Windows™ 95 also supports DOS™, should we feel the requirement to utilise the fact. Also, nowadays Windows™ 95 is the entry-level operating system for the majority of the world's computers, so is very accessible.

Now knowing our platform we must assemble our development packages necessary for the development of the simulator.

Windows™ 95, and its associated programming libraries, are programmed to be C++ native. Consequently, we must use a C++ compiler; this is also true of the DirectX™ and Direct3D™ libraries. (From here on, all references to data types and functions will be made as C/C++ data types and functions).

We also require an object design package, to construct our graphic 3D objects, and also a paint package to design the texture maps and background bitmaps.

A resource we may need in the future will be a DOS™ assembler. This may be utilised in the construction of the driver required to mimic the action of the START.COM program.

Therefore, the requirements for our chosen platform can be listed in the following table, together with our chosen packages.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows™ 95</td>
<td>DirectX™/Direct3D™ Microsoft Corporation.</td>
</tr>
<tr>
<td>C++ compiler</td>
<td>Borland C++ 5.0 ©Borland Inc.</td>
</tr>
<tr>
<td>3D object design software</td>
<td>Lightwave 5.0 ©NewTek.</td>
</tr>
</tbody>
</table>
Apart from DirectX™/Direct3D™, the remaining packages were just arbitrary choices based on the resources available to us. They in no way represent some compatibility factor, or bias on my part; that is, for example, Visual C++ (the Microsoft C++ compiler) could be used instead of Borland C++, without hassle.

The remainder of this section will focus entirely on the programming libraries of DirectX™ and Direct3D™, as they play such a major part in the development of the simulator. As we will see, these libraries contain almost all the functions required for 3D manipulation and display.

It should be noted that due to the size and complexity of the DirectX™/Direct3D™ libraries it would be impossible to explain all its functions. As such, we have chosen to focus only on the functions related to our immediate needs.

### 4.1 What is DirectX™?

Before we begin we must point out that this document refers to DirectX™ 5.0.

Following is how Microsoft® describe their programming library:

*The Microsoft® DirectX™ Programmer's Reference provides a finely tuned set of application programming interfaces (APIs) that provide you with the resources you need to design high-performance, real-time applications. DirectX™ technology will help build the next generation of computer games and multimedia applications.*

*Microsoft® developed DirectX™ so that the performance of applications running in the Microsoft® Windows™ operating system can rival or exceed the performance of applications running in the MS-DOS® operating system or on game consoles.*

In fact, this is what DirectX™ actually is. It is a programming interface between a Windows™ program and a graphics layer, whether it be hardware supported or not. DirectX™ has the capability of utilising hardware-accelerated functions, should they be present, or emulating them if not, completely transparent to the user. That is, the user need not specify that they possess an accelerated video card, DirectX™ will automatically detect and use it.
Although DirectX™ contains other APIs that deal with sound and the like, we will only be utilising the API of Direct3D™, for the purpose of this project.

### 4.2 Introducing Direct3D™

Direct3D™ is the 3D programming interface provided by DirectX™, for the purpose of displaying and manipulating 3 dimensional objects, in either real-time or non-real-time situations. Microsoft state the mission of Direct3D™ as...

... to provide device-dependent access to 3-D video-display hardware in a device-independent manner. Simply put, Direct3D™ is a drawing interface for 3-D hardware.

Direct3D™ is further separated into two available modes, which are Immediate mode and Retained mode.

Immediate mode operates as a low level 3D programming interface, between the program and the hardware abstraction layer or, alternatively, the hardware emulation layer, should no video accelerator be installed. Using this mode means that all objects must be constructed in real-time, using geometric primitives.

Retained mode is a high level 3D programming interface, operating above immediate mode. Therefore, when a retained mode function is called, it is first decoded and then passed on to the immediate mode function to execute. This results in a slightly slower function, but at the advantage of not needing so much code to display and manipulate a 3D object. For this reason, we have chosen to use the programming interface of retained mode.

### 4.3 Useful Direct3D™ data types

Following is a list of data types, together with an explanation of their function, grouped together into related categories. These data types are the types provided by Direct3D™, which we will use extensively throughout our coding procedure, and are linked to by placing the line of code,

```c
#include <d3drmwin.h>
```

, into the top of our source file.

Note that with a lot of the data types, just an addition of LP to the front of the name changes the data type into a pointer data type. For example, D3DVECTOR is a structure, while LPD3DVECTOR is a pointer to the D3DVECTOR structure.
4.3.1 General Direct3D™ data types

**D3DVALUE**

This data type forms the basis for all 3D calculations and can be simply modelled by the following C source code.

```c
typedef float D3DVALUE;
```

That is, it is simply a floating point value.

**D3DVECTOR**

This defines a structure used in specifying vectors in 3D space, and appears as,

```c
typedef struct _D3DVECTOR
{
    union
    {
        D3DVALUE x, dvX;
    };
    union
    {
        D3DVALUE y, dvY;
    };
    union
    {
        D3DVALUE z, dvZ;
    }
} D3DVECTOR;
```

Notice that this structure's variables use the D3DVALUE data type introduced above.

4.3.2 DirectDraw™ data types

**LPDIRECTDRAWCLIPPER**

This data type is used to declare a variable which holds the pointer to an IDirectDrawClipper interface.

A clip can be defined as a rectangle which describes the visible areas of the surface. The data type appears as,

```c
typedef struct IDirectDrawClipper FAR* LPDIRECTDRAWCLIPPER;
```

**LPDIRECTDRAW**
This represents a pointer to a valid IDirectDraw object, which was initialised using the DirectDrawCreate interface, to be discussed later on.

Therefore, the data type appears as,

```c
typedef struct IDirectDraw FAR *LPDIRECTDRAW;
```

**LPDIRECT3D**

This data type is used to declare a variable which holds the pointer to an IDirect3D interface. The IDirect3D interface is used to create Direct3D™ objects and set up the environment.

It appears as,

```c
typedef struct IDirect3D *LPDIRECT3D;
```

### 4.3.3 Direct3D™ device data types

**D3DFINDDEVICESEARCH**

This is a structure which defines parameters wanted in the Direct3D™ device. It is best explained by looking at its structure,

```c
typedef struct _D3DFINDDEVICESEARCH
{
    DWORD dwSize;
    DWORD dwFlags;
    BOOL bHardware;
    D3DCOLORMODEL dcmColorModel;
    GUID guid;
    DWORD dwCaps;
    D3DPRIMCAPS dpcPrimCaps;
} D3DFINDDEVICESEARCH;
```

where `dwSize` is the size of the structure,
`dwFlags` is the type of device wanted,
`bHardware` is true for hardware support,
`dcmColorModel` decides on either ramp or RGB colour model,
`guid` is the global unique identifier of device,
`dwCaps` is reserved,
and `dpcPrimCaps` defines the device’s capabilities.

**D3DFINDDEVICESRESULT**
This defines a structure for a device that was found by calling the FindDevice function. This is tightly related to the above structure.

```c
typedef struct _D3DFINDDEVICERESULT
{
    DWORD dwSize;
    GUID guid;
    D3DDEVICEDESC ddHwDesc;
    D3DDEVICEDESC ddSwDesc;
} D3DFINDDEVICERESULT;
```

where `dwSize` is the size of the structure, `guid` is the global unique identifier, and `ddHwDesc` & `ddSwDesc` are structures which describe the hardware and software devices found.

**4.3.4 Retained mode display data types**

**D3DRMCOLORMODEL**

This is an enumerated data type, which serves to identify the current system's colour model. It may take the value of `D3DCOLOR_MONO` to identify ramp emulation, or `D3DCOLOR_RGB` to identify full colour emulation, and appears as,

```c
typedef enum _D3DCOLORMODEL
{
    D3DCOLOR_MONO = 1,
    D3DCOLOR_RGB = 2,
} D3DCOLORMODEL;
```

**D3DRMRENDERQUALITY**

This is a constant value made up of a combination of shading, filling and lighting descriptions for the current device. It exists as,

```c
typedef DWORD D3DRMRENDERQUALITY;
```

**D3DRMVECTOR4D**

This data type is linked almost exclusively to the Transform and Inverse Transform functions, in that it represents the destination coordinates for Transform, and the source coordinates for Inverse Transform.
typedef struct _D3DRMVECTOR4D
{
    D3DVALUE x;
    D3DVALUE y;
    D3DVALUE z;
    D3DVALUE w;
} D3DRMVECTOR4D;

4.3.5 Retained mode data types

LPDIRECT3DRMFRAME

The variable associated with this data type holds the pointer to an IDirect3DRMFrame interface, which is responsible for interacting with frames.

typedef struct IDirect3DRMFrame *LPDIRECT3DRMFRAME;

LPDIRECT3DRM

The variable associated with this data type holds a pointer to a valid Direct3DRM pointer.

typedef struct IDirect3DRMDevice *LPDIRECT3DRMDEVICE;

LPDIRECT3DRMVIEWPORT

The variable associated with this data type holds the address that will be filled with a pointer to an IDirect3DRMViewport interface.

typedef struct IDirect3DRMViewport *LPDIRECT3DRMVIEWPORT;

LPDIRECT3DRMWINDEVICE

The variable associated with this data type holds the pointer to an IDirect3DRMWinDevice interface, which is responsible for activating and updating the current window.

typedef struct IDirect3DRMWinDevice *LPDIRECT3DRMWINDEVICE;
**LPDIRECT3DRMMESHBUILDER**

The variable associated with this data type holds the address that will be filled with a pointer to an IDirect3DRMMeshBuilder interface.

typedef struct IDirect3DRMMeshBuilder *LPDIRECT3DRMMESHBUILDER;

**LPDIRECT3DRMTEXTURE**

The variable associated with this data type holds the address that will be filled with a pointer to an IDirect3DRMTexture interface.

typedef struct IDirect3DRMTexture *LPDIRECT3DRMTEXTURE;

**LPDIRECT3DRMLIGHT**

The variable associated with this data type holds the address that will be filled with a pointer to an IDirect3DRMLight interface.

typedef struct IDirect3DRMLight *LPDIRECT3DRMLIGHT;

**LPDIRECT3DRMMATERIAL**

The variable associated with this data type holds the address that will be filled with a pointer to an IDirect3DRMMaterial interface.

typedef struct IDirect3DRMMaterial *LPDIRECT3DRMMATERIAL;

### 4.4 Useful Direct3D™ functions

Following is a list of functions, together with an explanation of their purpose, grouped together into related Direct3D™ interfaces. These functions are those provided by Direct3D™ for retained mode programming, which we will use extensively throughout our coding procedure, and are linked to by placing the line of code,

```
#include <d3drmwin.h>
```

, into the top of our source file.
4.4.1 IDirect3DRM interface functions

IDirect3DRM->Direct3DRMCreate
(
LPDIRECT3DRM FAR *lp/pD3DRM
);

**Purpose:** Creates a single instance of the Direct3DRM object specified in lp/pD3DRM.

IDirect3DRM->LoadTexture
(
const char *lpFileName,
LPDIRECT3DRMTEXTURE *lpD3DRMTexture
);

**Purpose:** Loads in a texture from the file whose name is given in lpFileName. This texture file can exist in two formats; bitmap file or portable pixmap file. Both file formats may have a resolution of either 8, 24 or 32 bits per pixel. The variable lpD3DRMTexture contains a pointer to a valid Direct3DRMTexture, should the function call succeed.

IDirect3DRM->CreateFrame
(
LPDIRECT3DRMFRAME lpD3DRMFrame,
LPDIRECT3DRMFRAME *lp/pD3DRMFrame
);

**Purpose:** Accepts a parent frame in lpD3DRMFrame and then creates a new child frame of that parent frame, and stores a pointer to it in lp/pD3DRMFrame.

IDirect3DRM->CreateLightRGB
(
D3DRMLIGHTTYPE ltLightType,
D3DVALUE vRed,
D3DVALUE vGreen,
D3DVALUE vBlue,
LPDIRECT3DRMLIGHT *lp/pD3DRMLight
);

**Purpose:** Creates a new light source with RGB colour specified by vRed, vGreen and vBlue. ltLightType holds one of the available lighting modes of AMBIENT, POINT, SPOT, DIRECTIONAL or
PARALLELPOINT. _lpD3DRMLight holds the pointer to the new interface.

_IDirect3DRM->SetSearchPath
{
  LPCSTR lpPath
};

Purpose: This function accepts a string (lpPath) containing one or more directories, separated by semicolons, where Direct3D™ should search when looking for textures.

_IDirect3DRM->CreateMeshBuilder
(
  LPDIRECT3DRMMESHBUILDER *lpD3DRMMeshBuilder
);

Purpose: Creates a new mesh builder object and stores the pointer to the new interface in _lpD3DRMMeshBuilder.

_IDirect3DRM->CreateMaterial
(
  D3DVALUE vPower,
  LPDIRECT3DRMMATERIAL *lpD3DRMMaterial
);

Purpose: Creates a new material with a specular property specified by vPower. The variable _lpD3DRMMaterial holds the pointer to the new interface.

_IDirect3DRM->CreateDeviceFromClipper
(
  LPDIRECTDRAWCLIPPER lpDDClipper,
  LPGUID lpGUID,
  int width,
  int height,
  LPDIRECT3DRMDEVICE *lpD3DRMDevice
);

Purpose: From a clipper object given in _lpDDClipper, creates a Direct3DRM Windows™ device whose interface pointer is stored in _lpD3DRMDevice. The width and height of the device to be created is held in width and height.
**IDirect3DRM->SetDefaultTextureShades**

( 
DWORD dwShades
);

**Purpose**: Sets the number of default shades to be used for any Direct3DTexture object to the number held in `dwShades`.

**IDirect3DRM->SetDefaultTextureColors**

( 
DWORD dwColors
);

**Purpose**: Sets the number of default colours to be used for any Direct3DTexture object to the number held in `dwColors`.

### 4.4.2 IDirectDraw interface functions

**IDirectDraw->DirectDrawCreateClipper**

( 
DWORD dwFlags,
LPDIRECTDRAWCLIPPER FAR *lpDDClipper,
IUnknown FAR *pUnkOuter
);

**Purpose**: This function creates a single instance of another DirectDrawClipper object, which has no associations with the current object, and returns the pointer to the new interface in `lpDDClipper`. `pUnkOuter` must be NULL and `dwFlags` must be 0.

**IDirectDraw->QueryInterface**

( 
REFIID riid,
LPVOID *obp
);

**Purpose**: This function has the ability to determine whether the supplied object supports a particular COM interface, given in `riid`. If it does, then the interface can be used immediately. The variable `obp` points to the new interface.

### 4.4.3 IDirectDrawClipper interface functions

**IDirectDrawClipper->SetHWnd**

(
DWORD dwFlags,
HWND hWnd
);

Purpose: This function allocates the window handle, given in hWnd, to the clip list, to obtain clipping information. The variable dwFlags must be 0.

4.4.4 IDirect3DRMFrame interface functions

IDirect3DRMFrame->GetPosition
(
LPDIRECT3DRMFRAME lpRef,
LPD3DVECTOR lprvPos
);

Purpose: This function retrieves the position of the frame given in lpRef, relative to the current frame, and stores the result as a vector in lprvPos.

IDirect3DRMFrame->SetPosition
(
LPDIRECT3DRMFRAME lpRef,
D3DVALUE rvX,
D3DVALUE rvY,
D3DVALUE rvZ
);

Purpose: This function sets the position of the current frame to a position relative to the frame of reference, given in lpRef. That is, the function positions the current frame at a distance of (rvX, rvY, rvZ) from the origin of the reference frame. It should also be noted that when a child frame is created from within a parent frame, it is placed initially at the default origin position; that being (0, 0, 0) in the parent frame.

IDirect3DRMFrame->Transform
(
D3DVECTOR *lpd3dVDst,
D3DVECTOR *lpd3dVSrc
);

Purpose: Transforms the vector variable given in lpd3dVSrc, which is specified in model coordinates, into world coordinates and returns the result in the variable lpd3dVDst.
*Direct3DRMFrame->InverseTransform*

(D3DVECTOR *lprvDst,
D3DVECTOR *lprvSrc)

**Purpose:** Transforms the vector variable given in *lprvSrc*, which is specified in world coordinates, into model coordinates and returns the result in the variable *lprvDst*.

*Direct3DRMFrame->SetOrientation*

(LPDIRECT3DRMFRAME lpRef,
D3DVALUE rvDX,
D3DVALUE rvDY,
D3DVALUE rvDZ,
D3DVALUE rvUX,
D3DVALUE rvUY,
D3DVALUE rvUZ)

**Purpose:** This function has the effect of aligning the current frame so that its Z direction vector points along the supplied direction vector (*rvDX*, *rvDY*, *rvDZ*) and its Y direction vector points along the supplied direction vector (*rvUX*, *rvUY*, *rvUZ*). All of these transformations are undertaken relative to the frame given in *lpRef*.

*Direct3DRMFrame->SetRotation*

(LPDIRECT3DRMFRAME lpRef,
D3DVALUE rvX,
D3DVALUE rvY,
D3DVALUE rvZ,
D3DVALUE rvTheta)

**Purpose:** This function allows the rotation of the current frame by some angle *rvTheta* relative to the reference frame *lpRef*. The direction vector specified in (*rvX*, *rvY*, *rvZ*), which exists in *lpRef*, defines the vector about which rotation will take place. Motion will take place each time the IDirect3DRMFrame->Move or IDirect3DRMFrame->Tick function is called.

*Direct3DRMFrame->Move*

(}
D3DVALUE delta
);

**Purpose:** This function has the ability to apply the transformations already taken place (those such as SetRotation, above) on the rotations and velocities governed by the current object hierarchy. The *delta* variable defines the actual amount to change the velocity and rotation.

```c
IDirect3DRMFrame->AddScale
(  
D3DRMCOMBINETYPE rctCombine,
D3DVALUE rvX,
D3DVALUE rvY,
D3DVALUE rvZ
);
```

**Purpose:** This function has the effect of scaling the current frame’s local transformation by *(rvX, rvY, rvZ)*. The enumerated data type given in *rctCombine* defines how to combine the new scale with any previous frame transformation, and may take one of the possible values of D3DRMCOMBINE_REPLACE, D3DRMCOMBINE_BEFORE or D3DRMCOMBINE_AFTER.

```c
IDirect3DRMFrame->AddLight
(  
LPDIRECT3DRMLIGHT lpD3DRMLight
);
```

**Purpose:** Adds a light object to the current frame, whose address is held in the variable *lpD3DRMLight*.

```c
IDirect3DRMFrame->SetSceneBackgroundImage
(  
LPDIRECT3DRMTEXTURE lpTexture
);
```

**Purpose:** This function allows the application to specify an image to be the background image for the current scene. The address of the created Direct3DRMTexture object that will contain the new background scene is held in *lpTexture*.

```c
IDirect3DRMFrame->AddVisual
(  
LPDIRECT3DRMVISUAL lpD3DRMVisual
);
```
**Purpose**: Adds a visual object to a frame. The address of the variable that represents the Direct3DRMVisual object to be added to the frame is held in lpD3DRMVisual.

### 4.4.5 IDirect3DRMWInDevice interface functions

**IDirect3DRMWInDevice->HandlePaint**

```c
(HDC hDC);
```

**Purpose**: This function is designed to respond to a Windows™ 95 WM_PAINT system message. By calling the Windows™ 95 function BeginPaint, which should always be called whenever a repaint is commanded for, we obtain the PAINTSTRUCT structure. From this structure we can extract the hDC parameter needed for this function. The purpose of using this function and the above-mentioned Windows™ 95 function is to ensure that whenever a repaint is requested, our application will repaint only those areas defined within the viewports we created on the device.

### 4.4.6 IDirect3DRMViewport interface functions

**IDirect3DRMViewport->Clear**

```c
()
```

**Purpose**: This function has the effect of clearing the current viewport by filling it to the current background colour.

**IDirect3DRMViewport->Render**

```c
(LPDIRECT3DRMFRAME lpD3DRMFrame);
```

**Purpose**: This function represents the workhorse of the Direct3D™ library in that it is given the responsibility of rendering the frame given in lpD3DRMFrame to the current viewport. All visuals and any child frames attached to the supplied frame are rendered.

