Robotic Haptics: Retrofitting a Pick and Place Manipulation Arm to Haptic Input Device

A thesis presented in partial fulfilment of the

requirements for a degree of

Master of Engineering

Mechatronics

at

Massey University,

Albany, New Zealand

Courtney C. de Lautour

2009

Masters Abstract

Robotic haptics has been and continues to be an area of intense research, primarily in medical and exploration industries. This is due to an ability to provide high data throughput between human and machine. In medical applications, it is possible to detect and compensate errors such as a hand tremor in a surgeon. It is possible to apply scaling factors to assist in microsurgery situations, and can allow leading experts to perform procedures from anywhere on the globe.

As part of a collaboration to develop a robotic method of femur fracture realignment between Auckland University, Auckland District Health Board, and Massey University, the project seeks to provide a haptic driven HMI for the realignment system.

To reduce construction required, an existing manipulation arm (Mitsubishi RV-M1) is used as the hardware interface device. A new motor controller is designed to provide additional functionality as the standard controller provides no force control or real-time feedback of position.

A software interface is developed (using version 3 of the C# programming language, developed by Microsoft, and version 3.5 of the Microsoft .NET Framework) with the ultimate specification of becoming being the primary interface platform for the realignment system. The interface has been implemented to the point of providing a simulated environment for the haptic device.

It was found that the configuration of the RV-M1 provides a tight area of high dexterity as a haptic device, and as such, similar kinematic configurations are poor candidates for practical implementation. The implication of which, is that a new manipulator should be designed which grants a larger volume of high dexterity space.

Acknowledgments

I would like to thank everyone who has helped me though this project. I would firstly like to thank my partner Monika Szejna, who has provided me with support and assistance in whatever forms necessary, whenever I have required it. Next I would like to thank my family for providing me with encouragement and financial support throughout university, and for making sure I believed I could achieve whatever I put my mind to.

To the Massey University staff of the engineering department and my supervisors, Prof. Peter Xu, and Dr. Johan Potgieter whom deserve a big thank you for their help and guidance over the course of the project, and Gordon Hain who has provided me with invaluable help and expertise with electronics design.

Finally I would like to thank my friends and fellow students, Lim Kang, Baden Rielly, Daniel and Mathew Plane, Greg Hancock, and Arek Szejna; whom allowed me to discuss problems with them, provided another view point on solutions, and helped me relax when things weren't going right.

Table of Contents

List of Figures

List of Tables

List of Code Snippets

Chapter 1: Introduction

Project Background and Objectives

The project has consisted of developing a six axis haptic device from an existing manipulator (Mitsubishi RV-M1). This device acts as a Human Machine Interface (HMI), enabling the user to input motions to a computerised simulation. In return, the simulation is able to provide resistance forces to the user as a form of feedback.

The primary design purpose of the haptic device is to enable a surgeon to perform a simulated long bone realignment procedure before it is carried out on the patent via a Stuart Platform manipulator. “Haptics is the science and technology of experiencing and creating touch sensations in human operators” (Siciliano & Khatib, 2008). By using a haptic input device, it is possible to relay information in a more natural way between the surgeon and robotic interface, thus increasing reliability and speed.

Placing an additional layer between operator and patient, reduces implications of fatigue, grants higher manipulation control, repeatability and accuracy (Tavakoli, Patel, Moallem, & Aziminejad, 2008). Through the implementation of a femur realignment simulation, it is possible to provide multiple visual aids and statistics, such as tissue stress relative to a safe threshold depending on; patent age, gender, and health conditions. This allows the surgeon to predetermine any problematic areas within the procedure and perform counter-measures without endangering the health of the patient.

The haptic interface is part of a larger body of work, the Robotically Enhanced Femur Re-Alignment (RIFRA) project, conducted by collaboration between Auckland University, Massey University and the Auckland District Health Board. The interface is intended to provide a HMI platform from which Femur Fracture and Muscle Interaction Simulation, Stuart Platform Control, and Fracture Imaging, can easily be accessed and utilised.

During the project, it became apparent that there are many issues involved in developing a comprehensive haptic input system. The most significant of these is caused from the forward kinematic layout implemented. This design introduces many areas of operation where correct force feedback cannot be produced requiring an approximation. Due to the nature of the operational environment, accuracy must be placed at a premium and further research should be carried out into developing a manipulator with higher dexterity.

As the project evolved, it became apparent that the existing manipulator controller was insufficient and it would be required for a custom controller to be designed. Unfortunately due to time constraints, this was unable to be physically implemented.

Objectives

The primary objective of the project is to develop a three-dimensional input device to be used for remote re-alignment of long bone fractures. This consists of the following sub-objectives;

- Modify the RV-M1 to enable force detection capabilities.
- Interface the modified RV-M1 with a standard desktop computer.
- Model the physical characteristics of the RV-M1 so environmental forces can be counteracted.
- Develop a user interface to link the haptic input device, long-bone manipulation platform and simulation of soft tissues (within the leg) into a concise application.

Thesis Outline

Chapter 2 introduces and provides a background of the technologies used within the project. It provides examples of real world uses for haptic interfaces.

Chapter 3 describes the software interface which was designed for the project, and requirements of developing a user friendly environment, including the implementation details of rendering three-dimensional objects in real-time.

Chapter 4 details the implementation of simulating a custom haptic device, including forward and reverse kinematics.

Chapter 5 provides details on the design of a new controller for an existing robotic manipulator, adding functionality that is required for providing force feedback, and finally;

Chapter 6 provides a conclusion and possible areas of further work.

Chapter 2: Literature Review

Haptic Exploration

"Haptic exploration is a mechanism by which humans learn about the surface properties of unknown objects. Through the sense of touch, we are able to learn about attributes such as object shape, surface texture, inertia, and stiffness" (Jarvis & Zelinsky, 2003). Haptic interfaces are related to the ability of humans to perform haptic exploration. These virtual haptic environments would ideally feel identical to real world environments; however, limitations in hardware and modelling prevent a perfect representation (Jarvis & Zelinsky, 2003).

Despite the current limitations with haptic technologies, research has sought to provide another channel of sensory feedback that may enhance user interfaces. This feedback includes force feedback; a presentation of forces to the user, and tactile feedback (through the use of vibrotactile stimuli), which provides the user with a sense of surface texture (Jacko & Sears, 2003).

A challenge to haptic feedback is the tendency for visual feedback to dominate other senses. In cases where visual and haptic feedbacks do not match, performance of the user decreases (Jacko & Sears, 2003). However, studies where haptic and visual textures match, show (that by the addition of haptic feedback), performance increased (Jacko & Sears, 2003; Gerovichev, Marayong, & Okamura, 2002). This requires that care be taken to provide non conflicting information to the user via separate sensory channels.

Haptics Consumer Products

Touch has been described as the most profound of the senses and is becoming a must have feature for high-tech consumer devices such as mobile phones. One of the most well known products of today, the iPod, utilises touch in the form of its touch sensitive scroll wheel. It is believed that the success of the iPod has "triggered a user-interface revolution" (Paterson, 2007).

Haptic controllers have been provided for video games for a substantial period of time, in the form of force feedback joysticks and steering wheels. Vibrotactile controllers are now commonplace among gaming console controllers, initially released with Sony's PlayStation 2. They are now prevalent among major consoles such as PlayStation 3, Microsoft's Xbox 360, and Nintendo's Wii. These controllers provide haptic feedback in the form of vibrotactile stimuli and input via the users movement of the controller. As such, these haptic controllers offer a more active exploration to users, not just to see objects, but to feel and interact with them (Paterson, 2007; Bicchi, Buss, Ernst, & Peer, 2008).

Advanced Haptic Interfaces

PHANToM

The PHANToM haptic interface is produced by SensAble technologies. This device is able to measure a users' finger tip position and exert a precisely controlled force vector on the fingertip. As many haptic interactions, using the finger does not involve significant torque. A three-DOF gimbal has been used, allowing the user's finger to assume any comfortable position. Since the three rotations about the finger are neither measured nor actuated by the PHANToM, the fingertip can be modelled as a point, or frictionless sphere within a virtual environment (Massie & Salisbury, 1994).

Figure 1: PHANToM Haptic Device

The PHANToM has been designed such that the transformation matrix between motor rotations and end point is near to diagonal. When the motors are decoupled, they produce a desirable amount of back-drive friction and inertia. The PHANToM is also statically balanced.

This means there is no requirement to compromise dynamic range by applying biasing torques (Massie & Salisbury, 1994).

Delta.6

Force Dimension produce a number of haptic devices. One of which is the Delta.6. This device offers six active degrees-of-freedom in both translation and rotation. It uses a parallel design layout which allows a large range of forces and torques over a larger workspace than non parallel devices.

Figure 2: DOF.6 DELTA Haptic Device

Omega.6

Figure 3: Omega.6 Haptic Device

Also manufactured by Force Dimension is the Omega.6. This device has a pen form input and can provide either passive or force feedback modes in six DOF. It features full gravity compensation; however this is at the cost of dynamic range. The range of motion from the Omega.6 is designed to be close to that of human wrist movements.

Medical Haptics

Training

Training surgical residents can add a substantial cost to health care, some of which can come from inefficient use of operating room time and equipment. Residents are currently trained using a variety of methods, which include working on plastic models, cadavers, and live animal and human patients (Wnek & Bowlin, 2008; Batteau, Liu, Maintz, Bhasin, & Bowyer, 2004). It is more likely that a new surgeon will make an error compared to an expert, which may have dire economical, legal and societal costs (Wnek & Bowlin, 2008).

Medical simulators take inspiration from that used by airline and military aviation industries to train residents in a virtual reality environment. This enables training in difficult scenarios or on difficult anatomy without endangering lives. In addition, simulations are able to provide a structured level of working, providing levels of difficulty (Wnek & Bowlin, 2008; Batteau, Liu, Maintz, Bhasin, & Bowyer, 2004), and a means of scoring how well a resident has preformed (Wnek & Bowlin, 2008). This also reduces the requirement on operating on animals and cadavers which provide both ethical and financial benefits (Wnek & Bowlin, 2008).

Simulation

The simulation typically utilises both specialised hardware and software components. While low end simulations only employ position sensors, there are advanced haptic interfaces. These provide force feedback as the virtual tools interact with the simulated medium (Wnek & Bowlin, 2008).

Soft tissues are simulated through the use of spring-mass and finite element models (Wnek & Bowlin, 2008; Batteau, Liu, Maintz, Bhasin, & Bowyer, 2004). For haptic solutions to exhibit high fidelity, it is generally accepted that an update rate of at least 1000Hz is required. This leads to using simplified models to achieve the required update rate (Batteau, Liu, Maintz, Bhasin, & Bowyer, 2004). These models are considered an oversimplification of soft tissues since they do not represent the non-linear, heterogeneous, viscoelastic behaviour exhibited in soft tissues (Wnek & Bowlin, 2008).

Remote Surgery

Reviews of surgical robotics and their roll in presently available configurations demonstrates that a surgeon can safely be removed from the immediate surgical local and still maintain interaction (Wnek & Bowlin, 2008).

At present, broad range implementation is prohibited due to a lack of supporting technologies, thus increasing setup and operational time as a result of complex interfaces. These interfaces make it is difficult to perform simple modifications to the system such as moving the robot or changing surgical tools (Wnek & Bowlin, 2008).

Manipulation Arms

Robotic manipulators are serial, multiple link, kinematic chains, where each lower-pair joint is powered via a motor. This manipulator is a unique development in Man's history in that it is a general-purpose tool. When coupled with sensors and an information processing computer, the resulting system can embody the characteristics that distinguish man from the rest of the animal kingdom (Ellery, 2000).

To function to maximum effect, a manipulator must be able to provide six degrees of freedom (DOF) in both position and orientation. This requires that there must be a minimum of six joints present to produce any arbitrary position. It is possible to de-couple the first three-DOF into arm position variables and the final three into an orientating wrist. This allows for decomposition of the six-DOF problem into two, three-DOF problems which reduce computational complexity (Ellery, 2000).

Three Dimensional Graphics APIs

There are a number of libraries or Application Programming Interfaces (APIs), which specifically act as a low-level interface for three dimensional rendering, typically accelerated via dedicated 3-D hardware. These can be thought of as a mediator, allowing the application and the graphics device to communicate smoothly (Luna, 2006; Chen, 2003). Each library provides a defined set of interfaces and functions which are exposed to the application/programmer. These represent the entire feature set in which the current version of the API is capable of supporting. This should not be interpreted, that because the API is capable of a feature, the installed hardware supports it (Luna, 2006; Chen, 2003).

The libraries communicate with the Graphics Processing Unit (GPU) via another layer known as the Hardware Abstraction Layer (HAL). The HAL is a set of device-specific instruction sets which perform common operations. This prevents the library from having to know specific details of a graphics device, and it can be developed independently of hardware devices (Luna, 2006).

Direct3D 9 was chosen for use with the haptic simulator over other APIs listed below. This was due to prior experience of using an earlier version of the API, reducing the ramp up time

associated with learning a new API. In addition, a managed wrapper (Managed DirectX (MDX)), is provided for version 9, dramatically increasing productivity by allowing the interface to be designed in a managed language (C#).

OpenGL

OpenGL is the most widely adopted device-independent 3-D Graphics API. OpenGL was originally developed by SGI, which built on its already existing device dependent Graphics Library (GL). The technology standard is now controlled by the OpenGL Architecture Review Board (ARB) with input from leading computer soft/hardware companies (Chen, 2003).

Mesa

Originally developed by Brian Paul, Mesa is designed to simulate OpenGL functions on UNIX platforms which did not support OpenGL. Mesa is currently the most convenient OpenGL API on Linux and is an open software implementation for learners to study (Chen, 2003).

Direct3D

Direct3D forms part of Microsoft's DirectX API. It is the de facto standard API for use on Windows systems. It is comparable to OpenGL in terms of feature sets. It is widely supported by hardware graphics card vendors and is developed to work with hardware many years before it is released (Luna, 2006; Chen, 2003).

QuickDraw3D

QuickDraw3D is a library developed by Apple Computer, implemented on top of QuickTime by Apple Computer (Chen, 2003), It was originally designed for Macintosh computers, however was delivered as a cross-platform system. QuickDraw3D was abandoned in 1998, when Apple announced that future 3-D support would be based on OpenGL.

PHIGS / GKS-3D

These are standards which were defined within the 1980's. There have been some high-level graphics packages developed using PHIGS or GSK-3D. Many of the functions used by OpenGL and Direct3D have evolved from these early standards (Chen, 2003).

Graphics Shaders

Shaders were developed by computer graphics developers who discovered that photo-realistic rendering has so many variables that it cannot be expressed as a simple set of equations and fixed functions. Due to this, developers needed a way to fully control the rendering process so they could implement complex algorithms, which are required for realistic renderings. One of the first architectures, RenderMan, was developed by Pixar Animation Studios (St-Laurent, 2004).

RenderMan

RenderMan is a standard that was developed in the early 1980's. It specifies an information exchange scheme which allows compatibility between 3-D authoring software and the actual renderer. Along with this interface, it defines a programmable approach to materials, which allows developers to create functions specifying how to render a surface (St-Laurent, 2004).

Although RenderMan is only a standard, Pixar has developed a software render (PhotoRealistic RenderMan (PRMan)) based on it. Through the use of PRMan, Pixar proved the viability of shader rendering and as such, has been used in large volumes of computer generated movies. RenderMan was never designed to be used for real-time rendering and lends itself to rending movie-grade graphics (St-Laurent, 2004).

Due to the success of RenderMan and the flexibility it provided, hardware makers wanted to include the same flexibility for real-time, hardware-accelerated graphics. However, due to a lack of processing power, it was not feasible to implement a shader approach and a fixed-

function pipeline was required. As advancements in silicon chip increased performance, hardware implemented shaders became a possibility (St-Laurent, 2004).

The release of Microsoft's DirectX 8.0 Software Development Kit (SDK) hailed the arrival of hardware-accelerated shaders. nVidia released a GPU which supported the vertex and pixel 1.1 shader standard. This early standard had limited functionality and didn't include conditional statements or looping (St-Laurent, 2004).

Today, processing power has enabled shaders to take leaps and bounds, and is now demonstrating near photorealistic real-time rendering. Beyond the scope of this thesis, it may be possible to incorporate advanced shader based rendering techniques to the user interface, allowing a more detailed and natural view to surgeons using the application.

Chapter 3: Software Interface

Graphical User Interface (GUI) Design Principles

One of the most important features of any user presented software is the user interface. This interface has to enable the user to access the information required in a natural way. To reduce 'clutter', the GUI should make commonly used controls easily accessible and hide those not used until required. This can be achieved through the use of separate 'advanced' dialog controls (Johnson, 2000).

It is also possible to develop intuitive methods for highlighting data. For instance, *Figure 4* shows the simulation of the RV-M1. With all information displayed at once, it is very difficult to determine what is actually being displayed. *Figure 5* displays the same configuration, however, hiding information which is not relevant to the kinematic linkage selected. This makes it is easier to interpret the information being displayed, which is: the axis of rotation, gravitational torque (which must be countered), torque being applied from the joint, and forces exerted by the end effecter.

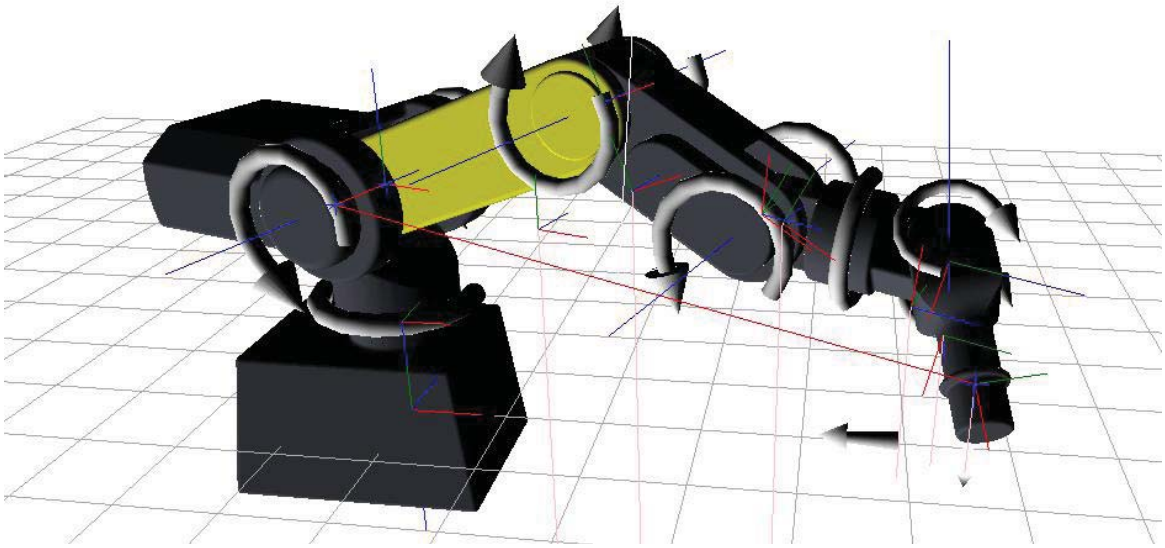


Figure 4: Cluttered Display

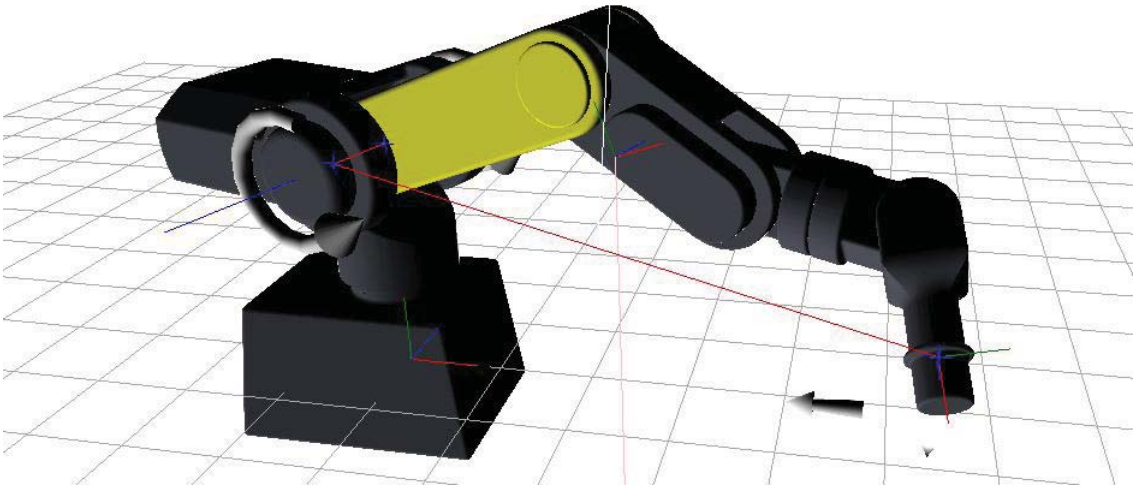


Figure 5: Clean Display

One of the most overlooked aspects of GUI design is that of the user's eye movements. Interfaces which require excessive eye movement will generate fatigue. The main cause of this is placing two important displays, which must be continuously monitored a large distance away from each other. This causes the user to have to continually 'search' for the visual area in which information is placed, placing a requirement on the eyes to perform many large saccades. It is also helpful to guide the eye within the viewport to important areas; this can be done through the use of colour, animation and intuitive placement of controls. Similarly it is important to not distract the user. Where animation is required, motions should remain consistent to avoid drawing the eye when the user's attention is placed elsewhere within the interface.

A method to keeping data displayed concisely, is taken from the military in the form of a Heads-Up-Display (HUD); where important information is overlayed, often semi-transparent, over the main view port. This approach has been implemented in the LONGBONE GUI and can be seen in *Figure 5* as only information relevant to the highlighted section of the manipulator is displayed.

LONGBONE Interface Brief

The primary LONGBONE interface is designed to have minimal controls on-screen at any given time and focuses on intuitive use. This is achieved through the use of Hot-ToolHUDs, hotkeys, and four main panels; which can be seen from the design sketch *Figure 6*. These panels display related information in a concise and intuitive fashion. All panels are resizable to allow maximum flexibly to the user. In addition, each panel (with the exception of the graphic panel) can be quickly minimised or maximised.

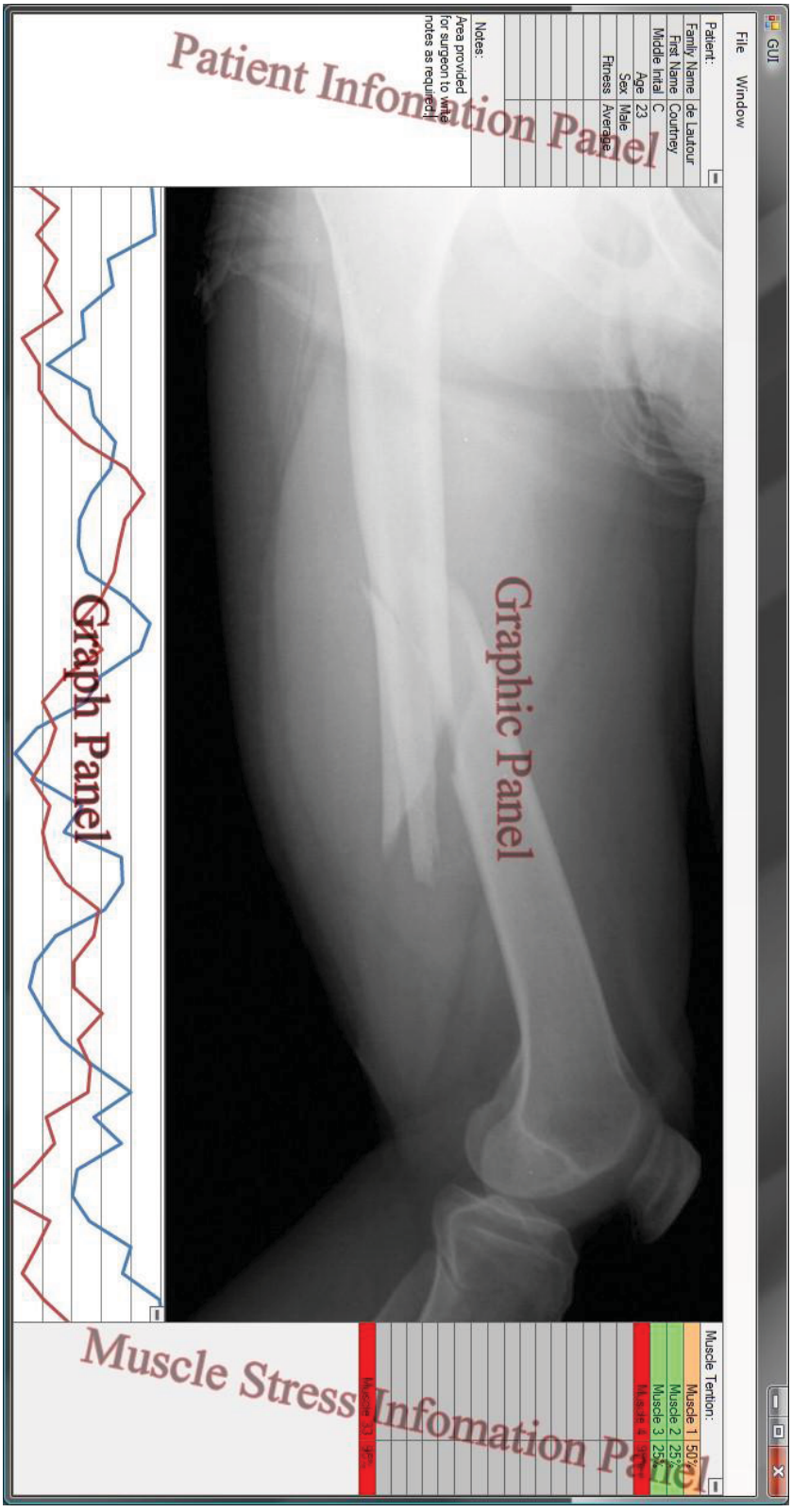


Figure 6: Proposed Primary Interface

Hot-ToolHUDs

Hot-ToolHUDs provide the user with actions related to a specific panel. They are accessed through placing the mouse cursor over the desired panel and holding SPACE. The Hot-ToolHUD will become visible near the cursor. Releasing SPACE or invoking an action will hide the ToolHUD.