### 4.4.7 IDirect3DRMDevice interface functions

**IDirect3DRMDevice->Update**
Purpose: This function usually follows the Render function, above, in that it has the task of copying the rendered image to the display. It also said to “provide a heartbeat function to the device driver”.

```
IDirect3DRMDevice->SetShades
(DWORD ulShades);
```

Purpose: When running Direct3D™ in ramp colour mode, calling this function allows the altering of the number of shades residing in the ramp of colours used for shading. The new number of shades is held in `ulShades` and must be a power of 2, with the default being 32.

```
IDirect3DRMDevice->SetDither
(BOOL bDither);
```

Purpose: This function sets or clears the dither flag for the current device. The variable `bDither` can be either TRUE for dithering on or FALSE for dithering off. The default is TRUE.

```
IDirect3DRMDevice->GetWidth
();
```

Purpose: Retrieves the width of the current device, in pixels.

```
IDirect3DRMDevice->GetHeight
();
```

Purpose: Retrieves the height of the current device, in pixels.

4.4.8 IDirect3DRMMeshBuilder interface functions

```
IDirect3DRMMeshBuilder->Load
( LPVOID lpvObjSource,
  LPVOID lpvObjID,
```
D3DRMLOADOPTIONS d3drmLOFlags, 
D3DRMLOADTEXTURECALLBACK d3drmLoadTextureProc, 
LPVOID lpvArg 
);

Purpose: This function loads a Direct3DRMMeshBuilder object from the current IDirect3DRMMeshBuilder interface and is the function we use to load our 3D objects into Direct3D. The name of our file containing our 3D object is passed through lpvObjSource, which can also represent a resource. In the case of multiple objects existing in the one object file, the variable lpvObjID should hold a number corresponding to the required object number in the file.

The d3drmLoadTextureProc variable points to a callback procedure to be called to handle the loading in of associated textures, with a variable pointed to by lpvArg to be passed to it.

The d3drmLOFlags variable should contain one of the constants D3DRMLOAD_FROMFILE, D3DRMLOAD_FROMRESOURCE, D3DRMLOAD_FROMMEMORY, D3DRMLOAD_FROMURL, D3DRMLOAD_BYNAME, D3DRMLOAD_BYPOSITION, D3DRMLOAD_BYGUID, D3DRMLOAD_INSTANCEBYREFERENCE, D3DRMLOAD_INSTANCEBYCOPYING or D3DRMLOADASYNCHRONOUS.

IDirect3DRMMeshBuilder->Scale 
( 
D3DVALUE sx, 
D3DVALUE sy, 
D3DVALUE sz 
); 

Purpose: We use this function to scale the current Direct3DRMMeshBuilder object by the scaling factors given in (sx, sy, sz), in model coordinates.

IDirect3DRMMeshBuilder->SetMaterial 
( 
LPDIRECT3DRMMATERIAL lpIDirect3DRMmaterial 
); 

Purpose: This function sets the material of all the faces of a Direct3DRMMeshBuilder object and applies the interface given in lpIDirect3DRMmaterial to it.
4.4.9 IDirect3D interface functions

IDirect3D->FindDevice
(
LPD3DFINDDEVICESEARCH lpD3DFDS,
LPD3DFINDDEVICERESULT lpD3DFDR
);

Purpose: This function uses a method whereby it finds a device with specified characteristics and retrieves a description of it. The variable structure lpD3DFDS contains the necessary information of the device to be found and the variable structure lpD3DFDR contains information of the device, if found.

IDirect3D->CreateViewport
(
LPDIRECT3DVIEWPORT2 *lpD3DViewport2,
Iunknown *pUnkOuter
);

Purpose: This function creates a Direct3DViewport object, with a pointer to the new interface stored in lpD3DViewport2 and the value of pUnkOuter NULL.
5.0 Implementation

This section deals with the implementation of all the theory and background knowledge accumulated thus far. Using the resources listed earlier we must transform the theoretical into the literal.

Before beginning coding we must remember that we had decided to code the simulator using Windows™ 95 as the platform and C++, with perhaps assembly language, as the programming language of choice. Remember also that we had introduced a whole selection of DirectX™/Direct3D™ library functions to aid us in our venture.

One factor we must point out is that neither this section, nor any other following sections, represents a tutorial into Windows™ 95 programming. We will assume a knowledge of how to perform such tasks as opening windows and message queuing, as they form part of the default Windows™ libraries and are irrelevant to our problem to be solved. As such we will not be discussing those library functions in this implementation section.

Note that accompanying this document is the full source code to the working simulator, which can be compiled and executed as is. This source code is extremely well commented and should be kept in reach whilst reading this section.

We will follow the implementation procedure by once again looking at the project objectives and designing functions to accommodate them. We will also be reproducing quite verbosely code from the source code listing to aid in our explanations and, wherever possible, actual screen dumps from the running simulator.

Please note that in the explanations to come, DirectX™ can refer to DirectX™, Direct3D™ or both. Functions labelled DirectX™ have already been discussed and explained in the section Application Technology, so will not be explained again here.

Explanations of the message queuing variables and data formats can be found in the file TSGlobal.h.

Also, note that in the structure diagrams to follow, variables surrounded by ( ) represent output variables, and variable surrounded by [ ] represent input variables. For example, the function GetX (Y) should be interpreted as the variable Y returned from the call to GetX and the function GetX [Z] should be interpreted as the variable Z used in the function call to GetX.

5.1 A graphical display

All of the functions necessary for the construction of the graphical display can be found in the source files ToySim.h and ToySim.cpp.
The functions which are of relevance to us here are

- FindDevice
- CreateDevice
- CreateScene
- LoadTextures
- AlterRenderQuality
- Render
- Routines found in WindowProc

5.1.1 FindDevice

The FindDevice function is called to return a valid device, based on the desired search criteria, which is, in itself, based on the colour model we are to be using.

As we introduced earlier in this document we have two options when it comes to deciding the colour model, MONO or RGB. That is, a colour model based on a ramp or a colour model based on true RGB.

This function will perform a device search based on the colour model and, if a valid device is found, return its unique ID number to the calling function or NULL otherwise.

Subjects of significance addressed by this function include

**DirectX:**
- DirectDrawCreate,
- LpDD->QueryInterface,
- LpD3D->FindDevice.

![Figure 5.1.1 FindDevice](image)
FUNCTION: FindDevice

INPUTS: D3DCOLORMODEL cm - The colour model specified for searching for a compatible device.

OUTPUTS: LPGUID result.guid - Found identifier for device.

AFFECTED: N/A

PURPOSE: The purpose of this function is to perform a device search such that the found device will be able to handle the requested colour model specified in the variable, cm.

LPGUID FindDevice(D3DCOLORMODEL cm)

LPDIRECTDRAW lpDD; // DirectDraw object pointer.
LPDIRECT3D lpD3D; // Direct3D object pointer.

D3DFINDDEVICESEARCH search; // Holds the search criteria.
static D3DFINDDEVICESEARCHRESULT result; // Holds the result of the search.

RESULT error; // The result obtained from Direct3D calls.

HDC hdc; // The handle to the current device context.
int bpp; // Holds the current bits-per-pixel of the display.

hdc = GetDC(NULL); // Obtain the current device context handle.
bpp = GetDeviceCaps(hdc, BITSPIXEL); // Get the current bits-per-pixel.
ReleaseDC(NULL, hdc); // Release the device context handle.

// Attempt to create a DirectDraw object.
if (DirectDrawCreate(NULL, &lpDD, NULL))
    return NULL;

// Do the same for the Direct3D object.
if (lpDD->QueryInterface(IID_IDirect3D, (void**)&lpD3D))
    lpDD->Release(); // On error, release the object.
    return NULL;

// Initialize the search structure.
memset(&search, 0, sizeof(search); // First clear it.
search.dwSize = sizeof(search); // Save the size of the structure.
search.dwFlags = D3DFDS_COLORMODELSpecify the colour model.
// Which colour model are we interested in?
search.dmColorModel = (cm == D3DCOLOR_MONO) ? D3DCOLOR_MONO : D3DCOLOR_RGB;

// Initialize the result structure.
memset(&result, 0, sizeof(result); // First clear it.
result.dwSize = sizeof(result); // Save the size of the structure.

// Do Direct3D call to search for required device.
error = lpD3D->FindDevice(&search, &result);

if (error == DD_OK)
    // Do search again, this time excluding hardware devices.
    if (result.ddwDesc.dwFlags & (result.ddwDesc.dwDeviceRenderBitDepth & bppToDdbd(bpp))
        search.dwFlags = D3DFDS_HARDWARE;
        search.ddwDesc.dwDeviceRenderBitDepth = FALSE; // No hardware check.
        memset(&result, 0, sizeof(result));
        result.dwSize = sizeof(result);
        error = lpD3D->FindDevice(&search, &result);

lpD3D->Release(); // Release the previously assigned objects.
lpDD->Release();

if (error)
    return NULL; // Return on error.
else
    return &result.guid; // Return on success.
5.1.2 CreateDevice

The CreateDevice function is called after the FindDevice function has found the required device.

This function uses the ID returned by the FindDevice function to create the required Direct3D™ device. That is, initialises the number of colours on screen, sets the quality of the screen, initialises the dither constant and opens a new viewport.

Subjects of significance addressed by this function include

Windows: GetDeviceCaps,

This function is called to obtain the current screen density. That is the bits-per-pixel rating.

DirectX

GetD3D DRV->CreateViewport.

static BOOL CreateDevice(HWND win, ApplicationInfo *info)
{
    RECT r; // Rectangle structure, for screen co-ordinates.
    int bpp; // Storage for the current bits-per-pixel screen value.
    HDC hdc; // Handle of the current device context.

    GetClientRect(win, &r); // Obtain the current window extremes.

    // Based on the above rectangle, create a clipper device.
    if (FAILED(lpD3D DRV->CreateDeviceFromClipper(lpD3D DRV, FindDevice(info->model), r.right, r.bottom, &info->dev)))
        goto generic_error;

    // Just inserting some special fx ...
Section 5.0 - Page 65

info->dev->SetRenderMode(D3DRENDERMODE_BLEND|D3DRENDERMODE_SCIANGRAPHIC);  
if (FAILED(info->dev->QueryInterface(IID_IDirect3DDevice, (IID *)&info->dev)))  
goto generic_error;

hdc = GetDC(win);  // Obtain the current device context handle.
bpp = GetDeviceCaps(hdc, BITSPIXEL);  // Get the current bits-per-pixel.
ReleaseDC(win, hdc);  // Release the device context handle.

// Do action depending on the value of bpp.
switch (bpp)  
{  
case 1:  // 1 bit-per-pixel, MONO.
    if (FAILED(info->dev->SetShades(4)))
        goto generic_error;
    // Set the number of independent shades for textures to 4.
    if (FAILED(lpD3DRM->SetDefaultTextureShades(4)))
        goto generic_error;
    break;

case 16:  // 16 bits-per-pixel, 65536 colours.
    if (FAILED(info->dev->SetShades(16)))
        goto generic_error;
    // Set the number of independent shades for textures to 256.
    if (FAILED(lpD3DRM->SetDefaultTextureColors(256)))
        goto generic_error;
    break;

case 24:  // 24 bits-per-pixel, 16.8 million colours.

case 32:  // 32 bits-per-pixel, 4.3 billion colours.
    if (FAILED(info->dev->SetShades(32)))
        goto generic_error;
    // Here we take 24-bit and 32-bit to be the same. Normally
    // your eyes can't tell the difference between 16.8 million
    // colours and 4.3 billion colours.
    if (FAILED(lpD3DRM->SetDefaultTextureColors(256)))
        goto generic_error;
    break;

default:  // Unknown bpp value.
    if (FAILED(info->dev->SetShades(1)))
        goto generic_error;
    if (FAILED(info->dev->SetDefaultTextureColors(1)))
        goto generic_error;
    break;
}

// Initially start off in wireframe mode.
if (FAILED(info->dev->SetQuality(D3DQUAL_WIREFRAME)))
    goto generic_error;

// Attempt to create the scene.
if (!CreateSceneInfo())
    goto ret_with_error;

// Create a viewport for the scene, using the default camera.
if (FAILED(lpD3DRM->CreateViewport(info->dev, info->camera, 0, 0, info->dev->GetWidth(), info->dev->GetHeight(), &info->view)))
    goto generic_error;

// Set the background.
if (FAILED(info->view->SetBackColor(D3DVAL(5000.0))))
    goto generic_error;

return TRUE;  // All OK!

// Exit immediately.
generic_error:
DisplayMsg("An error occurred while creating the needed Direct3D device.
");

ret_with_error:
return FALSE;  // Error.
}
5.1.3 CreateScene

The CreateScene function is the primary function to call for the creation of the main scene and loading and initialising of the 3D objects.

This function is really 5 different functions in one; it has the tasks of creating the default scene, creating the default frame, creating the pointer frame, loading & initialising the objects and creating the camera frame.

At the time of coding it was seen to be advantageous to have some scheme of determining the coordinates of any point on the current scene. It was decided to introduce a new frame called the pointer frame, which could be manipulated on screen, using the mouse, and return its current position to the main program. Such is the flexibility of DirectX™ that this was achieved quite easily, as will be seen in the function below.

Subjects of significance addressed by this function include

**DirectX:**
- lpD3DRM->SetSearchPath
- info->scene->SetSceneBackgroundImage
- builder->Load

**C++:**

ObjectPointer data structure

```c++
struct _OBJECTPOINTERTYPE{
    // Declares a structure for the robot-tip pointer.
    LPDIRECT3DRMFRAME parent;
    // Direct3D pointer to parent frame.
    double X, Y, Z;
    // The (X,Y,Z) co-ordinate of the pointer.
    typedef _OBJECTPOINTERTYPE _OBJECTPOINTER;
    // Robot-tip pointer type.
}
```

ObjectList data structure

```c++
# define MAXOBJECTS 16
struct _OBJECTTYPE{
    BOOL ACTIVE;
    // True if object initialised
    char NAME[32];
    // The object file, filename.
    double IX, IY, IZ;
    // Starting position of object in x,y,z format
    double PITCH; // Initial pitch angle in degrees
    double BANK; // Initial banking angle in degrees
    double YAW; // Initial yaw angle in degrees
    double SX, SY, SZ;
    // Initial scaling factors for x,y,z planes
    double JX, JY, JZ;
    // Point in object where joint can occur, in x,y,z
    int LINK;
    // Object no. to which this object links to {-1-none}
    typedef _OBJECTTYPE _OBJECTLIST[MAXOBJECTS];
    // List of allowed objects
}
```

ObjOffList data structure
Create Default Scene

CreateFrame(info->scene)

CreateLightRGB[DIRECTIONAL]
(light1)

CreateLightRGB[AMBIENT]
(light2, light3)

CreateFrame[info->scene]
(light)

light->SetPosition[info->scene]

light->AddLight[light1, light3]

info->scene->AddLight[light2]

Create Frames

LoadTexture(DefaultTexture)

info->scene->SetSceneBackground[DefaultTexture]

CreateFrame[X]
(Y)

Y->SetRotation[X]

Y->SetPosition[X]

CreateMeshBuilder(builder)

builder->Load("grid.x")

builder->Scale

CreateMaterial(mat)

builder->SetMaterial[mat]

Y->AddVisual[builder]
/* FUNCTION: CreateScene */
/* INPUTS: ApplicationInfo *info - Our Direct3D scene structure. */
/* OUTPUTS: static bool - FALSE if error occurred. */
/* AFFECTED: ObjectPointer, ObjOffList. */
/* PURPOSE: As the name suggests, this function initializes and creates the */
/* scene specified in the *info variable. It involves creating the */
/* frames, the lights, the colours and the position of the main */
/* scene. It also loads in the background bitmap, the grid, the */
/* pointer, and all of the objects required to build the robot. */
/*==========================================*/

static BOOL CreateScene(ApplicationInfo* info)
{
    LPDIRECT3DFRAME2 frame = NULL; // Declare/init. frame object.
    LPDIRECT3DFRAME2 light = NULL; // Declare/init. light object.
    LPDIRECT3DFRAME2 parent = NULL; // Declare/init. Parent object.
    LPDIRECT3DTEXTURE2 DefaultTexture = NULL; // Declare/init. default texture.
    LPDIRECT3DTEXTURE2 builder = NULL; // Declare/init. builder object.
    LPDIRECT3DLIGHT2 light1 = NULL; // Declare/init. 1st light.
    LPDIRECT3DLIGHT2 light2 = NULL; // Declare/init. 2nd light.
    LPDIRECT3DLIGHT2 light3 = NULL; // Declare/init. 3rd light.
    LPDIRECT3DMATERIAL mat = NULL; // Declare/init. material.

    RESULT rvil; // Result obtained from Direct3D calls.
    int i; // Dummy loop variable.

    // The following represents the steps needed to create the default scene.
    //----------------------------------------------------------------------
    // Attempt to create the default scene.
    if (FAILED(lpD3DRM->CreateFrame(NULL, info->scene)))
        goto generic_error;

    // Attempt to create the 1st light.
    if (FAILED(lpD3DRM->CreateLightRGB(D3DRMLIGHT DIRECTIONAL, D3DVAL(1.0),
                                        D3DVAL(1.0), D3DVAL(1.0), &light1)))
        goto generic_error;

    // Attempt to create the 2nd light.
    if (FAILED(lpD3DRM->CreateLightRGB(D3DRMLIGHT AMBIENT, D3DVAL(0.2),
                                        D3DVAL(0.2), D3DVAL(0.2), &light2)))
        goto generic_error;

    // Attempt to create the 3rd light.
    if (FAILED(lpD3DRM->CreateLightRGB(D3DRMLIGHT AMBIENT, D3DVAL(0.2),
                                        D3DVAL(0.2), D3DVAL(0.2), &light3)))
        goto generic_error;

    // Attempt to create the default light.
    if (FAILED(lpD3DRM->CreateFrame(info->scene, &light)))
        goto generic_error;

    // Set the position of the default scene.
    if (FAILED(light->setPosition(info->scene, D3DVAL(1.0), D3DVAL(2.0),
                                   D3DVAL(5.0))))
        goto generic_error;

    // Set the orientation of the default scene.
    if (FAILED(light->setOrientation(info->scene, D3DVAL(-1.0), D3DVAL(-1.0),
                                       D3DVAL(1.0), D3DVAL(0.0), D3DVAL(0.0), D3DVAL(0.0))))
        goto generic_error;

    // Add the 1st light to the scene.
    if (FAILED(light->addLight(light1)))
        goto generic_error;

    // Add the 3rd light to the scene.
    if (FAILED(light->addLight(light3)))
        goto generic_error;

    // We no longer need the following light objects.
    RELEASE(light1);
    RELEASE(light3);

    // Add the 2nd light to the scene.
    if (FAILED(info->scene->addLight(light2)))
        goto generic_error;

    // We no longer need the following light objects.
    RELEASE(light2);
    RELEASE(light);

    //----------------------------------------------------------------------
}
The following represents the steps needed to create the default frame.

---

Initialize the default texture search paths
lpD3DRM->SetSearchPath(TEXTUREPATH); // This is where Direct3D will look
// when attempting to load textures.

Load the background image.
rval = lpD3DRM->LoadTexture("ToySim.bmp", &DefaultTexture);
if (rval == D3DRM_OK) // Display it as the background.
rval = info->GetScene->SetSceneBackgroundImage(DefaultTexture);
if (rval != D3DRM_OK) // If error occurred, notify user.
    DisplayMsg("Unable to load 'ToySim.bmp.\n\s", TEXTUREPATH,
        ErrorToString(rval));
    goto ret_with_error; // Exit immediately!

Start setting up the default frame...

Create the default frame inside the current scene.
if (FAILED(lpD3DRM->CreateFrame(info->scene, info->defaultframe)))
    goto generic_error;

Initialize the rotation of the frame.
if (FAILED(info->defaultframe->SetRotation(info->scene, D3DVAL(1.0),
D3DVAL(1.0), D3DVAL(1.0), D3DVAL(-0.02)))
    goto generic_error;

Initialize the position of the frame.
if (FAILED(info->defaultframe->SetPosition(info->scene, D3DVAL(0.0),
D3DVAL(0.5), D3DVAL(5.0)))
    goto generic_error;

Obtain the current mesh builder object.
if (FAILED(lpD3DRM->CreateMeshBuilder(builder)))
    goto generic_error;

Load in the 1st 3D object to the frame.

This is the grid that appears underneath the robot.
rval = builder->Load("grid.x", NULL, D3DRM_LOAD_FROMFILE, loadTextures, builder);
if (rval != D3DRM_OK) // If error loading then notify user.
    DisplayMsg("Unable to load 'grid.x.\n\s", ErrorToString(rval));
    goto ret_with_error; // Exit immediately!

Scale this object to fit the current scene.
// Initially this should be a scale of 1:1.
rval = builder->Scale(D3DVAL(1.0), D3DVAL(1.0), D3DVAL(1.0));

Create the associated material.
if (FAILED(lpD3DRM->CreateMaterial(D3DVAL(10.0), &mat)))
    goto generic_error;

Use the mesh builder to set this material.
if (FAILED(builder->SetMaterial(mat)))
    goto generic_error;
RELEASE(mat); // The material object is no longer needed.

Add the visual device, i.e. build the objects.
if (FAILED(info->defaultframe->AddVisual(builder)))
    goto generic_error;

RELEASE(builder); // The mesh builder object is no longer required.

End default frame

---

The following represents the steps needed to create the pointer frame.

---

Initialize the ObjectPointer structure.
ObjectPointer.parent = info->defaultframe; // Save the parent frame.
ObjectPointer.x = -0.05; // Initial X position.
ObjectPointer.y = 0; // Initial Y position.
ObjectPointer.z = 0; // Initial Z position.

Create a frame with the parent equal to the variable, parent.
parent = ObjectPointer.parent;
if (FAILED(lpD3DRM->CreateFrame(parent, info->pointer)))
goto generic_error;

// Set up the rotation for the pointer frame.
if (FAILED(info->pointer->SetRotation(parent, D3DVAL(0.0), D3DVAL(0.0), D3DVAL(0.0))))
goto generic_error;

// Set up the position for the pointer frame.
if (FAILED(info->pointer->SetPosition(parent,D3DVAL(ObjectPointer.Y), D3DVAL(ObjectPointer.Z), D3DVAL(ObjectPointer.X))))
goto generic_error;

// Get builder object to create our 3D object.
if (FAILED(lpD3DRM->CreateMeshBuilder(&builder)))
goto generic_error;

// Actually load in the pointer 3D object.
rvval = builder->Load("pointer.x", NULL, D3DLOAD_PROFILE, loadTextures, NULL);
if (rvval != D3DRM_OK) // Notify user of loading error.
    DisplayMsg("Unable to load pointer.x\nWarning:", ErrorToString(rvval));
goto ret_with_error; // Exit immediately.

// Setup the scaling for the pointer frame.
rvval = builder->Scale(D3DVAL(0.5), D3DVAL(0.5), D3DVAL(0.5));

// Create an associated material.
if (FAILED(builder->CreateMaterial(D3DVAL(10.0), &mat)))
goto generic_error;

// Set the material to the frame.
if (FAILED(builder->SetMaterial(mat)))
goto generic_error;

RELEASE(mat); // The material object is no longer needed.

// Actually build the objects on screen.
if (FAILED(info->pointer->AddVisual(builder)))
goto generic_error;

RELEASE(builder); // The builder object is no longer needed.

// The following represents the steps needed to create the objects needed
to recreate the robot on screen, in 3D, inside the default frame.
// Loop through all possible objects.
for (i = 0; i < MAXOBJECTS; i++)
    if (ObjectList[i].ACTIVE) // Only process if object is valid/active.
        if (ObjectList[i].LINK == -1) // link to defaultframe if no link
            parent = info->defaultframe; // specified.
        else parent = info->frame/ObjectList[i].LINK; // Else link to
            // specified link.

    // Start initialization of new object.
    ObJOffList[i].X = 0; ObJOffList[i].Yinc = 0; ObJOffList[i].XF = 0;
    ObJOffList[i].Y = 0; ObJOffList[i].Yinc = 0; ObJOffList[i].YF = 0;
    ObJOffList[i].Z = 0; ObJOffList[i].Zinc = 0; ObJOffList[i].ZF = 0;
    ObJOffList[i].P = 0; ObJOffList[i].Pinc = 0; ObJOffList[i].PF = 0;
    ObJOffList[i].B = 0; ObJOffList[i].Binc = 0; ObJOffList[i].BF = 0;
    ObJOffList[i].parent = parent;

    // Create a frame for the current object.
    if (FAILED(lpD3DRM->CreateFrame(parent, info->frame[i])))
goto generic_error;

    // Adjust the rotation for the current object.
    if (FAILED(info->frame[i]->SetRotation(parent, D3DVAL(0.0), D3DVAL(0.0), D3DVAL(0.0))))
goto generic_error;

    // Adjust the position for the current object.
    if (FAILED(info->frame[i]->SetPosition(parent, D3DVAL(ObjectList[i].Y), D3DVAL(ObjectList[i].Z), D3DVAL(ObjectList[i].X))))
goto generic_error;
Create a builder object for the 3D object.
if (FAILED(lpD3DRM->CreateMeshBuilder(&builder)))
    goto generic_error;

// Actually load in the object.
rval = builder->load(ObjectList[i].NAME, NULL, D3DMLOAD_FROMFILE,
                       loadTextures, NULL);

if (rval != D3DRM_OK) // Notify the user of any error loading.
    { DisplayMsg("Unable to load %s
\n", ObjectList[i].NAME,
                 ErrorToString(rval));
        goto ret_with_error; // Exit immediately.
    }

// Adjust the scaling for the current object.
rval = builder->Scale(D3DVAL(ObjectList[i].SX),
                       D3DVAL(ObjectList[i].SZ), D3DVAL(ObjectList[i].EX));

// Do if object is pointer.
if (i = MAXOBJECTS-1)
    info->frame[i]->AddScale(D3DMCOMBINE_REPLACE, D3DVAL(0.0),
                              D3DVAL(0.0), D3DVAL(0.0));

// Create an associated material object.
if (FAILED(lpD3DRM->CreateMaterial(D3DVAL(0.0), &mat)))
    goto generic_error;

// Set the material to the object.
if (FAILED(builder->SetMaterial(mat)))
    goto generic_error;

RELEASE(mat); // The material object is no longer needed.

// Actually build the 3D object.
if (FAILED(info->frame[i]->AddVisual(builder)))
    goto generic_error;

RELEASE(builder); // The builder object is no longer needed.

// The following represents the steps needed to create the camera frame.
// -----------------------------------------------
// Create the default camera for the scene.
if (FAILED(lpD3DRM->CreateFrame(info->scene, &info->camera)))
    goto generic_error;

// Set the position for the camera frame.
if (FAILED(info->camera->SetPosition(info->scene, D3DVAL(0.0),
                                     D3DVAL(0.0), D3DVAL(0.0))))
    goto generic_error;

return TRUE; // All OK!

// This is for an immediate exit, due to fatal error!

return FALSE; // Return with error!

5.1.4 LoadTextures

The LoadTextures function is actually a callback function that is called whenever DirectX™ attempts to load in an object with an associated texture map.
At the moment this function does nothing except return the information it received through the callback procedure. In the future this function could be used for translating texture maps from file formats other than the standard .bmp or .ppm formats, which are the only formats currently supported by DirectX™.

```
/*-----------------------------------------------*/
/* FUNCTION: LoadTextures */
/* INPUTS: char *name - Name of texture to load. */
/* LPDIRECT3DMESHESHORDER builder - Mesh builder object. */
/* LPDIRECT3DMESHTEXTURE *tex - Texture object. */
/* OUTPUTS: HRESULT - An open handle to texture. */
/* AFFECTED: N/A */
/* PURPOSE: This is a callback function used to load textures associated */
/* with an object being loaded in. */
/*-----------------------------------------------*/
HRESULT loadTextures(char *name, LPDIRECT3DMESHBUILDER builder, LPDIRECT3DMESHTEXTURE tex)
{
    // Load in the associated texture.
    // Assign it to the texture object, tex.
    return lpD3DRM->LoadTexture(name, tex);
}
```

### 5.1.5 AlterRenderQuality

The AlterRenderQuality function serves as an interface between the window menu and the DirectX™ function to change the quality of the scene. Remember that the available quality modes are WIREFRAME, UNLITFLAT, FLAT, GOURAUD and PHONG.

The menu in question appears as

![Render Quality Menu](image)

Subjects of significance addressed by this function include

**DirectX: info->dev->SetQuality**

```
/*-----------------------------------------------*/
/* FUNCTION: AlterRenderQuality */
/* INPUTS: HWND win - Handle to current window. */
/* ApplicationInfo *info - Pointer to scene structure. */
/* D3DRMRENDERQUALITY RenderQuality - The render quality we want. */
/* OUTPUTS: static bool - TRUE on success */
/* AFFECTED: N/A */
/* PURPOSE: This function comes as a result of the user selecting one of */
/* the sub-menu items from the RENDER popup menu. This involves */
/* checking for the validity of the new quality, changing the */
/* quality, and placing a check next to the currently active view */
/* quality in the sub-menu. */
/*-----------------------------------------------*/
static BOOL AlterRenderQuality(HWND win, ApplicationInfo *info, D3DRMRENDERQUALITY RenderQuality)
{
    HMENU Menu; // Handle to the current window menu.
    HRESULT rval; // Result returned from Direct3D calls.
    int OldMenuItem; // Index of old menu item.
```
int NewMenuitem; // Index of new menu item.

// Do action depending on requested render quality.
switch (RenderQuality) {
    case D3DRMRENDER_WIREFRAME: // Wireframe mode.
        NewMenuitem = 0; // It's menu item 0.
        break;
    case D3DRMRENDER_UNLITFLAT: // Solid mode.
        NewMenuitem = 1; // It's menu item 1
        break;
    case D3DRMRENDER_FLAT: // Flat shading mode.
        NewMenuitem = 3; // It's menu item 3
        break;
    case D3DRMRENDER_GOURAUD: // Gouraud shading mode.
        NewMenuitem = 4; // It's menu item 4
        break;
    case D3DRMRENDER_PHONG: // Phong shading mode.
        NewMenuitem = 5; // It's menu item 5
        break;
    default:
        return FALSE; // Unknown render quality, return error!
}

// Since new render quality, we must now adjust the menu look.
menu = GetMenu(win); // Get handle to current menu.
menu = GetSubMenu1menu, MENU_POPUP_RENDER); // Get handle to current RENDER
submenu.

// Uncheck all entries in this submenu
for (OldMenuitem = 0; OldMenuitem <= 5; OldMenuitem++)
    CheckMenuitem(menu, OldMenuitem, MF_BYPOSITION | MF_UNCHECKED);

// Check only the new menu item.
CheckMenuitem(menu, NewMenuitem, MF_BYPOSITION | MF_CHECKED);

// Attempt to change the render quality.
rval = info->dcv->SetQuality(RenderQuality);
if (rval != D3DRM_OK) // notify the user of error.
    DisplayMsg("Unable to alter the render quality,\n", ErrorToString(rval));
    return FALSE; // Return error.

return TRUE; // All OK!

Examples of the different render qualities now follow,
Figure 5.1.5.3
Flat

Figure 5.1.5.4
Gouraud

5.1.7 Routines found in Win32

These routines are used to provide an interface for the pointer object and its control. It interfaces with the mouse on either a left mouse button click or a mouse movement.

Mouse movement between different routines in changes in the YZ plane, with a right mouse button click results in changes in the XZ plane.

These routines use the Transform/Inverse Transform function pairs provided by DirectX to convert the screen coordinates to model space coordinates.
5.1.6 Render

The Render function is called to render the current scene. It uses a floating point variable called RenderRotation to determine by how much the move will be scaled. Therefore, a value of 1.0 would mean a move ratio of 1:1.

Subjects of significance addressed by this function include

**DirectX**: `Active_Window->defaultframe->Move`

**DirectX**: `Active_Window->view->Clear`

**DirectX**: `Active_Window->view->Render`

**DirectX**: `Active_Window->dev->Update`

```c
// FUNCTION: Render
// INPUTS: N/A
// OUTPUTS: static bool - TRUE if no error occurred.
// AFFECTED: RenderRotation.
// PURPOSE: This function performs the necessary Direct3D calls to render the current scene. Uses the RenderRotation variable to decide on how much the scene should be rotated. A value of 0.0 means no rotation, while 1.0 means a ratio of 1:1.

static BOOL Render()
{
    // Update all the objects in the current scene.
    if (FAILED(Active_Window->defaultframe->Move(D3DVAL(RenderRotation))))
        return FALSE; else RenderRotation = (double)0.0;

    // Clear the current view.
    if (FAILED(Active_Window->view->Clear()))
        return FALSE;

    // Render the scene.
    if (FAILED(Active_Window->view->Render(Active_Window->Scene)))
        return FALSE;

    // Update the device.
    if (FAILED(Active_Window->dev->Update()))
        return FALSE;

    return TRUE; // All OK!
}
```

5.1.7 Routines found in WindowProc

These case statements provide the necessary interface for the pointer object and its control. It causes the main program to activate on either a left mouse button click, right mouse button click or any mouse movement.

Mouse movement while the left button is clicked results in changes in the YZ plane, while movement with the right button clicked results in changes in the XY plane.

These routines use the Transform/Inverse Transform function pairs provided by DirectX\textsuperscript{TM}, to convert the screen coordinates to model
coordinates and back again. This is needed for the coordinate conversion and error checking.

\[
\text{mouse coords} \rightarrow \text{Button Pressed} \rightarrow \text{input} \rightarrow \text{last X, last Y, X, Y} \rightarrow \text{change in X(lastX - dx)} \rightarrow \text{change in Y(lastY - dy)} \rightarrow \text{LastX} < X \rightarrow \text{LastY} < Y \rightarrow \text{GetPosition(info->defaultframe)} \rightarrow \text{(point1)} \rightarrow \text{Transform[point1]} \rightarrow \text{(point2)} \rightarrow \text{scale point2.x} \rightarrow \text{scale point2.y} \rightarrow \text{InverseTransform[point1][point2]} \rightarrow \text{check point in envelope} \rightarrow \text{yes} \rightarrow \text{replot pointer} \rightarrow \text{no}
\]

This same general algorithm is used by both the left and right button routines.

case WM_LBUTTONDOWN: // left mouse button pushed.
    {  
        int x = LOWORD(lparam); // X position of mouse pointer.
        int y = HIWORD(lparam); // Y position of mouse pointer.
        Last_x = x; // Save last X position.
        Last_y = y; // Save last Y position.
        left_drag = TRUE; // We are dragging the mouse pointer.
        SetCapture(win); // Allocate mouse to this window.
    } break;

case WM_LBUTTONUP: // left mouse button released.
    ReleaseCapture(); // deallocate mouse from this window.
    left_drag = FALSE; // No longer dragging mouse.
    break;

case WM_RBUTTONDOWN: // Right mouse button pushed.
    {  
        Last_x = LOWORD(lparam); // Save last X position.
        Last_y = HIWORD(lparam); // Save last Y position.
        right_drag = TRUE; // We are dragging the mouse pointer.
        SetCapture(win); // Allocate mouse to this window.
    } break;

case WM_RBUTTONUP:
    ReleaseCapture(); // Deallocate mouse from this window.
    right_drag = FALSE; // No longer dragging mouse.
    break;

case WM_MOUSEMOVE: // the mouse is moving.
    {  
        // Do if mouse moving and left button pushed.
        if ((lparam & MK_LBUTTON)) & left_drag == TRUE
        {  
            double delta_x, delta_y; // The change in X and Y.
            D3DXVECTOR2 point1; // Holds position of pointer object.
            D3DXVECTOR2 point2; // Holds transformed position of pointer.
            double delta_x, delta_y; // The change in X and Y.
            delta_x = SIG Extend(LOWORD(lparam)) - Last_x;
            delta_y = SIG Extend(HIWORD(lparam)) - Last_y;
            // Re-assign last (X, Y) position.
            Last_x = SIG Extend(LOWORD(lparam));
            Last_y = SIG Extend(HIWORD(lparam));
            // Get current co-ordinates of pointer object.
            info->point->GetPosition(info->defaultframe, &point1);
            // Transform the co-ordinates.
            info->view->Transform(&point2, &point1);
        }
    }
Figure 5.1.7
Main Screen.

The above functions, when appropriately linked to dependent functions, result in the following main screen,

Notice how the default quality for the robot is WIREFRAME. Also notice the pointer object positioned at (0,0,0). (The robot has been shifted to the right corner for the purpose of the screen dump. Its default position is the centre of the screen).
5.2 Immediate motion

All of the functions necessary for immediate motion can be found in the source files \texttt{ObjMenu.h} and \texttt{ObjMenu.cpp}.

The routines which are of relevance to us here are found in the \texttt{WindowProc} function. The module represented by \texttt{ObjMenu} is made up, mostly of initialisation functions for things such as window creation and initialisation of data structures.

A new concept is introduced here, which refers to our 'auto-fire' routine. The mouse-button system messages that Windows™ 95 sends to our program are only sent once for each event. Thus, even if the button is kept pressed only one message is sent. Our routines overcome this limitation by using a timer interrupt to 'click-off the mouse and send another message whenever the left mouse button is clicked; the right mouse button is used as a non-auto-fire button.