Rather than display all tools in a linear strip, the Hot-ToolHUD arranges tools in a circular fashion about a central point. Where possible, the HUD centres about the mouse cursor when it is invoked. This is so that the user has minimal distance required to move the mouse to any given action. In cases where centring the HUD over the mouse would result in part of the HUD being 'off-screen', it is simply shifted towards the centre of the display such that it remains completely visible.

Due to the circular layout scheme of the Hot-ToolHUD, radial positions vary depending on the number of tools displayed on the HUD. As there are some common actions between panels (such as minimise), these actions are kept as close to the same radial position as possible to reduce searching on the users behalf.

Patient Information Panel

The patient information panel displays all personal data about the patient which may be of use during the surgery; it provides no method to modify this, as this data will likely be drawn from the health database. However, it is modifiable in the interim through use of a secondary window.

The Patient Hot-ToolHUD has two actions:

Reduces the panel to its minimum size. The panel becomes a thin strip large enough to house a maximise button, effectively removing it from the screen.

Figure 7: Hot-ToolHUD for Patient Info Panel

Opens the 'modify patient data dialog'. This should be removed once the system is linked into a health database.

Muscle Stress Panel

Muscle tensions are displayed in numerical form via the Muscle Stress Information Panel. The panel provides a grid view of muscles and stress each is subjected to. The panel can display stress data either as maximum reached during the operation, stress at a given point within the simulation procedure, or real-time during the procedure. Safe limits of stress within each muscle are determined from patent demographics and statistical models; the stress of each muscle is then compared with this limit, and colour coded for quick optical recognition. Muscles are able to be selected for use with the graphing panel and can be quickly added to the graph via the Hot-ToolHUD.

The Muscle stress Hot-ToolHUD has four actions:

Reduces the panel to its minimum size.

Figure 8: Hot-ToolHUD for Muscle Stress Panel

Sorts the list of muscles by name alphabetically. Invoking the action when already sorted A to Z will reverse the order to become Z to A.

Sorts the list based on percentage of stress a muscle can safely handle. Invoking the action when already sorted will reverse the order placing most critical values first.

Places selected muscles onto the graphing panel.

Graph Panel

In order to provide a graphical representation of muscle stress and movement operations, a graphing panel is implemented. Each visual representation of a parameter being graphed will be known as a 'trace'.

The left hand side (LHS) of the graph has a list of the three axis of force and three moments of torque which have been recorded for the Stuart Platform to perform during the procedure. These are toggle-able in selection and when toggled on, the related force/moment will be displayed on the graph.

The graphing panel enables the user to view the most number of critical muscle stresses, with the number displayed remaining constant. However, different muscles displayed across the graph vary depending on stress level. In synergy with the graphing panel, this provides the ability to modify control trajectories in non real-time. This allows the surgeon to analytically smooth motions and refine the path to generate an optimal trajectory after they have been recorded using the haptic device.

The panel can be panned and scaled along both the x and y axis. Panning is achieved via holding ALT+LEFT MOUSE and dragging; scaling is implemented in a similar fashion except the key combination is ALT+RIGHT MOUSE. An auto-fit feature is invoked by ALT+DOUBLE CLICK; the auto-fit feature will centre the graph on the selection and scale accordingly. Should no trace be selected, either in full or partially, the graph will fit to all displayed traces.

Trace Selection/ De-selection

Traces displayed on the graph can be selected through use of the drag select tool implemented. The tool is invoked by holding the CTRL button while the mouse is over the panel. The tool is designed to act intuitively and has two modes:

Mode 1: The user 'Draws' by holding the LEFT MOUSE button - the tool is not cancelled by releasing the button. Any traces which are drawn over will have the selection status inverted. In cases where a trace is crossed multiple times, the section in-between crossings will be inverted. Where the trace is drawn over a non-even number of times, the final crossing will invert the entire section. This is visualised in *Figure 9*, where a previously unselected trace is selected, in part, as the user draws over it a number of times (selected regions of the trace are outlined).

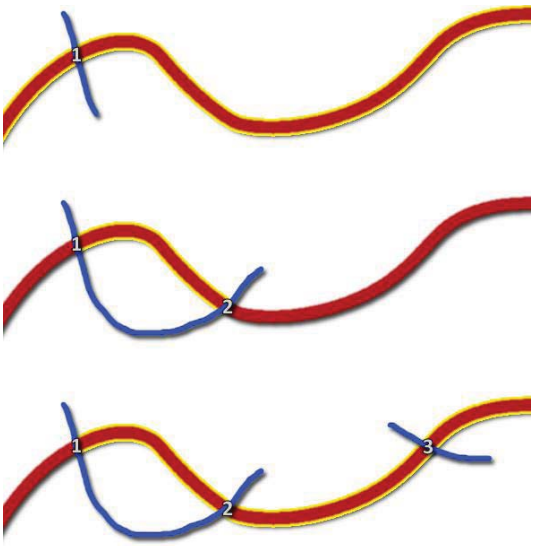


Figure 9: Trace Selection

Mode2: The user drags a horizontal band by holding the RIGHT MOUSE button. Multiple bands can be drawn by releasing then re-holding the mouse button; any section of a trace within a band will be selected. It is not possible to select an entire trace (without dragging along the entire length) using this mode, as there will always be an even number of crossings between the bands and trace.

The Graph Hot-ToolHUD has five actions:

Figure 10: Hot-ToolHUD for Graph Panel

Reduces the panel to its minimum size.

Cancel Selected: Removes selected traces from the graph.

Display Critical: Toggles Critical display mode, this mode will graph a number of muscles which have the highest stress/safe stress ratio. The muscles displayed do not remain constant across the graph.

Smooth: Smooths the selected region, through an implementation of a moving average algorithm.

Graphic Panel

The most used panel is the graphic pane, and as such is the most complex. In addition to the Hot-ToolHUD, the panel can be split into four displays by tapping SPACE. Each of these sub-panes acts as an independent camera on the environment Figure 11, enabling the surgeon to have multiple viewpoints simultaneously. Each pane can be configured to display in one of six orthographic modes; Top, Bottom, Left, Right, Back and Front, or free perspective view. Tapping space while the mouse is over any sub-pane will expand it to fill the panel enabling a more detailed display.

This panel will be able to display a number of items, including rendered 3-D placement of the bone and simulated muscle, real-time fluoroscopy images, and still x-rays. These are controlled via the Hot-ToolHUD.

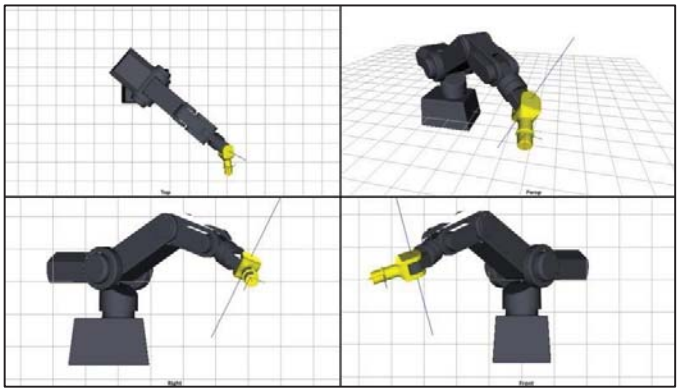


Figure 11: Four Independent Displays

Similar to the graph panel Hot-ToolHUD, the graphic HUD has five actions.

Reduces the panel to its minimum size.

Toggles display of reconstructed 3-D Bone.

Toggles display of simulated muscles.

Toggles display of X-Ray: Will disable fluoroscopy display if active.

Toggles display of fluoroscopy: Will disable X-Ray display if active.

Figure 12: Hot-ToolHUD for Graphic Panel

Three Dimensional Rendering

Model Data

In order to display the simulation data, a three dimensional display is utilised. This requires the implementation of a number of hardware and software algorithms. First we must understand the principles behind 3-D computer graphics. There are two main methods of describing objects within a scene:

Solid-Body or volumetric

Solid bodies are traditionally used for non graphical purposes. This is because it is generally too slow to render, preventing real time displays. They are typically used for mathematical simulations where the internal structure of the body is deforming, such as stress analysis. Another application is medical imaging, such as Computed Axial Tomography (CAT) scans.

Shell or Boundary

These models are in effect hollow. They consist of an infinitesimally thin shell which is rendered to the user. Since models are (in the general case) closed volumes, it appears that they are solid, since one cannot see within the shell. Data for the model comprises of points, known as vertices, which have been sampled around the boundary of the object, creating an approximation of the surface. The point cloud of vertices is typically referred to as the model mesh for pure polygon rendering or the control cage when Non-Rational Uniform B-Spline (NURBS) patches are used.

These vertices must then be interpolated to define the surface between them; this can be done via, or simple linear interpolation or implementation of a NURBS patch. This patch requires a number of points to define a surface. The surface can then be sampled at intervals along it to provide a tessellation into polygon primitives, which then can be rendered via linear interpolation. Figure 13 shows a NURBS patch which has been tessellated into triangular polygons for rendering via linear interpolation.

NURBS provides the advantage that smoothness of objects can be modified simply by changing the sampling interval, however tessellation is currently CPU bound which in many cases draws processing time away from other tasks required. The use of NURBS modelling is rare within real-time rendering applications due to a lack of hardware support. However some next generation GPUs which are currently in development will have NURBS support.

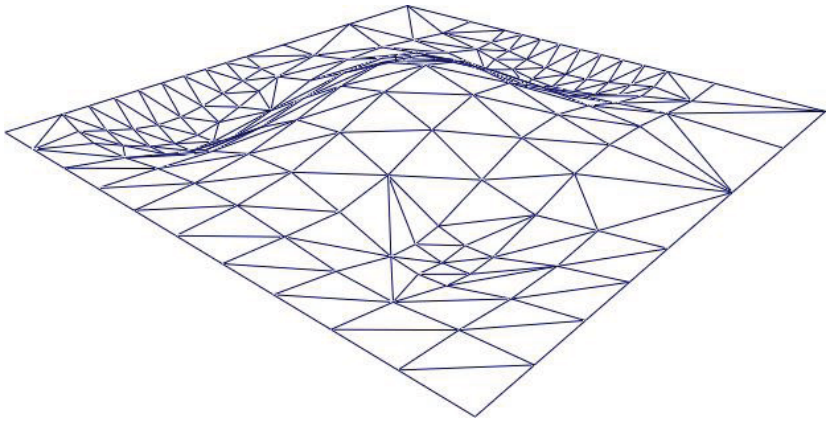


Figure 13: Tessellated NURBS Patch - Wire Frame

The most common form of rendering uses a triangle the fundamental primitive and is in fact the only one supported by graphics hardware. This is because it is the only polygon which is guaranteed to have all vertices on the same plane, simplifying interpolation greatly (St-Laurent, 2004).

The haptic device simulation software implements a boundary model approach using pure linear interpolation. This is realised through the use of Direct3D9, an Application Programming Interface (API), developed by Microsoft for implementing three dimensional graphics.

Each link of the model is assigned a frame; vertices are then described using local co-ordinates. This makes it possible to move each link independently of each other and implement a forward kinematic chain[1].

1

Forward Kinematics (p. 27)

Lighting

Effects of Lighting

Lighting plays a vital role in adding realism to a scene. Bellow in *Figure 14* we see a non planar surface (wireframe seen in *Figure 13*), rendered without lighting. This appears to be a flat square as we have no visible information about the surface gradient.

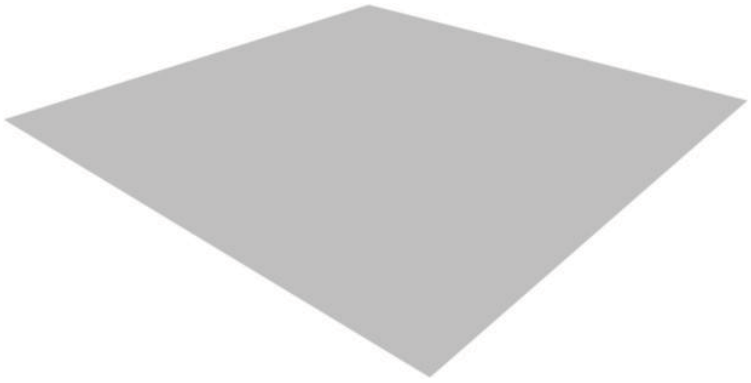


Figure 14: Non Planar Surface without Lighting

Figure 15 shows how the additional lighting allows us to interpret the shape of an object when the silhouette remains constant, adding a huge amount of realism to the scene. As humans, we are easily able to determine the shape of this surface, as we deal with objects which receive varying amounts of lighting at every moment we have our eyes open in day to day life.

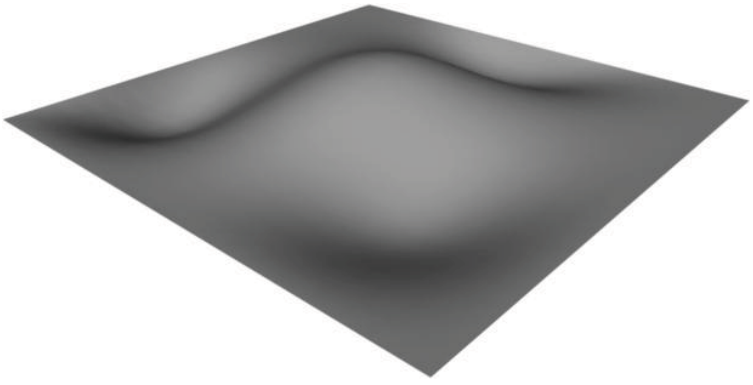


Figure 15: Non Planar Surface with Lighting

Types of light

There are a number of commonly implemented light models which are simple to implement:

- Ambient Light – This is a light value which is applied constantly over the scene, it is typically used to prevent areas from being too dark when no other light source is affecting the vertex / pixel. This lighting technique is very simple to implement as it involves only addition.
- Directional Light – This model is used to mimic large sources of light which are a long distance away, an example of which is sunlight. All rays of light are treated as having the same direction.
- Point Light – A point light is used to simulate small light sources, such as a light bulb. All rays are treated as emanating from a single point of origin and are given the same value of luminescence. These sources may have a fall off curve if desired, preventing objects which are too far from the source from receiving lighting.
- Spot Light – The most complicated light to compute. These lights have a single point of origin and direction. If a ray deviates from the direction more than a specified amount the luminescence begins to fall-off until it has no effect.

Creating Visible Light

Lighting can be implemented either per vertex or per pixel, depending on which shader the algorithm is implemented. In either case, the theory remains the same. When a light ray hits a surface, the more direct the angle of incidence, the more lighting the surface should receive. Similarly, grazing angles should receive less. This means that the lighting intensity should be multiplied by the cosine of θ, where θ is the

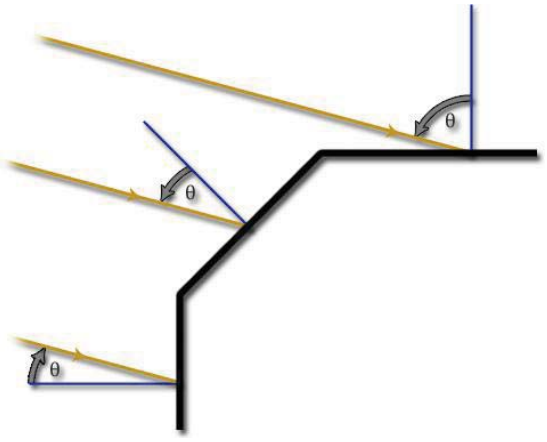


Figure 16: Light and Surface Normals

angle between the normal of the surface and the incoming light ray (*Figure 16*). We can model a ray of light simply as a three dimensional vector and the same can be said for a surface normal.

We can find θ from rearranging the equation below.

$$Ray \cdot Normal = |Ray||Normal|cos(\theta)$$

$$cos(\theta) = \frac{Ray \cdot Normal}{|Ray||Normal|}$$

However, recall that the value we are looking for (intensity of light to apply) is directly proportional to $cos(\theta)$.

$$Visable\ Intencity = SourceIntencity \cdot cos(\theta)$$

We can simplify this to enhance computational performance; however this is not without a compromise. The equation used to calculate visible light intensity (which is proportional to the length of both the ray and normal vectors) is:

$$Visable\ intensity = Ray \cdot Normal$$

Typically these vectors will be pre-normalized to unit length so that $|Ray||Normal|$ will equate to 1, providing a light intensity which will range between 0 and 1. This value can then be used to provide shading.

Defining the Surface Normal

We have previously used the surface normal to determine the amount of light a point should receive. However, we must still define the normal. There are two primary methods for doing this; the easiest to understand is per face (*Figure 17*). The normal for a face is simple to compute, since we are using triangles as our primitive, and we therefore know that each vertex will coincide with a single plane. Defining a plane from three points becomes trivial as we can retrieve two vectors which lie on the same plane. Cross multiplication yields a vector which is

perpendicular to both, which intern, is the normal to the plane. Due to the simplifications used for calculating light intensity, we should normalise it to unit length before use.

let:

p_1, p_2, and p_3 be three vertices defining a plane.

then:

$$a = p_2 - p_1$$

$$b = p_3 - p_1$$

$$N = \frac{a \times b}{\|a \times b\|}$$

This normal is then stored in a data structure defining a vertex, which will now contain two vectors, position and normal. The normal must be included with the vertex due to the GPU pipeline, which can only perform operations on a per vertex operation, with the exception of specialised operations (such as rasterisation).

If each plane consists of three vertices, each with a normal which is equal to that of the plane, then the normal will be constant over the plane when interpolated between vertices. This means that the face will receive a constant amount of light from directional lights, and very little variation from point and spot lights. In turn this creates sharp lines at the edges of triangles, as the lighting factor abruptly changes as does the normal.

It is possible to average the normals for all vertices which are at the same location, resulting in smooth shading between the faces. This effect is useful when one wishes to create smooth looking curves, as it eliminates harsh changes in lighting at edges of polygons.

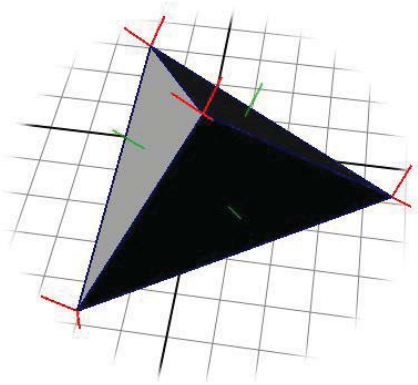


Figure 17: Face Normals

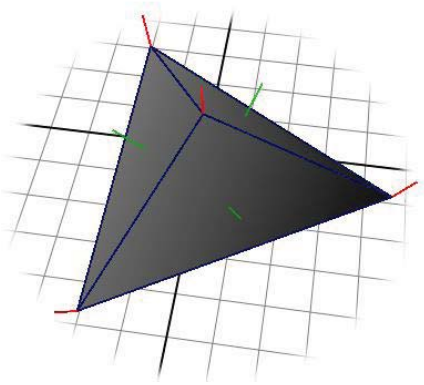


Figure 18: Averaged Normals

Normals can also be applied on a per-pixel basis through the use of texture mapping. This technique is known as normal mapping, where the angle of the surface is recorded in a bitmap image. The normal of the pixel is retrieved as a conventional texture. Normals are stored using the colour channels within the bitmap *(Figure 19)*, and can be in either object space or tangent space. Tangent space normals represent the angle of the surface relative to the triangle plane, while object space normals are defined relative to the mesh frame.

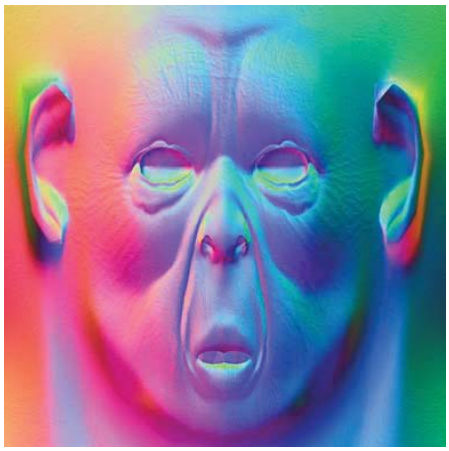


Figure 19: Normal Map

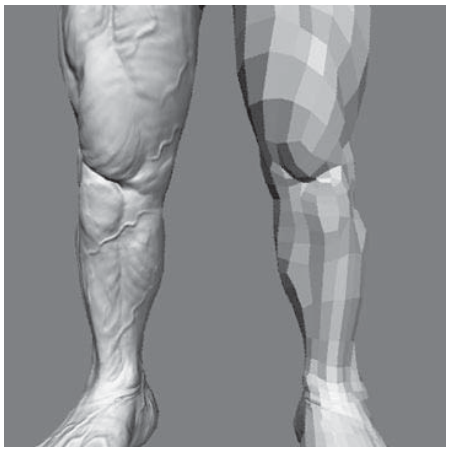


Figure 20: Normal mapped vs. flat shading Mapping (Jian, 2005)

The typical application for normal mapping is in games, where low render times are required. Advantages of normal mapping are that it is able to provide a surface with much higher detail than that described by the polygon mesh (*Figure 20*). However, the silhouette of objects will expose the illusion. It is possible to achieve better results using a technique known as Parallax Occlusion Mapping (POM), which utilises a height field to dynamically modify texture co-ordinates which are then used for standard normal mapping and texturing, this creates the illusion of visible hollows within a planar polygon. The implementation of POM is beyond the scope of this thesis.

Figure 21: Parallax Occlusion Mapping

Rendering Pipeline

The sequence of events used to transform mesh data into a two dimensional image to be presented to the user is referred to as the rendering pipeline, and is preformed in a multiprocessor environment. The Central Processing Unit (CPU) performs modifications to mesh data as required, and alters render constants before passing data to the GPU to perform vertex transformations and interpolation. This process is visible in (*Figure 22*). Due to the parallel processing nature of the pipeline, once constants and vertex data has been sent to the GPU, the CPU is free to perform other computations.

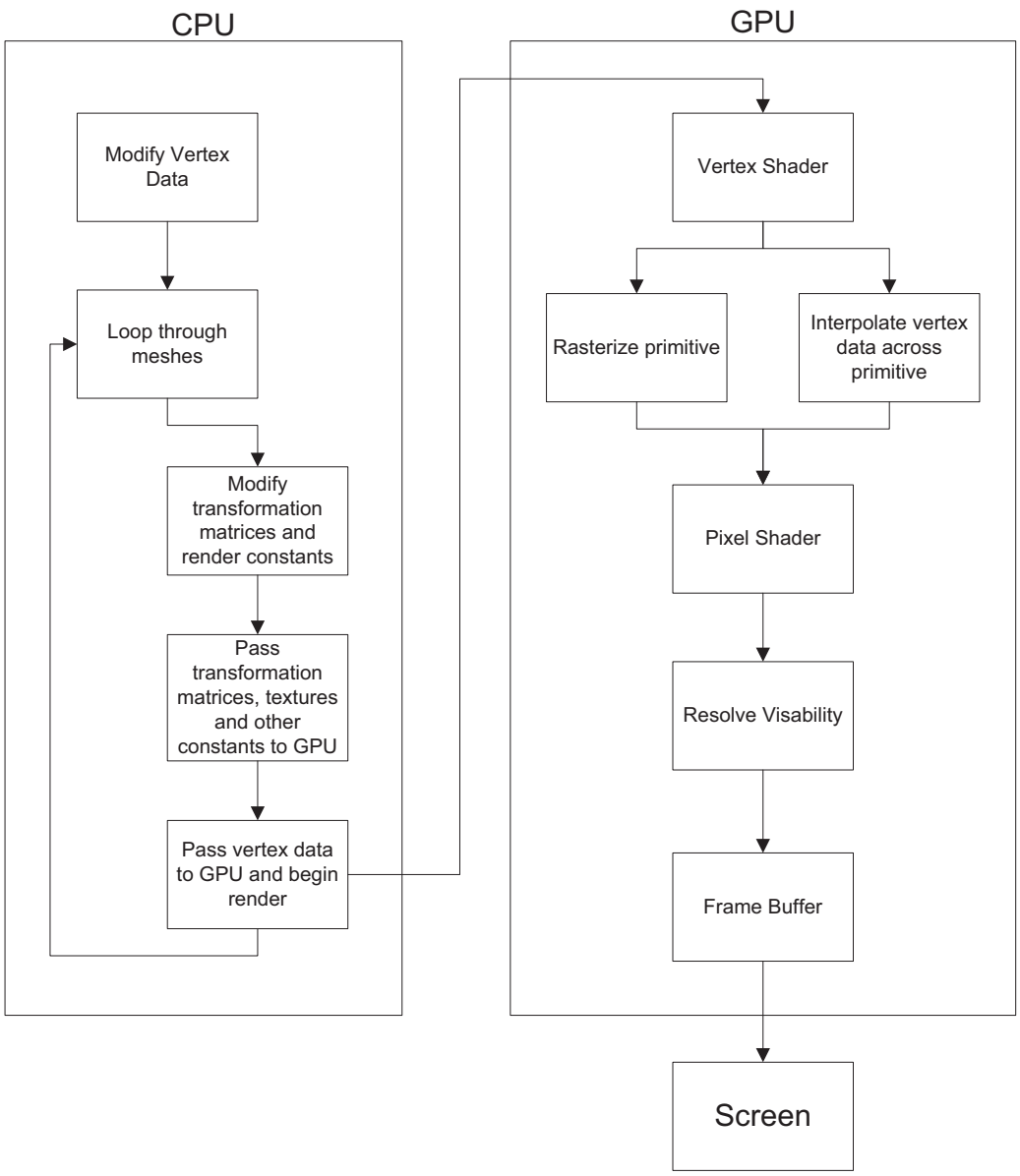


Figure 22: Rendering Process Diagram

CPU Pipeline

Three transformation matrices are used for the conversion from three-dimensional co-ordinates to two-dimensional screen space; these matrices are computed by the CPU and then

transferred to the GPU, which applies the transformation on a per vertex basis via a vertex shader[2] . *Figure 23* shows the order in which transformations are applied.