This module is accessed via the main menu through the menu item, \textit{Manipulate View}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.2}
\caption{Manipulate Menu.}
\end{figure}

Subjects of significance addressed by this function include

\textbf{C++: ObjMenu constants}
// Defines and types for the ObjMenu module
#define LBUTTONTIMER 1  // Unique identifier for button timer.
#define LBUTTONTIMEOUT 20 // Time (ms) between button pushes.
#define PANU 0  // The button identifiers.
#define PAN 1
#define PANL 2
#define PANR 3
#define ZOOMIN 4
#define ZOOMOUT 5
#define ROTU 6
#define ROTD 7
#define ROTL 8
#define ROTR 9
#define ROTDL 10
#define ROTOR 11
#define VIEWMAXPAN (double)4.9 // Max/min values for ObjMenu.
#define VIEWMINPAN (double)-4.9
#define VIEWMAXZOOM (double)9.9
#define VIEWMINZOOM (double)0
#define VIEWMAXANGLE (double)3.12
#define VIEWMINANGLE (double)-3.12

\textbf{C++: ManipScene data structure}
struct VIEWSCENETYPE{
  double PANY, PANZ; // The y and z values for panning +-10
  double ZOOM; // The current zoom factor +-10
  double PITCH; // The current pitch angle +-2
  double BANK; // The current banking angle +-2
  double HEADING; // The current heading +-2
};
case WM_RBUTTONDOWN: // The user has pressed one of the mouse buttons.
case WM_LBUTTONDOWN:
  i = 0; // Check if user has clicked on a valid button.
  while (i <= MAXBUTTONS)
  {
    if (MouseX >= ObjBs[i][0] && MouseY >= ObjBs[i][1] && MouseX <= ObjBs[i][2] && MouseY <= ObjBs[i][3])
    {
      // Turn ON button.
      PressButtonBMP(win, 1, i, ObjBs[i][0], ObjBs[i][1], ObjBs[i][2], ObjBs[i][3]);
      LastButtonDown = i; // Save the last button pressed.
      switch (i) // Do action depending on button clicked.
      {
        case PANU: // Pan UP.
          if (ManipScene.PANZ < VIEWMAXPAN)
            [ManipScene.PANZ += 0.1; InLimits = TRUE;]
          break;
        case PAND: // Pan DOWN.
          if (ManipScene.PANZ > VIEWMINPAN)
            [ManipScene.PANZ -= 0.1; InLimits = TRUE;]
          break;
        case PANL: // Pan LEFT.
          if (ManipScene.PANY > VIEWMINPAN)
            [ManipScene.PANY -= 0.1; InLimits = TRUE;]
          break;
        case PANR: // Pan RIGHT.
          if (ManipScene.PANY < VIEWMAXPAN)
            [ManipScene.PANY += 0.1; InLimits = TRUE;]
          break;
        case ZOOMIN: // Zoom IN.
          if (ManipScene.ZOOM < VIEWMINZOOM)
            [ManipScene.ZOOM += 0.1; InLimits = TRUE;]
          break;
        case ZOOMOUT: // Zoom OUT.
          if (ManipScene.ZOOM > VIEWMAXZOOM)
            [ManipScene.ZOOM -= 0.1; InLimits = TRUE;]
          break;
        case ROTU: // Rotate UP.
          if (ManipScene.BANK < VIEWMAXANGLE)
            [ManipScene.BANK += 0.02; InLimits = TRUE;]
          break;
        case ROTL: // Rotate LEFT.
          if (ManipScene.BANK > VIEWMAXANGLE)
            [ManipScene.BANK -= 0.02; InLimits = TRUE;]
          break;
        case ROTO: // Rotate DOWN.
          if (ManipScene.PITCH < VIEWMAXANGLE)
            [ManipScene.PITCH += 0.02; InLimits = TRUE;]
          break;
        case ROTR: // Rotate RIGHT.
          if (ManipScene.PITCH > VIEWMAXANGLE)
            [ManipScene.PITCH -= 0.02; InLimits = TRUE;]
          break;
      }
      if (InLimits)
      {
        // Do only if values within limits.
        if (InLimits)
        {
          // Notify the parent that the robot must be updated?
          PostMessage(Parenthwnd, WM_OBJMENUCHANGED, (WORD)i, (LPARAM)&ManipScene);
          if (msg == WM_LBUTTONDOWN)
            ObjMenuUpdate = 1; // Do not update.
          else ObjMenuUpdate = -1; // Update possible.
          OldMouseXY = lparam; // Save old mouse (X,Y).
          else ObjMenuUpdate = -1; // Update possible.
          if (i < MAXBUTTONS)
            TextOutBMenu(win); // Update view.
          i++;
        }
        break;
      }
      break;
    }
    i++;
  } // Keep checking until all buttons checked.
break;
case WM_RBUTTONDOWN:
case WM_LBUTTONDOWN:
    if (LastButtonDown != -1) // Do only if last button clicked was valid.
    {
        // Turn OFF button.
        PressButtonBMP(win, 0, LastButtonDown, ObjBs[LastButtonDown][0], ObjBs[LastButtonDown][1]);
        LastButtonDown = -1; // Reset variable to NONE.
        TextOutObjMenu(win); // Update view.
        ObjMenuUpdate = -1; // Do not update.
    }
    break;

case WM_TIMER: // A timer event was registered.
    // Do only if it is the correct timer event for this module.
    if (wparam == LBUTTONTIMER && ObjMenuUpdate != -1)
    {
        MouseX = LOWORD(OldMouseXY);
        MouseY = HIWORD(OldMouseXY);
        mouse_event(MOUSEEVENTF_LEFTDOWN, MouseX, MouseY, 0,
                    GetMessageExtraInfo());
    }
    // This has the effect of an 'auto-fire' for the left mouse button.
    break;

The immediate motion window appears in our application as,

![Immediate Motion Window](image)

Immediate motion, via the manipulate view window (above) allows us to PAN UP, PAN DOWN, PAN LEFT, PAN RIGHT, ZOOM IN, ZOOM OUT, ROTATE UP, ROTATE DOWN, ROTATE LEFT and ROTATE RIGHT, all at the click of a mouse button. Another two possibilities, ROTATE DIAGONAL RIGHT and ROTATE DIAGONAL LEFT are also included.

Also available are the current values for the Y offset, Z offset and zoom factor, which is actually the X offset. The alternate axis of (heading, pitch, banking) is also specified, with values in the range -pi..pi.

### 5.3 Indirect motion

The indirect module hinges entirely on the ability of the user's program being able to communicate effectively with our simulator. Earlier on we stipulated that we could achieve this by intercepting the interrupt vector at 78HEX, which the START.COM program uses to interface with the user's program.
What we never discussed was how to then pass this information to our simulator.

The users' programs will always be executed in DOS mode, in something like a DOS window from within Windows™ 95, so the idea of intercepting the interrupt is still valid, as long as we write an assembly language program to accomplish this. The main flaw in our thinking is assuming that a DOS program can efficiently communicate with a Windows™ 95 program.

When attempting to face this problem a number of alternatives emerged. One possibility was using a swap file on disk to pass commands, but this was omitted due to the slowness of the response. Another possibility was the very real option of writing a Windows™ 95 device driver (.Vxd) to handle the communication. However, this idea was passed over when it was realised that to achieve this we would require the purchase of a special programming library, at approximately $500.

Convinced an alternative was available a search was undertaken to find a possible method. Whilst examining the list of DOS interrupts an interrupt entitled WinOldAp was found at interrupt vector 2fHEX. Apparently this interrupt still remains, for compatibility reasons, a functional part of the Windows™ operating system, although it was primarily used in earlier versions of windows, which were 16-bit based.

The interrupt is a hook into the clipboard provided by the Windows™ operating system. By using this interrupt we can efficiently pass information from our DOS program to our Windows™ 95 simulator and back again.

Windows™ allows a type tag to be attached to clipboard data to identify the type of data on the clipboard. Rather than create our own type, which may conflict with another program's custom type, we are going to use the CF_TIFF type, which is used for .TIF graphic files. This was chosen due to its lack of popularity in computing circles. Regardless of this fact the user should refrain from using clipboard functions whilst simulating, as errors and perhaps lockups, may occur.

When it came to a protocol, it was decided to mimic a very simple modem protocol of using a flag character to signal the start of transmission and another flag character to signal receipt of transmission. We don't need a flag to signal end of transmission, as each transmission will be the same length!

These flags are entitled STX and ACK respectively.

This Windows™ module is accessed via the main menu through the menu item, Drivers -> RTX Driver.
The format of the driver module is dependent on the menu item entitled *Programming*, which can either be *online* or *offline*.

The online module’s window looks like,

![Driver Online](image)

while the offline module’s window looks like,

![Driver Offline](image)

The online and offline modules differ only in the fact that the online module has six extra LEDs and four extra buttons, entitled *com1*, *com2*, *com3* and *com4*, which are clicked on to choose a communication port for robot-computer communications.

The window displays all the motor register values for the motor chosen by clicking on the two buttons to the right of the *Motor Data* heading. Also displayed is the currently active motor, designated by an LED corresponding to that particular motor, in the box entitled *Motors*. 

**Implementation**
The reset button resets all motor registers to zero and redraws the arm. The step button causes the system to enter debug mode, prompting for a mouse click every time a robot command is processed. The error button is used for testing the user’s program by introducing ‘fake’ errors. Finally, the ACK button causes an ACK command to be sent to the DOS driver, in case of system lockups or hangs.

All of the functions necessary for the construction of this module can be found in the source files TSWinDrv.h, TSWinDrv.cpp and TSDOSDRV.ASM.

The functions which are of relevance to us meeting our objectives here are

- TSDOSDRV.ASM (DOS TSR driver)
- ProcessCOMEvent
- WaitForCOMEvent
- WriteBytesToCOM
- ReadBytesFromCOM
- UpdateCOMButtons
- UpdateMotorsAfterMove
- AdjustMotorMode
- ForceClipACK
- RTX_ARM_GO
- UpdateClipboard
- Routines found in WindowProc

5.3.1 TSDOSDRV.ASM

![Flowchart](image)

Figure 5.3.1 TSDOSDRV.ASM.

---

Program: TSDOSDRV.EXE

Purpose: This program is a TSR which utilizes the 'WinOldAp' interface provided by Windows, for the purpose of character-mode app access to the clipboard. This has the effect of allowing DOS access to the clipboard and its functions, via INT 0x2F. NOTE: Executing these functions outside of a Windows DOS shell may yield unpredictable results.

Tech: The clipboard format is CF_TIFF. Data is in form of a program string, consisting of a leading protocol identifier followed by the command and data. The leading identifier can be either 'STX' or 'ACK', STX for
**Section 5.0**

```asm
.model small
.stack 200h
.code
386

; GLOBAL DEFINES

; STX EQU 01h ; signals beginning of program string
; ACK EQU 02h ; signals program string received
; NEW_INTERRUPT EQU 07h ; signals beginning of program string
; DATA_BUFFER_SIZE EQU 100h ; user specified max. size of data allowable
; BUFFER_SIZE EQU 07h ; user defined min. size of data allowable
; Clip consists of ... [0] : ACK or STX
; [1] : service type
; [2..17] : data

; D_CLIPFORMAT EQU CF_TIFF ; The clip format we will be filtering
; D_IDENTIFY EQU 1700h ; Service parameter: Identify WinOldAp
; D_OPENCLIP EQU 1701h ; Service parameter: Open clipboard
; D_CLOSECLIP EQU 1702h ; Service parameter: Close clipboard
; D_EMPTYCLIP EQU 1703h ; Service parameter: Empty clipboard
; D_WRITECLIP EQU 1704h ; Service parameter: Write to clipboard
; D_READCLIP EQU 1705h ; Service parameter: Read from clipboard

; MAIN FUNCTION

; PROCEDURE: TSDOSDRV (AX, BX, CX); / PURPOSE: The main function. / INPUTS: AX = Service number ... / BX = Offset of parameters. / CX = Segment of parameters. / OUTPUTS: AX = 0 - NO ERROR / AX <> 0 - ERROR /
; () NOTICE(): Parameters format - int Params[8] /

; PROC FAR
jmp @InstallInterrupt ; Install the interrupt proc.

@InterruptStart:
  db Oeh, offset @CopyrightJmp-6 ; Go to start of start of code!
  db Och, 'c UMI 1996' ; Message for compatibility
@CopyrightJmp:
  jmp @StartHere ; Go to start of code!

@Int2fVectorizer: dw 0, 0 ; Vector to hold pointer to INT 2Fh
@WinOldApVer: dw 0, 0 ; MajorVersion.MinorVersion
@InstallState: db 0 ; =1 if WinOldAp installed

@TSBUFFER: db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ; Buffer for incoming and,
  db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ; outgoing data.

@StartHere:
  push ax
  push bx
  push cx
  cmp byte ptr @InstallState[0], 0 ; Check if WinOldAp installed
  jnz @WinOldApYes ; Yes, WinOldAp installed
  mov ax, 0170h ; AX=0170h "Identify WinOldAp"
  int 2Fh ; Do service call, AX=RESULT
  mov word ptr @WinOldApVer[0], ax ; Save version of WinOldAp
```
emp
jnz
mov
iret

@WinOldApYes:
mov
cmp
iret
 commodifying

@WinOldApYes:
mov
push
cmp
j1
mov
iret

---

Section 5.0 - Page 86

; Check if right version/exists
jnz @WinOldApYes
; Yep, start code proper
pop bx
pop bx
pop ax
mov ax,1
iret

; Exit interrupt NOW!

@WinOldApYes:
mov byte ptr @InstallState[0],1
; Flag that WinOldAp installed
pop cx
; Load important regs
pop bx
pop ax
push ax
pop bx
pop bx
push bx
mov ax, bx
pop bx
; Now, AX and BX have swapped!
cmp bx,16
jl @GetClipboardData
; Check if service supported
mov ax,1
; Yep, do it!
iret
; Exit interrupt NOW!

; Set important registers
; ES ~ CX - @BUFFERS
; ES: DS - Parameters
; Size of Parameters structure
; Forward
; Copy parameters into @BUFFERS
; Indicate service type
mov byte ptr @BUFFERS+1,bl
int 2fh

; Open the clipboard
push cs
mov bx,offset @BUFFERS
push cs
mov ax,D_WRITECLIP
; Specify we want to write clip
cmp bx,D_CLIPFORMAT
; Type of clip data
xor si,si
mov cx,NORMCLIPSIZE
int 2fh
; Do call (AX=0 ERROR)
mov ax,D_CLOSECLIP
int 2fh
pop bx
pop ax

; Get the data from the clipboard, if
; there is any available. Does not
; include ACK/STX or service type.
@GetClipboardData:
oparse error

---

@GetClipboardDataWait:
mov ax,D_WRITECLIP
int 2fh

push cs
pop es

; Save important registers
push cx

push ax
; Save important registers

push ax

push ax

mov bx, offset @TSBUFFER
mov ax, D_READCLIP ; Specify we want to read clip
mov dx, D_CLIPFORMAT
xor si, si ; SI must be zero (64Kb segments)
mov cx, NORMCLIPSIZE ; SI: CX = size of receive buffer
int 2fh ; Do call (AX=0 ERROR)
push ax
mov ax, D_CLOSECLIP ; Close the clipboard
int 2fh
pop ax
cmp ax, 0 ; Check if any data in clipboard
jz @GetclipboardDataWait
cmp byte ptr @TSBUFFER, ACK ; Yep, check if it contains ACK
jnz @GetclipboardDataWait
pop ecx ; Load important registers
pop ax
push cs
pop ds
mov si, offset @TSBUFFER+2
mov dx, cx
mov cx, NORMCLIPSIZE-2
rep movsb
iret ; Return from interrupt

******************************************************************************
; END
******************************************************************************

@InstallInterrupt:
mov ah, 35h ; DOS function, get int.
mov al, 2fh ; Interrupt 2fh
int 21h ; Get interrupt vector
mov word ptr @Int2fVector, bx ; Store int vector
mov word ptr @Int2fVector+2, es
push cs
pop ds ; Let DS = CS
mov dx, offset @InterruptStart ; DS: DX -> your interrupt
mov al, NEW_INTERRUPT
mov ah, 25h
int 21h
mov dx, offset @InstallInterrupt+1024 ; CS: DX -> end of interrupt
int 27h

TSDOSDRV ENDP
END TSDOSDRV

5.3.2 ProcessCOMEvent

The ProcessCOMEvent function is a function that accepts an event number and lights up the appropriate LED in the window and then checks the status of the comm port and lights up the other respective LEDs. This, of course, only applies to the online module.

The available event numbers are EV_RXCHAR and EV_TXEMPTY, which correspond to character received and transmit buffer empty, respectively.

The comm port status registers we are using are MS_CTS_ON, MS_DSR_ON, MS_RING_ON and MS_RLDS_ON, which correspond to clear-to-send on, data-set-ready on, ring-indicator on and carrier-detect active, respectively.

Subjects of significance addressed by this function include Windows: GetCommModemStatus
/* FUNCTION: ProcessCOMEvent */
/* INPUTS: HWND hwnd - Handle to current window. */
/* DWORD EventNumber - Identifier of comm event. */
/* OUTPUTS: N/A */
/* AFFECTED: N/A */
/* PURPOSE: This function processes a COM event, as specified by the value in EventNumber. */

void ProcessCOMEvent(HWND hwnd, DWORD EventNumber)
{
    HDC hdc; // Handle to current device context.
    int i; // Dummy loop variable.
    DWORD ModemStatus; // Holds the current status of the serial port.

    hdc = GetDC(hwnd); // Get the current device context handle.
    SelectObject(hdc, hClipLEDOFFBrush); // Activate LED OFF brush.
    for (i = 0; i < MAXCOMLEDS; i++) // Turn off all LEDs on screen.
    {
        Rectangle(hdc, COMLEDs[i][0], COMLEDs[i][1], COMLEDs[i][2], COMLEDs[i][3]);
    }
    DeleteObject(hClipLEDOFFBrush); // Release LED OFF brush.

    SelectObject(hdc, hClipLEDONBrush); // Activate the LED ON brush.
    switch (EventNumber) // Process the event, and turn on LEDs.
    {
    case EV_RXCHAR: // Char received.
        i = ONLINE_RXLED;
        Rectangle(hdc, COMLEDs[i][0], COMLEDs[i][1], COMLEDs[i][2], COMLEDs[i][3]);
        break;
    case EV_TXEMPTY: // Transmit buffer empty.
        i = ONLINE_TXLED;
        Rectangle(hdc, COMLEDs[i][0], COMLEDs[i][1], COMLEDs[i][2], COMLEDs[i][3]);
        break;
    }

    GetCommModemStatus(hPort, &ModemStatus); // Get the status of the serial port.
    if (ModemStatus & MS_CTS_ON) // Clear to send.
        Rectangle(hdc, COMLEDs[i][0], COMLEDs[i][1], COMLEDs[i][2], COMLEDs[i][3]);
    if (ModemStatus & MS_DSR_ON) // Data set ready.
        Rectangle(hdc, COMLEDs[i][0], COMLEDs[i][1], COMLEDs[i][2], COMLEDs[i][3]);
    if (ModemStatus & MS_RING_ON) // Ring indicator.
        Rectangle(hdc, COMLEDs[i][0], COMLEDs[i][1], COMLEDs[i][2], COMLEDs[i][3]);
    if (ModemStatus & MS_RLSD_ON) // Carrier detect.
        Rectangle(hdc, COMLEDs[i][0], COMLEDs[i][1], COMLEDs[i][2], COMLEDs[i][3]);
    DeleteObject(hClipLEDONBrush); // Release the LED ON brush.

    ReleaseDC(hwnd, hdc); // Release the device context handle.
}

5.3.3 WaitForCOMEvent

The WaitForCOMEvent function is a function initiated by the fact that polling the serial port tends to waste a lot of time, as we wait for incoming data and can even cause lockups while the system waits.

This function is designed to run in its own thread and never exit, unless expressly done so by an external function terminating the thread, and simply keeps infinitely polling the serial port until such time that there
is an event, whereby it sends the data off to be processed and then continues polling again.

Subjects of significance addressed by this function include
**Windows:**  **WaitCommEvent**

```
// Function: WaitForCOMEvent
// Inputs: hWnd - Window handle
// Outputs: N/A
// Purpose: This function is designed to run independently in its own thread, in a never-ending loop. It just waits for a comm event to occur, and then dispatches the event number to another function for processing.

void WaitForCOMEvent(HWND hWnd)
{
    DWORD COMEvent = 0;  // Storage for the current event number.
    while (1) // Never-ending loop.
    {
        if (hPort != NULL) // Only wait if we are actually online.
            WaitCommEvent(hPort, &COMEvent, NULL);
        else COMEvent = 0;
        // Comm event occurred; Process the information.
        if (hPort != NULL) ProcessCOMEvent(hWnd, COMEvent);
    }
}
```

### 5.3.4 WriteBytesToCOM

The WriteBytesToCOM function is a simple function for transmitting a maximum of 3 supplied bytes through the serial port.

Subjects of significance addressed by this function include
**Windows:**  **WriteFile**

**C++:**  **COMBuffer.InEvent variable**

This variable is set to TRUE whilst in a communications event, so as to avoid conflicts with parallel running threads.

```
// Function: WriteBytesToCOM
// Inputs: hWnd - Handle to current window.
// short int NumBytes - Number of bytes to write to serial port.
// short int B1 - Byte 1.
// short int B2 - Byte 2.
// short int B3 - Byte 3.
// Outputs: Bool - TRUE if no error occurred.
// Affected: COMBuffer.
// Purpose: Sends bytes to the currently open serial port. It will send B1, B1 & B2, or B1 & B2 & B3, depending on the value of NumBytes.

bool WriteBytesToCOM(HWND hWnd, short int NumBytes, short int B1, short int B2, short int B3)
```
DWORD BytesWritten; // The number of bytes actually written.
BYTE TempBuff[3]; // Temporary buffer.

COMBuffer.InEvent = TRUE; // Signal that we are in a comm event.
TempBuff[0] = LOBYTE(B1); // Fill temporary buffer with our data.
TempBuff[1] = LOBYTE(B2);
TempBuff[2] = LOBYTE(B3);

// Send the bytes to the serial port.
WriteFile(hPort, &TempBuff, NumBytes, &BytesWritten, NULL);

COMBuffer.InEvent = FALSE; // No longer in a comm event.
if (BytesWritten != NumBytes) // Error writing the bytes.
    return FALSE;

ProcessCOMEvent(hwnd, 0); // Process this event.
return TRUE; // All OK!

**5.3.5 ReadBytesFromCOM**

The ReadBytesFromCOM function is a simple function to read a maximum of 3 bytes through the serial port.

Subjects of significance addressed by this function include

Windows: ReadFile

C++: COMBuffer.InEvent variable

This variable is set to TRUE whilst in a communications event, so as to avoid conflicts with parallel running threads.

```c
bool ReadBytesFromCOM(HWND hwnd, short int NumBytes, short int *B1, short int *B2, short int *B3)
{
    DWORD BytesRead; // Actual number of bytes read.
    BYTE TempBuff[3]; // Temporary buffer.

    COMBuffer.InEvent = TRUE; // Signal that we are in a comm event.
    // Attempt to read 3 bytes from serial port.
    ReadFile(hPort, &TempBuff, 3, &BytesRead, NULL);

    COMBuffer.InEvent = FALSE; // No longer in comm event.
    if (BytesRead == 0) // Error occurred.
        return FALSE;

    *NumBytes = (short int)BytesRead; // Return actual number of bytes read.
    *B1 = (short int)TempBuff[0]; // Assign the bytes read.
    *B2 = (short int)TempBuff[1];
    *B3 = (short int)TempBuff[2];
}
```
ProcessCOMEvent(hwnd, 0); // Process this event.
return TRUE; // All OK!

5.3.6 UpdateCOMButtons

The purpose of the UpdateCOMButtons is not solely as the name would suggest. This function performs the functions of allowing the user to choose a comm port to open, creates the thread to the WaitForCOMEvent function, opens the specified comm port, sets the comm speed and various timeout parameters, and performs general housekeeping such as lighting the correct LEDs and changing the motor speed variable to reflect the online mode.

In order to keep the robot moving at the quickest pace possible, which is needed for real-time communications, we set the speed to its fastest value, specified in the constant ONLINE_MOTOR_SPEED.

Subjects of significance addressed by this function include

Windows: CreateThread

Windows: TerminateThread

Windows: CreateFile

Windows: BuildCommDCB

Windows: GetCommTimeouts

Windows: SetCommTimeouts

Windows: SetCommMask

Figure 5.3.6 UpdateCOMButtons

window handle —— UpdateCOMButtons

port already open? — true — close port.
false — terminate thread

open port

set port params[9600,n,8,1]

set timeout values

CreateThread[Wai1forCOMEvent]

ProcessCOMEvent

/* FUNCTION: UpdateCOMButtons
 */
/* INPUTS: HWND hwnd - Handle to current window. */
/* int P  - The port number to be highlighted. */
/* int LP  - The previously highlighted port number. (-1=None). */
void UpdateCOMButtons(HWND hwnd, int Port, int LP)
{
    COMTIMEOUTS COMTimeout; // The serial port timeout structure.
    HDC hdc; // Handle to the current device context.
    DCB NewDCB; // Our new DCB structure.
    DWORD ThreadID; // Storage for the newly created thread ID.
    int i; // Dummy loop variable.

    hdc = GetDC(hwnd); // Assign a handle to the current device context.

    if (LP != -1) // If a comm port was previously highlighted, do.
    {
        SelectObject(hdc, hClipLEDOffBrush); // Turn off LED for that comm port.
        Rectangle(hdc, COMLeds[LP][0], COMLeds[LP][1], COMLeds[LP][2], COMLeds[LP][3]);
        DeleteObject(hClipLEDOffBrush);
    }

    switch(P) // Get name of chosen port.
    {
        case ONLINECOM1:
            strcpy(COMNumber, "COM1");
            break;
        case ONLINECOM2:
            strcpy(COMNumber, "COM2");
            break;
        case ONLINECOM3:
            strcpy(COMNumber, "COM3");
            break;
        case ONLINECOM4:
            strcpy(COMNumber, "COM4");
            break;
        default:
            break;
    }

    if (hPort != NULL) // Do if a port already open.
    {
        // Kill the thread, so that another one can be assigned.
        if (COMThread != NULL) TerminateThread(COMThread, 0);
        COMThread = NULL;
        CloseHandle(hPort); // Close the currently open serial port.
    }

    // Open a serial port for reading and writing.
    hPort = CreateFile(COMNumber, GENERIC_READ|GENERIC_WRITE, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hPort == INVALID_HANDLE_VALUE) // Error occurred.
    {
        // Alert user.
        DisplayMsg("Could not open Port: Is\a Serial transfer cannot occur!", COMNumber);
        hPort = NULL; // Nullify the hPort value.

        SelectObject(hdc, hClipLEDOffBrush); // Turn off all comn LEDs.
        for (i=0; i<MAXCOMLEDs; i++)
        {
            Rectangle(hdc, COMLeds[i][0], COMLeds[i][1], COMLeds[i][2], COMLeds[i][3]);
            DeleteObject(hClipLEDoffBrush);
        }

        // Blank the area in the window that says 'ONLINE'.
        SelectObject(hdc, hClipBrush);
        Rectangle(hdc, BlankArea[0], BlankArea[1], BlankArea[2], BlankArea[3]);
        DeleteObject(hClipBrush);
    }
    else // No error occurred.
    {
        COMBuffer.InEvent = FALSE; // Not in a comm event.

        GetCommState(hPort, &NewDCB); // Get current comm state.
        BuildDCB(NewDCB, COMSTATE, &NewDCB); // Initialise it with our own values.
        SetCommState(hPort, &NewDCB); // Do it.

        GetCommTimeouts(hPort, &COMTimeout); // Fill our timeout structure.
        COMTimeout.ReadIntervalTimeout = 0; // 0ms read interval.
        COMTimeout.ReadTotalTimeoutMultiplier = 0; // 0ms multiplier.
        COMTimeout.ReadTotalTimeoutConstant = COMTIMEOUT; // Timeout value.
COMTimeout.WriteTotalTimeoutMultiplier = 0; // Oms write multiplier.
COMTimeout.WriteTotalTimeoutConstant = 0; // Oms timeout value.
SetCommTimeouts(hPort, &COMTimeout); // Do it.

SetCommMask(hPort, COMEventMask); // Tell the system on what events we
// want to be alerted.

if (COMThread == NULL) // Assign a new thread to WaitForCOMEvent.
    COMThread = CreateThread(NULL, 0, WaitForCOMEvent, (LPVOID)hwnd,
        NULL, HIGH_PRIORITY_CLASS, &ThreadID);

// Display the 'ONLINE' banner.
BitBlt(hdc, 0, 0, ClipBackground.bWidth, ClipBackground.bHeight,
    GeneralMemory, 0, 0, SRCCOPY);
TextOut(HWND, ClipLastRec, ClipLastTran);
// Update view.
SelectObject(hdc, &ClipLEDOnBrush); // Turn ON appropriate comm. LED.
Rectangle(hdc, COMs[P][0], COMs[P][1], COMs[P][2], COMs[P][3]);
DeleteObject(&ClipLEDOnBrush);

ProcessCOMEvent(hwnd, 0); // Process this event.

MOTOR_SPEED = ONLINE_MOTOR_SPEED; // Change motor speed to suit.

ReleaseDC(hwnd, hdc); // Release the current device context handle.

5.3.7 UpdateMotorsAfterMove

The UpdateMotorsAfterMove function is called after a system
message has been received stating that a particular motor has finished
moving. This function updates the register corresponding to current
position with the robot's current position, quite appropriately.

The function also has the responsibility of deciding whether the motor
has reached its limit and then alters the robot variable
ARM_MOTOR_STATUS to match. It decides this by comparing the
robot's current position to the maximum and minimum values held in
the constant array called MotorLimits.

Subjects of significance addressed by this function include

C++:

    motor structure
    // The enumerated type relating to the available
    // registers within the RTX arm.
    typedef enum { DC_ERROR, DC_CURRENT_POSITION,
        DC_ERROR_LIMIT, DC_NEW_POSITION, DC_SPEED, DC_KP, DC_KI,
        DC_KD, DC_DEAD_BAND, DC_OFFSET, DC_MAX_FORCE,
        DC_CURRENT_FORCE, DC_ACCELERATION_TIME, DC_USER_RAN,
        DC_USER_ID, DC_ACTUAL_POSITION, DC_PREVIOUS_POSITION,
        DC_NONE, DC_MATE, DC_ALL, DATA_CODE};
    typedef short int DATA_CODE[DC_ALL];
    // Variable version of above.

    // Structure for each motor.
    struct MOTOR { int current;
        int current_data_code[TOTALMOTORS];
        DATA_CODE data_code[TOTALMOTORS]; } motor;

C++:

    MotorLimits structure
    // Minimum/maximum values for the motor encoder counts.
    const int MotorLimits[TOTALMOTORS][2] = {
        [-2630, 2206], [-2630, 2630], [-3554, 0], [-2642, 1081], [-3560, 4882],
        [-1071, 1071], [-30, 1200]};

    // FUNCTION: UpdateMotorsAfterMove
5.3.8 AdjustMotorMode

The AdjustMotorMode function accomplishes the task of a MANUAL_GO. We have seen earlier that the RTX can be in either NUMERIC_GO or MANUAL_GO and since most of the simulator’s development was under numeric control, the manual control appeared as though it would be a problem.

That is, we had already in place routines to divide up the numeric movements into timed frames and were working fine. Manual control, however, requires that the motors be constantly moving. We achieve this fact through ‘sleight of hand’ by checking whether we are in manual mode and, if we are, set the new position register to its limit. This has the effect of continuous movement.

All we really have to decide is whether we choose the minimum limit or the maximum limit, which depends on the GO_BITS variable.

Subjects of significance addressed by this function include

C++: GO_BITS variable

Defines which motors to move and in what direction.

C++: MOTOR_BITS structure

```c++
#define ELBOW_BITS 0x0003
// The bits that need to be affected in order to
#define SHOULDER_BITS 0x000c
// move that particular motor.
#define ZED BITS 0x0030
#define WRIST1_BITS 0x00c0
#define WRIST2_BITS 0x0300
#define YAW BITS 0x0600
#define GRIP_BITS 0x3000
// Array of pre-calculated bits for all the motors, for
// all possible options.
const WORD MOTOR_BITS[TOTALMOTORS] =
    {ELBOW_BITS, 0x0000, 0x0001, 0x0002, 0x0003},
```
Section 5.9 - Page 95

[SHOULDER_BITS, 0x0000, 0x0004, 0x0008, 0x000c],
[ZED_BITS , 0x0000, 0x0010, 0x0020, 0x0030],
[WRAW_BINTS , 0x0000, 0x0004, 0x0008, 0x000c],
[WRAWIPE_BINTS , 0x0000, 0x0004, 0x0008, 0x000c],
[YAW_BITS , 0x0000, 0x0040, 0x0080, 0x0100],
[GRIP_BITS , 0x0000, 0x0004, 0x0008, 0x000c]);

#define BITS_BITS 0
// All motor bits at index 0, above.
#define BITS_POWERSTOP 1
// Powerstop bits at index 1, above.
#define BITS_FWD 2
// Motor forward bits at index 2, above.
#define BITS_REV 3
// Motor reverse bits at index 3, above.
#define BITS_FREESTOP 4
// Freestop bits at index 4, above.

C++:

motor structure (see 5.3.6)

/**
 * FUNCTION: AdjustMotorMode
 * INPUTS: MOTOR - The motor to be worked on.
 * short int Varint - The motor variable to be adjusted.
 * OUTPUTS: short int Varint - The adjusted input variable.
 * AFFECTED: N/A
 * PURPOSE: Accomplishes the task of a MANUAL_GO by setting the input variable (Varint) to the extreme for that motor.
 */

short int AdjustMotorMode(MOTOR, short int Varint)
{
    WORD TempBITS; // Temporary storage for GO_BITS.
    // Only do if in MANUAL GO mode.
    if (GO_MODE == MANUAL_GO)
    {
        // Alter Varint to the motor extremes, depending on whether it is moving forwards or backwards.
        TempBITS = GO_BITS & MOTOR_BINTS[MOTOR][BITS_BITS];
        if (TempBITS == MOTOR_BINTS[MOTOR][BITS_FWD])
        {
            Varint = (short int)MotorLimits[MOTOR][MOTOR_MIN];
        } else if (TempBITS == MOTOR_BINTS[MOTOR][BITS_REV])
        {
            Varint = (short int)MotorLimits[MOTOR][MOTOR_MAX];
        }
        // Motor should not move.
        if (TempBITS == MOTOR_BINTS[MOTOR][BITS_POWERSTOP])
        {
            Varint = motor.data_code[MOTOR][DC_CURRENT_POSITION];
        } else if (TempBITS == MOTOR_BINTS[MOTOR][BITS_FREESTOP])
        {
            Varint = motor.data_code[MOTOR][DC_NEW_POSITION] = Varint;
        }
        return Varint; // Return the altered variable.
    }

5.3.9 ForceClipACK

The ForceClipACK function forces the ACK character to be sent to the user program by placing it in the clipboard and notifying the DOS driver. Useful for lockups or system hangs.

Subjects of significance addressed by this function include

Windows: EmptyClipboard
Causes the clipboard to be cleared.

Windows: SetClipboardData
Fill the clipboard with data of specified type.

Windows: CloseClipboard
C++: SetMemory array

BYTE *SetMemory;
// Pointer to memory for our transmitted clipboard data.

/* FUNCTION: ForceClipACK */
/* INPUTS: HWND hwnd - Handle to current window. */
/* OUTPUTS: N/A */
/* AFFECTED: SetMemory. */
/* PURPOSE: Forces an ACK character to be sent to the DOS driver, in the */
/* event that a lock-up, or system hang, has occurred. */

void ForceClipACK(HWND hwnd)
{
    // Do only if clipboard is open and clip memory has been allocated.
    if (OpenClipboard(hwnd) && hGlobalSet != NULL)
    {
        EmptyClipboard(); // Clear the clipboard.
        SetMemory[0] = 2; // The command byte is the ACK character.
        SetMemory[1] = 0;
        SetMemory[16] = 1; // Error flag.
        SetMemory[17] = 0;
        SetClipboardData(CF_TIFF, hGlobalSet); // Place data on clipboard.
        CloseClipboard(); // Close the clipboard.
    }
}

5.3.10 RTX_ARM_GO

The RTX_ARM_GO function is called whenever the user issues an
ARM_GO command or the user clicks on the RESET button.

It’s primary function is to rearrange the encoder counts found in the
robot registers into numbers which can be interpreted by the robot
control module, situated in the main ToySim module.

This process appears as,

1. All motors are checked whether they lie outside of their
   motor limits and are clipped if they do.
2. We pass control, for the moment, to the AdjustMotorMode
   function to adjust the motor counts for the manual go
   situation.
3. The motor speed is calculated using a complicated
   algorithm
4. The data for each motor is placed on the message queue
   together with the speed, to be picked up by the ToySim
   module
5. The variable ARM_GENERAL_STATUS is adjusted to
   symbolise that at least one motor is moving.

Subjects of significance addressed by this function include

C++: MAX_FACTOR constant array

// TEncoder counts for each motor extreme to extreme.
const WORD MAX_FACTOR[TOTALMOTORS] = 
{4836, 5260, 3554, 2750, 8442, 2142, 1230};

C++: motor structure (already defined)

C++: MotorLimits structure (already defined)
Windows: SendMessage

The algorithm for calculating the speed uses the values in the MAX_FACTOR array, which represent the total encoder counts for each motor.

Firstly, we divide the current motor speed (MOTORSPED) by the value in the MAX_FACTOR array that corresponds to the motor in question.

Secondly, we divide the maximum speed allowed (MAX_SPEED) by the user supplied speed for the motor, OR-ed with a slack variable (SPEED_SLACK). This slack variable ensure that the result will NEVER be an infinite number.

Thirdly, The value for the current position is subtracted from the previous position, to obtain the change in position.

Finally, the motor speed is calculated through the multiplication of the above three numbers, OR-ed with a frame-slack variable (AT_LEAST_FRAMES). This slack variable ensures that there will always be at least AT_LEAST_FRAMES to be rendered, to stop the robot jittering between moves.

The actual conversion of encoder counts for each motor is covered in the source below.

```c
window handle -- calculate motor speed

i<0

case[i]

elbow shoulder zed wrist2 wrist1 yaw grip

SEND SEND SEND SEND SEND SEND SEND

WM_OBJECTMOVEH WM_OBJECTMOVEZ WM_OBJECTMOVEP WM_OBJECTMOVEV

WM_OBJECTMOVEH WM_OBJECTMOVEZ WM_OBJECTMOVEP WM_OBJECTMOVEV

i=i+1

p<6 no

yes

send WM_OBJECTMOVEDO

adjust ARM_GENERAL_STATUS
```

Figure 5.3.10 RTX_ARM_GO.
void RX_ARM_GO(HWND hwnd)
{
    short int i; // Dummy loop variable.
    short int Varint, Elbowint, Tempint; // Temporary value storage.
    WORD N_MOTORSPED; // The new adjusted motor speed.
    float FracMotor, FracDiff, FracSpeed; // Fractions of speed.

    // Go through all available motors.
    for (i = 0; i <= 6; i++)
    {
        // Get the required new position for motor.
        Varint = motor.data_code[i][DC_NEW_POSITION];

        // Adjust it for relative mode, if necessary.
        if (motor.data_code[i][DC_MODE] == RELATIVE_MODE)
            Varint = motor.data_code[i][DC_CURRENT_POSITION];
        else if (Varint > MotorLimits[i][MOTORMAX])
            Varint = (short int)MotorLimits[i][MOTORMAX];
        else if (Varint > MotorLimits[i][MOTORMAX])
            Varint = (short int)MotorLimits[i][MOTORMIN];
        Varint = AdjustMotorMode(_MOTORs, i, Varint); // Adjust for MANUAL_GO.

        // Calculations for obtaining the new motor speed.
        FracMotor = (float)((float)MOTORSPED/(float)MAX_FACTOR[i]);
        FracSpeed = (float)((float)MAX_SPEED/ (float)motor.data_code[i][DC_SPEED][SPEED_SLACK]) -
                    FracDiff = (float)(abs(motor.data_code[i][DC_PREVIOUS_POSITION] - Varint));
        motor.data_code[i][DC_PREVIOUS_POSITION] = Varint;
        N_MOTORSPED = (BYTE)(FracMotor* FracDiff* FracSpeed) AT_LEAST_FRAMES;

        // Process depending on which motor is to be moved.
        switch(i)
        {
            case ELBOW:
                SendMessage(Parenthwnd, WM_OBJECTMOVEH, (WORD)3 |
                            (LPARAM)&(Dummydouble=(double)Varint/(double)ENCODER_ELHOM)*pi_331);
                break;
            case SHOULDER:
                SendMessage(Parenthwnd, WM_OBJECTMOVEH, (WORD)2 |
                            (LPARAM)&(Dummydouble=(double)Varint/(double)ENCODER_SHOULDR*pi_180));
                break;
            case ZED:
                SendMessage(Parenthwnd, WM_OBJECTMOVEH, (WORD)1 |
                            (LPARAM)&(Dummydouble=(double)Varint/(double)ENCODER_ZED));
                break;
            case WRIST2:
                Tempint = (short int)(Varint - motor.data_code[i-1][DC_NEW_POSITION]);
                if (Tempint > Motorlimits[4][0] && Tempint < Motorlimits[4][1])
                    SendMessage(Parenthwnd, WM_OBJECTMOVEH, (WORD)6 |
                                (LPARAM)&(Dummydouble=(double)Tempint/(double)ENCODER_PITC)*pi_102));
                break;
            case WRIST1:
                Tempint = (short int)(Varint +
                                    motor.data_code[i+1][DC_NEW_POSITION]);
                if (Tempint > Motorlimits[3][0] && Tempint < Motorlimits[4][1])
                    SendMessage(Parenthwnd, WM_OBJECTMOVEH, (WORD)5 |
                                (LPARAM)&(Dummydouble=(double)Tempint/(double)ENCODER_EGHz)*pi_131));
                break;
            case YAW:
                Elbowint = motor.data_code[0][DC_CURRENT_POSITION];
                if (Elbowint != motor.data_code[0][DC_NEW_POSITION])
                    Elbowint = motor.data_code[0][DC_NEW_POSITION];
                Dummydouble = -(double)Elbowint/(double)ENCODER_ELHOM)*pi_331/2 +
                                (double)Varint/(double)ENCODER_YAW)*pi_232);
                SendMessage(Parenthwnd, WM_OBJECTMOVEH, (WORD)4 |
                                (LPARAM)&(Dummydouble));
                break;
            case GRIP:
                break;
        }
    }
}
Section 5.3.11 UpdateClipboard

The UpdateClipboard function is where the simulation comes into action. This function reads the data off the clipboard and performs the relevant action or stores values in the robot registers.