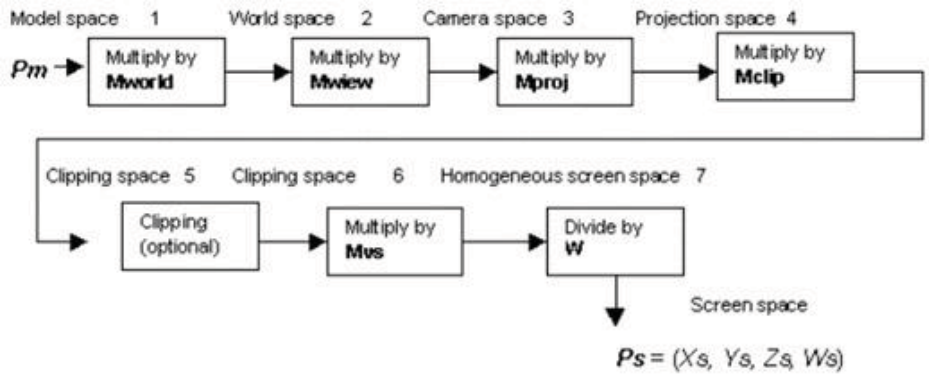


Figure 23: Transformation Pipeline (Microsoft Corporation, 2008)

Of these transformations, the Direct3D API will handle stages 4 and onwards by default. However, this still leaves three transformation matrices[3] which must be defined.

World Matrix

The world matrix, referred to within the Direct3D API as 'Mworld', is used to transform mesh vertices from local mesh space to global space. This moves meshes off the world origin. It specifies location, rotation and scaling for individual mesh bodies.

View Matrix

This matrix allows us to move a virtual camera throughout the scene. The view matrix defines what is viewed by the camera; this is done by providing the cameras' position, target to look at, and direction to be treated as 'up' (Cawood & McGee, 2007). These values are converted into a transformation matrix using the following steps.

[2] Simple Vertex Shader (p. 46)
[3] 45Frame Transformations (p. 17)

From (Microsoft Corporation, 2008):

let:

At, Eye, *and* Up *be vectors of three-dimensions.*

$$zaxis = \frac{(At - Eye)}{\|At - Eye\|}$$
$$xaxis = \frac{(Up \times zaxis)}{\|Up \times zaxis\|}$$
$$yaxis = zaxis \times xaxis$$

then:

$$M_{View} = \begin{bmatrix} xaxis_x & yaxis_x & zaxis_x & 0 \\ xaxis_y & yaxis_y & zaxis_y & 0 \\ xaxis_z & yaxis_z & zaxis_z & 0 \\ -xaxis \cdot Eye & -yaxis \cdot Eye & -zaxis \cdot Eye & 1 \end{bmatrix}$$

Figure 24: View Matrix

Projection Matrix

The projection matrix describes how vertex position is converted into screen space. There are two types of projection; orthographic and perspective.

Orthographic views are typically used when sizes on screen must provide an accurate representation of size, typically Computer Aided Design (CAD) applications. The disadvantage to orthographic projection is that it provides no information about the distance an object is from the view point.

Perspective views are used to create a more realistic view of objects. Objects which are further away from the view point appear smaller than those which are closer. The reason this effect is noticeable is demonstrated in *Figure 25*. As the object moves away from the view point, the angle between respective rays which pass through the same

part of the object decreases. As lenses (found in cameras and the human eye) modify vertical and horizontal deviations into angular displacements, we in fact see this angle rather than an orthographic projection of the object.

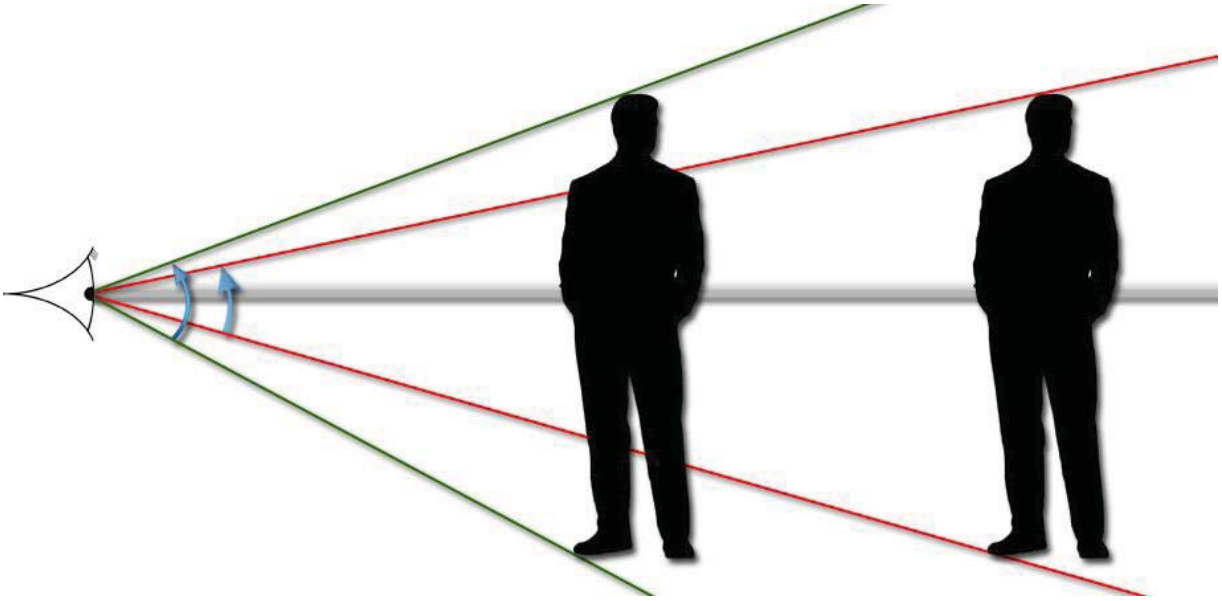


Figure 25: Perspective

As there are two different methods of projection, we must define two matrices which describe these transformations.

From (Microsoft Corporation, 2008):

Let:

- S_w = screen window width in camera space in near clipping plane.
- S_h = screen window height in camera space in near clipping plane.
- Z_n = distance to the near clipping plane along z-axis in camera space.
- Z_f = distance to the far clipping plane along z-axis in camera space.

$$M_{Orth} = \begin{bmatrix} \frac{2}{S_w} & 0 & 0 & 0 \\ 0 & \frac{2}{S_h} & 0 & 0 \\ 0 & 0 & \frac{1}{Z_f - Z_n} & 0 \\ 0 & 0 & -\frac{Z_f}{Z_f - Z_n} & 1 \end{bmatrix}$$

Equation 1: Orthographic Projection Matrix

$$M_{Persp} = \begin{bmatrix} \frac{2 * Z_n}{S_w} & 0 & 0 & 0 \\ 0 & \frac{2 * Z_n}{S_w} & 0 & 0 \\ 0 & 0 & \frac{Z_f}{Z_f - Z_n} & 1 \\ 0 & 0 & -\frac{Z_f * Z_n}{Z_f - Z_n} & 0 \end{bmatrix}$$

Equation 2: Perspective Projection Matrix

Transforming points by this matrix projects the three dimensional image onto a 2-D plane known as the near clipping plane, which is a specified distance *(Z_n)* from the camera. This is illustrated in *Figure 26*.

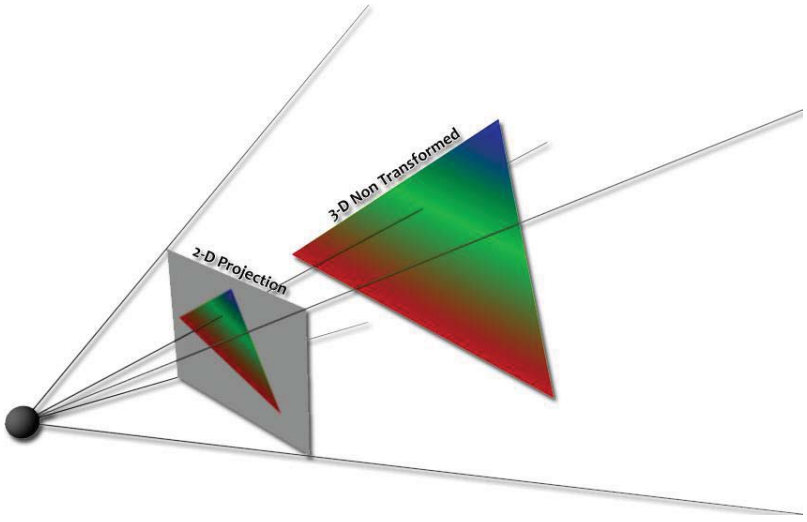


Figure 26: 2-D Projection

GPU Pipeline

Shaders

On modern GPUs, this is implemented via a programmable function known as a Shader. Older GPUs used a fixed function pipeline which reduced flexibility in rendering techniques. Direct3D 9 provides a default shader which mimics the actions of a fixed function GPU. There are a number of shader programming languages in use depending on the API used. High Level Shader

Language (HLSL), which is based on Ansci-C, has been developed for use with Direct3D, while OpenGL Shader Language (GLSL) has been developed for the OpenGL API. The shader language used varies with the API since the shader is compiled at run time by the application then transferred to the GPU during initialisation.

Shaders are implemented in various parts of the GPU pipeline. It is important to note that the GPU has multiple processing cores and that shader functions do not behave as a conventional program function would. Each shader is invoked via hardware to act on a specific set of data. Standard implementation utilises two shaders, the vertex and pixel/fragment shader. New generation GPUs are being released with an additional shader stage known as the geometry shader. This new shader comes first in the sequence and can modify the vertex stream before it is passed to the vertex shader by either removing or adding vertices.

Since we may not always want the mesh to be rendered the same way as a pervious mesh, HLSL provides a method of implementing different techniques. Each technique can have multiple passes if required. A simple technique is shown below in *Snippet 1*.

```
technique Simple
{
    pass Pass0
    {
        VertexShader = compile vs_1_1 SimpleVertexShader();
        PixelShader = compile ps_1_1 SimplePixelShader();
    }
}
```

Snippet 1: Simple HLSL Technique

Vertex Shader

The vertex shader performs operations on each vertex in the vertex stream sent from the CPU or geometry shader if the GPU has one. The function of the vertex shader is to transform vertices from local co-ordinate systems to two-dimensional screen space. This is achieved by multiplying the vertex co-ordinates with the transformation matrices provided from the CPU. The shader is also capable of calculating various properties for the vertex; these include colour,

lighting, texture co-ordinates, and anything else which may be required for the pixel shader to function as intended. The vertex shader requires the transformation matrices and all data which is not allocated per vertex to be passed as constants before rendering is initiated.

```
float4x4 xWorld;
float4x4 xViewProjection;
float3   xLightDirection;
```

Snippet 2: Shader Constants

```
struct VertexToPixel
{
    float4 Position     : POSITION;
    float4 Color        : COLOR0;
};
```

Snippet 3: Vertex Shader Output Structure

```
VertexToPixel SimpleVertexShader(   float4 inPos : POSITION,
                                    float4 inColor : COLOR0,
                                    float3 inNormal : NORMAL0)
{
     VertexToPixel Output = (VertexToPixel)0;

// Preshader
     float4x4 preWorldViewProjection = mul (xWorld, xViewProjection);


// Transformations
     float3 Normal = normalize(mul(inNormal, (float3x3)xWorld));
     Output.Position = mul(inPos, preWorldViewProjection);

// Lighting
     Output.Color = inColor * dot(xLightDirection, Normal) ;

// Finished
     return Output;
 }
```

Snippet 4: Simple Vertex Shader

Above is the code required to implement a simple vertex shader. *Snippet 2* shows required constants which need to be passed to the shader before rendering is initiated. These are two transformation matrices and a three dimensional vector for a directional light to be implemented. View and projection matrices are combined into a single transformation matrix. It should be noted that we do not modify normals by the view or projection matrix, but only by the world. As such, we must pass both matrices to the vertex shader.

Snippet 3 shows the structure which we use to pass data from the vertex shader to the pixel shader. It is a requirement that a position be output; however any additional parameters are optional, in this case we also want to pass a colour.

Implementation of the vertex shader is shown in *Snippet 4*. First in the function header we describe which properties we will be using in the shader, and what the output format will be. In this case we are using the position, normal and colour of the vertex.

This shader utilises a technique known as pre-shading. When the shader is compiled, it will be detected that "float4x4 preWorldViewProjection = mul (xWorld, xViewProjection);" is the same for every vertex rendered. Rather than perform this multiplication many times, the compiler will remove this from the vertex shader and perform the operation on the CPU and send the result to the GPU only once.

Transformations are applied as required; this converts the world space co-ordinates of the vertex into screen space and modifies normals such that they remain 'fixed' to the mesh being rendered.

The lighting factor (as per Creating Visible Light p. 25) for the vertex is calculated by performing the dot product. The vertex colour is then assigned to be inColour*lighting.

Pixel Shader

The pixel shader is implemented for every pixel which is within the bounds of the primitive being drawn. Values passed are interpolated to achieve a smooth gradient between the vertices of a primitive.

```
struct PixelToFrame
{
    float4 Color        : COLOR0;
};
```

Snippet 5: Pixel Shader Output Structure

```
PixelToFrame SimplePixelShader(VertexToPixel PSIn)
{
     PixelToFrame Output = (PixelToFrame)0;
     Output.Color = PSIn.Color;
     return Output;
};
```

Snippet 6: Pixel Shader

In the same way as we did with the vertex shader, we must define an output for our shader. *Snippet 5* shows the structure to be used; in this case it is simply the colour of the pixel.

The shader itself is simple as well, taking the colour which has been passed to it from the interpolator and then returning it as the output.

Picking 3-D Objects via Mouse

When designing an interface which implements 3-D graphics, it is useful for the user to be able to interact with rendered objects using the mouse. Determining if the mouse is 'over' a rendered object is known as picking, and is a multi stage process. Firstly a ray (mono directional 3-D line) which passes through the mouse and camera origin must be determined. This ray must then be tested against each object to see if it intersects with any polygons.

Converting Mouse to Ray

The first of which involves creating a ray, which in global space passes through the mouse and camera origin. As the mouse is a two dimensional point on the screen, it must first be converted to a three dimensional point. This initially involves transforming to a point in projection space. Since screen co-ordinates are constrained by:

$$0 \leq x \leq Width$$
$$0 \leq y \leq Height$$
$$0 \leq z \leq 1$$

and we know the corresponding projection space locations for these bounds:

$$\begin{pmatrix} x = 0 \\ y = 0 \\ z = 0 \end{pmatrix}_{Screen} \rightarrow \begin{pmatrix} x = -1 \\ y = 1 \\ z = Z_n \end{pmatrix}_{Projection}$$

$$\begin{pmatrix} x = Width \\ y = Height \\ z = 1 \end{pmatrix}_{Screen} \rightarrow \begin{pmatrix} x = 1 \\ y = -1 \\ z = Z_f \end{pmatrix}_{Projection}$$

where:

- Z_n = distance to the near clipping plane along z-axis in camera space.
- Z_f = distance to the far clipping plane along z-axis in camera space.

a transformation matrix can be deduced such as follows:

$$M_{S \rightarrow P} = \begin{bmatrix} \frac{2}{Width} & 0 & 0 & -1 \\ 0 & -\frac{2}{Height} & 0 & -1 \\ 0 & 0 & z_f - z_n & z_n \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It is now possible to transform a point on the viewport to projection space. However, there is no information available on what the depth should be. However at this point it is irrelative, so we choose it to be maximum '1' to reduce floating point errors.

$$P_{Screen} = \begin{pmatrix} Mouse_x \\ Mouse_y \\ 1 \\ 1 \end{pmatrix}$$

While the point is in projection space it is still of little use, so conversion to object space is required, we know from *CPU Pipeline (p. 30)* that:

$$P_{Projection} = M_{World} M_{Object} M_{LookAt} M_{Projection} P_{World}$$

It is efficient to use the ray in object space to reduce transformations, as there are normally many hundreds of vertices per object, but only one ray.

$$P_{Projection} = M_{S\to P} P_{Screen} = M_{LookAt} M_{Projection} P_{Object}$$

rearranging:

$$P_{Mouse\ Object} = M_{Projection}^{-1} M_{LookAt}^{-1} M_{S\to P} P_{Mouse\ Screen}$$

This gives us the position of the mouse cursor as a 3-D point. We can now transform the camera position to object space:

$$P_{Camera\ Object} = M_{World} P_{Camera\ World}$$

This can now be used to determine a ray passing from the camera through the mouse.

$$P_{Ray} = P_{Camera} + t(P_{Mouse} - P_{Camera})$$

for:

$$t \geq 0$$

Equation 3: Picking Ray Equation

Intersection Testing

It is possible to determine if the ray has intersected an object, and thus the mouse is over it, by testing the ray against each polygon within the object. This is done using the equation for a point within a triangle specified by three points and the equation for the ray.

$$P_{InTri}(u,v) = (1-u-v)P_{Tri_0} + uP_{Tri_1} + vP_{Tri_2}$$

for:

$$u \geq 0$$
$$v \geq 0$$

$$u + v \leq 1$$

Equation 4: Point in Triangle

Substituting *Equation 3* into *Equation 4*:

$$P_{Camera} + t(P_{Mouse} - P_{Camera}) = (1 - u - v)P_{Tri_0} + uP_{Tri_1} + vP_{Tri_2}$$

We can then say, that if there exists a triplet (t,u,v) that meets the above requiems of t, u, and v, then there is an intersection.

The algorithm to solve this from (Möller & Haines, 2002, pp. 578-581):

1. Find vectors from P_{Tri_1} and P_{Tri_2} to P_{Tri_0}

   $V_1 = P_{Tri_1} - P_{Tri_0}$, $V_2 = P_{Tri_2} - P_{Tri_0}$
2. Take the cross product between the direction of the ray and one of the vectors found in step 1:

   $H = Ray_{Direction} \times V_2$
3. Take the dot product of the vector found from step 2, and the vector found in step 1 which was not used in step 2:

   $A = V_1 \cdot H$
4. If *A* is near zero ($-0.00001 < A < 0.00001$) then there is no intersection.
5. Find the vector from point 0 of the triangle to the origin of the ray:

   $S = Ray_{Origin} - P_{Tri_0}$
6. Find *u*, using the dot product from vectors found in steps 2 and 5 and the dot product found in step 3.

   $u = \frac{S \cdot H}{A}$
7. If not $0 \leq u \leq 1$ then there is no intersection.
8. Find the cross product between the vector from P_{Tri_0} to Ray_{Origin} and the vector from P_{Tri_1} to P_{Tri_0}:

   $Q = S \times V_1$

9. Find *v*, using the dot product of the ray direction and the vector found in step 8, and the value found from step 3.

$$u = \frac{Q \cdot H}{A}$$

10. If $V < 0$ or $u + v \leq 1$ then there is no intersection.
11. Find *t*, using the dot product of the vector found in Step 8 and the vector with the value found from P_{Tri_2} to P_{Tri_0} , and the value found from step 3.

$$t = \frac{V_2 \cdot Q}{A}$$

12. If $t \geq 0$ there is an intersection.

If a triangle is hit, we store *t*. This enables us to perform a test if another triangle is intersected. The intersection with the lowest *t* value is the closest polygon to the user.

After this algorithm is preformed for every primitive within the scene, we can determine which object the user has the mouse over (if any), using the ownership of the primitive which has the lowest *t* value overall.

Chapter 4: Haptic Simulation

In order to determine the feasibility of the Haptic input device, and later to remove mechanical effects of the manipulator from the 'feel' of inputs, it was determined that a simulation should be created.

The simulation was realised, using C#, to the point of calculating forces and torques provided from the output of servos within the linkage on the end effectors, along with the effects of gravity relative to each link in a static configuration. Effects which remain to be implemented occur relative to a dynamic configuration, these include momentum and friction components.

It was decided that a rigid body simulation would suffice, as any deformation in linkage components would be negligible.

Rigid Bodies

Mathematically, a rigid body is a simplification of a real world object which remains near to a constant shape. Given any two particles (P and Q) within a rigid body, it is assumed that no change will occur between their relative positions as the body moves.

It is convenient to be able to define points within a rigid body in a local co-ordinate system, known as a frame, which is associated with the body (Parent, 2002). It becomes apparent that the transference of a point from one co-ordinate space to another is a common task both within kinematics, physical simulation, and three dimensional computer graphics. As such, it is imperative to have a well defined method for translating co-ordinates from one frame to another (Parent, 2002). And as later used in forward kinematics, we find that we must transform co-ordinates from one rigid body frame to another.

As previously mentioned, there are many situations where we wish to be able to translate between one frame and another. A transformation mapping (g) function is considered a rigid body transformation if it meets the following properties (Murray, Li, & Sastry, 1994):

For all points within the frame:

$$P_n = \begin{pmatrix} P_{n_x} \\ P_{n_y} \\ P_{n_z} \end{pmatrix}, P_m = \begin{pmatrix} P_{m_x} \\ P_{m_y} \\ P_{m_z} \end{pmatrix}, P_o = \begin{pmatrix} P_{o_x} \\ P_{o_y} \\ P_{o_z} \end{pmatrix}$$

and vectors between any two points:

$$v = p_n - p_m \, , w = p_m - p_o$$

1. The distance between the points must remain constant:

$$\left\| g\,(P_n) - g\,(P_m) \right\| = \left\| P_n - P_m \right\| = Constant$$

2. The cross product of vectors must remain constant:

$$g \cdot (v \times w) = g \cdot (v) \times g \cdot (w) = Constant$$

Equation 5: Rigid Body Transformation Function Requirements (Murray, Li, & Sastry, 1994)

Kinematics

Frame Transformations

Translation

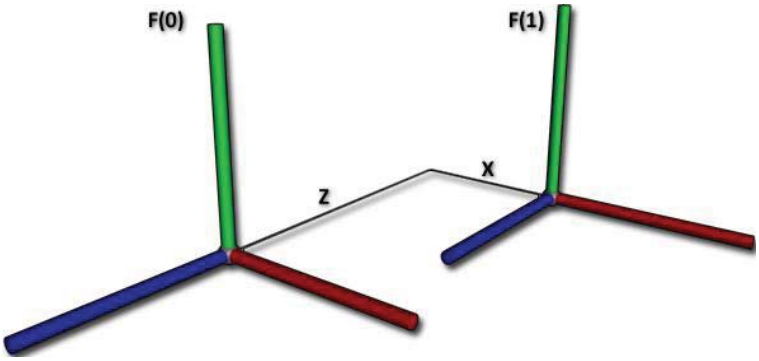


Figure 27: Frame Translation

If we wish to move the origin of a frame in relation to another, we must perform a vector operation. Figure 27, shows the translation of $F_{(1)}$ in relation to $F_{(0)}$. In this case, $F_{(1)}$ is translated along both the X and Z axes, while Y remains constant. We can show this translation mathematically via the vector $V^{Translate}$ where:

$$V^{Translate} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} = \begin{pmatrix} X \\ 0 \\ Z \end{pmatrix}$$

The translation between $F_{(1)}$ to $F_{(0)}$ is given by:

$$P_{F(0)} = P_{F(1)} + V^{Translate}$$

Scaling

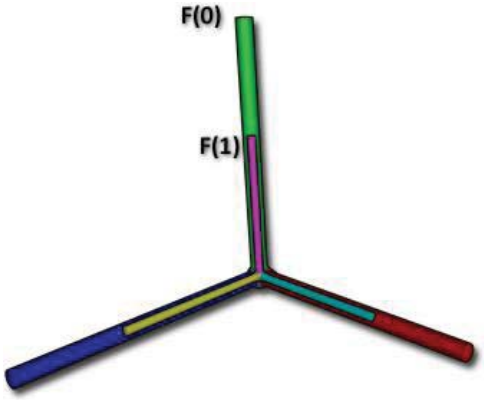


Figure 28: Frame Scaling

It is relatively simple to scale one co-ordinate frame to another. In order to perform the scaling, all that is required is to multiply each co-ordinate by the desired scaling factor where:

$$S = \begin{pmatrix} S_x \\ S_y \\ S_z \end{pmatrix}$$

then:

$$P_{F(0)} = S \times P_{F(1)}$$

While for pure scaling, a vector implementation is sufficient once we want to combine multiple transformations. It becomes apparent that a scaling matrix becomes more viable. This matrix is defined as:

$$\boxed{M^S = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_x \end{bmatrix}}$$

Rotation

Rotation is a more complicated transformation than that of translation or scaling. It should be noted that rotation of a frame consists of one rotation about each axis and the order of rotation affects the resulting transformation. In *Figure 30* we see $F_{(1)}$ rotated about its y-axis by θ_y and then around its x-axis by θ_x; while in *Figure 29* the rotation order is reversed so that $F_{(1)}$ is first rotated around its x-axis by θ_x and then about its y-axis by θ_y. Additionally, the owner of the axis being rotated about affects rotational transformation; *Figure 29* and *Figure 30* demonstrate rotation about local axis while the following example rotates about axes within the target frame. This effect must be kept in mind when assigning frames to bodies which will have rotational transformations applied.

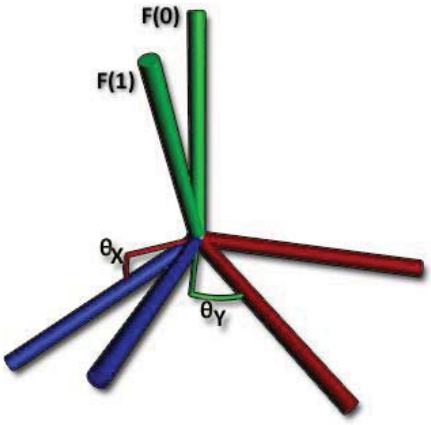


Figure 29: Frame Rotation Order X-Y

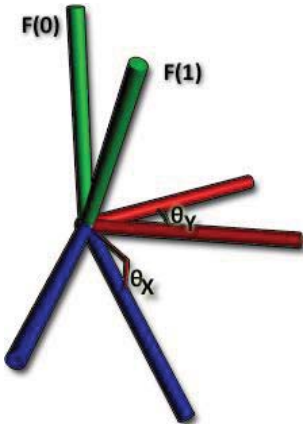


Figure 30: Frame Rotation Order Y-X

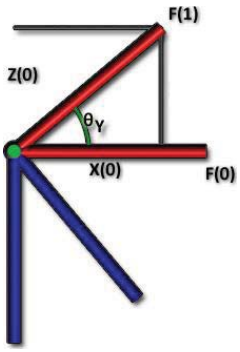
We wish to mathematically transform a point $P_{F(1)}$ from frame $F_{(1)}$ to $F_{(0)}$ space.

Let:

$$P_{F(1)} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \theta_z = 0^\circ, \ \theta_y = 40^\circ, \text{ and } \theta_x = 30^\circ$$

To keep the desired rotation order (z-y-x), we must first rotate about the z-axis, by applying basic trigonometry functions:

$$P^z_{F(0)} = \begin{pmatrix} Cos(\theta_y) \cdot P_{F(1)_x} - Sin(\theta_y) \cdot P_{F(1)_y} \\ Cos(\theta_y) \cdot P_{F(1)_y} + Sin(\theta_y) \cdot P_{F(1)_x} \\ P_{F(1)_z} \end{pmatrix} = \begin{pmatrix} P_{F(1)_x} \\ P_{F(1)_y} \\ P_{F(1)_z} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$



As we would expect there is no change to the point, as there is no rotation about the z-axis specified.

Figure 31: Orthographic View of Rotation about the Y-Axis

Rotation about the y-axis yields:

$$P^y_{F(0)} = \begin{pmatrix} Cos(\theta_y) \cdot P^z_{F(0)_x} + Sin(\theta_y) \cdot P^z_{F(0)_z} \\ P_{F(1)_y} \\ Cos(\theta_y) \cdot P^z_{F(0)_z} - Sin(\theta_y) \cdot P^z_{F(0)_x} \end{pmatrix} = \begin{pmatrix} 0.77 \cdot P^z_{F(0)_x} + 0.64 \cdot P^z_{F(0)_z} \\ P^z_{F(0)_y} \\ 0.77 \cdot P^z_{F(0)_z} - 0.64 \cdot P^z_{F(0)_x} \end{pmatrix} = \begin{pmatrix} 0.77 \\ 1 \\ -0.64 \end{pmatrix}$$

Rotation about the x-axis yields:

$$P^x_{F(0)} = \begin{pmatrix} P^z_{F(0)_x} \\ Cos(\theta_x) \cdot P^y_{F(0)_y} - Sin(\theta_x) \cdot P^y_{F(0)_z} \\ Cos(\theta_x) \cdot P^y_{F(0)_z} + Sin(\theta_x) \cdot P^y_{F(0)_y} \end{pmatrix} = \begin{pmatrix} P^z_{F(0)_x} \\ 0.87 \cdot P^y_{F(0)_y} - 0.5 \cdot P^y_{F(0)_z} \\ 0.87 \cdot P^y_{F(0)_z} + 0.5 \cdot P^y_{F(0)_y} \end{pmatrix} = \begin{pmatrix} 0.77 \\ 1.19 \\ -0.057 \end{pmatrix}$$

Figure 32: Rotation Steps

While it is possible to perform rotations using vector operations, this does not carry over into practical implementation, as rotation about each axis would require an individual implementation. However, looking at the vector equations, we can see that we are dealing with a problem which could be solved using a matrix equation. Since there are three axes of

rotation, we require three matrices for each axis rotated about, and as we are dealing with three dimensions these matrices should be 3-by-3:

$$M_x^{Rot}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & cos(\theta) \end{bmatrix}$$

$$M_y^{Rot}(\theta) = \begin{bmatrix} cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & cos(\theta) \end{bmatrix}$$

$$M_z^{Rot}(\theta) = \begin{bmatrix} cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

These matrices can be combined into one by multiplying them, following the order of rotation. For a rotation order of z-y-x, this yields:

$$\boxed{M_{zyx}^{Rot} = M_z^{Rot} M_y^{Rot} M_x^{Rot}}$$

We can now transform a point in $F_{(m)}$ to $F_{(n)}$ by multiplying the rotation matrix and original point:

$$\boxed{P_{F(n)} = M_{zyx}^{Rot\ m \to n} P_{F(m)}}$$

Equation 6: Rotational Transform

Combined Transformations

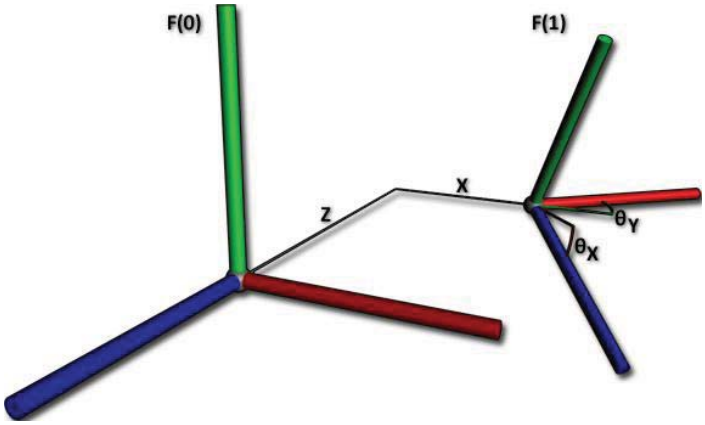


Figure 33: Frame Translation and Rotation Combined

While we can now transform our frame via Translation, Scaling, or Rotation, we need to combine this into a single action. Once again the order in which we proceed affects the overall transformation. For instance, if we are transforming from $F_{(1)}$ to $F_{(0)}$ and were to apply a translation before rotation, then the centre of rotation would be the origin of $F_{(0)}$ rather than that of $F_{(1)}$. With this in mind, our order of transformation will be Rotate-Scale-Translate. This will create a transformation which rotates and scales about the origin of $F_{(1)}$ and then translate from the origin of $F_{(0)}$. To do this, we must modify the format of data storage.

The transformation matrix contains rotation, scaling and translation data; while scaling and rotation may be combined into a 3-by-3 matrix, translation, being a vector operation would need to be excluded. As a result, a 4-by-4 transformation matrix is used.

Combining Scaling and Rotation:

$$M^{RS} = M^{R}_{(Order)} M^{S}$$

then:

$$M^{Transform} = \begin{bmatrix} & M^{RS} & & V^{Translate} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 7: Transformation Matrix

In order to make the co-ordinates of points within a frame compatible with this matrix, we must modify the vector to contain four elements rather than three:

$$\begin{pmatrix} P_z \\ P_y \\ P_z \end{pmatrix} \rightarrow \begin{pmatrix} P_z \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

This validates the equation:

$$\boxed{P_n = T_n^m P_m}$$

where:

P_n is a point in frame F_n

P_m is a point in frame F_m

T_n^m is the transformation matrix from F_m to F_n.

Forward Kinematics

Forward Kinematics utilises rigid bodies to determine the location of links within a kinematic chain. This is achieved through the use of transformation matrices by assigning frames to each link in the chain and establishing a multiplication hierarchy. Movements in 'base' frames are carried forward throughout the chain. As it is possible to transform between co-ordinate spaces by a matrix multiplication, we could implement the chain by performing a transformation and then transforming again until the chain is complete. This is implemented using the equation:

$$P_{n-1} = T_{n-1}^{n} P_n$$

While this will yield the correct result, as the number of points being transformed increases, it is more efficient to determine the transformation from the local frame to base frame directly. This is simply implemented by multiplying the transformation matrices.

$$T_n^m = T_{m-1}^m \cdots T_n^{n+1}$$

The resulting matrix can be used to transform directly from frame F_m to frame F_n.

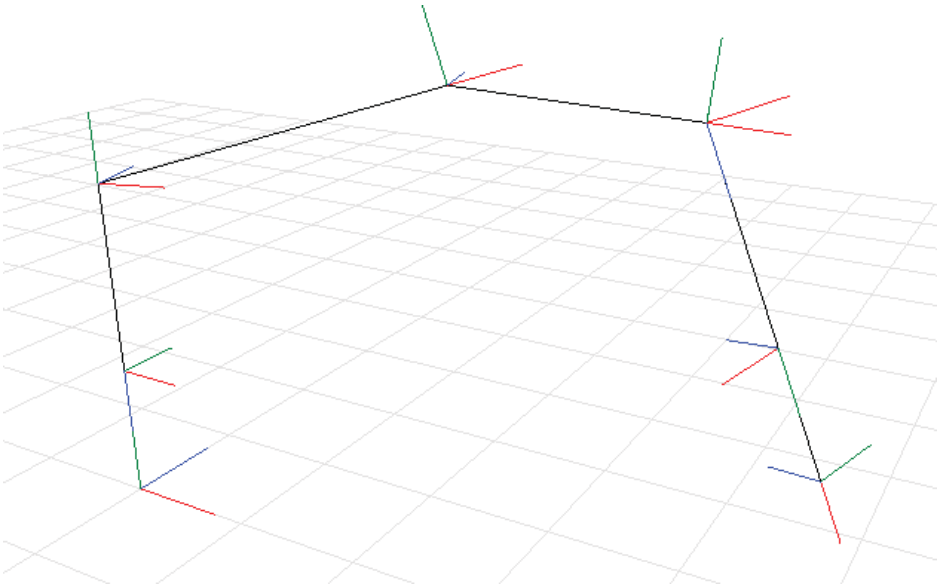


Figure 34: Kinematic Diagram of Manipulator

Denavit and Hartenberg Convention

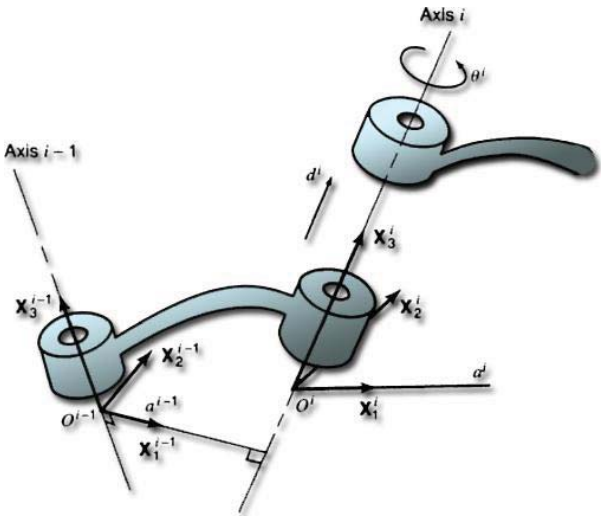


Figure 35: D-H Link (Shabana A. A., 2005)

As seen in *Figure 34*, the origins of each frame describe provide a visual representation of one link of the manipulation arm. Here we can see that the orientation of each origin has been defined such that any rotations of the physical joint will always occur around the z (Blue) axis. This follows the Denavit and Hardenberg (D-H) convention, which allows for the kinematic system to be described with fewer parameters than if origins were chosen arbitrarily. The D-H convention describes each link in the chain (relative to the previous link), with only four parameters per link; θ^i (Rotation about z-axis, seen as rotation in the joint), α^{i-1} (Rotation about x axis, seen as twist within the link), di (Distance, along the z^i-axis, between x-axes of joints), and a^{i-1} (Distance, along the x^{i-1}-axis, between Z-axes of joints) (Shabana, 2005).

$$T_{i-1}^{i} = \begin{bmatrix} cos(\theta^i) & -sin(\theta^i)cos(\alpha^{i-1}) & sin(\theta^i)sin(\alpha^{i-1}) & d^i \cdot cos(\alpha^{i-1}) \\ sin(\theta^i) & cos(\theta^i)cos(\alpha^{i-1}) & -cos(\theta^i)sin(\alpha^{i-1}) & d^i \cdot sin(\alpha^{i-1}) \\ 0 & sin(\alpha^{i-1}) & cos(\alpha^{i-1}) & a^{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equation 8: Transformation Matrix using D-H Parameters

Confusion may arise with the use of a D-H table in some configurations (one of which is present in the RV-M1). *Figure 34* shows two origins located at the same position, although the physical joints are separated by some distance in the RV-M1. This model remains valid since the axis of rotation between the model and physical system remains co-linear despite the offset along the Z-axis.

```
public void FromDHParam(float theta, float alpha, float a, float d)
        {

this.FromRowVec(

Vector.Vec4(Math.Cos(theta), -Math.Sin(theta) * Math.Cos(alpha),
Math.Sin(theta) * Math.Sin(alpha), d * Math.Cos(theta)),

Vector.Vec4(Math.Sin(theta),Math.Cos(theta) * Math.Cos(alpha),-
Math.Cos(theta) * Math.Sin (alpha),    d * Math.Sin(theta)),

Vector.Vec4(0, Math.Sin(alpha), Math.Cos(alpha),a),

Vector.Vec4(0, 0, 0,1));
}
```

Snippet 7: Matrix from D-H Table

I_0 represents the initial rotation of the base frame relative to the world space co-ordinates used by the rest of the software.

Table 1 shows the D-H table used for the forward kinematics of the manipulator in the home position, while *Equation 8* shows how these parameters can be converted into a matrix which is suitable for forward kinematic multiplication.

i	a (mm)	d (mm)	alpha	theta (offset)
1	0	0	90	0
2	-142	0	-90	0
3	0	260	0	0
4	0	160	0	0
5	0	0	90	90
6	152	0	90	-90
7	0	100	0	90

Table 1: D-H Parameters for RV-M1

Software Implementation

The link table is loaded from `'Link Tabel.txt'`, located in the base project directory; this contains data required to construct the kinematic chain. This includes how many links and servos the simulation contains, as well as D-H parameters, graphical parameters, physical mass properties, for each link.

Forward kinematics are implemented using the `FowardKinimatics` function found within the `ArmModel` class, seen below in *Snippet 8*. This function updates the transformation tables of all links in the arm, updates physical simulation values for the links, and then calculates forces applied by each link on the end effecter.

```
        public void FowardKinimatics()
        {

            this.Transformations();
            Link.UpdateLinks();
            this.Forces();

        }
```

Snippet 8: ArmModel.FowardKinimatics()

```
for (int i = 0; i < Globals.RobotProps.num_Links; i++)
            {
                float ServoAngle = 0;

                if (Globals.RobotProps.linkTable[i].ServoID >= 0)
                {
                    ServoAngle =
Globals.RobotProps.servoTable[Globals.RobotProps.linkTable[i].ServoID].Angle;
                }
                di = Globals.RobotProps.linkTable[i].di;
                ai = Globals.RobotProps.linkTable[i].ai;
                alpha = Globals.RobotProps.linkTable[i].alpha;
                theta_Offset = Globals.RobotProps.linkTable[i].thetaOffset;


Globals.RobotProps.linkTable[i].Local_TransformMat.FromDHParam
                (ServoAngle + theta_Offset, alpha, ai, di);
            }
```

Snippet 9: Update Transformation Matrix for each link.

Snippet 9 shows how transformation tables are updated; this is done by combining both the DH-Table which was loaded from `Link Table.txt` and the angle of the servo attached to the

link. Once all the transformation matrices in the links table have been recreated, the chain is multiplied in order by the code in *Snippet 10*.

```
            GroundFrame.ToMatrix(tmp);

            for (int i = 0; i < Globals.RobotProps.num_Links; i++)
            {
                Matrix4.Multiply(tmp,
Globals.RobotProps.linkTable[i].Local_TransformMat, CompleteTrans);
                CompleteTrans.ToMatrix(tmp);

CompleteTrans.ToMatrix(Globals.RobotProps.linkTable[i].Global_TransformMat);

                if (ClassVerbose && LocalVerbose)
CompleteTrans.WriteToConsole();
            }


            if (ClassVerbose && LocalVerbose) CompleteTrans.WriteToConsole();
            if (ClassVerbose && LocalVerbose) Console.Write("\n\n");

            CompleteTrans.ToMatrix(Grip.Transform);
```

Snippet 10: Multiplication of kinematic chain transforms.

These matrices are then used for force calculations and rendering transformations.

End Effecter Forces

Calculating the force on the End Effecter is done via a multi stage process where the force contributed from each servo is accumulatively added to obtain the final result.

In order to find the force applied by the servo to the end point, multiple sub-steps are taken:

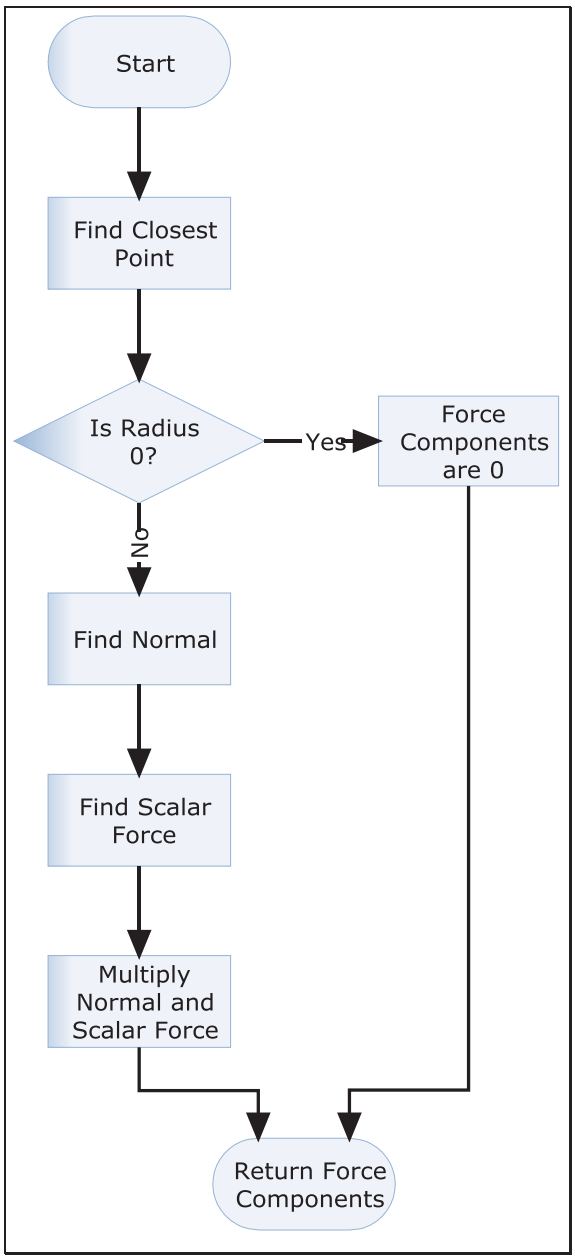


Figure 36: Process of force calculation

1. Find the point which is located on the axis of rotation for the given joint and origin of the end effecter. This can be done by:

   Treating the axis as a line in three dimensions, it can be described mathematically by the function:

   Equation 9

   $$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + t \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

   Where:

   - (x, y, z) is an unknown point on the line
   - (x_o, y_o, z_o) is a known point which the line passes through
   - t is the distance along the line from the known point
   - (a, b, c) is a vector representing the direction of the line

   The distance between any given point on the line and another point can be given via:

   Equation 10

   $$d = \sqrt{(x - e_x)^2 + (y - e_y)^2 + (z - e_z)^2}$$

   Where:

   - (x, y, z) is a point on the line from *Equation 9*
   - (e_x, e_y, e_z) is the point representing the origin of the end Effecter

   While this function represents the true Euclidian distance between the two points, we do not require the true distance for this situation. Thus, the square root can be dropped since the minima will still be at the same value of t and thus simplifying the problem to:

Equation 11

$$\boxed{d = (x - e_x)^2 + (y - e_y)^2 + (z - e_z)^2}$$

The minima can be found by solving the derivative of *Equation 11* in relation to `t`.

$$\boxed{\frac{dd}{dt} = \frac{d}{dt}(x - e_x)^2 + (y - e_y)^2 + (z - e_z)^2 = 0}$$

Giving:

Equation 12

$$\boxed{t = -\frac{ax_0 - ae_x + by_0 - be_y + cz_0 - ce_z}{a^2 + b^2 + c^2}}$$

We can now substitute `t` from *Equation 12* back into *Equation 9*, thus yielding the point we were originally looking for.

2. If the point we have found from step 1 is the same as the End Effecter, we can deduce quickly that the radius is zero with no further calculations and thus no force is applied.

3. We must now find the vector which the torque will produce a force on the end effecter. To do this, we find the normal vector to the plane which passes through two points on the axis and the end effecter.

   Since the normal remains constant (despite any offsets in the plane), we can take any three points on the plane and convert them into two vectors, which will describe a parallel plane which passes though the origin.

   These vectors can be obtained from:

Equation 13

$$\boxed{\begin{matrix} a = v - u \\ b = w - u \end{matrix}}$$

Where:

- a and b are vectors which describe the parallel plane
- u, v, and w are discrete points which lie on the original plane

To find the vector normal to this plane, requires only to find the cross product of a and b from *Equation 13*.

Equation 14

$$n = a \times b$$

While this vector is in the direction of the force it may still be of any length so we must normalize before using it in any further force calculations.

Equation 15

$$\hat{n} = \frac{n}{\sqrt{(n_x^2 + n_y^2 + n_z^2)}}$$

4. To find the force applied to the End Effecter, torque provided from the servo is divided by the minimum distance between the axis and the End Effecter.

Equation 16

$$F = \frac{\tau}{\sqrt{(p_x - e_x)^2 + (p_y - e_y)^2 + (p_z - e_z)^2}}$$

Where:

- τ is the torque from the servo
- p is the point which was found from step 1
- e is the End Effecter location

5. We can now scale the vector found in *Equation 15* by the force found in *Equation 16* to obtain the force components applied by the servo.

Equation 17

$$F_{servo} = F \times \hat{n}$$

With the force components calculated for each servo we may now find the sum in order to arrive at the total force on the End Effecter.

Equation 18

$$F_{total} = \sum_{x=0}^{x=5} F_{servo\,x}$$

Inverse Kinematics

Calculating the required torque from each servo within the chain is challenging. This is due to the multi variable nature of the system (6 DOF). This entails the solving of six equations simultaneously, as we know the contribution from each motor to the overall force via performing a forward kinematic algorithm. These equations can be composited into matrix form:

$$\begin{bmatrix} Force_{d_1 m_1} & \cdots & Force_{d_1 m_6} \\ \vdots & \ddots & \vdots \\ Force_{d_6 m_1} & \cdots & Force_{d_6 m_6} \end{bmatrix} = \begin{pmatrix} DesiredForce_{d_1} \\ \vdots \\ DesiredForce_{d_6} \end{pmatrix}$$

where:

d_a is the a'th dimension.

m_b is the b'th motor.

An equation of this form can be solved using Gaussian elimination, so long as the matrix is rank sufficient. Unfortunately, there are many locations within the workspace of arm where dexterity is lost. An example of lost dexterity is when the arm is at maximum reach as seen in *Figure 37* (while in this configuration, it is impossible to produce a force which pushes out from the manipulator).

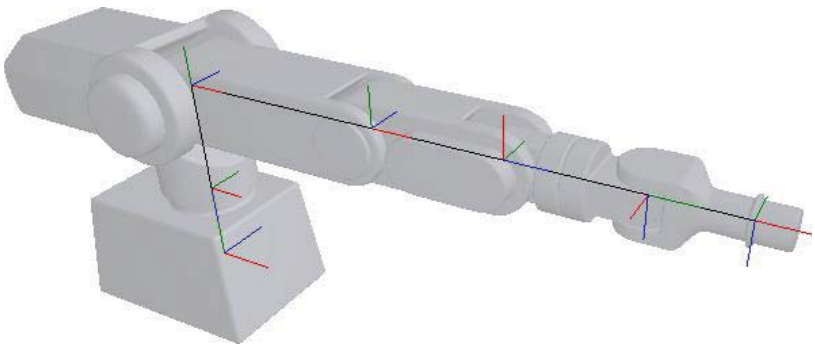


Figure 37: Arm configuration with low dexterity

Due to this nature, there are many cases where the solution matrix is rank deficient, which renders standard Gaussian elimination programmatically unstable (since an exact solution does not exist). This can be illustrated in a two dimensional example. *Figure 38* illustrates a simple bar and rotary joint (rotated 45˚), where we wish to find a torque which will result in a force along the Y axis, and zero force along the X axis. Due to the rank deficient nature of this problem, a solution does not exist.

Figure 38 shows the inversely proportional relationship between the errors of actual force to that which is desired in each dimension (blue and green). While it is impossible to achieve the desired forces, we can minimise the error of the overall system (shown in red). This is achieved by minimising the equation given by:

$$error = \sqrt{(actual_x - desired_x)^2 + \left(actual_y - desired_y\right)^2}$$

Where:

$$actual = \begin{pmatrix} Sin\left(\frac{\pi}{4}\right) \\ Cos\left(\frac{\pi}{4}\right) \end{pmatrix} \cdot \tau = \begin{pmatrix} 0.707 \\ 0.707 \end{pmatrix} \cdot \tau$$

Equation 19: Squared Error

Looking at the graph in *Figure 38*, we can see the squared error is at its minimum when both forces are equal in both dimensions.

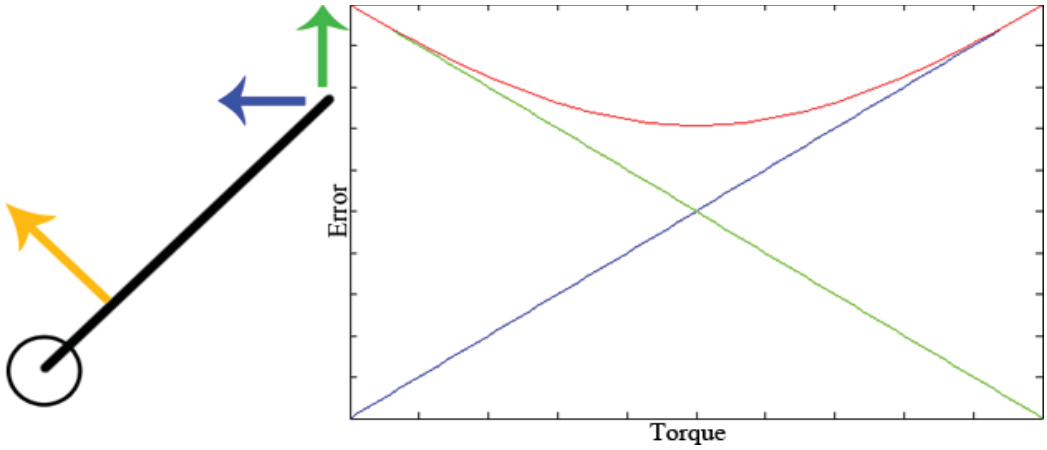


Figure 38: Two dimensional forces and least mean squared error

As the dimensions increase, it becomes increasingly complex to locate the minimum local. To solve for the minimum error in multiple dimensions, we must find the generalised pseudo-inverse matrix, also known as the Moore-Penrose Pseudo Inverse; to quote (Ellery, 2000, p. 173):

> There exists a generalised pseudo-inverse Jacobian which provides a useful least squares solution such that:
>
> $$\dot{\theta} = J^{+}\dot{q}$$
>
> The least-squares estimator can be used to find the generalised inverse. The error to be minimised with respect to $\dot{\theta}$:
>
> $$\begin{matrix} \dot{\theta} = J^{T}\lambda \\ \dot{q} = J\dot{\theta} \end{matrix} \rightarrow \dot{q} = JJ^{T}\lambda \rightarrow \lambda = (JJ^{T})^{-1}\dot{q}$$
>
> Now,
>
> $$\dot{\theta} = J^{T}\lambda \rightarrow \dot{\theta} = J^{T}(JJ^{T})^{-1}\dot{q}$$
>
> Where
>
> $$J^{+} = J^{T}(JJ^{T})^{-1}$$
>
> This is the Moore-Penrose pseudo inverse which is optimal in terms of the minimum covariance of $\dot{\theta}$ and follows the following properties:
>
> (i) $JJ^{+}J = J^{+};$
> (ii) $J^{+}JJ^{+} = J;$
> (iii) $(J^{+}J)^{T} = J^{+}J;$
> (iv) $(JJ^{+})^{T} = JJ^{+}$

Equation 20: Moore-Penrose Pseudo Inverse

Implemented via a compiled MatLab function (using the Pseudo Inverse), we can find the solution which has the least mean squared error. That is the solution which, in all six degrees of freedom, is the closest to that specified.