The clipboard data is held in the ClipData and ClipKeep variable arrays, which have the format,

- `ClipData[0]` Protocol character. Only perform simulation if this character is the STX constant.
- `ClipData[1]` Command byte. This byte signifies which RTX command is to be simulated.
- `ClipData[3]*256+ClipData[2]` Parameter 1 (integer)
- `ClipData[7]*256+ClipData[6]` Parameter 3 (integer)
- `ClipData[9]*256+ClipData[8]` Parameter 4 (integer)
- `ClipData[11]*256+ClipData[10]` Parameter 5 (integer)
- `ClipData[13]*256+ClipData[12]` Parameter 6 (integer)
- `ClipData[15]*256+ClipData[14]` Parameter 7 (integer)
- `ClipData[17]*256+ClipData[16]` Error code (integer)

For a much better explanation a good look through the source below will suffice.

Subjects of significance addressed by this function include

Windows: Selected Clipboard functions (defined earlier)

C++: SetMemory array (defined earlier)
C++: motor structure (defined earlier)
C++: ARM_MOTOR_STATUS (defined earlier)
C++: ARM_GENERAL_STATUS (defined earlier)

Figure 5.3.11
Update Clipboard.

window handle - Update Clipboard
check format[CF_TIFF]
open clipboard[window handle]
ClipKeep = ClipData - data from clipboard
ClipKeep[0] = STX?

no

yes

case ClipKeep[1]
do operation
place ACK on clipboard
send data
close clipboard

/*=================================*/
/* FUNCTION: UpdateClipboard */
/* INPUTS: HWND hwnd - Handle to the current window. */
/* OUTPUTS: N/A */
/* AFFECTED: ClipHandle, ClipData, ClipKeep, ClipErrorCode, SetMemory, */
/* motor structure, GO_BITS, GO_MODE. */
/* PURPOSE: This function reads data off the clipboard and processes the */
/* information, returning the result back onto the clipboard. */
/* Explanations of the data format can be found in the RTX user's */
/* manual. */
/*=================================*/

void UpdateClipboard(HWND hwnd)
{
    short int i; // Dummy loop variable.
    short int VarInt, DataInt, TempInt; // Temporary value storage.
    short int B0, B1, B2, B3; // Needed for serial port communications.
    WORD TEMP_GO_MODE, TEMP_GO_BITS, TEMP_MOTOR_SPEED;
    // Temporary storage for GO_MODE, GO_BITS and MOTOR_SPEED.

    // Do only if clipboard data signalled as TIFF format.
    if (IsClipboardFormatAvailable(CF_TIFF))
    {
        if (OpenClipboard(hwnd)) // Do only if clipboard is open.
        {
            ClipHandle = GetClipboardData(CF_TIFF); // Obtain handle to clipboard.
            ClipData = (BYTE *)GlobalLock(ClipHandle); // Get the data.
            if (ClipData[0] == 1) // This is the start of transmission character.
            {
                memcpy(&ClipKeep, clipData + 1); // Save the data.
                GlobalUnlock(ClipHandle); // Clear clipboard handle.
                EmptyClipboard(); // Empty the clipboard.
                switch (ClipKeep[1]) // Process depending on the command byte.
                {
                case 0: // ARM_RAW_COMMAND.
                    // Do only if we are actually online.
                    if (hPort != NULL)
                    {
                        PurgeCOM(); // Empty the comm buffer.
                        if ((TempInt == 0) || (TempInt == 1))
                        {
                            /* Do only if the command is valid. */
                            /* Process the command. */
                        }
                        else
                        {
                            /* Error handling. */
                        }
                    }
                }
            }
        }
    }
}
if ((DataInt >= 1) && (DataInt <= 3))
{
  // Send the bytes to the serial port.
  if (WriteBytesToCOM(hwnd, DataInt, B1, B2, B3))
    ClipErrorCode = 0x0011; // Signal error.
  else ClipErrorCode = 0x0053; // Error.
}
else ClipErrorCode = 0x0053; // Error.

// Update view.
TextOutClipWindow(hwnd, "ARM_RAW_COMMAND", "ACK");
break;

case 1: // ARM_RAW_RESPONSE
  // Do only if we are actually online.
  if (hPort != NULL)
  {
    if ((TempInt == 0) || (TempInt == 1))
    {
      // Read bytes from serial port.
      if (ReadBytesFromCOM(hwnd, &B0, &B1, &B2, &B3) == TRUE)
      {
        // Get them ready to be sent back to user.
        SetMemory[15] = HIBYTE(B3);
        SetMemory[14] = LOBYTE(B3);
        SetMemory[13] = HIBYTE(B2);
        SetMemory[12] = LOBYTE(B2);
        SetMemory[11] = HIBYTE(B1);
        SetMemory[10] = LOBYTE(B1);
        SetMemory[9] = HIBYTE(B0);
        SetMemory[8] = LOBYTE(B0);
      }
    }
  }
  else ClipErrorCode = 0x0011; // Error occurred.
  else ClipErrorCode = 0x0053; // Error.

  // Update view.
  TextOutClipWindow(hwnd, "ARM_RAW_RESPONSE", "ACK");
  break;

case 2: // ARM_INIT_COMMS - NOT IMPLEMENTED.
  // Update view.
  TextOutClipWindow(hwnd, "ARM_INIT_COMMS", "ACK");
  break;

case 3: // ARM_VERSION - NOT IMPLEMENTED.
  // Update view.
  TextOutClipWindow(hwnd, "ARM_VERSION", "ACK");
  break;

case 4: // ARM_RESTART.
  // Tell parent to stop moving the robot.
  if (RobotMoving == TRUE)
    SendMessage(Parenthwnd, WM_OBJECTMOVEDONT, 0, 0);
  // Update view.
  TextOutClipWindow(hwnd, "ARM_RESTART", "ACK");
  break;

case 5: // ARM_DEFINE_ORIGIN - NOT IMPLEMENTED.
  // Update view.
  TextOutClipWindow(hwnd, "ARM_DEFINE_ORIGIN", "ACK");
  break;

case 6: // ARM_RELOAD_PIDS - NOT IMPLEMENTED.
  // Update view.
  TextOutClipWindow(hwnd, "ARM_RELOAD_PIDS", "ACK");
  break;

case 7: // ARM_SET_MODE.
  // Change the current GO MODE.
if (Tempint < TOTALMOTORS & Dataint < DC_ACTUAL_POSITION)
    motor.data_code[Tempint][DC_MODE] = Dataint;

    // Update view.
    TextOutClipWindow(hwnd, "ARM_SET_MODE", "ACK");
    break;

case 8: // ARM_STOP.
    // Tell parent to stop moving the robot.
    if (RobotMoving -- TRUE)
        SendMessage{ParentHand, WM_OBJECTMOVED, 0, 0);

    // Update view.
    TextOutClipWindow(hwnd, "ARM_STOP", "ACK");
    break;

case 9: // ARM_GO.
    // Move the robot arm.
    RTI_ARM_GO(hwnd);

    // Update view.
    TextOutClipWindow(hwnd, "ARM_GO", "ACK");
    break;

case 10: // ARM_MOTOR_STATUS.
    // Retrieve the status of a particular motor.
    // Update view.
    TextOutClipWindow(hwnd, "ARM_MOTOR_STATUS", "ACK");
    SetMemory[14] = LOBYTE(ARM_MOTOR_STATUS[Tempint]);
    SetMemory[15] = HIBYTE(ARM_MOTOR_STATUS[Tempint]);
    break;

case 11: // ARM_GENERAL_STATUS.
    // Receive the status of all the motors.
    // Update view.
    TextOutClipWindow(hwnd, "ARM_GENERAL_STATUS", "ACK");
    SetMemory[14] = LOBYTE(MOTOR_GENERAL_STATUS);
    SetMemory[15] = HIBYTE(MOTOR_GENERAL_STATUS);
    break;

case 12: // ARM_READ.
    // Read a particular motor register.
    if (Tempint < TOTALMOTORS & Dataint < DC_ACTUAL_POSITION)
        [motor.current = Tempint;
        motor.data_code[Tempint][Dataint] = Varint;

        // Update view.
        TextOutClipWindow(hwnd, "ARM_READ", "ACK");
        SetMemory[14] = LOBYTE(Varint);
        SetMemory[15] = HIBYTE(Varint);
        break;

case 13: // ARM_WRITE.
    // Write data to a particular motor register.
    if (Tempint < TOTALMOTORS & Dataint < DC_ACTUAL_POSITION)
        [motor.current = Tempint;
        if (Dataint == DC_SPEED)
            if (Varint < 0) Varint = 0;
            else if (Varint > MAX_SPEED) Varint = MAX_SPEED;
        if (Dataint == DC_NEW_POSITION)
            motor.data_code[Tempint][DC_PREVIOUS_POSITION] = motor.data_code[Tempint][DC_ACTUAL_POSITION];
            motor.data_code[Tempint][Dataint] = Varint;
            motor.current.data_code[Tempint] = Dataint;
// Update view.
TextOutClipWindow(hwnd, "ARM_WRITE", "ACK");
}
break;

case 14: // ARM_INTERPOLATE.
// Perform an arm interpolation.
for (i=0; i <= 6; i++)
  motor.data_code[i][DC_NEW_POSITION] = (short int)(
    motor.data_code[i][DC_CURRENT_POSITION] +
    (short int)((ClipKeep[i*2+3]<<8) + ClipKeep[i*2+2]));
  TEMP_MOTORSPEED = MOTORSPEED; // Save current values.
  TEMP_GO_MODE = GO_MODE;
  TEMP_GO_BITS = GO_BITS;
  MOTORSPEED = INTERPOLATE_MOTORSPEED;
  GO_MODE = NUMERIC_GO;
  GO_BITS = 0x1555; // Move all motors.
  RTX_ARM_GO(hwnd);
  GO_BITS = TEMP_GO_BITS; // Retrieve previously saved values.
  GO_MODE = TEMP_GO_MODE;
  MOTORSPEED = TEMP_MOTORSPEED;

// Update view.
TextOutClipWindow(hwnd, "ARM_INTERPOLATE", "ACK");
break;

case 15: // ARM_SOAK - NOT IMPLEMENTED.
// Update view.
TextOutClipWindow(hwnd, "ARM_SOAK", "ACK");
break;

default: // UNKNOWN_COMMAND
// Update view.
TextOutClipWindow(hwnd, "UNKNOWN_COMMAND", "");

if (hGlobalSet != NULL)
{
  SetMemory[0] = 2; // This is the ACK character.
  SetMemory[1] = ClipKeep[1];
  SetMemory[17] = HIBYTE(ClipErrorCode); // The error code
  SetMemory[16] = LOBYTE(ClipErrorCode);
  SetClipboardData(CF_TIFF, hGlobalSet); // Place on clipboard.
  if (ClipSTEP == TRUE) // Do only if in step mode.,
    DebugMsg("continue debug?\n");
}
else GlobalUnlock(ClipHandle); // Release handle to clipboard.
CloseClipboard(); // Close the clipboard.
}

5.3.12 Routines found in WindowProc

The following two routines are executed whenever the main ToySim module signals that a motor has either completed its motion or that a motor is part-way completing its motion.

WM_OBJECTMOVEDONE is used to signal that the motor specified in lpParam has completed its motion so should have its current position register updated.

If the value of lpParam is 65536 then this means that ALL motors have completed their motion, so we should update the RobotMoving variable to FALSE and adjust the ARM_GENERAL_STATUS variable.
WM_OBJECTMOVEPART is used to signal that part of a motion has been completed, with a pointer to a CurrentObjects structure embedded in the Iparam parameter. This message is sent so that we keep our actual position register updated with the most recent number. This does not need to be supported, as it slows down the whole simulation process, but is provided for completeness; and it looks better having the display dynamically updating.

Subjects of significance addressed by this function include

Windows:

**CurrentObjects structure**

```c
#define MAXOBJECTS 16
// Envelope object always LAST object!! i.e. 15!!
struct OBJECTTYPE {
    BOOL ACTIVE; // True if object initialised
    char NAME[32]; // The object file, filename.x
    double IX, IY, IZ; // Starting position of object in x,y,z format
    double PITCH; // Initial pitch angle in degrees
    double BANK; // Initial banking angle in degrees
    double YAW; // Initial yaw angle in degrees
    double SX, SY, SZ; // Initial scaling factors for x,y,z planes
    double JX, JY, JZ; // Point in object where joint can occur, in x,y,z planes
    int LINK; // Object no. to which this object links to (-1=none)
};
typedef OBJECTTYPE OBJECTLIST[MAXOBJECTS]; // List of allowed objects
```

```c
switch(lparam) // Process the result; lparam relates to the motor!
{
    case WM_OBJECTMOVEDON: // Received when a previously instructed robot move has completed.
        // Update the motor variables following the move.
        case 1:
            UpdateMotorsAfterMove(ZED);
            break;
        case 2:
            UpdateMotorsAfterMove(SHOULDER);
            break;
        case 3:
            UpdateMotorsAfterMove(ELBOW);
            break;
        case 4:
            UpdateMotorsAfterMove(YAW);
            break;
        case 5:
            UpdateMotorsAfterMove(WRIST1);
            break;
        case 6:
            UpdateMotorsAfterMove(WRIST2);
            break;
        case 7:
            break;
        case 8:
            UpdateMotorsAfterMove(GRIP);
            break;
        case 65536: // All motors finished moving.
            RobotMoving = FALSE; // No longer moving.
            break;
        default: // Get the current object information.
            memcpy(&CurrentObjects, (LPVOID)lparam, sizeof(_OBJECTOFFSETLIST));
            break;
    }
    break;
    case WM_OBJECTMOVEPART: // Part of a robot move was completed. This could be omitted for an increase in speed, at cost of loss in accuracy.
        // Get the current object information.
        memcpy(&CurrentObjects, (LPVOID)lparam, sizeof(_OBJECTOFFSETLIST));
        break;
    }
    break;
```
motor.data_code[SHOULDER][DC_ACTUAL_POSITION] =  
(short int)((CurrentObjects[2].H*ENCODER_SHOULDER)/pi_180);
// ELBOW.
motor.data_code[ELBOW][DC_ACTUAL_POSITION] =  
(short int)((CurrentObjects[3].H*ENCODER_ELBOW)/pi_331);
// YAW
motor.data_code[YAW][DC_ACTUAL_POSITION] =  
(short int)((CurrentObjects[2].H*ENCODER_YAW/pi_220)* 
((motor.data_code[ELBOW][DC_ACTUAL_POSITION]*pi_331)/(1-ENCODER_ELBOW*2)));
// PITCH, [WRIST1 & WRIST2].
motor.data_code[WRIST1][DC_ACTUAL_POSITION] =  
(short int)((CurrentObjects[5].P*ENCODER_PITCH/pi_102) - 
motor.data_code[WRIST1][DC_ACTUAL_POSITION]);
// ROLL, [WRIST1 & WRIST2].
motor.data_code[WRIST1][DC_ACTUAL_POSITION] =  
(short int)((CurrentObjects[6].P*ENCODER_ROLL/pi_313) + 
motor.data_code[WRIST2][DC_ACTUAL_POSITION]);
// GRIP
motor.data_code[GRIP][DC_ACTUAL_POSITION] =  
(short int)(CurrentObjects[8].Y*60*ENCODER_GRIP);
break;

5.4 Robot specifications

All of the functions necessary for the parsing of the ROBOT.SPEC configuration file can be found in the source files TSParse.h and TSParse.cpp.

We have already discussed at length the concepts behind this module and quoting source code will not make it more exciting. It is simply a very simple text parser module to read in and decipher the ROBOT.SPEC file.

*Remember, of course, that the full source to this and all modules IS supplied with this thesis and should be read to gain a better understanding.*

A reproduction of the ROBOT.SPEC file appears here,
This is a demo ROBOT.SPEC file!

Note: Comment lines begin with a '#'.
Only ONE command per line!

Example INIT block

```c
INIT <block number> ; Identifies the current initialisation block (0..15)
NAME <filename.x> ; The filename of the .x object file
NAME: If you use textures in your object files,
they MUST be a power of 2 in size,
64x64, 128x128, 256x256, etc...
POSN <x> <y> <z> ; The initial x,y,z co-ordinates of the object
PITCH <radians> ; The initial pitch angle (-pi..pi)
BANK <radians> ; The initial banking angle (-pi..pi)
YAW <radians> ; The initial yaw angle (-pi..pi)
SCALE <x> <y> <z> ; Scales the object in any/all of the 3 dimensions
JOINT <x> <y> ; Point inside object where a joint can occur, other
than the pivot. At the moment all objects are
limited to only two joints per object, i.e. the
pivot and one other joint.
LINK <block number> ; Link this objects pivot to the specified objects
joint ... as described by the JOINT command. At
the moment an object may be linked to only one
other object. -1 for no link, i.e. links to the
default frame.
ENDINIT ; Ends the current block.
```

The following are the default values, should any of the above
commands be omitted ...
NAME = "DEFAULT.x"
POSN = (0.0,0.0,0.0)
PITCH = 0.0
BANK = 0.0
YAW = 0.0
SCALE = (1.0,1.0,1.0)
JOINT = (0.0,0.0,0.0)
LINK = -1

Initialisation for the envelope object

```c
INIT -1
NAME ..\objects\x\envelope.x
POSN 0.0 0.0 0.0
SCALE 5.0 5.0 4.0
ENDINIT
```

Initialisation for the robot column

```c
INIT 0
NAME ..\objects\x\column.x
POSN 0.0 0.0 0.0
PITCH 0.0
BANK 0.0
YAW 0.0
SCALE 1.0 1.0 1.0
JOINT 0.0 0.0 0.0
LINK -1
ENDINIT
```

Initialisation for the robot shoulder

```c
INIT 1
NAME ..\objects\x\shoulder.x
POSN -0.18 -0.09 1.03
PITCH 0.0
BANK 0.0
YAW 0.0
SCALE 1.0 1.0 1.0
JOINT 0.0 0.0 0.0
LINK 0
ENDINIT
```

Initialisation for the robot upper-arm

```c
INIT 2
NAME ..\objects\x\upperarm.x
POSN -0.025 0.04 0.55
PITCH 0.0
BANK 0.0
YAW 0.0
SCALE 1.0 1.0 1.0
JOINT 0.0 0.0 0.0
LINK 1
ENDINIT
```
5.5 Robot control

As we continued along programming the simulator it became apparent that facets of the robot control objectives were appearing in other modules. It soon
became obvious that it was going to be impossible to separate out the functions into a module in its own right.

Part of our rules for designing the simulator stated that the simulator was to be a device-independent package, whereby with the alteration of the ROBOT.SPEC file and the inclusion of another module in place of the TSDriver module, we would be able to simulate a completely different robot. In order for this to occur the rest of the package had to remain largely independent and the only way this could be achieved was to place some functions in the TSDriver module and the rest in the ToySim main module.

We have already examined how the TSDriver module addresses the robot control question, as it translates the robot encoder counts and calculates the necessary speed and frame rates. These are then sent via the Windows™ message queue to the ToySim main module, for the rest of the robot control routines to process.

The remainder of the robot control routines, introduced above, are found in the WindowProc function of the ToySim.cpp file.

Subjects of significance addressed by this function include

**Windows:**  
**ObjOffList structure**

```c
struct OBJECTOFFSETTYPE
{  
LPDIRECT3DRMFRAME parent;
// Direct3D pointer to parent object
double X, Xinc, XF;
// X offset, No. of factors, factor offset value
double Y, Yinc, YF;
// Y offset, No. of factors, factor offset value
double Z, Zinc, ZF;
// Z offset, No. of factors, factor offset value
double P, Pinc, PF;
// Heading offset, No. of factors, factor offset value
double B, Binc, BF;
// Bank offset, No. of factors, factor offset value
};
typedef OBJECTOFFSETTYPE OBJECTOFFSETLIST[MAXOBJECTS];
// List of object offsets
```

**Windows:**  
**WM_OBJECTMOVEY system message**

This message is sent by the driver module to signal that a move is desired in the Y direction. The ID number of the object to be moved (the number taken from the ROBOT.SPEC file) is held in the low byte of wparam, and the speed is held in the high byte of wparam. The pointer to the ObjOffList structure element YF, which at the moment represents encoder counts, is held in lparam and so is copied down.

The YF element becomes the factor that is added to the object's current Y value at every frame interval, while the Yinc element denotes the number of frames.
Windows: **WM_OBJECTMOVEZ** system message

This message is sent by the driver module to signal that a move is desired in the Z direction. The ID number of the object to be moved (the number taken from the ROBOT.SPEC file) is held in the low byte of wparam, and the speed is held in the high byte of wparam. The pointer to the ObjOffList structure element ZF, which at the moment represents encoder counts, is held in lparam and so is copied down.

The ZF element becomes the factor that is added to the object’s current Z value at every frame interval, while the Zinc element denotes the number of frames.

Windows: **WM_OBJECTMOVEH** system message

This message is sent by the driver module to signal that a rotation is desired about the H axis. The ID number of the object to be moved (the number taken from the ROBOT.SPEC file) is held in the low byte of wparam, and the speed is held in the high byte of wparam. The pointer to the ObjOffList structure element HF, which at the moment represents encoder counts, is held in lparam and so is copied down.

The HF element becomes the factor that is added to the object’s current H value at every frame interval, while the Hinc element denotes the number of frames.

Windows: **WM_OBJECTMOVEP** system message

This message is sent by the driver module to signal that a rotation is desired about the P axis. The ID number of the object to be moved (the number taken from the ROBOT.SPEC file) is held in the low byte of wparam, and the speed is held in the high byte of wparam. The pointer to the ObjOffList structure element PF, which at the moment represents encoder counts, is held in lparam and so is copied down.

The PF element becomes the factor that is added to the object’s current P value at every frame interval, while the Pinc element denotes the number of frames.

Windows: **WM_OBJECTMOVEB** system message

This message is sent by the driver module to signal that a rotation is desired about the B axis. The ID number of the object to be moved (the number taken from the ROBOT.SPEC file) is held in the low byte of wparam, and the speed is held in the high byte of wparam. The pointer to the ObjOffList structure element BF, which at the moment represents encoder counts, is held in lparam and so is copied down.

The BF element becomes the factor that is added to the object’s current B value at every frame interval, while the Binc element denotes the number of frames.
at the moment represents encoder counts, is held in lpam and so is copied down.

The BF element becomes the factor that is added to the object's current B value at every frame interval, while the Binc element denotes the number of frames.

**Windows:** **WM_OBJECTMOVEDO system message**
This message is received after the driver module has received the RTX ARM_GO command. This routine simply accepts this message and initiates the system timer to interrupt at every frame interval, defined by the value in the variable ObjectDelay.

**Windows:** **WM_OBJECTMOVEDONT system message**
This message is received to instruct us to stop the movement of the robot immediately. Here, we simply reset the ObjOffList structure variables Xinc, Yinc, Zinc, Hin, Pinc, Binc to zero. This will take effect the next time the timer interrupts, whereby our associated routine, explained below, will deduce that the motion has been completed and stop the arm.

**Windows:** **WM_TIMER system message**
This is the message that is received every time the system timer interrupts, as specified by the ObjectDelay variable.

The routine updates the elements in the ObjOffList structure, and decrements the increment counters (Xinc, Yinc, etc.). When one of these increment counters goes to zero, this motor is deduced to have completed its motion and so the driver module is notified. Also, if all counters are zero, the driver is notified that all motion has stopped.

The routine uses the DirectX™ functions of SetRotation, SetPosition and SetOrientation to prepare the objects to be moved the next time the Render() function is called.

**Figure 5.5**

<table>
<thead>
<tr>
<th>WM_OBJECTMOVEA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=Y,Z,H,P or B</td>
</tr>
<tr>
<td>ObjOffList[object number].AF=ObjOffList[object number].A</td>
</tr>
<tr>
<td>ObjOffList[object number].Ainc=speed</td>
</tr>
<tr>
<td>ObjOffList[object number].AF=ObjOffList[object number].AF/speed</td>
</tr>
</tbody>
</table>
Section 5.0 - Page 111

Figure 5.5.1

WM_TIMER

\[ \text{WM TIMER} \]

### WM_TIMER

\[ \text{iC=0} \]

\[ A=Y.Z,H,P \text{ or } B=\begin{cases} \text{injected} & \text{ObjOffList[i].Ainc<=0} \\ \text{true} & \text{otherwise} \end{cases} \]

\[ \text{ObjOffList[i].A} \leftarrow \text{ObjOffList[i].A} + \text{ObjOffList[i].AF} \]

\[ \text{ObjOffList[i].Ainc} \leftarrow \text{ObjOffList[i].Ainc} - 1 \]

\[ \text{Y or Z - SetPosition} \]

\[ \text{H,P or B} \leftarrow 0 \]

\[ \text{SetOrientation} \]

\[ \text{H,P} \text{ or } \text{B} \leftarrow 0 \]

\[ \text{SetRotation} \]

\[ \text{H,P} \text{ and } \text{B} \leftarrow 0 \]

\[ \text{Move} \]

---

**case WM_OBJECTMOVEY:** // The object needs to be moved in the Y direction.

\[ \text{ObjectNum} = \text{wparam} \& 0xff; \]

\[ \text{wparam} = \text{wparam} >> 8; \]

\[ \text{// Only do if object number is valid and object is active.} \]

\[ \text{if} \begin{cases} \text{ObjectNum < MAXOBJECTS} & \& \text{ObjectList}[(\text{ObjectNum})].ACTIVE \end{cases} \]

\[ \text{// Get object information.} \]

\[ \text{memcpy}(&\text{ObjOffList}[(\text{ObjectNum})].YF, \text{LPOID} \text{lparam, sizeof(double)}); \]

\[ \text{// Calculate the increments needed to accomplish the move.} \]

\[ \text{ObjOffList[ObjectNum].Y} \leftarrow \text{ObjOffList[ObjectNum].Y} + \text{wparam}; \]

\[ \text{ObjOffList[ObjectNum].Yinc} \leftarrow \text{wparam}; \]

\[ \text{ObjOffList[(ObjectNum)].Y} /= \text{(double)} \text{wparam}; \]

\[ \text{break;} \]

**case WM_OBJECTMOVEZ:** // The object needs to be moved in the Z direction.

\[ \text{ObjectNum} = \text{wparam} \& 0xff; \]

\[ \text{wparam} = \text{wparam} >> 9; \]

\[ \text{// Only do if object number is valid and object is active.} \]

\[ \text{if} \begin{cases} \text{ObjectNum < MAXOBJECTS} & \& \text{ObjectList}[(\text{ObjectNum})].ACTIVE \end{cases} \]

\[ \text{// Get object information.} \]

\[ \text{memcpy}(&\text{ObjOffList}[(\text{ObjectNum})].ZF, \text{LPOID} \text{lparam, sizeof(double)}); \]

\[ \text{// Calculate the increments needed to accomplish the move.} \]

\[ \text{ObjOffList[ObjectNum].Z} \leftarrow \text{ObjOffList[ObjectNum].Z} + \text{wparam}; \]

\[ \text{ObjOffList[ObjectNum].Zinc} \leftarrow \text{wparam}; \]

\[ \text{ObjOffList[(ObjectNum)].Z} /= \text{(double)} \text{wparam}; \]

\[ \text{break;} \]

**case WM_OBJECTMOVEH:** // The object needs its heading changed.

\[ \text{ObjectNum} = \text{wparam} \& 0xff; \]

\[ \text{wparam} = \text{wparam} >> 9; \]

\[ \text{// Only do if object number is valid and object is active.} \]

\[ \text{if} \begin{cases} \text{ObjectNum < MAXOBJECTS} & \& \text{ObjectList}[(\text{ObjectNum})].ACTIVE \end{cases} \]

\[ \text{// Get object information.} \]

\[ \text{memcpy}(&\text{ObjOffList}[(\text{ObjectNum})].HF, \text{LPOID} \text{lparam, sizeof(double)}); \]

\[ \text{// Calculate increments needed to accomplish the move.} \]

\[ \text{ObjOffList[ObjectNum].H} \leftarrow \text{ObjOffList[ObjectNum].H} + \text{wparam}; \]

\[ \text{ObjOffList[ObjectNum].Hinc} \leftarrow \text{wparam}; \]

\[ \text{ObjOffList[(ObjectNum)].H} /= \text{(double)} \text{wparam}; \]

\[ \text{break;} \]

**case WM_OBJECTMOVEP:** // The object needs its pitch changed.

\[ \text{ObjectNum} = \text{wparam} \& 0xff; \]

\[ \text{wparam} = \text{wparam} >> 8; \]

\[ \text{// Only do if object number is valid and object is active.} \]

\[ \text{if} \begin{cases} \text{ObjectNum < MAXOBJECTS} & \& \text{ObjectList}[(\text{ObjectNum})].ACTIVE \end{cases} \]

\[ \text{// Get object information.} \]

---
memcpy(&ObjOffList[ObjectNum].PF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].PF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].BF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].BF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].PF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].PF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].BF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].BF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].PF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].PF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].BF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].BF = (double)wparam;

case WM_OBJECTMOVETO: // The object needs to change its banking value.  
ObjectNum = wParam & 0xff; // Obtain the object number.
wparam = wParam>>8; // Obtain the speed.  
// Only do if object number is valid and object is active.  
if (ObjectNum < MAXOBJECTS & ObjectList[ObjectNum].ACTIVE)  
// Get object information.
memcpy(&ObjOffList[ObjectNum].BF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].BF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].PF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].PF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].BF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].BF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].PF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].PF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].BF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].BF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].PF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].PF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].BF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].BF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].PF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].PF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].BF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].BF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].PF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].PF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].BF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].BF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].PF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].PF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].BF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].BF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].PF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].PF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].BF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].BF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].PF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].PF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].BF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].BF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].PF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].PF = (double)wparam;

// Get object information.
memcpy(&ObjOffList[ObjectNum].BF, (LPVOID)lpam, sizeof(double));

// Calculate increments needed to accomplish the move.
ObjOffList[ObjectNum].BF = (double)wparam;
5.6 Robot extras

Here we look at some extra functions that were added along the way to either improve the performance of the simulator or improve its appearance.

5.6.1 The SNAP function

The SNAP function is implemented as a dialog box, which alters the snap setting used in moving the pointer object. The SNAP setting simply decides the accuracy of the pointer's movement. The higher the value the less accurate the movement, but the faster it will move.

SNAP is accessed via the main menu,
/* FUNCTION: SnapDlgProc */
/* INPUTS: HWND win - Handle to current window. */
/* unsigned msg - Returned windows message. */
/* WORD wparam - WIN95 variable. */
/* LONG lparam - WIN95 variable. */
/* OUTPUTS: BOOL - TRUE if message valid. */
/* AFFECTED: TIPScale. */
/* PURPOSE: Function for the 'Snap' dialog box. Just processes the messages*/
/* that are directed towards this dialog box, such as the OK or */
/* CANCEL buttons being clicked, or the Snap UP/DOWN buttons being*/
/* clicked. Consequently, the TIPScale variable is altered! */

BOOl FAR PASCAL SnapDlgProc(HWND win, unsigned msg, WORD wparam, LONG lparam)

1param = lparam; // Dummy assign.

switch(msg) // Check system message for compliance with accepted value.
{
    case WM_INITDIALOG: // Dialog window first opened.
        TempTIP = (int)TIPScale; // Backup the Scale value.
        sprintf(TIPScaleString, "%d to 1 ratio\0", TempTIP); // Show it.
        SetDlgItemText(win, IDC SNAPEDIT, TIPScaleString);
        return TRUE; // All OK so far.
        break;

    case WM_COMMAND:// Command message received.
        switch(wparam) // Process this message.
        {
            case IDOK: // OK button clicked.
                TIPScale = (D3DVALUE)TempTIP; // Re-assign TIPScale.
                break;

            case IDCANCEL: // CANCEL button clicked.
                PostDialog(win, TRUE); // Close dialog window.
                return TRUE; // All OK.
                break;

            case IDC SNAPBUTTON1: // SNAP button UP clicked.
                if (TempTIP < 100) TempTIP += 1; // Maximum value is 100.
                sprintf(TIPScaleString, "%d to 1 ratio\0", TempTIP); // Show it.
                SetDlgItemText(win, IDC SNAPEDIT, TIPScaleString);
                return TRUE; // All OK.
                break;

            case IDC SNAPBUTTON2: // SNAP button DOWN clicked.
                if (TempTIP > 1) TempTIP -= 1; // Minimum value is 1.
                sprintf(TIPScaleString, "%d to 1 ratio\0", TempTIP); // Show it.
                SetDlgItemText(win, IDC SNAPEDIT, TIPScaleString);
                return TRUE; // All OK.
                break;
        }
        break;

    return FALSE; // Unknown message received!
}

5.6.2 The Dynamic Trails module

This module can be best explained by the following screen dump,
It's purpose is to plot a continuous line corresponding to the movement of the robot tip in the relevant axis plane, while allowing the user to alter the pen colour at any time during plotting.

The functions which are of relevance to us here are

- **ShowFrontPlane**
- **ShowSidePlane**
- **ShowTopPlane**

```c
/** FUNCTION: ShowFrontPlane */
/** INPUTS: HWND win - Handle to current window. */
/** WORD CenterX, CenterY - 'Raw' co-ordinates for plot. */
/** OUTPUTS: N/A */
/** AFFECTED: N/A */
/** PURPOSE: Displays the plot associated with the FRONT-VIEW, originating from the center co-ordinates (Cx, Cy). */
/** */
void ShowFrontPlane(HWND win, 
                    WORD CenterX, 
                    WORD CenterY)
{
    HDC hdc; // Handle to current device context.
    int Cx, Cy; // Processed co-ordinates for plot.

    hdc = GetDC(win); // Get current device context.

    Cx = TSVector.X*ScaleFactorX + CenterX; // 'Cook' the center co-ordinates.
    Cy = CenterY - TSVector.Y*ScaleFactorY; // Plot only if center co-ordinates within allowable region.

    if (Cx > CenterX-CENTER_WIDTH && Cx < CenterX+CENTER_WIDTH &&
        Cy > CenterY-CENTER_HEIGHT && Cy < CenterY+CENTER_HEIGHT)
    {
        // Change to current drawing colour.
        SelectObject(hdc, hDTrailsPens[DTailPenColor]); // Select object.
        Rectangle(hdc, Cx, Cy, Cx+PEN_THICK, Cy+PEN_THICK); // Plot 'dot' onto screen.
        DeleteObject(hDTrailsPens[DTailPenColor]); // Release colour-object.

        // Again, change to current drawing colour.
        SelectObject(hdc, hDTrailsPens[DTailPenColor]); // Select object.
        Rectangle(hdc, Cx, Cy, Cx+PEN_THICK, Cy+PEN_THICK); // Plot 'dot' onto video swap buffer.
        DeleteObject(hdc, hDTrailsPens[DTailPenColor]); // Release colour-object.
    }

    ReleaseDC(win, hdc); // Release device context handle.
}
```

```c
/** FUNCTION: ShowSidePlane */
/** INPUTS: HWND win - Handle to current window. */
/** WORD CenterX, CenterY - 'Raw' co-ordinates for plot. */
/** OUTPUTS: N/A */
/** AFFECTED: N/A */
/** PURPOSE: Displays the plot associated with the SIDE-VIEW, originating from the center co-ordinates (Cx, Cy). */
/** */
/** NOTE: Identical to previous function except in the way the Cx & Cy */
/** ------ variables are 'cooked'. */
/** */
void ShowSidePlane(HWND win, 
                   WORD CenterX, 
                   WORD CenterY)
{
    HDC hdc; int Cx, Cy;

    hdc = GetDC(win);

    Cx = TSVector.X*ScaleFactorX + CenterX; // ... Note the differences here!
    Cy = CenterY - TSVector.Y*ScaleFactorY;

    if (Cx > CenterX-CENTER_WIDTH && Cx < CenterX+CENTER_WIDTH &&

```
5.6.3 About and Display Envelope

These two facets of our simulator don’t really warrant a section of their own, suffice to say that we have an ‘about’ box which accessed via the main menu,
The envelope display is simply a 3D object which was specified in the ROBOT.SPEC file. The actual range checking of the working envelope is performed in the driver module, via the function TIPinEnvelope. 

```c
// FUNCTION: TIPinEnvelope
// INPUTS: D3DVECTOR point - The current (X,Y,Z) of the pointer.
//         D3DVECTOR EnvVector - The envelope vector.
//         char *CoOrdString - Variable containing the pointer co-ord
// OUTPUTS: bool - TRUE if pointer inside envelope.
// AFFECTED: FutureVector.
// PURPOSE: This function uses the values in EnvVector and point to decide
//          whether the pointer is inside the envelope. If it is, the
//          function returns TRUE, and FutureVector is updated, as well as
//          CoOrdString, which forms part of the main window title.

bool TIPinEnvelope (D3DVECTOR point, D3DVECTOR EnvVector, char *CoOrdString)
{
    // Radius from pointer to (0,0,0).
    double Pradius = sqrt((point.x*point.x)+(point.z*point.z));
    // Radius of envelope to (0,0,0).
    double Eradius = sqrt((EnvVector.dvX*EnvVector.dvX)+(EnvVector.dvZ*EnvVector.dvZ));
    char S1[5], S2[5], S3[5]; // Temporary strings for CoOrdString.
    // Initialise CoOrdString with default value.
    strcpy(CoOrdString, ".. Pointer= (X: E, Y: E, Z: E)\0");
    if ((point.y < 0 || point.y > EnvVector.y) || point.z > -0.05)
    {
        return FALSE; // Pointer outside of envelope.
    }
    if (Pradius <= Eradius) // Pointer inside envelope.
    {
        // Adjust FutureVector to suit.
        FutureVector[0] = point.x*SCREEN_TO_RTX_X;
        FutureVector[1] = point.z*SCREEN_TO_RTX_Z;
        FutureVector[2] = point.y*SCREEN_TO_RTX_Y;
        FutureVector[3] = Pradius*SCREEN_TO_RTX_R;
        // Convert values to strings.
        itoa((int)FutureVector[0], S1, 10);
        itoa((int)FutureVector[1], S2, 10);
        itoa((int)FutureVector[2], S3, 10);
        strcpy(CoOrdString, ".. Pointer= (X: ");
        strcat(CoOrdString, S1);
        strcat(CoOrdString, ", Y: ");
        strcat(CoOrdString, S2);
        strcat(CoOrdString, ", Z: ");
        strcat(CoOrdString, S3);
        strcat(CoOrdString, "\0");
        return TRUE; // All OK!
    }
    else
    {
        return FALSE; // Error occurred.
    }
} 
```

The envelope will resemble the following screen dump, when activated,
Figure 5.6.3.3
Envelope Display.
6.0 Verification

To date in this document we have introduced theory, spoken about current application technologies and implemented our code. At this time we come to the final testing phase of the project.

We call this the ‘final’ phase because throughout the development of the software we have been testing the individual modules, to check for functionality and bugs, but it is now that we must declare the software a fully working solution. We achieve this by introducing the programs tested with our software, their results and whether or not the actual output coincided with the expected output. By stating exactly what is tested and the expected results, we provide a basis for future development and debugging.

6.1 Verification expectations

Before entering into this project we first decided that this package should be aimed directly at students studying robotics and so should be able to simulate, at least, all of the companion programs that come bundled with the RTX robot.

The rationalisation behind this concept is that we expect the bundled programs to show the complete range of properties of the robot. Thus, if our simulator can simulate all of the bundled programs then our simulator can be regarded as functionally equivalent to the RTX; out-of-the-box at least.

It is impossible to be 100% certain to the functionality of our software. Every piece of software goes through various stages of testing, usually starting at the alpha stage, then following on to the beta stage and then, hopefully, the final production stage.