In the haptic environment, it may be that all degrees of freedom should not receive an equal weighting of importance. However under the current implementation, weighting is not possible and is an area for further research.

It should be noted that during computation, some values which should compute to zero will have a small value caused by computational discrepancies through the use of floating point values. These should be detected and set to zero before the inverse is applied to increase stability of the solution (Aster, Thurber, & Borchers, 2005).

Simulator Outcome

A mechanism which can be 'plugged-in' to the LONGBone interface has been developed allowing users to see the state of the haptic input device. While the graphical interface may not be used, the mechanical simulation data can be used to reduce artefacts felt by the user of the input device. It may be that the simulator could be re-written as a low level driver if the device were to advance to a 'plug and play' device.

The simulation has indicated that the manipulator, in its current configuration, is lacking dexterity. This is visible via the effect of limited positions where the manipulator can create the desired output forces, without substantial error of haptic feedback in one or more DOF.

Chapter 5: Motor Controller

Board Layout

The controller which comes standard with the RV-M1 provides insufficient control and feedback parameters to implement a haptic system. Namely the controller has been implemented to perform simple pick and place operations, with no control over torque exerted from each joint, and no feedback of position, velocity or acceleration is provided.

It was decided that modifying the exiting controller would be infeasible; as such getting the required features would require a new controller to be designed and built from the ground up. The following requirements were identified:

- RS-232 Communication to PC.
- Varying supply voltages depending on drive configuration.
- Sufficient power through-put for each motor.
- Current sensing and control for each joint.
- Position feedback.
- Scalability.

It was originally decided that the best design method would be to design a master board (*Figure 39*), which would provide the all external connection sockets, voltage regulation, communications interface, and six sockets for child boards. These child boards would then contain localised communication, processing and motor driver circuitry for individual joints within the RV-M1. However, later considerations of production and component sourcing made it more realistic to place all circuitry on a single PCB.

Unfortunately, the controller was never physically realised due to manufacturing constraints which were initially overlooked. Furthermore, time constraints prevented re-design and manufacture of the controller and as such it was decided that additional effort should be given

to software components of the project. Due to lack of a physical system to develop against, firmware for the controller was not developed to any considerable point.

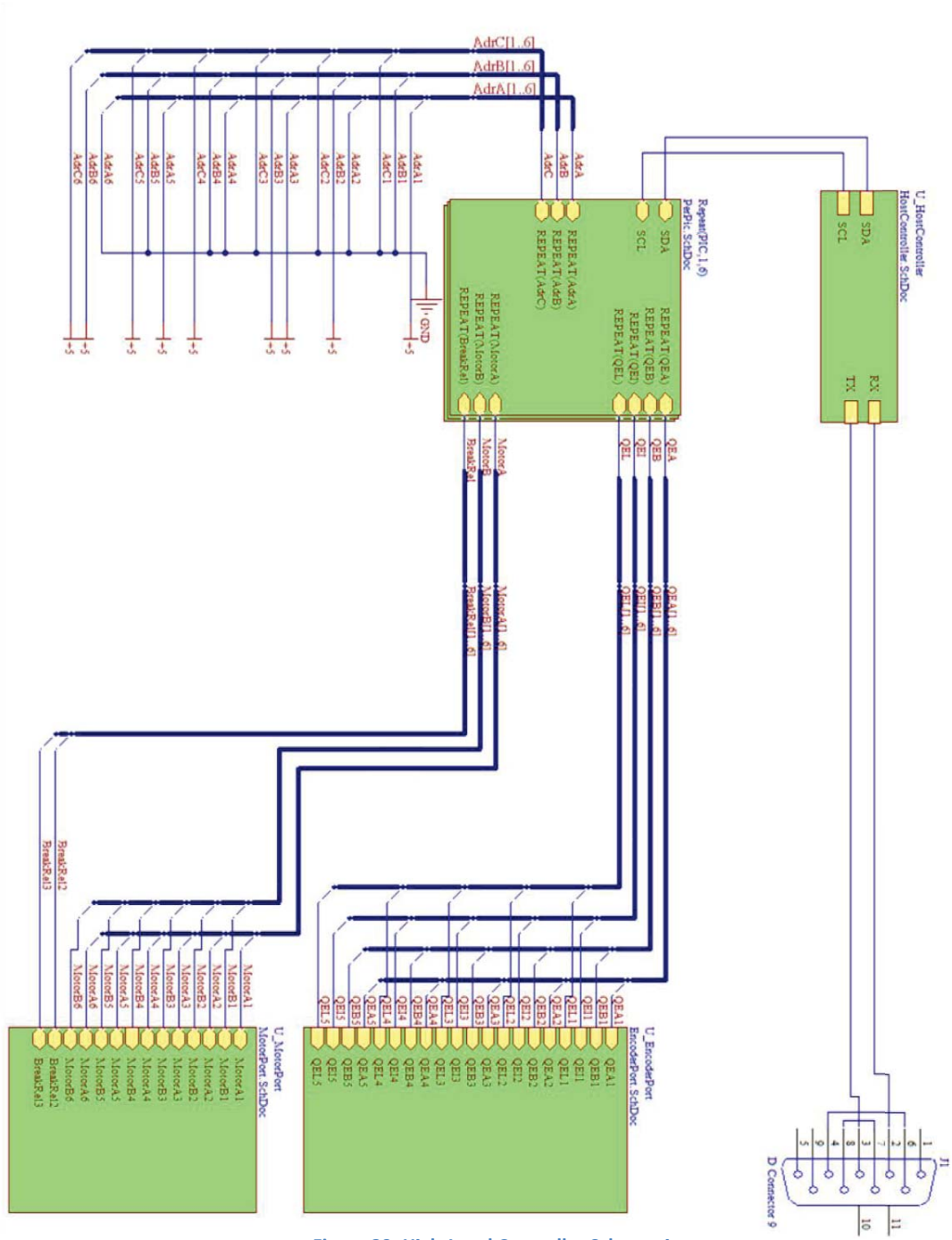


Figure 39: High-Level Controller Schematic

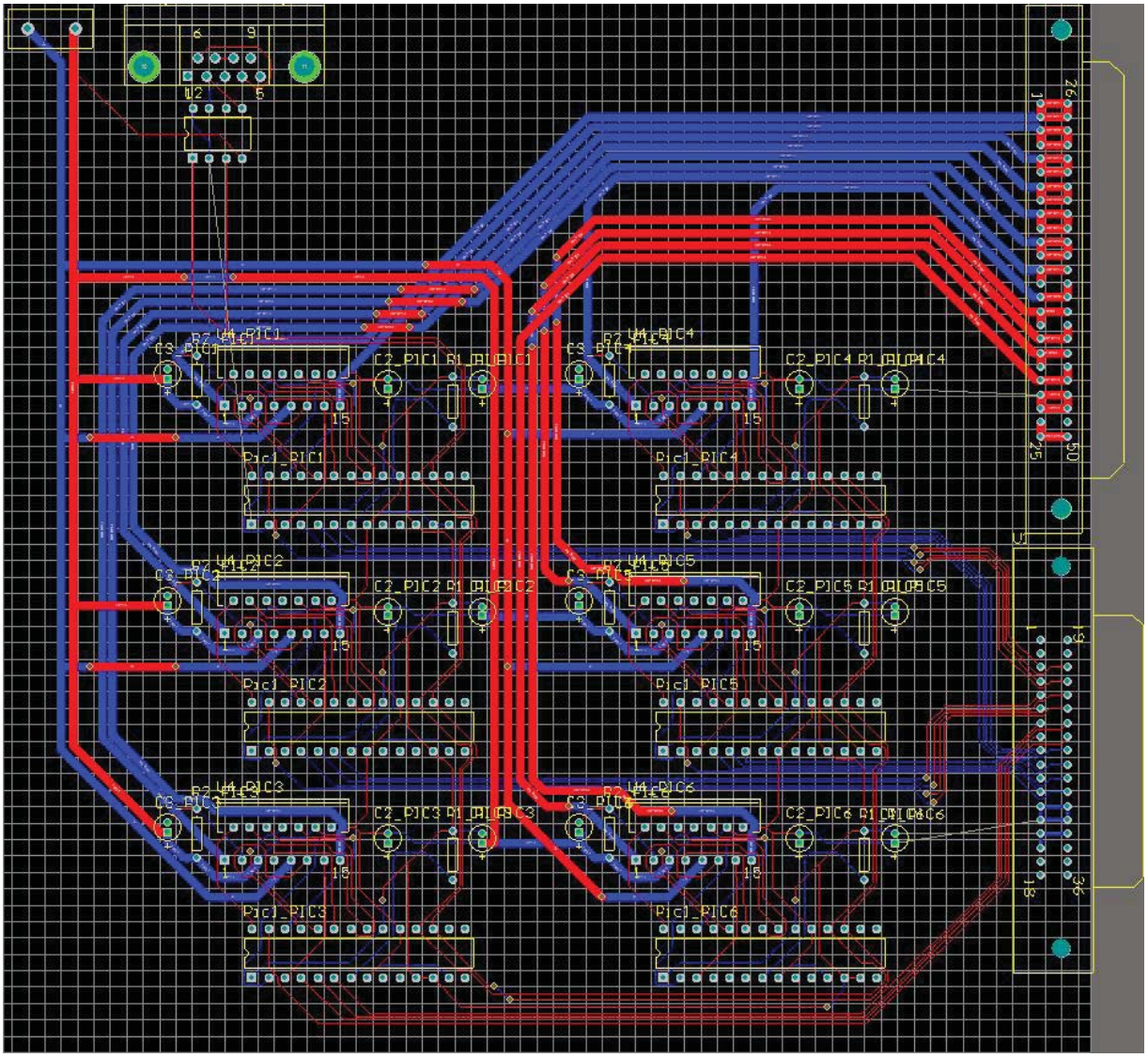


Figure 40: Controller PCB

Communications

Initial design of the controller board (*Figure 40*) is designed such that the host PC will control all joints individually via an implementation of a RS-232 bus. This involves all motor controllers being placed on the same RX/TX lines with flow control implemented from the PC via an address-request-response protocol. Advancement on this technique (which is seen in *Figure 39*), would be the implementation of an I^2C bus within the control board which allows for additional re-useability in code and hardware designs.

All communications with the controller board from the host PC must follow the specified format. This format is address-instruction-data-response-ACK/NAK. Depending on the instruction, there may be no data or response expected. The ACK byte is 0x99 (10011001), while the NAK byte is 0x66 (01100110). Instructions and expected data/responses are detailed in *Table 2*.

Instruction	Data	Resp 1	Resp 2	Description
0x01	0-3	-	-	Set motor direction; 0 Lock, 1 Clockwise, 2 Anti-Clockwise
0x03	0-255	-	-	Set motor current; 0 Min, 255 Max
0x04	0-255	-	-	Set max motor current; Amps * 50 (5.1A max)
0x05	-	MSB	LSB	Motor position
0x06	-	MSB	LSB	Motor current
0x07	-	LSB	-	Max motor current

Table 2: RS-232 Instruction Set

Future I²C internal communication implementation

It would be beneficial to move low level control over the motor drivers from the host PC to a single master PIC16F690; this would decrease RS-232 and allow internal polling for data, thus reducing response delays. Internal communications between child modules and the aforementioned PIC16F690 should be implemented via an Inter-Integrated Circuit (I^2C) bus, utilising the PIC16F690 as the I^2C master device.

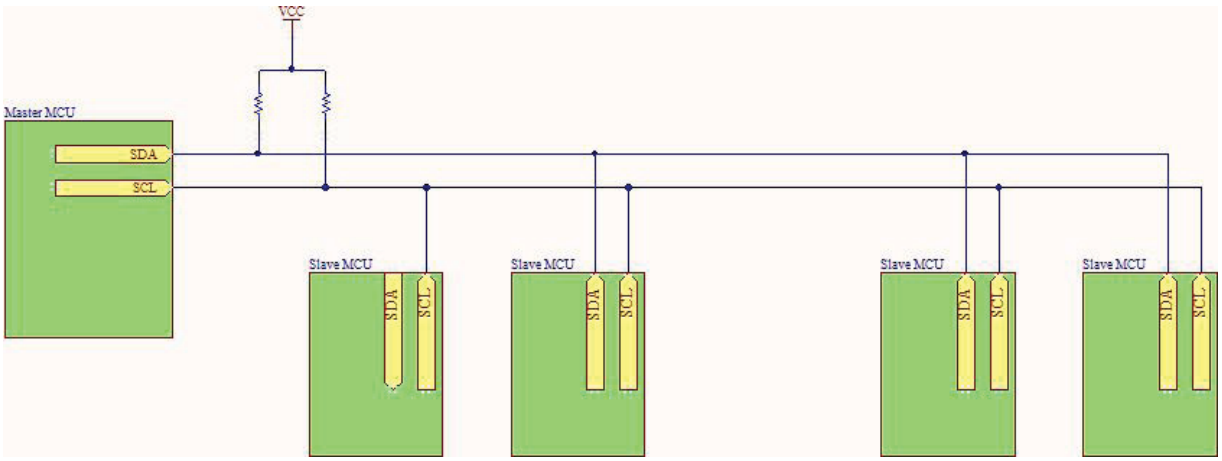


Figure 41: I²C Master-Slave Intercommunications

"The two wires used to interconnect with I²C are **SDA** (serial data) and **SCL** (serial clock). Both lines are open-drain. They are connected to a positive supply via a pull-up resistor and therefore remain high when not in use." (Catsoulis, 2005).

The I²C protocol requires that a start sequence be sent from the master before any transmission is initiated and a stop sequence once terminated. This is due to the multi-master capability of I²C. These two sequences are the only occasion on which the data line should change state while the clock is high. A start sequence involves the data line being pulled low whilst the clock is active. While a stop sequence is the opposite; the data line is released while the clock is high, *Figure 42* shows these sequences.

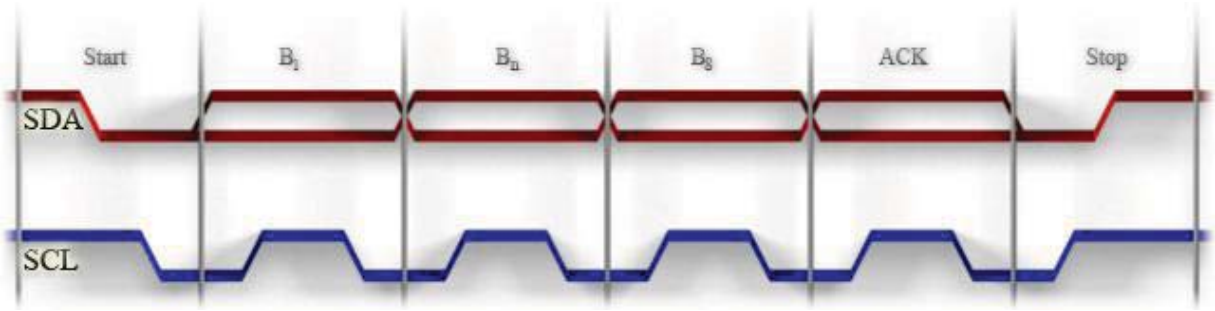


Figure 42: I²C Waveform Protocol

After the transmission of a byte, an ACK bit is expected to be returned from the receiving MCU. This is due to the passive high nature of I²C. It is expected that a stop sequence be present once the transaction is fully complete, rather than on the conclusion of each byte.

Typical transmission sequences can be seen in *Figure 43* and *Figure 44*. In some devices, it is expected that a NAK be transmitted before the Stop sequence to indicate that reading has ceased. It is common for a write sequence, to inform the slave device, which registers must be read from, to precede the read sequence.

Figure 43: Typical I²C Write Sequence

Figure 44: Typical I²C Read Sequence

Above *Figure 43* and *Figure 44* demonstrate which device has control over the SDA line during the transmission sequence; red designates the master MCU, while blue indicates the slave. *Table 3* shows the SDA line levels associated with defined bits within the read/write sequences.

TAG	SDA Line Level
ACK	Low
NAK	High
Write Flag	High
Read Flag	Low

Table 3: I²C Designated Values

Each child module has a hardwired address (*Figure 39*), allowing the same software to be implemented on each microcontroller within the motor controller modules. This hardware address should be used to determine the 7-Bit I²C address, which begin from Hex 0x30 and go through to 0x40, as seen in *Table 4*.

It is important that the I²C address is not an odd value; that is to say, the least significant bit (LSB) of the binary address is always low. This is required due to the software implementation of the I²C protocol using 7-Bit addressing.

Adr A	Adr B	Adr C	I²C Adr (HEX)	I²C Adr (Binary)	I²C Adr (DEC)

1	0	0	0x30	0011 0000	48
0	1	0	0x32	0011 0010	50
1	1	0	0x34	0011 0100	52
0	0	1	0x36	0011 0110	54
1	0	1	0x38	0011 1000	56
0	1	1	0x40	0011 1010	58

Table 4: Child Module I²C Addressing

Since the PIC16F690 does not have hardware support for implementation of an I²C master device, all communications must be handled via a software implementation (Bit-Banging). The I²C bus operates at approximately 100 kHz; well within the internal oscillator frequency of 8 MHz. However, it remains impractical to implement a method using timer interrupts. This requires the use of hard-coded routines which take a known period of time in order to provide delays within the send/receive sequence. Unfortunately, this approach prevents additional processing on the MCU during I²C communications.

```
char i2csendbyte(char Byte)
{
 char count;
 SDA=I2CLOW;
 SCL=I2CLOW;

 i2cdelay(I2CCLOCKLOW);          //-- Minimum Clock Low Time

 for(count=8;count>0;count--)    //-- Send 8 bits of data
 {
  if( (Byte & 0x80)== 0)         //-- Get the Bit
  {
   SDA=I2CLOW;                   //-- Ensure Port pin is low
   SDA_TRIS=I2CLOW;              //-- Lower pin if bit = 0
  }
  else
  {
   SDA_TRIS=I2CHIGH;             //-- Release pin if bit = 1
  }
  Byte=Byte<<1;                  //-- Shift next bit into position
  i2cclock();                    //-- Pulse the clock
 }
 SDA_TRIS=I2CHIGH;               //-- Release data pin for ACK
 return(1);
}
```

Snippet 11: I²C Send Byte

```
void i2cclock(void)
{
 DelayUs(I2CDATASETTLE);          //-- Minimum Clock Low Time
 SCL_TRIS=I2CHIGH;                //-- Release clock
 DelayUs(I2CCLOCKHIGH);           //-- Minimum Clock High Time
 SCL_TRIS=I2CLOW;                 //-- Lower the clock
 DelayUs(I2CCLOCKLOW);            //-- Minimum Clock Low Time
}
```

Snippet 12: Generate I²C Clock Pulse

```
#define      DelayUs(x)   { unsigned char _dcnt; \
                    _dcnt = (x)/((12MHZ)/(XTAL_FREQ))|1; \
                    while(--_dcnt != 0) \
                          continue; }
#endif
```

Snippet 13: Delay Loop

As seen from i2cclock in *Snippet 12*, there are three calls to a delay macro (*Snippet 13*). This function simply loops until the specified time has elapsed. The number of times to loop is determined by the selected clock frequency `(XTAL_FREQ)` and known properties of the PIC16F690 microcontroller.

Chapter 6: Conclusion and Future Recommendations

The project to retrofit the Mitsubishi RV-M1 into a haptic input device has proved difficult to complete. The initial problem faced is the design of the RV-M1. As the manipulator is designed for pick and place operations, it provides only five DOF, rather than the required six. In order to achieve the desired feedback, addition of an additional drive linkage to the end effecter would be required.

The manipulator provides a high drive-back force and inertia components when the device is decoupled. This is caused by the gearing ratios used within the device, and has the effect that the arm is difficult move with any precision. Further to this, resolution of forces applied to the arm would be too low to function as an effective input device.

As the controller which is supplied with the RV-M1 does not provide any method of force control or measurement, a new controller has been designed. However, as time constraints prevented the design to be implemented, functionality still remains unknown. A lack of controller has prevented further development in linking the physical device and simulation.

Simulation of the manipulator has successfully demonstrated the ability of the manipulator to behave as a force feedback device. However, the workspace where forces are rendered (without introduction of error) is too small for the femur realignment scenario. This suggests that the use of a parallel device, similar to that of the Delta.6[4], would be a more effective configuration.

The design of GUI for integration with other elements developed outside of the project has produced an effective means of presenting the user with data in an aesthetically pleasing manner.

[4] Delta.6 (p. 6) - Appendix C

Further work should be carried out on designing a novel haptic platform which provides a large work area with a high level of dexterity. This device should have low drive-back forces and be as close to statistically balanced as possible to maintain dynamic range.

The electrical controller for this device should be designed to interface to PC via modern methods (USB) and feature plug and play type operation. This would serve in reducing complexity and make progress towards developing an easily integrated system.

Glossary

- API — Application Programming Interface
- C# — C# programming language
- CPU — Central Processing Unit
- GUI — Graphical User Interface
- GPU — Graphics Processing Unit
- SDK — Software Development Kit. Provides examples and required files for a developer to learn to use an API.
- Shader — A short program executed on the GPU to define how an object is rendered
- Haptics — The means by which humans and other animals communicate via touch.
- USB — Universal Serial Bus
- HMI — Human – Machine interface. The interface which humans and machine communicate.
- HUD — Heads up display. An image which is overlayed a display
- RV-M1 — The model of the Mitsubishi manipulation arm used for the project
- I2C — Inter-Integrated Circuit. A communication specification for communication between integrated circuits.
- HLSL — High Level Shader Language. Shader programming language which is compatible with Microsoft DirectX.
- NURBS — NonUniform Rational B-Spline.
- PCB — Printed Circuit Board.
- GLSL — Shader language compatible with the OpenGL library.
- OpenGL — 3D Graphics interface API.
- Direct3D — 3D Graphics interface API, developed by Microsoft.
- D-H Link — Danevert Burnburg transformation link.

References

Aster, R. C., Thurber, C. H., & Borchers, B. (2005). *Parameter estimation and inverse problems.* Academic Press.

Batteau, L. M., Liu, A., Maintz, A. A., Bhasin, Y., & Bowyer, M. W. (2004). A Study on the Perception of Haptics in Surgical Simulation. *International Symposium on Medical Simulation* (pp. 185-192). Verlag Berlin Heidelberg: Springer.

Bicchi, A., Buss, M., Ernst, M. O., & Peer, A. (2008). *The Sense of Touch and Its Rendering: Progress in Haptics Research.* Springer.

Catsoulis, J. (2005). *Designing embedded hardware* (2, illustrated ed.). Sebastopol, CA, USA: O'Reilly.

Cawood, S., & McGee, P. (2007). *Microsoft XNA game studio creator's guide.* McGraw-Hill Professional.

Chen, J. X. (2003). *Guide to graphics software tools.* Springer.

Ellery, A. (2000). *An Introduction to Space Robotics.* Springer.

Fernando, Randy; Harris, Mark; Wloka, Matthias; Zeller, Cyril. (2004). *Introduction to the Hardware Graphics Pipeline.* Retrieved 2009, from nVidia Developer Zone: http://developer.nvidia.com/object/eg_2004_presentations.html

Gerovichev, O., Marayong, P., & Okamura, A. M. (2002). The Effect of Visual and Haptic Feedback on Manual and Teleoperated Needle Insertion. *Medical Image Computing and Computer-assisted Intervention MICCAI 2002, 5th International Conference, Tokyo, Japan, September 25-28, 2002: Proceedings* (pp. 147-154). Verlag Berlin Heidelberg: Springer.

Jacko, J. A., & Sears, A. (2003). *The human-computer interaction handbook: fundamentals, evolving technologies, and emerging applications.* Kawrence Erlbaum Associates.

Jarvis, R. A., & Zelinsky, A. (2003). *Robotics research: the tenth international symposium.* Springer.

Jian, Z. (2005, April 12). *Normal Mapping: Theory and Practice*. Retrieved 2009, from HIGHEND3D: http://www.highend3d.com/maya/tutorials/texturing/229-2.html

Johnson, J. (2000). *GUI Bloopers.* Morgan Kaufmann.

Lengyel, E. (2004). *Mathematics for 3D game programming and computer graphics.* Cengage Learning.

Luna, F. D. (2006). *ntroduction to 3D game programming with DirectX 9.0c: a shader approach.* Wordware Publishing, Inc.

Massie, T. H., & Salisbury, J. K. (1994, November). *The PHANTOM Haptic Interface: A Device for Probing Virtual Objects.* Retrieved Feburary 2009, from Haptic Projects & Papers: http://www.sensable.com/documents/documents/ASME94.pdf

Microsoft Corporation. (2008, April). *The Direct3D Transformation Pipeline*. Retrieved January 14, 2009, from Microsoft Accessibility Developer Center: http://msdn.microsoft.com/en-us/library/bb206260(VS.85).aspx

Möller, T., & Haines, E. (2002). *Real-time rendering* (2nd Edition ed.). A K Peters, Ltd.

Murray, R. M., Li, Z. S., & Sastry, S. (1994). *A Mathematical Introduction to Robotic Manipulation.* CRC Press.

Parent, R. (2002). *Computer animation: algorithms and techniques.* Morgan Kaufmann.

Paterson, M. (2007). *The Senses of Touch: Haptics, Affects and Technologies.* Berg Publishers.

Shabana, A. A. (2005). *Dynamics of multibody systems.* Cambridge University Press.

Siciliano, B., & Khatib, O. (2008). *Springer Handbook of Robotics.* Springer.

St-Laurent, S. (2004). *Shaders for Game Programmers and Artists.* Cengage Learning.