Our simulator package can be thought of as being at the beta stage of testing. That is, for complete testing the package must be installed at a number of sites being used in everyday situations, attempting a range of operations, for a period of time that we see as sufficient to ‘draw out’ the bugs.

6.2 The companion programs

In this subsection we name the companion programs bundled with the RTX and examine the expected results and the steps followed in the testing phase for a particular program.

The programs that will be tested are

- START.COM
6.2.1 START.COM

Perhaps not surprisingly, this program is incompatible with our software. When it is first executed it overwrites the vector at interrupt 78H, leaving our inserted interrupt, for our DOS™ to Windows™ driver, inoperative.

This program will wreak havoc in simulating the robot and should be avoided.

6.2.2 INIT.COM

The purpose of the INIT.COM program is to initialise the robot for use and should be executed at least once every time the robot is first switched on.

The relevance of such a program to our package is limited, as we can simply initialise our robot by clicking on the RESET button in the Windows™ driver window. However, the INIT.COM program introduces factors such as MANUAL_GO motor movements and range checking, which are not usually addressed through simpler programs, which, more often than not, utilise the NUMERIC_GO method of motor control.

After a successful run, the computer screen should resemble the following screen dump,
This program is non-interactive, in that it does not accept any user input.

A typical program run would involve the following actions taking place:

(i) The ZED axis is initialised by moving the shoulder up to its limit, whereby the axis is reset.
(ii) The gripper is then closed.
(iii) The arm is first bent slightly to the left at the elbow and then the wrist is rotated so that the gripper is pointing down.
(iv) The shoulder is then moved to its limit, on its left.
(v) The elbow is moved to its limit by rotating towards the center of the robot. That is, rotating to the robot’s right.
(vi) The Z axis is once again checked.
(vii) Finally, the robot arm is moved to its initialised position of a straight, horizontal arm.

6.2.3 TEST.COM

The TEST.COM program achieves what its name suggests; it is a program to test all facets of the RTX robot arm.

When first executed, the screen should resemble,
This menu links to other sub-menus, by pressing the respective key and then hitting the enter key. As we explain the menu options, the words (SIM OK!) appended to a menu item means that this option is supported by the simulator and simulated perfectly.

Through option 1 we are presented with another menu,

1. Initialisation, etc (SIM OK!)
2. Mode Control (SIM OK!)
3. Motor Stops (SIM OK!)
4. Turn Motor Drivers ON (SIM OK!)

Option 1 allows us to initialise the robot, 2 allows us to change the mode control (force, position, etc.), 3 defines stop modes (dead stop, free stop, etc.), 4 turns motors on.

Through option 2 we are presented with another menu,

1. Elbow (SIM OK!)
2. Zed (SIM OK!)
3. Wrist-1 (SIM OK!)
4. Wrist-2 (SIM OK!)
5. Yaw (SIM OK!)
6. Gripper (SIM OK!)

This allows us to read and write the IP variables associated with each motor in the menu.
Through option 3 we are presented with another menu,

1. Manual Control (SIM OK!)
2. Numeric Control (SIM OK!)

These options allow the user to initiate a manual go, or numeric go. That is, allows the user to make the arm move by pressing the keypad keys or by entering a position count.

Through option 4 we are presented with another menu,

1. General Status (SIM OK!)
2. Motor status (SIM OK!)

Allows the user to view either the general status or the status of a particular motor.

Through option 5 we are presented with another menu,

1. Elbow (SIM OK!)
2. Zed (SIM OK!)
3. Wrist-1 (SIM OK!)
4. Wrist-2 (SIM OK!)
5. Yaw (SIM OK!)
6. Gripper (SIM OK!)

This menu allows us to select which motor we will be performing interpolation on. Another menu will appear with variables such as acceleration time and the like. Remember we mentioned the problems in calculating acceleration. As such, this simulation will not be an accurate representation.

Through option 6 we are presented with another menu,

1. Initialise (SIM OK!)
2. Soak
3. Initialise & Soak
4. Stop Soak

From this menu we can initialise and soak test the robot arm. Notice that options 2 to 4 are not supported in the current version of our simulator. This has no bearing on program development, with the simulator returning a valid code, to make the calling program believe that the action actually took place.
Through **option 7** we are presented with another menu,

1. **Send 1 Byte** *(SIM OK!)*
2. **Send N Bytes** *(SIM OK!)*
3. **Send 1 Command** *(SIM OK!)*
4. **Send N Commands** *(SIM OK!)*

This menu gives us the option of sending raw commands and then receiving a raw response. Please note that in the version of this program tested here (version number given in screen dump) a bug exists, in that it will keep reading raw responses even after the valid number of responses have been read. This bug does not show up when using on the real robot. This just shows sloppy programming by the programmer who originally coded the TEST.COM program.

### 6.2.4 DRIVE.COM

The DRIVE.COM program presents us with a menu allowing us to change the encoder counts relating to each motor and then to perform the movement. The screen should resemble,

![Figure 6.2.4 DRIVE.COM Screendump.](image)

This program was, in fact, the program used throughout the construction of the simulator. Its operation is well-assured, with the expected results occurring without flaw.
6.2.5 DEMO.COM

The DEMO.COM program was just added to see the result of a real-life application, rather than just a test program. This program ran flawlessly in the test, and simulates the towers of Hanoi program even down to the gripper opening and closing, expecting a piece of the puzzle to be there.

6.3 Custom Programs

In order to test fully the process of interpolation and raw communications, we introduce two custom programs, programmed in PASCAL, entitled Interpolate and RawCommunications.

The source files for these two programs are listed here, together with a brief explanation.

6.3.1 Interpolate program

Interpolation represents a unique situation in that it allows us to move one or a number of motors by some increment. This has the effect of allowing us to configure factors such as acceleration and path generation.

The following commented source code is an example of interpolation, whereby it simply sends some arbitrary number to the robot for us to see what actually is happening.

PROGRAM Interpolate;

{*******************************************************************************
FILE: Interpol.pas
AUTHOR: Anthony Rodriguez,
ECU STUDENT No. 0943335
DESCRIPTION: This file is simply a demonstration program for use in testing the
interpolation function of the ToySim robot simulator.

It simply passes an interpolation array to the simulator to process and waits
for a key to exit.
*******************************************************************************}

USES dos, crt;
\$1 ARMP.INC \\

(System variables.)

Var
   OK : Integer;
      (Used for RTX error checking.)
   InterData : I_DATA_T;
      (This is the interpolation array.)
   Ch : Char;
      (Used in reading keyboard input.)

\[\text{**=============================**}\]
\text{*/
\text{FUNCTION: LIBRARY INSTALLED */
\text{** INPUTS: N/A */
\text{** OUTPUTS: Boolean - True if library installed. */
\text{** AFFECTED: OK. */
\text{** PURPOSE: Simple function to initialise communications with the robot and check if the library is installed. */
\text{**=================================**}\}

Function LIBRARY INSTALLED : Boolean;

Begin
   ARM_INIT_COMMS (TOGGLE_OFF, 0, OK);  
      (Initialise communications.)
   If OK = -1
      Then LIBRARY INSTALLED := False  
         (Library not installed.)
   Else Begin
      LIBRARY INSTALLED := True;  
         (Library IS installed.)
      If OK = ARM_RESTARTED
         Then Begin
         Write ('Arm restarted');
         OK := 0;
         ARM_RESTART(OK);  
            (Restart the arm.)
      End;
   End;

End;

\[\text{**=================================**}\]
\text{*/
\text{PROCEDURE: Main procedure */
\text{** INPUTS: N/A */
\text{** OUTPUTS: N/A */
\text{** AFFECTED: OK, InterData structure. */
\text{** PURPOSE: This is the main procedure, which initialises the robot and then keeps passing an interpolation array to */
\text{**=================================**}\}
Begin
ClrScr;
Normvideo;
Writeln;
Writeln(' U.M.I. RTX Robot Arm ');
Writeln(' ==================== ');
Writeln(' Interpolation Program Demo');
Writeln;
Lowvideo;

If Not LIBRARY_INSTALLED
Then Begin
Write ('^G');
Writeln ('The library must be');
Writeln ('installed with the');
Writeln ('START command for this');
Writeln ('program to run. ');
End
Else Begin
Writeln('Interpolating');
Writeln('Press Any key to exit');

{Fill the interpolation array with}
{some dummy data, just to see }
{something happening on screen. }
OK := 0;
InterData[ELBOW] := 0;
InterData[SHOULDER] := 0;
InterData[ZED] := -8;
InterData[YAW] := 0;
InterData[WRIST1] := 0;
InterData[WRIST2] := 0;
InterData[GRIP] := 0;
Ch := #0;
While Ch = #0 Do
Begin
ARM_INTERPOLATE(InterData, OK);
{Send to robot.}

If Keypressed
Then Ch := Readkey;
{Wait for a keypress.}
End;
Writeln('Done.');
End;

Writeln;
End.
6.3.2 RawCommunications program

This program was designed to test the ARM_RAW_COMMAND and ARM_RAW_RESPONSE functions inside the simulator.

As we know, we gave our simulator the ability to communicate with the real robot through the RTX functions of ARM_RAW_COMMAND and ARM_RAW_RESPONSE. As we noted, performing communications this way meant that the program would run 'normally' on the real robot, but then on the simulated robot would provide some way of exchanging information between the real robot and the simulated robot.

Introducing robot communications meant we had a new set of problems to deal with and subsequently test. Now we are dealing with serial communications, having to take notice of such things as timing, parity, error checking and protocols. For this reason we must have a dedicated test program.

The commented source code below simply describes a way of reading the current position from the real robot and then transferring the information to the simulated robot, so that the simulated robot can mimic the movements of the real robot.

Program RawCommunications;

{******************************~***********~**************}
{ *
FILE: RawComm.pas
AUTHOR: Anthony Rodriguez,
ECU STUDENT No. 0943335
DESCRIPTION: This file is simply a demonstration program for use in testing the raw communications capabilities of the ToySim robot simulator.

It simply reads the position of the motors from the real robot and then passes on these values to the simulated robot, which mimics the real robot's movements.
*
{******************************~***********~**************}

Uses Dos, Crt;

{$I ARMP.INC} {The RTX robot include file.}

[Constants that apply to the RTX command bytes.] Const

R_TOGGLE_OFF = $28;
{Turn IP toggling off.}
R_TOGGLE_ONCE = $29;
    {Toggle the IP once.}

R_REQUEST_ID = $01;
    {Get IP number.}

R_RESET_IP = $20;
    {Reset the current IP.}

R_STOP_ALL = $a0;
    {Stop all motors.}

R_GO_ACTIVE = $00;
    {Go active.}

R_DEFINE_HOME = $21;
    {Define current position as home.}

R_IMM_READ = $40;
    {Code for an immediate read.}

R_ALL_GO = $1555;
    {Code for all motors moving.}

{Various variables used in this program.}
Var
  Ch  : Char;
    {Used to determine if a key was pressed.}

  OK  : Integer;
    {Used by RTX for error signalling.}

  BytesRead  : Integer;
    {Number of bytes read from robot.}

  B1, B2, B3  : Integer;
    {Temp storage for R/W bytes.}

  SP, EP, ZP, YP : Integer;
    {Temp storage for motor values.}

*----------------------------------------------------------------------------*/
/* PROCEDURE: RawInitialise */
/*  INPUTS: N/A */
/*  OUTPUTS: N/A */
/*  AFFECTED: B1, B2, B3, OK */
/*  PURPOSE: This procedure performs an */
/*  initialisation of the simulated */
/*  robot via raw commands. Rather than */
/*  storing the command bytes in an */
/*  array, we have left them as separate*/
/*  commands to make it easy to read and*/
/*  understand. */
Procedure RawInitialise;

Begin

{Initialise error.}
OK := 0;

ARM_INIT_COMMS(TOGGLE_ON, 0, OK);
{Do conventional reset.}
ARM_RESTART(OK);

{Perform an initialisation using RAW methods.}
ARM_RAW_COMMAND(0, 1, R_TOGGLE_OFF, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_TOGGLE_ONCE, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_TOGGLE_OFF, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_REQUEST_ID, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_RESET_IP, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_STOP_ALL, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_TOGGLE_ONCE, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_GO_ACTIVE, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_TOGGLE_ONCE, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_RESET_IP, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_STOP_ALL, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_TOGGLE_ONCE, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_REQUEST_ID, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_TOGGLE_OFF, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_DEFINE_HOME, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_TOGGLE_ONCE, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_TOGGLE_OFF, 0, 0, OK);
ARM_RAW_COMMAND(0, 1, R_DEFINE_HOME, 0, 0, OK);

{Make sure we are currently on IP0.}
ARM_RAW_COMMAND(0, 1, R_REQUEST_ID, 0, 0, OK);
ARM_RAW_RESPONSE(0, BytesRead, B1, B2, B3, OK);
If B1 <> 33
  Then
    ARM_RAW_COMMAND(0, 1, R_TOGGLE_ONCE, 0, 0, OK);
End;

{==================================--==================*/
/* PROCEDURE: Main procedure */
/* INPUTS: N/A */
/* OUTPUTS: N/A */
/* AFFECTED: B1, B2, B3, SP, EP, ZP, YP, OK. */
/* PURPOSE: This is the main procedure, which */
/* calls Rawinitialise and then enters */
/* a loop whereby it is constantly */
/* polling the robot for changes in its */
/* position, sending the data back to */
/* the simulator for interpretation. */
/*==================================--==================*/

Begin
Clrscr;
WriteLn ('Raw communications demo,');  
WriteLn ('Press any key to exit. ');  

RawInitialise;  {Perform initialisation.}

Ch := #0;
While Ch = #0 Do
  Begin
    [Read RAW ELBOW motor counts.]  
    ARM_RAW_COMMAND(0, 1, R_IMM_READ, 0, 0, OK);

    ARM_RAW_RESPONSE(0, BytesRead, B1, B2, B3, OK);
    EP := -(B3 Shl 8)+B2;

    [Read RAW SHOULDER motor counts.]  
    ARM_RAW_COMMAND(0, 1, R_IMM_READ Or 1, 0, 0, OK);

    ARM_RAW_RESPONSE(0, BytesRead, B1, B2, B3, OK);
    SP := -(B3 Shl 8)+B2;

    [Read RAW ZED motor counts.]  
    ARM_RAW_COMMAND(0, 1, R_IMM_READ Or 2, 0, 0, OK);

    ARM_RAW_RESPONSE(0, BytesRead, B1, B2, B3, OK);
    ZP := -(B3 Shl 8)+B2;
{Read RAW YAW motor counts.}
ARM_RAW_COMMAND(0,1,R_IMM_READ Or 3,0,0,OK);

ARM_RAW_RESPONSE(0,BytesRead,B1,B2,B3,OK);
YP := -(B3 Shl 8)+B2;

{Write out motor counts to the simulator.}
ARM_WRITE(ELBOW, NEW_POSITION, EP, OK);
ARM_WRITE(SHOULDER, NEW_POSITION, SP, OK);
ARM_WRITE(ZED, NEW_POSITION, ZP, OK);
ARM_WRITE(YAW, NEW_POSITION, YP, OK);

{Move the arm.}
ARM_GO(NUMERIC_GO, R_ALL_GO, OK);

If Keypressed
Then Ch := Readkey;
{Wait for any key to be pressed.}
End;
End.
7.0 Conclusion

This thesis has seen us journey from first principles, through to theory and finally to implementation and verification of the ToySim project. Through the introduction we assigned ourselves the tasks we wished to complete and throughout the project’s life have endeavoured to meet and fulfil those tasks. We’ve seen the project grow from a set of objectives into a fully-functional robot simulator that not only meets the objectives but, in some cases, surpasses them.

We remember the objectives we first decided on, before beginning the project, as

- A graphical display
- Immediate motion
- Indirect motion
- Robot specifications
- Robot control

Taking a look back at the sections making up this thesis we remember that in section 1.0 we introduced the objectives, in section 2.0 we embarked on a literature search, looking for ideas and alternate methods of satisfying the objectives and in Section 3.0 we once again introduced the above objectives, but this time in abstract terms, whereby we attempted to apply theory and programming methods. Section 4.0 saw us finally choose the platform the project would be based on, and saw us delve quite deeply into the inner-workings of DirectX™ and Direct3D™. Implementation of the project was undertaken in section 5.0, where we analysed important functions and quoted from the supplied source code listing, following on to section 6.0 where we performed verification, by being able to simulate the programs supplied with the RTX robot and also our own programs.

At this point we draw near to the completion of the project and so can reflect on what we’ve attempted and what we have actually achieved.

- The graphical display was originally defined in the introduction as,

  ... a 3D environment (i.e. 3 Dimensional environment), allowing the display of 3D objects in true 3D space. It should have the capability of translation, rotation, scaling, and animation within the 3D space, and should also show where the user is, within the 3D space, at any time, preferably using the easily recognisable (X,Y,Z) Cartesian coordinate format.

We accomplished the establishment of the 3D environment through DirectX™ and Direct3D™ and through the ObjMenu module we were able to, at the click of a mouse button, manipulate the robot in 3 dimensions, providing the facets of translation, rotation and scaling.
With the addition of new concepts, such as the pointer and envelope display, we were able to discover a method of obtaining the world coordinates of some object referenced to the object coordinates of the pointer, and apply these to the envelope display for range checking.

- **Immediate motion** was originally defined in the introduction as,

  ... the ability to manipulate the simulated robot in an 'immediate' fashion, so as to be able to program it and see the results immediately following.

  ... should appear to the user as an information-rich graphical display, and allow the user to specify where they want the robot to go.

This was achieved through the ObjMenu module, which we programmed to construct a window with buttons representing the movements of pan up, pan down, pan left, pan right, zoom in, zoom out, rotate up, rotate down, rotate left, rotate right, rotate left diagonal and rotate right diagonal. We also provided the display of the current Y and Z values, together with the zoom factor and the heading, pitch and banking values.

- **Indirect motion** was originally defined in the introduction as,

  ... the ability to manipulate the simulated robot in an 'indirect' fashion, so as to be able to program it and see the results immediately following.

  ... to satisfy the previous objective the user need not have a formal program procedure, whereas this is required to satisfy (this) objective ...

  ... an information-rich graphical user interface should appear and be a transparent interface between a user designed program and the simulator, acting on the commands received, within the graphical space.

  ... unobtrusive and not affect the user's program in any way. (ie. it would be ideal if the same program that ran with the hardware robot ran with the simulator, without alteration or recompilation).

We saw this objective come to fruition through the development of the TSDriver module. Through this module we were able to simulate the 'real' RTX robot and, with the introduction of our clipboard handling functions, were able to interface with DOS and ultimately the user's program, to exchange information and complete the simulation cycle.
We also saw the benefits of introducing new functions, such as the **step button**, for debugging users' programs, and the **ACK button** to prevent lockups.

- The *Robot specifications* objective was addressed via the use of a script language, organised into a configuration file called `ROBOT.SPEC`, prompted by the definition of *robot specifications* given in the introduction, which read,

  ... the simulator should be as device-independent as possible, allowing for the future injection of extra modules, for purposes such as simulating different types of robots, and for more manipulation tools.

  ... the simulator should be configurable, allowing a number of different robots to be implemented by simply providing a new set of design rules and 3D object modules;

We went through the process of identifying what exactly we needed to know about our robot and its constituent objects, using the information to design and implement the script language. We have intentionally left this module open, to facilitate the injection of new commands and data types into the script language, for future robot updates.

- *Robot control* was originally defined as,

  ... able to use the information entered by the user, either indirectly or immediately, to control the robot arm.

Starting off as a module in its own right we soon observed that it would be impossible to achieve this, so we facilitated the metamorphosis of the robot control system to a decentralised system of independent robot-specific routines which, when combined, formed a complete robot control system.

Looking at the above statements we can conclude that we have indeed met our objectives and therefore qualify for ending our role in this project.

Now that we have completed our project we must turn to future developments and implementations. To this end, we have supplied with this thesis a CD-ROM comprising all of the files required to compile and build the ToySim project, including all the object files and texture maps, and also the DirectX™ 5.0 software development kit, which of course must be installed before any development takes place.

For detailed information, please read the **README.TXT** file on the CD-ROM.
8.0 References


http://ourworld.compuserve.com/homepages/stefan_anton/