Tavakoli, M., Patel, R. V., Moallem, M., & Aziminejad, A. (2008). *Haptics for Teleoperated Surgical Robotic Systems.* World Scientific.

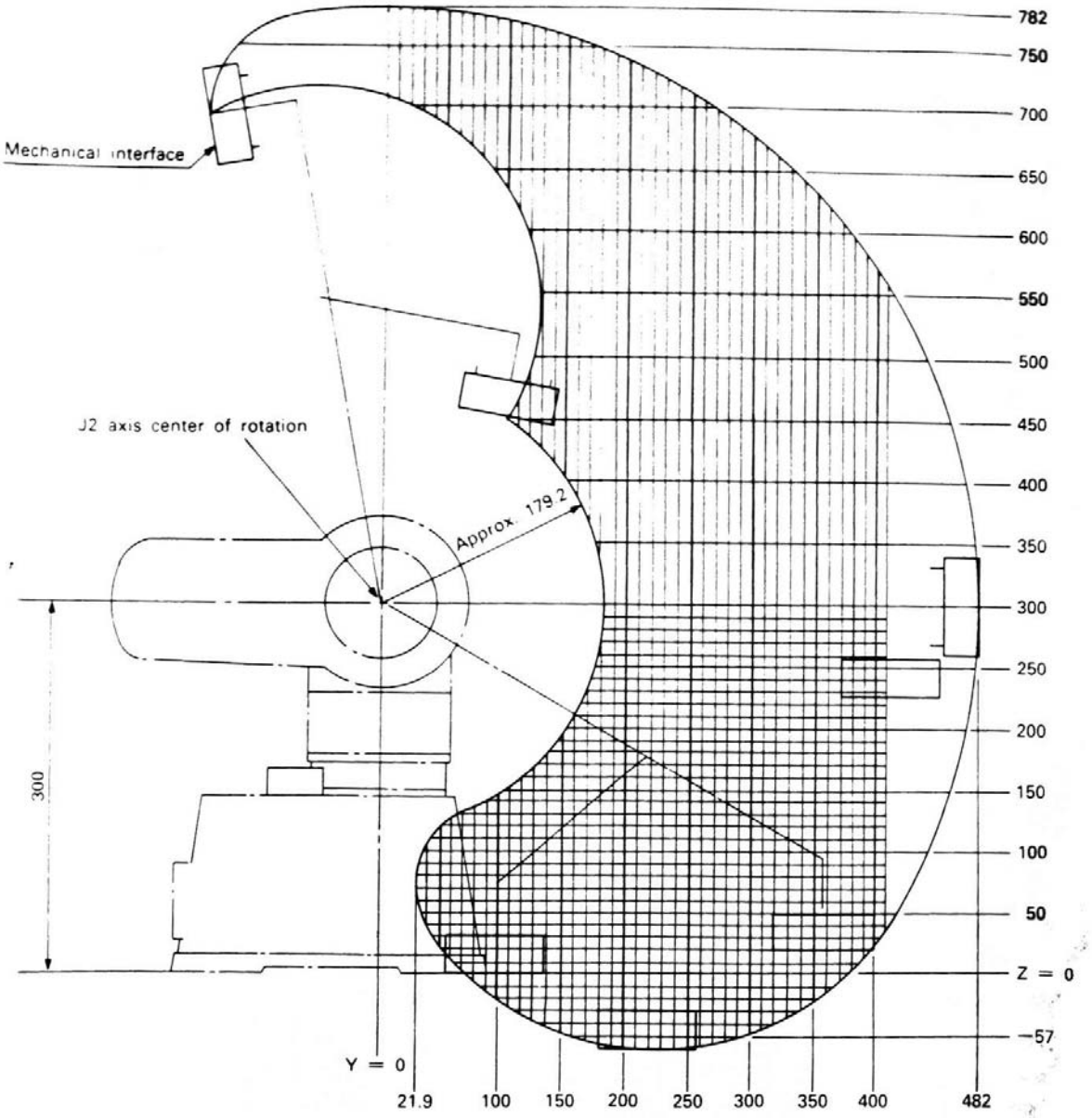
Udupa, J. K., & Herman, G. T. (1999). *3D Imaging in Medicine.* CRC Press.

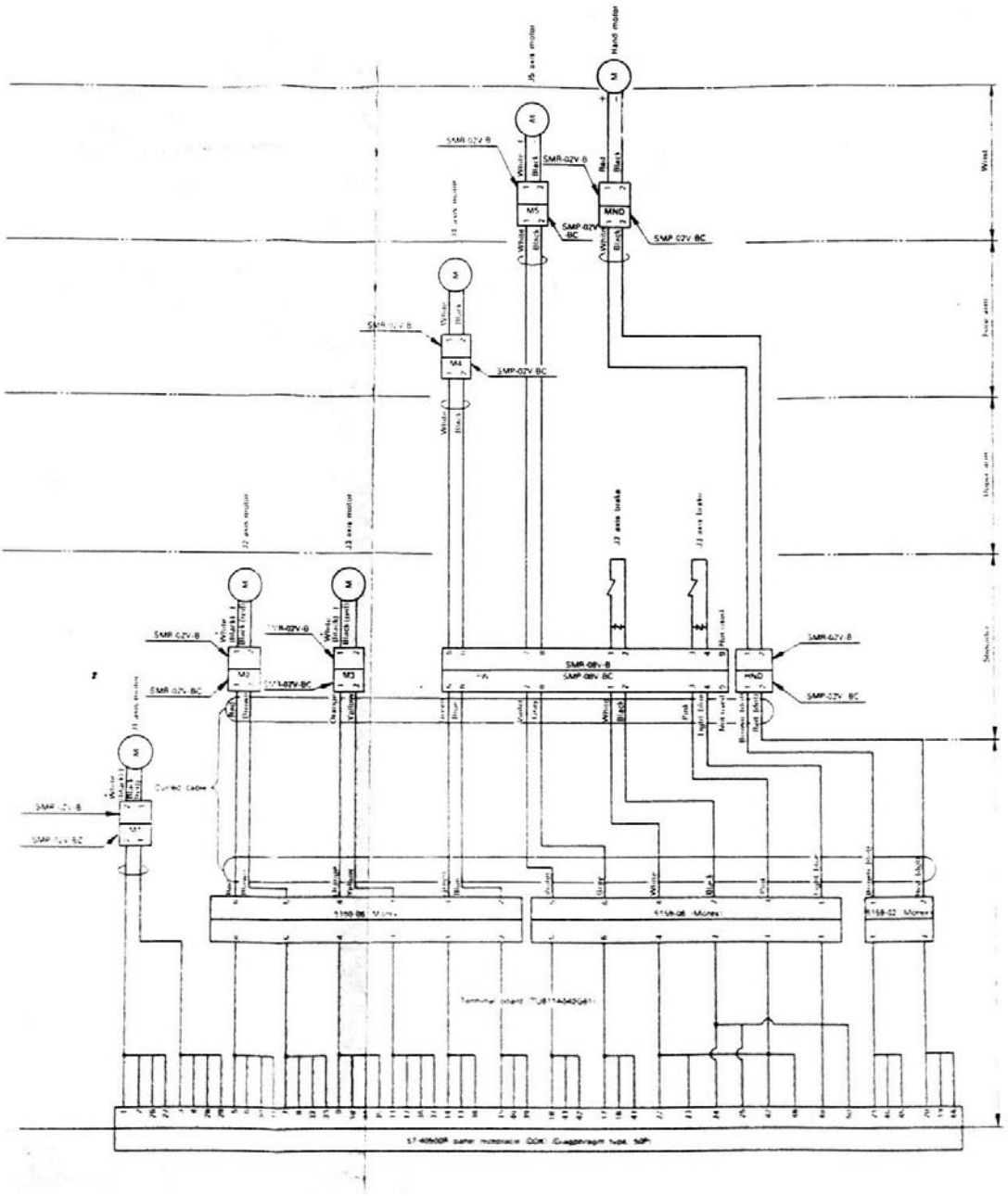
Weisstein, E. W. (n.d.). *Cartesian Coordinates.* Retrieved 2009, from MathWorld--A Wolfram Web Resource: http://mathworld.wolfram.com/CartesianCoordinates.html

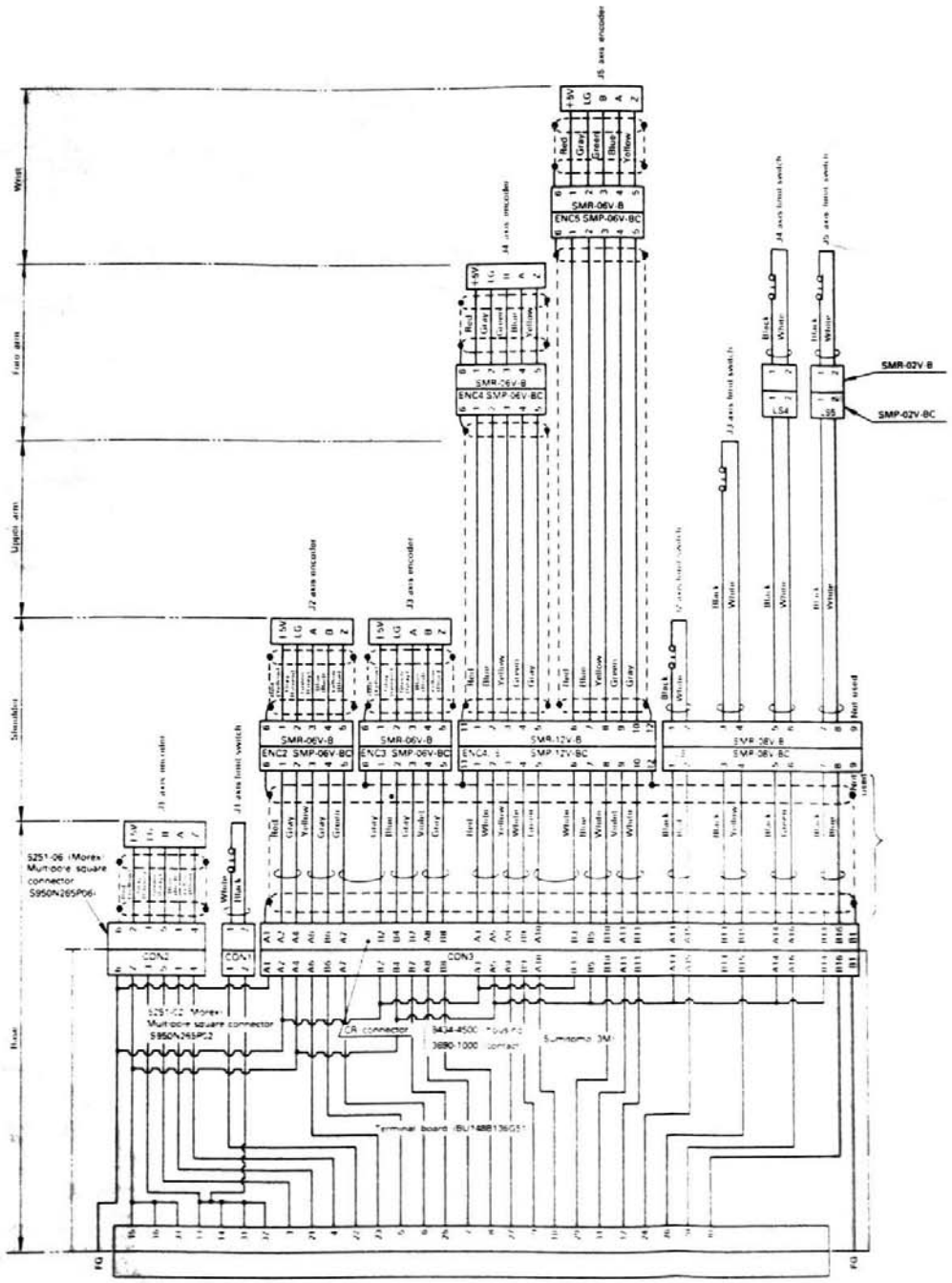
Wnek, G. E., & Bowlin, G. L. (2008). *Encyclopedia of Biomaterials and Biomedical Engineering.* Informa Health Care.

Appendices

Apendix A (RV-M1 Schematics)

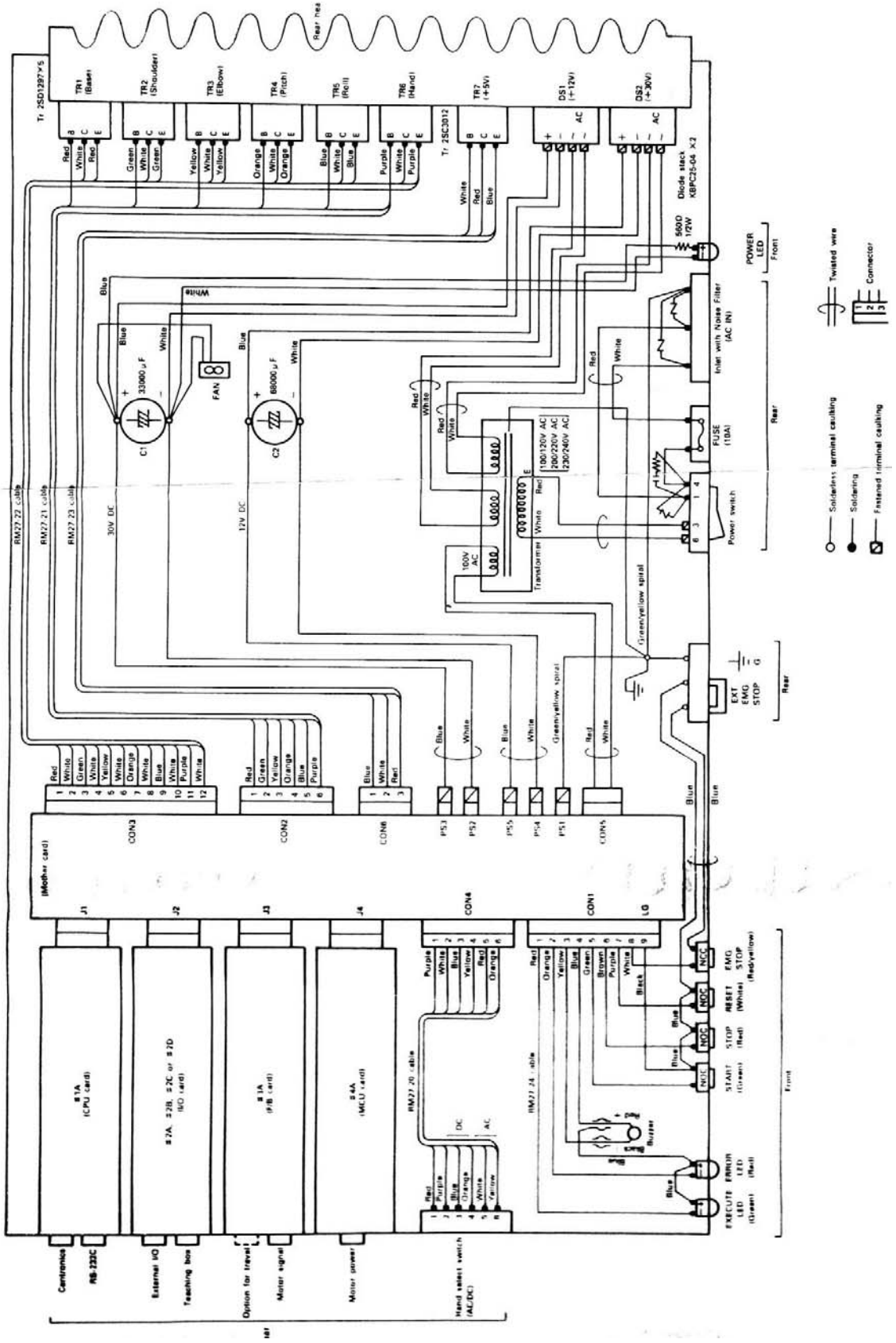



J6 axis motor
Hand motor
Wrist
J5 axis motor
SMR-02V-B
SMP-02V-BC
M5
MND
M4
J2 axis motor
J3 axis motor
J2 axis brake
J3 axis brake
M2
M3
SMR-08V-B
SMP-08V-BC
Upper arm
Fore arm
Shoulder
Current cable
M1
Terminal board



* The RV-MI's J1 to J3 axis motor builders depend on lots as follows:

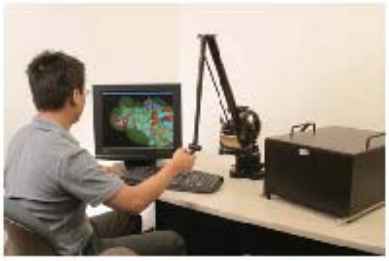
* Sanyo Electric Co., Ltd. motor (frame in black) wire color are indicated outside parentheses.
Daito Seisakusho Co., Ltd. motor (frame in gold) wire colors are indicated in parentheses.


Rear nea
TR1 (Base)
TR2 (Shoulder)
TR3 (Elbow)
TR4 (Pitch)
TR5 (Roll)
TR6 (Hand)
TR7 (+5V)
DS1 (+12V)
DS2 (+30V)
Tr 2SD1297*S
Tr 2SC3012
Diode stack KBPC25-04 ×2
560Ω 1/2W
POWER LED
Front
Inlet with Noise Filter (AC IN)
FUSE (10A)
Power switch
Rear
Transformer
100V AC
180/120V AC
200/220V AC
230/240V AC
33000 μF
68000 μF
C1
C2
FAN
30V DC
12V DC
RM27 22 cable
RM27 21 cable
RM27 23 cable
RM27 20 cable
RM27 24 cable
Green/yellow spiral
EXT EMG STOP
Twisted wire
Connector
Solderless terminal caulking
Soldering
Fastened terminal caulking
(Mother card)
CON3
CON2
CON6
CON4
CON1
CON5
PS3
PS2
PS5
PS4
PS1
J1
J2
J3
J4
LG
B1A (CPU card)
B2A, 2B, 2C or 2D (I/O card)
B3A (P/B card)
B4A (MCU card)
NCC
NOC
EMG STOP (Red/yellow)
RESET (White)
STOP (Red)
START (Green)
Buzzer
ERROR LED (Red)
EXECUTE LED (Green)
Centronics
RS-232C
External I/O
Teaching box
Option for travel
Motor signal
Motor power
Hand select switch (AC/DC)
Front

Apendix B (Phantom Premium Datasheet)

SensAble technologies

SPECIFICATIONS FOR THE PHANTOM® PREMIUM 3.0/6DOF HAPTIC DEVICE



Workspace	Translational	33 W x 23 H x 16 D inches 838 W x 584 H x 406 D mm
	Rotational Yaw Pitch Roll	 297 degrees / 5.18 radians 260 degrees / 4.54 radians 335 degrees / 5.85 radians
Footprint	Detachable portion	8 W x 8 D inches / 203 W x 203 D mm
	Electronics console (AMP box)	15 W x 20 D inches / 381 W x 508 D mm
Range of motion	Full arm movement pivoting at shoulder	
Nominal position resolution	Translational	> 1000 dpi / ~ 0.02 mm
	Rotational Yaw & Pitch Roll	 0.0023 degrees / 0.00004 radians 0.0080 degrees / 0.00014 radians
Backdrive friction **(x, y, z)**	0.75 oz 0.2 N	
Maximum exertable force and torque at nominal position (orthogonal arms)	Translational	4.9 lbf / 22N
	Rotational Yaw & Pitch Roll	 73 oz-in / 515 mNm 24 oz-in / 170 mNm
Continuous exertable force and torque at nominal position (orthogonal arms)	Translational	0.7 lbf / 3 N
	Rotational Yaw & Pitch Roll	 27 oz-in / 188 mNm 7 oz-in / 48 mNm
Stiffness	5.7 lbf in^{-1} / 1 N mm^{-1}	
Inertia (apparent mass at tip)	< 0.48 lbm / < 220 g	
Weight (device only)	Detachable portion	16 lb / 7.3 Kg
	Electronics console (AMP box)	53 lb / 24 Kg
Force feedback	x, y, z, Tx, Ty, Tz	
Position sensing	x, y, z, roll, pitch, yaw	
Interface	Parallel port	
Supported Platforms	Intel-based PCs	
GHOST® SDK Compatibility	Upon special request	
OpenHaptics™ Toolkit Compatibility	Yes	

'SenSitus' molecular docking software package shown above courtesy of Stefan Birmanns, Ph.D. of the Laboratory for Structural Bioinformatics, University of Texas Health Science Center at Houston.

Apendix C (Delta.6 Datasheet)

force dimension

Delta Haptic Device

6-DOF force feedback interface

The 6-DOF Delta Haptic Device is a high performance haptic device based on the Delta manipulator. It offers 6 active degrees-of-freedom in translation and rotation and was designed to display high-fidelity, high-quality kinesthetic and tactile information. Thanks to its unique parallel mechanical conception, the Delta Haptic Device can convey a large range of forces and torques over a large workspace, unlike other haptic structures which have either limited force capability or smaller workspace. In addition, parallel mechanics along with base-mounted actuators leads to high stiffness and very low inertia. These characteristics make the Delta Haptic Device one of the best haptic tools available today.

workspace	translation	Ø 360 mm x L 300mm
	rotation	± 20 deg / axis
forces	continuous	20.0 N
torques	continuous	0.2 Nm
resolution	linear	‹ 0.03 mm
	angular	‹ 0.04 deg
stiffness	closed loop	14.5 N/mm
dimensions	height	600 mm
	width	700 mm
	depth	400 - 750 mm
interface	standard	PCI I/O
power	universal	110V - 240V
OS	Windows	NT / 2000 / XP
	Linux	RH / Fedora
	Apple	OS-X
software	DHD-API	
calibration	precision reference point	
structure	parallel (delta-based)	
safety	velocity monitoring	
	electromagnetic brakes	

Force Dimension
PSE-C
CH-1015 Lausanne
Switzerland

t +41 21 693 1911
f +41 21 693 1910

www.forcedimension.com
info@forcedimension.com

Apendix D (Omega.6 Datasheet)

force dimension

omega.6 haptic device

force feedback interface

The **omega.6** is the most advanced pen-shaped force-feedback device available. Building on the omega.3 base, its design provides **perfect decoupling** of translations and rotations. The combination of **full gravity compensation** and **driftless calibration** contributes to greater user comfort and accuracy. Conceived and manufactured in Switzerland, the omega.6 is designed for demanding applications where **performance and reliability** are critical.

applications

The **omega.6** provides 3D active force feedback and passive rotation sensing for a wide range of applications:

- medical and space robotics
- micro and nano manipulators
- teleoperation consoles
- virtual simulations
- training systems
- research

force dimension

omega.6

workspace	translation	∅ 160 x L 110 mm
	rotation	240 x 140 x 240 deg
forces	continuous	12.0 N
resolution	linear	0.01 mm
	angular	0.09 deg
stiffness	closed loop	14.5 N/mm
dimensions	height	270 mm
	width	300 mm
	depth	350 mm

electronics

interface	standard	USB 2.0
power	universal	110V - 240V

software

platforms	Microsoft	Windows 2000 / XP / Vista
	Linux	kernel 2.4 / 2.6
	Apple	OS X
	QNX	Neutrino 6.3
SDK	DHD-API	haptic software library
	DRD-API	robotic software library

features

structure	delta-based parallel kinematics
	hand-centered, decoupled rotations
calibration	automatic, driftless
user input	1 programmable button
safety	velocity monitoring
	electronic damping
option	right or left-handed configuration

Force Dimension
PSE-C
CH-1015 Lausanne
Switzerland

t +41 21 693 1911
f +41 21 693 1910

www.forcedimension.com
info@forcedimension.com

Apendix E Software Documentation

Graph Panel Plotting

As the graph panel includes a pan and scrolling ability, it must be able to transform points it is plotting from global space to screen space. This is achieved using a transformation matrix. Each point to be graphed is stored in vector form $\begin{pmatrix} x \\ y \end{pmatrix}$. To perform operations, this vector must be transformed into $\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$ to enable multiplication with the 3x3 transformation matrix illustrated in *Equation 21*. It is possible to include a rotational transformation into the matrix. However, since rotation is not required for the graphing operation (as it adds computational complexly), it has been excluded. Due to the simplicity of this matrix, only four values need to be calculated when multiplying one transformation with another, this can be seen in *Equation 22*.

$$\begin{bmatrix} S_{x1} & 0 & O_{x1} \\ 0 & S_{y1} & O_{y1} \\ 0 & 0 & 1 \end{bmatrix}$$

Equation 21: Scaling and Translation Matrix

$$\begin{bmatrix} S_{x1} & 0 & O_{x1} \\ 0 & S_{y1} & O_{y1} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_{x2} & 0 & O_{x2} \\ 0 & S_{y2} & O_{y2} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} S_{x1}S_{x2} & 0 & S_{x1}O_{x2} + O_{x1} \\ 0 & S_{y1}S_{y2} & S_{xy}O_{y2} + O_{y1} \\ 0 & 0 & 1 \end{bmatrix}$$

Equation 22: Transformation Multiplication

Multiplication of the transformation matrix and a vector also prove trivial to implement:

$$\begin{bmatrix} S_x & 0 & O_x \\ 0 & S_y & O_y \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} xS_x + O_x \\ yS_y + O_y \\ 1 \end{pmatrix}$$

Equation 23: Transformation of 2D Point

Since many of the elements in the matrix remain constant, it is easy to implement a software class for 2D Transformation and Scaling. This emulates a 3x3 matrix using two 2x1 vectors as seen from Snippet 14.

```
public sealed class Matrix2D_ScaleTranslate
    {
```

```
        public Vector2D Scale;

        .
        .
        .

        public Vector2D Offset;

        .
        .
        .

        static public Matrix2D_ScaleTranslate operator
*(Matrix2D_ScaleTranslate m1, Matrix2D_ScaleTranslate m2)
        {
            Matrix2D_ScaleTranslate rtn = new Matrix2D_ScaleTranslate ();

            rtn.Scale = m1.Scale * m2.Scale;
            rtn.Offset = (m1.Scale * m2.Offset) + m1.Offset;

            return rtn;
        }
    }
```

Snippet 14: Scale-Translate Matrix Class Extract

Multiplication is handled through implementation of the overloaded operator '*'. This provides ease of integration when the class is instanced. The Vector2D class provides a number of overloaded operators to perform multiplication, addition and subtraction. Since it is rare for division to be utilised, it is un-implemented.

```
public sealed class Vector2D
    {
        public float X

        .
        .
        .

        public float Y

        .
        .
        .

        static public Vector2D operator *(Vector2D v1, Vector2D v2)
        {
            return new Vector2D(v1.X * v2.X, v1.Y * v2.Y);
        }

        .
        .
```

```
        .

        static public Vector2D operator *(Matrix2D_ScaleTranslate m, Vector2D
v)
        {
            return (m.Scale*v) + m.Offset ;
        }

        static public Vector2D operator +(Vector2D v1, Vector2D v2)
        {
            return new Vector2D(v1.X + v2.X, v1.Y + v2.Y);
        }

        static public Vector2D operator -(Vector2D v1, Vector2D v2)
        {
            return new Vector2D(v1.X - v2.X, v1.Y - v2.Y);
        }
    }
```

Snippet 15: Vector2D Class Extract

Using these two classes, it then becomes simple to transform graph points programmatically. *Snippet 16* shows a practical example using the classes:

A transformation matrix is initialised to scale x and y by 0.5 and 0.75 respectively and translate along the same axes by 10 and -5; a point is initialised to be at location 50, 20. This point is then transformed to become 35, 10.

```
// Define Variables
Matrix2D_ScaleTranslate ScaleTranslate;
Vector2D Point;

// Initalise ScaleTranslate Matrix
ScaleTranslate = new Matrix2D_ScaleTranslate();
ScaleTranslate.Scale = new Vector2D(0.5f, 0.75f);
ScaleTranslate.Offset = new Vector2D(10, -5);

// Initalise a Point as a Vector2D
Point = new Vector2D(50, 20);

// Transform Point
Point = ScaleTranslate * Point;
```

Snippet 16: Implementation of Point Transformation

It is easy to move from a single transformation to transforming all points which must be graphed.

Haptic Simulation

Below in Figure 45 the class structure of the haptic simulation is shown, as it forms part of the LongBone solution.

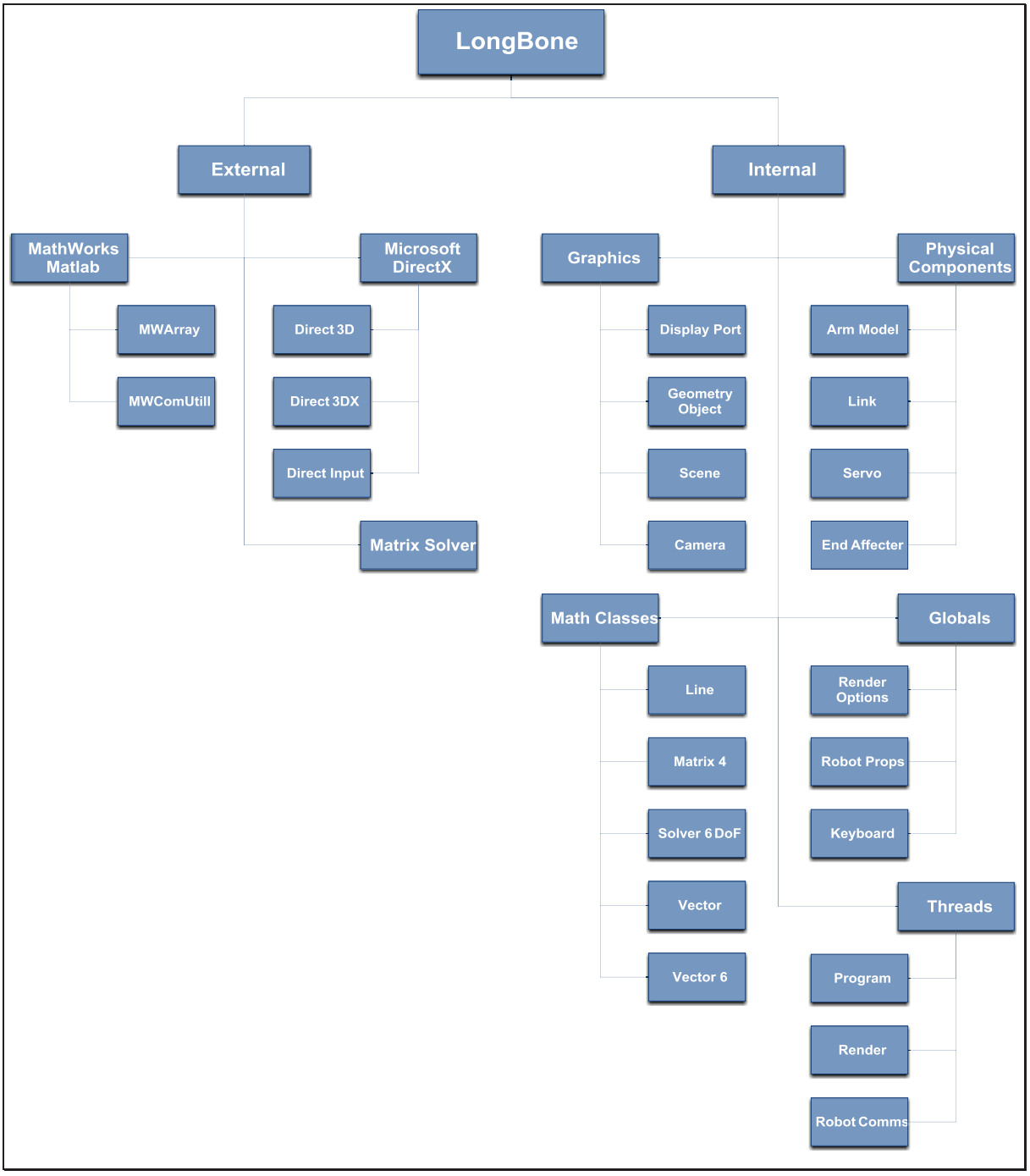


Figure 45: Haptic Simulation Namespace Diagram

frmMain

The main form provides the interface to the haptic simulation. It responds to use inputs and provides a method of displaying data.

Enums:

MoveDirection	Describes the direction the user is resizing the display panels

```
    public enum MoveDirecttion
    {
        None = 0,
        Both = 1,
        Vertical = 2,
        Horizontal = 4,
    };
```

Local Variables:

num_DisplayPorts	Number of display ports to render
DisplayResizeCursor	What cursor should be displayed
DisplayResize	Which direction are viewports being resized in
Arm	ArmModel representing the simulated device
FormMouseState	The position and state of the mouse over the form

Public Variables:

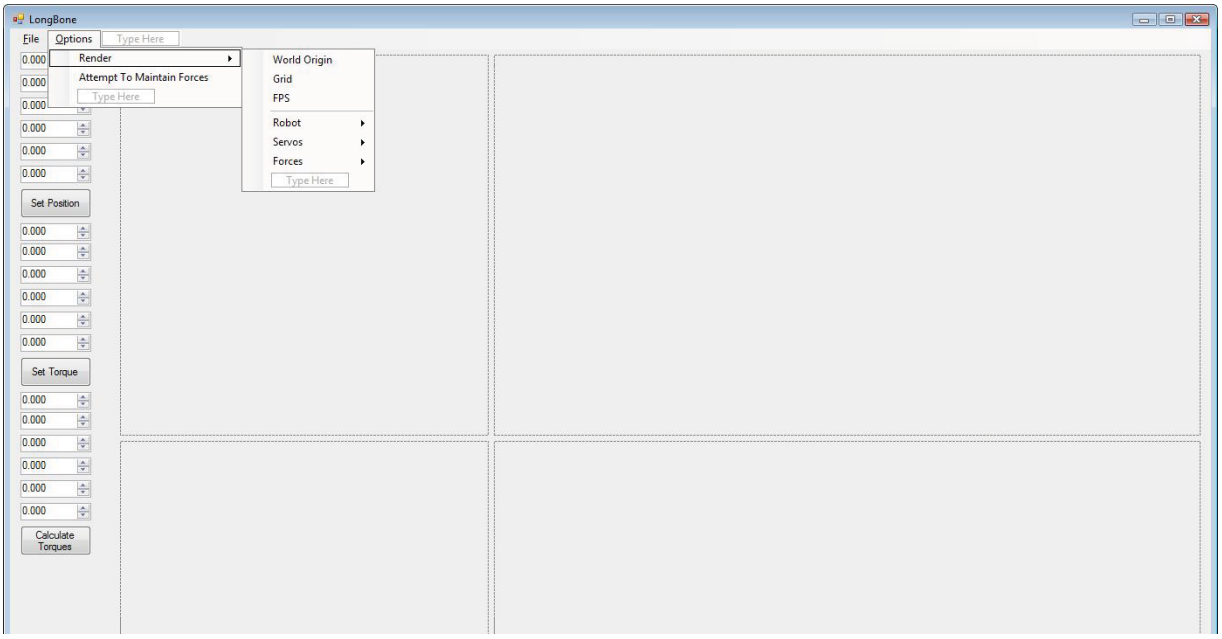
D3DPort	Array of render ports
RenderThread	Thread which handles rendering

```
    private const int num_DisplayPorts = 4;

        private MoveDirecttion DisplayResizeCursor = MoveDirecttion.None;
        private MoveDirecttion DisplayResize = MoveDirecttion.None;
        private ArmModel Arm = new ArmModel();
        private MouseEventArgs FormMouseState;

        public static Graphics.DisplayPort[] D3DPort = new
Graphics.DisplayPort[num_DisplayPorts];
        public Threads.Render RenderThread = new Threads.Render();
```

Form:




```
        public frmMain()
        {
            InitializeComponent();
        }

        private void frmMain_Load(object sender, EventArgs e)
        {
            InitializeGraphics();
            Globals.Keyboard.Initalise(this);

            CheckMenuItems();
        }


        private void frmMain_Activated(object sender, EventArgs e)
        {
            this.Focus();
            Globals.Keyboard.Aquire();

            if (!Program.Render)
            {
                Program.Render = true;
            }
        }
```

```
        private void frmMain_FormClosing(object sender, FormClosingEventArgs
e)
        {
            // Exit Looping Threads
            Program.Render = false;
            Program.Running = false;
        }

        private void frmMain_Deactivate(object sender, EventArgs e)
        {
            // Pause Rendering to prevent errors
            if (WindowState == FormWindowState.Minimized)
            {

                Program.Render = false;
            }
        }


        private void InitializeGraphics()
        {
            int port_Num;
            for (int i = 0; i < num_DisplayPorts; i++)
            {
                D3DPort[i] = new App.Graphics.DisplayPort();
            }


            port_Num = 0;

            D3DPort[port_Num].InitializeGraphics(D3DPortWindow1,
App.Graphics.RenderType.Orthographic | App.Graphics.RenderType.Top);
            D3DPort[port_Num].Camera.Set(Math_Classes.Vector.Vec3(0, 500, 0),
Math_Classes.Vector.Vec3(0, 0, 0), Math_Classes.Vector.Vec3(0, 0, 1));


            port_Num = 1;

            D3DPort[port_Num].InitializeGraphics(D3DPortWindow2,
App.Graphics.RenderType.Perspective);
            D3DPort[port_Num].Camera.Set(Math_Classes.Vector.Vec3(0, 100, 0),
Math_Classes.Vector.Vec3(50, 0, 0));
            D3DPort[port_Num].Camera.SetPosition(-45, 45, 100);


            port_Num = 2;

            D3DPort[port_Num].InitializeGraphics(D3DPortWindow3,
App.Graphics.RenderType.Orthographic | App.Graphics.RenderType.Left);
            D3DPort[port_Num].Camera.Set(Math_Classes.Vector.Vec3(0, 0, -
500), Math_Classes.Vector.Vec3(0, 0, 0), Math_Classes.Vector.Vec3(0, 1, 0));


            port_Num = 3;

            D3DPort[port_Num].InitializeGraphics(D3DPortWindow4,
App.Graphics.RenderType.Orthographic | App.Graphics.RenderType.Front);
```

```
            D3DPort[port_Num].Camera.Set(Math_Classes.Vector.Vec3(100, 0, 0),
Math_Classes.Vector.Vec3(0, 0, 0), Math_Classes.Vector.Vec3(0, 1, 0));

        }



        // ------------------------------------ Inputs ---------------------
------------------------
        #region Mouse Events


        //  ---------------------------------------- Form --------------------
------------------
        #region Form

        private void frmMain_MouseMove(object sender, MouseEventArgs e)
        {
            FormMouseState = e;

            // Is the mouse in the region of the display windows?
            if (e.Y > D3DPortWindow1.Top && e.Y < D3DPortWindow3.Bottom &&
e.X > D3DPortWindow1.Left && e.X < D3DPortWindow4.Right)
            {
                // Are we resizing?
                if (DisplayResize != MoveDirecttion.None)
                {
                    Point m = new Point();
                    #region "Resize Horizontal"
                    if (DisplayResize == MoveDirecttion.Horizontal ||
DisplayResize == MoveDirecttion.Both) // Resize Horizontal
                    {
                        if (e.X > D3DPortWindow1.Left + 20 && e.X <
D3DPortWindow2.Right - 20)  // Within size bounds?
                        {

                            if ((e.X - 20 < ((D3DPortWindow2.Right -
D3DPortWindow1.Left) / 2 + D3DPortWindow1.Left)) && (e.X + 20 >
(D3DPortWindow2.Right - D3DPortWindow1.Left) / 2 + D3DPortWindow1.Left))
                            {
                                m.X = (D3DPortWindow2.Right -
D3DPortWindow1.Left) / 2 + D3DPortWindow1.Left;
                            }
                            else
                            {
                                m.X = e.X;
                            }

                            D3DPortWindow1.Width = m.X - D3DPortWindow1.Left
- 1;

                            D3DPortWindow2.Width += D3DPortWindow2.Left - m.X
- 3;
                            D3DPortWindow2.Left = m.X + 3;

                            D3DPortWindow3.Width = m.X - D3DPortWindow3.Left
- 1;
```

```
                            D3DPortWindow4.Width += D3DPortWindow4.Left - m.X
- 3;
                            D3DPortWindow4.Left = m.X + 3;
                        }
                    }
                    #endregion

                    #region "Resize Vertical"
                    // Resize Vertical
                    if (DisplayResize == MoveDirecttion.Vertical ||
DisplayResize == MoveDirecttion.Both)
                    {
                        // Within size bounds?
                        if (e.Y > D3DPortWindow1.Top + 20 && e.Y <
D3DPortWindow3.Bottom - 20)
                        {
                            // Snap to center
                            if ((e.Y - 20 < (D3DPortWindow3.Bottom -
D3DPortWindow1.Top) / 2 + D3DPortWindow1.Top) && (e.Y + 20 >
((D3DPortWindow3.Bottom - D3DPortWindow1.Top) / 2 + D3DPortWindow1.Top)))
                            {
                                m.Y = (D3DPortWindow3.Bottom -
D3DPortWindow1.Top) / 2 + D3DPortWindow1.Top;
                            }
                            else
                            {
                                m.Y = e.Y;
                            }

                            // Move Windows
                            D3DPortWindow1.Height = m.Y - D3DPortWindow1.Top
- 1;

                            D3DPortWindow2.Height = m.Y - D3DPortWindow2.Top
- 1;

                            D3DPortWindow3.Height += D3DPortWindow3.Top - m.Y
- 3;
                            D3DPortWindow3.Top = m.Y + 3;

                            D3DPortWindow4.Height += D3DPortWindow4.Top - m.Y
- 3;
                            D3DPortWindow4.Top = m.Y + 3;
                        }
                    }
                    #endregion
                }

                else // Check to enable resize
                {
                    #region "Mouse over resize zone?"
                    // Are we sizeable in both directions?
                    if (e.Y > D3DPortWindow1.Bottom && e.Y <
D3DPortWindow3.Top && e.X > D3DPortWindow1.Right && e.X <
D3DPortWindow2.Left)
                    {
```

```
                        Cursor = Cursors.SizeAll;
                        DisplayResizeCursor = MoveDirecttion.Both;
                    }
                    // Only Up/Down?
                    else if (e.Y > D3DPortWindow1.Bottom && e.Y <
D3DPortWindow3.Top)
                    {
                        Cursor = Cursors.SizeNS;
                        DisplayResizeCursor = MoveDirecttion.Vertical;
                    }
                    // Only Left/Right?
                    else if (e.X > D3DPortWindow1.Right && e.Y <
D3DPortWindow2.Left)
                    {
                        Cursor = Cursors.SizeWE;
                        DisplayResizeCursor = MoveDirecttion.Horizontal;
                    }
                    #endregion
                }
            }
            else // We arn't in the resize region
            {
                if (DisplayResize == MoveDirecttion.None)    // Are we
already moving?
                {
                    Cursor = Cursors.Arrow;
                    DisplayResizeCursor = MoveDirecttion.None;
                }
            }
        }

        private void frmMain_MouseLeave(object sender, EventArgs e)
        {
            if (FormMouseState.Button != MouseButtons.Left)
            {
                Cursor = Cursors.Arrow;
            }
        }

        private void frmMain_MouseDown(object sender, MouseEventArgs e)
        {
            if (DisplayResizeCursor != MoveDirecttion.None)
            {
                DisplayResize = DisplayResizeCursor;
            }
        }

        private void frmMain_MouseUp(object sender, MouseEventArgs e)
        {
            if (DisplayResize != MoveDirecttion.None)   // Were we resizing
the display ports?
            {
                DisplayResize = MoveDirecttion.None;

                Program.Render = false;         // Pause rendering calls
while devices reset
```

```
                D3DPort[0].Reset();
                D3DPort[1].Reset();
                D3DPort[2].Reset();
                D3DPort[3].Reset();
                Program.Render = true;
            }
        }

        private void frmMain_Resize(object sender, EventArgs e)
        {
            this.D3DPortWindow1.Height = (this.Height - D3DPortWindow1.Top) /
2 - 8;
            this.D3DPortWindow1.Width = (this.Width - D3DPortWindow1.Left) /
2 - 8;

            this.D3DPortWindow2.Height = D3DPortWindow1.Height;
            this.D3DPortWindow2.Width = D3DPortWindow1.Width;
            this.D3DPortWindow2.Left = D3DPortWindow1.Right + 4;

            this.D3DPortWindow3.Height = D3DPortWindow1.Height;
            this.D3DPortWindow3.Width = D3DPortWindow1.Width;
            this.D3DPortWindow3.Top = D3DPortWindow1.Bottom + 4;

            this.D3DPortWindow4.Height = D3DPortWindow1.Height;
            this.D3DPortWindow4.Width = D3DPortWindow1.Width;
            this.D3DPortWindow4.Left = D3DPortWindow1.Right + 4;
            this.D3DPortWindow4.Top = D3DPortWindow1.Bottom + 4;
        }

        #endregion

        //  ------------------------------------- Top --------------------
------------------
        #region Top

        private void D3DPortWindow1_MouseMove(object sender, MouseEventArgs
e)
        {
            int deviceId = 0;

            Cursor = Cursors.Arrow;

            D3DPortWindow1.Focus();
            D3DPort[deviceId].mouseRay_Check = true;
            D3DPort[deviceId].MouseRay_Update(e.X, e.Y);

            if
(Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.LeftShift] ||
Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.RightShift])
            {
                if (e.Button == MouseButtons.Right)
                {
                    float moveX = (float)(FormMouseState.X - e.X);
                    float moveY = (float)(e.Y - FormMouseState.Y);

                    float scaleX = D3DPort[deviceId].Camera.OrthRange /
(D3DPort[deviceId].DispPort.Width / D3DPort[deviceId].Aspect);
```

```
                    float scaleY = D3DPort[deviceId].Camera.OrthRange /
D3DPort[deviceId].DispPort.Height;
                    D3DPort[deviceId].Camera.Pan(moveX * scaleX, moveY *
scaleY);

                }
            }


            FormMouseState = e;
        }

        private void D3DPortWindow1_MouseDown(object sender, MouseEventArgs
e)
        {
            FormMouseState = e;
        }

        private void D3DPortWindow1_MouseWheel(object sender, MouseEventArgs
e)
        {
            int deviceId = 0;
            if
(Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.LeftShift] ||
Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.RightShift])
            {
                D3DPort[deviceId].Camera.Zoom(e.Delta / 40);
            }
            else if ((Globals.RobotProps.mouseRay_Over >= 0) &&
(Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.LeftControl] ||
Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.RightControl]))
            {

Globals.RobotProps.servoTable[Globals.RobotProps.mouseRay_Over].ModifyCurrent
((float)e.Delta / 120);

                Arm.FowardKinimatics();

            }
            else if (Globals.RobotProps.mouseRay_Over >= 0)
            {

Globals.RobotProps.servoTable[Globals.RobotProps.mouseRay_Over].Angle +=
(float)e.Delta / 1200;

                Arm.FowardKinimatics();
            }
        }

        private void D3DPortWindow1_LostFocus(object sender, EventArgs e)
        {
            D3DPort[0].mouseRay_Check = false;
        }

        #endregion
```

```
        //  ------------------------------------- Persp ------------------
--------------------
        #region Persp

        private void D3DPortWindow2_MouseMove(object sender, MouseEventArgs
e)
        {
            int deviceId = 1;

            Cursor = Cursors.Arrow;

            D3DPortWindow2.Focus();
            D3DPort[deviceId].mouseRay_Check = true;
            D3DPort[deviceId].MouseRay_Update(e.X, e.Y);


            if
(Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.LeftShift] ||
Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.RightShift])
            {
                if (e.Button == MouseButtons.Left)
                {
                    float moveX = (float)(FormMouseState.X - e.X);
                    float moveY = (float)(e.Y - FormMouseState.Y);

                    float scaleX = 100 /
(float)(D3DPort[deviceId].DispPort.Width / D3DPort[deviceId].Aspect);
                    float scaleY = 100 /
(float)D3DPort[deviceId].DispPort.Height;

                    D3DPort[deviceId].Camera.Rotate(moveX * scaleX, moveY *
scaleY);
                }
                else if (e.Button == MouseButtons.Right)
                {
                    float moveX = (float)(FormMouseState.X - e.X);
                    float moveY = (float)(e.Y - FormMouseState.Y);

                    float scaleX =
D3DPort[deviceId].Camera.SphericalPosition.Z /
(D3DPort[deviceId].DispPort.Width / D3DPort[deviceId].Aspect);
                    float scaleY =
D3DPort[deviceId].Camera.SphericalPosition.Z /
D3DPort[deviceId].DispPort.Height;

                    D3DPort[deviceId].Camera.Pan(moveX * scaleX, moveY *
scaleY);

                }

            }
            FormMouseState = e;
        }

        private void D3DPortWindow2_MouseDown(object sender, MouseEventArgs
e)
        {
```

```
            FormMouseState = e;

        }

        private void D3DPortWindow2_MouseWheel(object sender, MouseEventArgs
e)
        {
            int deviceId = 1;
            if
(Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.LeftShift] ||
Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.RightShift])
            {
                D3DPort[deviceId].Camera.Zoom(e.Delta / 40);
            }
            else if ((Globals.RobotProps.mouseRay_Over >= 0) &&
(Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.LeftControl] ||
Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.RightControl]))
            {

Globals.RobotProps.servoTable[Globals.RobotProps.mouseRay_Over].ModifyCurrent
((float)e.Delta / 120);

                Arm.FowardKinimatics();

            }
            else if (Globals.RobotProps.mouseRay_Over >= 0)
            {

Globals.RobotProps.servoTable[Globals.RobotProps.mouseRay_Over].Angle +=
(float)e.Delta / 1200;

                Arm.FowardKinimatics();
            }
        }

        private void D3DPortWindow2_LostFocus(object sender, EventArgs e)
        {
            D3DPort[1].mouseRay_Check = false;
        }

        #endregion

        //  ------------------------------------- Left --------------------
--------------------
        #region Left

        private void D3DPortWindow3_MouseMove(object sender, MouseEventArgs
e)
        {
            int deviceId = 2;

            Cursor = Cursors.Arrow;

            D3DPortWindow3.Focus();
            D3DPort[deviceId].mouseRay_Check = true;
            D3DPort[deviceId].MouseRay_Update(e.X, e.Y);
```

```
            if
(Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.LeftShift] ||
Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.RightShift])
            {
                if (e.Button == MouseButtons.Right)
                {

                    float moveX = (float)(FormMouseState.X - e.X);
                    float moveY = (float)(e.Y - FormMouseState.Y);

                    float scaleX = D3DPort[deviceId].Camera.OrthRange /
(D3DPort[deviceId].DispPort.Width / D3DPort[deviceId].Aspect);
                    float scaleY = D3DPort[deviceId].Camera.OrthRange /
D3DPort[deviceId].DispPort.Height;
                    D3DPort[deviceId].Camera.Pan(moveX * scaleX, moveY *
scaleY);
                }
            }
            FormMouseState = e;
        }

        private void D3DPortWindow3_MouseDown(object sender, MouseEventArgs
e)
        {
            FormMouseState = e;
        }

        private void D3DPortWindow3_MouseWheel(Object sender, MouseEventArgs
e)
        {
            int deviceId = 2;
            if
(Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.LeftShift] ||
Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.RightShift])
            {
                D3DPort[deviceId].Camera.Zoom(e.Delta / 40);
            }
            else if ((Globals.RobotProps.mouseRay_Over >= 0) &&
(Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.LeftControl] ||
Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.RightControl]))
            {

Globals.RobotProps.servoTable[Globals.RobotProps.mouseRay_Over].ModifyCurrent
((float)e.Delta / 120);

                Arm.FowardKinimatics();

            }
            else if (Globals.RobotProps.mouseRay_Over >= 0)
            {

Globals.RobotProps.servoTable[Globals.RobotProps.mouseRay_Over].Angle +=
(float)e.Delta / 1200;

                Arm.FowardKinimatics();
            }
```

```
        }

        private void D3DPortWindow3_LostFocus(object sender, EventArgs e)
        {
            D3DPort[2].mouseRay_Check = false;

        }

        #endregion

        //  ------------------------------------- Front ------------------
--------------------
        #region Front

        private void D3DPortWindow4_MouseMove(object sender, MouseEventArgs
e)
        {
            int deviceId = 3;

            D3DPortWindow4.Focus();
            D3DPort[deviceId].mouseRay_Check = true;
            D3DPort[deviceId].MouseRay_Update(e.X, e.Y);

            if
(Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.LeftShift] ||
Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.RightShift])
            {
                if (e.Button == MouseButtons.Right)
                {
                    float moveX = (float)(FormMouseState.X - e.X);
                    float moveY = (float)(e.Y - FormMouseState.Y);

                    float scaleX = D3DPort[deviceId].Camera.OrthRange /
(D3DPort[deviceId].DispPort.Width / D3DPort[deviceId].Aspect);
                    float scaleY = D3DPort[deviceId].Camera.OrthRange /
D3DPort[deviceId].DispPort.Height;
                    D3DPort[deviceId].Camera.Pan(moveX * scaleX, moveY *
scaleY);

                }
            }
            FormMouseState = e;
        }

        private void D3DPortWindow4_MouseDown(object sender, MouseEventArgs
e)
        {
            FormMouseState = e;
        }

        private void D3DPortWindow4_MouseWheel(Object sender, MouseEventArgs
e)
        {
            int deviceId = 3;

            Cursor = Cursors.Arrow;
```

```
            if
(Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.LeftShift] ||
Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.RightShift])
            {
                D3DPort[deviceId].Camera.Zoom(e.Delta / 40);
            }
            else if ((Globals.RobotProps.mouseRay_Over >= 0) &&
(Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.LeftControl] ||
Globals.Keyboard.keys[Microsoft.DirectX.DirectInput.Key.RightControl]))
            {

Globals.RobotProps.servoTable[Globals.RobotProps.mouseRay_Over].ModifyCurrent
((float)e.Delta / 120);

                Arm.FowardKinimatics();

            }
            else if (Globals.RobotProps.mouseRay_Over >= 0)
            {

Globals.RobotProps.servoTable[Globals.RobotProps.mouseRay_Over].Angle +=
(float)e.Delta / 1200;

                Arm.FowardKinimatics();
            }
        }

        private void D3DPortWindow4_LostFocus(object sender, EventArgs e)
        {
            D3DPort[3].mouseRay_Check = false;
        }

        #endregion

        #endregion

        // ------------------------------------ Menu ----------------------
----------------------

        #region Menu Events

        private void Render_WorldOriginToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            Globals.RenderOptions.World_Origin =
Render_WorldOriginToolStripMenuItem.Checked;
        }

        private void Render_GridToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            Globals.RenderOptions.World_Grid =
Render_GridToolStripMenuItem.Checked;
        }

        private void Render_FPSToolStripMenuItem_Click(object sender,
EventArgs e)
```

```
        {
            Globals.RenderOptions.World_FPS =
Render_FPSToolStripMenuItem.Checked;
        }

        private void Render_Robot_physicalToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            Globals.RenderOptions.Robot_Physical =
Render_Robot_physicalToolStripMenuItem.Checked;
        }

        private void Render_Robot_kinimaticToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            Globals.RenderOptions.Robot_Kinimatic =
Render_Robot_kinimaticToolStripMenuItem.Checked;
        }

        private void Render_Servos_OriginsToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            Globals.RenderOptions.Servo_Origins =
Render_Servos_OriginsToolStripMenuItem.Checked;
        }

        private void Render_Servos_Axis_MouseToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            Globals.RenderOptions.Servo_Axis =
App.Globals.RenderOptions.ObjectRenderFlags.MouseOver;

            Render_Servos_Axis_AllToolStripMenuItem.Checked = false;
            Render_Servos_Axis_NoneToolStripMenuItem.Checked = false;
        }

        private void Render_Servos_Axis_AllToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            Globals.RenderOptions.Servo_Axis =
App.Globals.RenderOptions.ObjectRenderFlags.All;

            Render_Servos_Axis_MouseToolStripMenuItem.Checked = false;
            Render_Servos_Axis_NoneToolStripMenuItem.Checked = false;
        }

        private void Render_Servos_Axis_NoneToolStripMenuItem1_Click(object
sender, EventArgs e)
        {
            Globals.RenderOptions.Servo_Axis =
App.Globals.RenderOptions.ObjectRenderFlags.None;

            Render_Servos_Axis_MouseToolStripMenuItem.Checked = false;
            Render_Servos_Axis_AllToolStripMenuItem.Checked = false;
        }
```

```
        private void
Render_Servos_TorqueDirection_MouseToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            Globals.RenderOptions.Servo_TorqueDirection =
App.Globals.RenderOptions.ObjectRenderFlags.MouseOver;

            Render_Servos_TorqueDirection_AllToolStripMenuItem.Checked =
false;
            Render_Servos_TorqueDirection_NoneToolStripMenuItem.Checked =
false;

        }

        private void
Render_Servos_TorqueDirection_AllToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            Globals.RenderOptions.Servo_TorqueDirection =
App.Globals.RenderOptions.ObjectRenderFlags.All;

            Render_Servos_TorqueDirection_MouseToolStripMenuItem.Checked =
false;
            Render_Servos_TorqueDirection_NoneToolStripMenuItem.Checked =
false;
        }

        private void
Render_Servos_TorqueDirection_NoneToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            Globals.RenderOptions.Servo_TorqueDirection =
App.Globals.RenderOptions.ObjectRenderFlags.None;

            Render_Servos_TorqueDirection_MouseToolStripMenuItem.Checked =
false;
            Render_Servos_TorqueDirection_AllToolStripMenuItem.Checked =
false;
        }

        private void
Render_Servos_TorqueCalculationPintsToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            Globals.RenderOptions.Servo_TorqueCalcPoints =
Render_Servos_TorqueCalculationPintsToolStripMenuItem.Checked;
        }

        private void
Render_Forces_EndEffecter_TotalToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            Globals.RenderOptions.EndEffecter_ForceArrows =
App.Globals.RenderOptions.ObjectRenderFlags.All;

            Render_Forces_EndEffecter_NoneToolStripMenuItem.Checked = false;
            Render_Forces_EndEffecter_MouseToolStripMenuItem.Checked = false;
```

```
        }

        private void
Render_Forces_EndEffecter_MouseToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            Globals.RenderOptions.EndEffecter_ForceArrows =
App.Globals.RenderOptions.ObjectRenderFlags.MouseOver;

            Render_Forces_EndEffecter_NoneToolStripMenuItem.Checked = false;
            Render_Forces_EndEffecter_TotalToolStripMenuItem.Checked = false;

        }

        private void
Render_Forces_EndEffecter_NoneToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            Globals.RenderOptions.EndEffecter_ForceArrows =
App.Globals.RenderOptions.ObjectRenderFlags.None;

            Render_Forces_EndEffecter_MouseToolStripMenuItem.Checked = false;
            Render_Forces_EndEffecter_TotalToolStripMenuItem.Checked = false;
        }

        private void
Render_Forces_Internal_MouseToolStripMenuItem_Click(object sender, EventArgs
e)
        {
            Globals.RenderOptions.Servo_Internal =
App.Globals.RenderOptions.ObjectRenderFlags.MouseOver;

            Render_Forces_Internal_NoneToolStripMenuItem.Checked = false;
            Render_Forces_Internal_AllToolStripMenuItem.Checked = false;
        }


        private void Render_Forces_Internal_AllToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            Globals.RenderOptions.Servo_Internal =
App.Globals.RenderOptions.ObjectRenderFlags.All;

            Render_Forces_Internal_NoneToolStripMenuItem.Checked = false;
            Render_Forces_Internal_MouseToolStripMenuItem.Checked = false;
        }

        private void
Render_Forces_Internal_NoneToolStripMenuItem_Click(object sender, EventArgs
e)
        {
            Globals.RenderOptions.Servo_Internal =
App.Globals.RenderOptions.ObjectRenderFlags.None;

            Render_Forces_Internal_MouseToolStripMenuItem.Checked = false;
            Render_Forces_Internal_AllToolStripMenuItem.Checked = false;
```

```
        }

        #endregion


        private void CheckMenuItems()
        {
            Render_WorldOriginToolStripMenuItem.Checked =
Globals.RenderOptions.World_Origin;
            Render_GridToolStripMenuItem.Checked =
Globals.RenderOptions.World_Grid;
            Render_FPSToolStripMenuItem.Checked =
Globals.RenderOptions.World_FPS;

            Render_Robot_physicalToolStripMenuItem.Checked =
Globals.RenderOptions.Robot_Physical;
            Render_Robot_kinimaticToolStripMenuItem.Checked =
Globals.RenderOptions.Robot_Kinimatic;

            Render_Servos_OriginsToolStripMenuItem.Checked =
Globals.RenderOptions.Servo_Origins;

            Render_Servos_TorqueCalculationPintsToolStripMenuItem.Checked =
Globals.RenderOptions.Servo_TorqueCalcPoints;


            if (Globals.RenderOptions.Servo_Axis ==
App.Globals.RenderOptions.ObjectRenderFlags.All)
            {
                Render_Servos_Axis_AllToolStripMenuItem.Checked = true;
            }
            else if (Globals.RenderOptions.Servo_Axis ==
App.Globals.RenderOptions.ObjectRenderFlags.MouseOver)
            {
                Render_Servos_Axis_MouseToolStripMenuItem.Checked = true;
            }
            else
            {
                Render_Servos_Axis_NoneToolStripMenuItem.Checked = true;
            }


            if (Globals.RenderOptions.Servo_TorqueDirection ==
App.Globals.RenderOptions.ObjectRenderFlags.All)
            {
                Render_Servos_TorqueDirection_AllToolStripMenuItem.Checked =
true;
            }
            else if (Globals.RenderOptions.Servo_TorqueDirection ==
App.Globals.RenderOptions.ObjectRenderFlags.MouseOver)
            {
                Render_Servos_TorqueDirection_MouseToolStripMenuItem.Checked
= true;
            }
            else
            {
```

```
                Render_Servos_TorqueDirection_NoneToolStripMenuItem.Checked =
true;
            }


            if (Globals.RenderOptions.EndEffecter_ForceArrows ==
App.Globals.RenderOptions.ObjectRenderFlags.All)
            {
                Render_Forces_EndEffecter_TotalToolStripMenuItem.Checked =
true;
            }
            else if (Globals.RenderOptions.EndEffecter_ForceArrows ==
App.Globals.RenderOptions.ObjectRenderFlags.MouseOver)
            {
                Render_Forces_EndEffecter_MouseToolStripMenuItem.Checked =
true;
            }
            else
            {
                Render_Forces_EndEffecter_NoneToolStripMenuItem.Checked =
true;
            }


            if (Globals.RenderOptions.Servo_Internal ==
App.Globals.RenderOptions.ObjectRenderFlags.All)
            {
                Render_Forces_Internal_AllToolStripMenuItem.Checked = true;
            }
            else if (Globals.RenderOptions.Servo_Internal ==
App.Globals.RenderOptions.ObjectRenderFlags.MouseOver)
            {
                Render_Forces_Internal_MouseToolStripMenuItem.Checked = true;
            }
            else
            {
                Render_Forces_Internal_NoneToolStripMenuItem.Checked = true;
            }


        }

        private void button1_Click(object sender, EventArgs e)
        {
            Globals.RobotProps.servoTable[0].Angle =
(float)numericUpDown1.Value;
            Globals.RobotProps.servoTable[1].Angle =
(float)numericUpDown2.Value;
            Globals.RobotProps.servoTable[2].Angle =
(float)numericUpDown3.Value;
            Globals.RobotProps.servoTable[3].Angle =
(float)numericUpDown4.Value;
            Globals.RobotProps.servoTable[4].Angle =
(float)numericUpDown5.Value;
            Globals.RobotProps.servoTable[5].Angle =
(float)numericUpDown6.Value;
```

```
            Arm.FowardKinimatics();

        }

        private void button2_Click(object sender, EventArgs e)
        {


            Globals.RobotProps.servoTable[0].SetTorque
((float)numericUpDown7.Value);

Globals.RobotProps.servoTable[1].SetTorque((float)numericUpDown8.Value);

Globals.RobotProps.servoTable[2].SetTorque((float)numericUpDown9.Value);

Globals.RobotProps.servoTable[3].SetTorque((float)numericUpDown10.Value);

Globals.RobotProps.servoTable[4].SetTorque((float)numericUpDown11.Value);

Globals.RobotProps.servoTable[5].SetTorque((float)numericUpDown12.Value);

            Arm.FowardKinimatics();

        }

        private void but_Calculate_Click(object sender, EventArgs e)
        {
            Math_Classes.Vector6 Solution;
            float[] Coefficients = {
                10, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0,
                };
            float[] Constants = {
                10,
                0,
                0,
                0,
                0,
                0};

            Microsoft.DirectX.Vector3 Force = new
Microsoft.DirectX.Vector3((float)numericUpDown13.Value,
(float)numericUpDown14.Value, (float)numericUpDown15.Value);
            Microsoft.DirectX.Vector3 Torque = new
Microsoft.DirectX.Vector3((float)numericUpDown16.Value,
(float)numericUpDown17.Value, (float)numericUpDown18.Value);


            //Solution = Math_Classes.Solver6DoF.Solve(Coefficients,
Constants);

            Servo.SolveTorques(Force, Torque);
```

```
            Arm.FowardKinimatics();
        }

        private void menuStrip_ItemClicked(object sender,
ToolStripItemClickedEventArgs e)
        {

        }
```

Namespace App.Globals

Keyboard

Keyboard is a static class which provides a wrapper for handling all input from the keyboard via `DirectInput8`, for use with the three dimensional display ports. That is, detecting key presses when no windows control in the application has focus.

Local Variables:

device_Keyboard	Microsoft Direct Input Device.
initalised	Has the device been initalised(?).

Public Variables:

keys	Current key states.

```
        private static Device device_Keyboard;
        private static bool initalised = false;

        public static KeyboardState keys;
```

Public Functions:

void Initalise	Initialises the keyboard for use.
void Free	Frees the keyboard from the application.
void Aquire	Acquires the keyboard for the application.
void UpdateKeys	Updates the public variable '`keys`'.

```
        public static void Initalise(System.Windows.Forms.Control Handle)
        {

            CooperativeLevelFlags coopFlags;

            coopFlags = CooperativeLevelFlags.NonExclusive |
CooperativeLevelFlags.Background;

            device_Keyboard = new Device(SystemGuid.Keyboard);
            device_Keyboard.SetCooperativeLevel(Handle, coopFlags);


            try
            {
```

```
                device_Keyboard.Acquire();
            }
            catch
            {
            }

            initalised = true;
        }
```

```
        public static void Free()
        {
            device_Keyboard.Unacquire();
        }



        public static void Aquire()

        {
            if (initalised)
            {
                try { device_Keyboard.Acquire(); }
                catch (DirectXException) { }
            }
    }
```

```
public static void UpdateKeys()
        {
            if (!initalised)
            {
                return;
            }
            try
            {
                keys = device_Keyboard.GetCurrentKeyboardState();
            }
            catch (DirectXException)
            {
            }
        }
```

RenderOptions

This is a static data class which represents all the options regarding to what and how objects are rendered into the scene. This is also where the default properties are set for the program on initialisation.

Enums:

ObjectRenderFlags	Describes when an object is rendered.

```
public enum ObjectRenderFlags
{
    None = 0,
    All = 1,
    MouseOver = 2
}
```

Public Variables:

World_FPS	Frames Per Second.
World_Origin	Global Origin.
World_Grid	Grid.
Robot_Physical	CAD model of RV-M1.
Robot_Kinimatic	Kinematic model of RV-M1.
EndEffecter_Origin	Local origin of End Effecter.
EndEffecter_ForceArrows	Forces at End Effecter.
Servo_Origins	Local origins of serovs.
Servo_TorqueDirection	Torque direction.
Servo_Axis	Local axis of servo/s.
Servo_TorqueCalcPoints	Points used for calculation of forces.
Servo_Internal	Forces from gravity, friction and momentum.

```
        public static bool World_FPS = false;
        public static bool World_Origin = false;
        public static bool World_Grid = false;

        public static bool Robot_Physical = true;
        public static bool Robot_Kinimatic = false;

        public static bool EndEffecter_Origin = false;
        public static ObjectRenderFlags EndEffecter_ForceArrows =
ObjectRenderFlags.All;

        public static bool Servo_Origins = false;
        public static ObjectRenderFlags Servo_TorqueDirection =
ObjectRenderFlags.All;
        public static ObjectRenderFlags Servo_Axis =
ObjectRenderFlags.MouseOver;
        public static bool Servo_TorqueCalcPoints = false;
        public static ObjectRenderFlags Servo_Internal =
ObjectRenderFlags.None;
```

RobotProps

RobotProps is a static data class which is responsible for all data related to the robotic arm. Including cut off factors used in torque / force calculation.

Public Variables:

Gravity	Gravity Constant (-9.80661).
cutoff_Solver6DoF cutoff_TorqueRadius cutoff_OnAxis	Cut off values are used to prevent overwhelming values from being calculated from near zero divisions.
servoTable	Servos which are in the arm.
linkTable	Kinematic links for the arm.
endEffecter	Model of End Effecter.
num_Links	Number of links in the arm.
num_Servos	Number of servos in the arm.
mouseRay_Over	Link which the mouse pointer is over.

```
        public static Microsoft.DirectX.Vector3 Gravity = new
Microsoft.DirectX.Vector3 (0,-9.80661f,0);

        public static double cutoff_Solver6DoF = 0.00005;
        public static double cutoff_TorqueRadius = 0.005;
        public static double cutoff_OnAxis = 0.005;

        public static Physical_Components.Servo[] servoTable;
        public static Physical_Components.Link[] linkTable;
        public static Physical_Components.EndEffecter endEffecter;

        public static int num_Links = 0;
        public static int num_Servos = 0;
        public static int mouseRay_Over;
```

Namespace App.Graphics

Enums:

RenderType	Descriptions of camera type.

```
public enum RenderType
{
    Perspective = 1,
    Orthographic = 2,
    Front = 4,
    Top = 8,
    Left = 16,
}
```

Camera

A camera defines a view point which we view the scene from.

Local Variables:

Parent	Display port which owns the camera.

Public Variables:

RenderAs	Describes how the camera should render the scene.
SphericalPosition	Position of the camera relative to the target via yaw, pitch and radius.
Position	Cartesian co-ordinates for the position of the camera.
LookAt	Cartesian co-ordinates for the 'look at target' of the camera.
Up	Vector describing 'up'.
OrthRange	Distance the camera is from the orthographic plane, implements zoom.

```
    private DisplayPort Parent;

    public RenderType RenderAs;


    public Vector3 SphericalPosition = new Vector3();
    public float OrthRange;
```

```
        public Vector3 Position = new Vector3();
        public Vector3 LookAt = new Vector3();
        public Vector3 Up = new Vector3();
```

Public Functions:

Camera Constructor (DisplayPort Parent)	Creates a camera with a parent.
void SetLookAt (Vector3)	Sets the 'look at target' in Cartisian co-ordinates.
void SetLookAt (Vector3, bool)	Sets the 'look at target' in Cartisian co-ordinates, camera is moved as well if flagged true.
void set (Vector3, Vector3, Vector3)	Sets the properties of the camera.
void set (Vector3, Vector3)	Sets the properties of the camera and assumes up as (0, 1, 0) if camera is in perspective mode.
void SetPosition (Vector3)	Sets the position of the camera in Cartisian co-ordinates.
void SetPosition(float, float, float)	Sets the radial position of the camera, modifies up vector if required and updates Cartisian position, reflecting changes.
void Rotate (float, float)	Rotates the camera about the target.
void Zoom (float)	Moves the camera along screen space; z-axis.
void Pan(float,float)	Moves the camera along screen space; x and y axes.

```
        public Camera(DisplayPort Parent)
        {
            this.Parent = Parent;
        }
```

```
        public void SetLookAt(Vector3 p)
        {
            LookAt = p;
        }
```

```
        public void SetLookAt(Vector3 p, bool MaintainRelativePosition)
        {
            if (MaintainRelativePosition)
            {
                Vector3 differance = new Vector3();
                differance = Vector.Subtract(p, LookAt);
                Position = Vector.Add(Position, differance);
```

```
            }
            LookAt = p;
        }
```

```
        public void Set(Vector3 p, Vector3 At, Vector3 Up)
        {
            this.LookAt = LookAt;
            this.Position = Position;
            this.Up = Up;
        }
```

```
        public void Set(Vector3 p, Vector3 At)
        {
            this.LookAt = LookAt;
            this.Position = Position;

            if ((RenderAs & RenderType.Perspective) ==
RenderType.Perspective)
            {
                this.Up = Vector.Vec3(0, 1, 0);
            }
        }
```

```
        public void SetPosition(Vector3 p)
        {
            Position = p;
        }
```

```
        public void SetPosition(float yaw, float pitch, float radius)
        {

            float yawRad;
            float pitchRad;

            // Modulate angles to 360 degrees (non negative)
            yaw += 360;
            pitch += 360;
            yaw = yaw % 360;
            pitch = pitch % 360;

            SphericalPosition = Vector.Vec3 (yaw,pitch,radius);

            // Convert from degrees to radians
            yawRad = (float)(yaw * Math.PI / 180);
            pitchRad = (float) (pitch * Math.PI / 180);

            // Define the up vector
            if (pitch > 90 && pitch < 270)
            {
                Up = Vector.Vec3 (0,-1,0);
            }else if (pitch == 90){
                Up = Vector.Vec3(-Math.Cos (yawRad), 0, -Math.Sin (yawRad));
            }else if (pitch == 270)
            {
                Up = Vector.Vec3(Math.Cos(yawRad), 0, Math.Sin(yawRad));
            }else{
```

```
                Up = Vector.Vec3 (0,1,0);
            }

            Position.X = (float) ((Math.Cos (yawRad) * Math.Cos (pitchRad) *
radius) + LookAt.X);
            Position.Y = (float) ((Math.Sin(pitchRad) * radius) + LookAt.Y);
            Position.Z = (float)((Math.Sin(yawRad) * Math.Cos(pitchRad) *
radius) + LookAt.Z);

        }
```

```
        public void Rotate(float yaw, float pitch)
        {
            if ((RenderAs & RenderType.Perspective) ==
RenderType.Perspective)
            {
                SphericalPosition.X += yaw;
                SphericalPosition.Y += pitch;

                SetPosition(SphericalPosition.X, SphericalPosition.Y,
SphericalPosition.Z);
            }
        }
```

```
        public void Zoom(float factor)
        {
            if ((RenderAs & RenderType.Perspective) ==
RenderType.Perspective)
            {
                SphericalPosition.Z += factor;

                SetPosition(SphericalPosition.X, SphericalPosition.Y,
SphericalPosition.Z);
            }
            else
            {
                OrthRange += factor;

                Parent.device.Transform.Projection =
Matrix.OrthoLH(Parent.Aspect * OrthRange, OrthRange, 0.0f, 10000.0f);
            }
        }
```

```
        public void Pan(float x, float y)
        {

            if ((RenderAs & RenderType.Perspective) ==
RenderType.Perspective)
            {
                Vector3 transpose = new Vector3(-x,-y,0);
                Matrix4 viewMat = new Matrix4();

                viewMat = Matrix4.FromD3DMatrix( Matrix.TransposeMatrix (
Parent.device.Transform.View));

                transpose = Matrix4.RotateVector (viewMat, transpose);
```

```
                Position = Vector.Vec3(Position.X + transpose.X, Position.Y +
transpose.Y, Position.Z + transpose.Z);
                LookAt = Vector.Vec3(LookAt.X + transpose.X, LookAt.Y +
transpose.Y, LookAt.Z + transpose.Z);
            }
            else
            {
                if ((RenderAs & RenderType.Top) == RenderType.Top)
                {
                    Set(Vector.Vec3(Position.X + x, Position.Y, Position.Z +
y), Vector.Vec3(LookAt.X + x, LookAt.Y, LookAt.Z + y));
                }
                else if ((RenderAs & RenderType.Left) == RenderType.Left)
                {
                    Set(Vector.Vec3(Position.X + x, Position.Y + y, -100),
Vector.Vec3(Position.X + x, Position.Y + y, 0));
                }
                else if ((RenderAs & RenderType.Front) == RenderType.Front)
                {
                    Set(Vector.Vec3(100, Position.Y + y, Position.Z + x),
Vector.Vec3(0, Position.Y + y, Position.Z + x));
                }

            }
        }
```

DisplayPort

To render graphics, we require an interface between the GPU and our windows form. The DisplayPort class provides this and manages DirectX resources.

Public Variables:

Camera	Camera used for rendering to this port.
DispPort	Container to render to.
Aspect	Aspect ratio of the render region.
device	Direct3D device.
mouseX	X co-ordinate of the mouse.
mouseY	Y co-ordinate of the mouse.
mouseOver	Link the mouse is over.
mouseRay_Check	Check if mouse is over links(?).
presentParams	Parameters on how to present the scene.
scene	Scene to render.

```
public Camera Camera;
public System.Windows.Forms.PictureBox DispPort;
public float Aspect;

public Device device = null; // Our rendering device

public float mouseX;
public float mouseY;
public int mouseOver;
public bool mouseRay_Check;

PresentParameters presentParams = new PresentParameters();

Scene scene = new Scene();
```

Public Functions:

bool InitializeGraphics (PictureBox, RenderType)	Initialise the display port.

void onResetDevice(object, EventArgs)	Triggered on device reset, sets device properties.
void reset (void)	Forces a device reset.
void render(void)	Renders the scene.
void MouseRay_Update (float, float)	Updates the mouse co-ordinates.

```
        public bool InitializeGraphics(System.Windows.Forms.PictureBox w,
RenderType RenderInfo)
        {
            Camera = new Camera(this);
            Camera.RenderAs = RenderInfo;
            Camera.OrthRange = 100;


            DispPort = w;
            try
            {
                // Set Params
                presentParams.Windowed = true;
                presentParams.SwapEffect = SwapEffect.Discard;
                presentParams.EnableAutoDepthStencil = true;
                presentParams.AutoDepthStencilFormat = DepthFormat.D16;


                device = new Device(0, DeviceType.Hardware, DispPort.Handle,
CreateFlags.HardwareVertexProcessing, presentParams);
                device.DeviceReset += new
System.EventHandler(this.OnResetDevice);

                this.OnResetDevice(device, null);

                // Set Scene Properties
                scene.Initalise  (device, this);

                if ((Camera.RenderAs & RenderType.Left) == RenderType.Left)
                {
                    scene.SetGridTransform(Matrix.RotationX((float)Math.PI /
2));
                }
                else if ((Camera.RenderAs & RenderType.Front) ==
RenderType.Front)
                {
                    scene.SetGridTransform(Matrix.RotationZ((float)Math.PI /
2));
                }

                return true;
            }
            catch (DirectXException)
            {
                return false;
```

```
            }

        }
```

```
        public void OnResetDevice(object sender, EventArgs e)
        {

            Device dev = (Device)sender;

            dev.RenderState.ZBufferEnable = true;
            dev.RenderState.FillMode = FillMode.Solid;

            // Turn off culling, so we see the front and back of the triangle
            dev.RenderState.CullMode = Cull.None;
            dev.RenderState.Lighting = true;

            dev.Lights[0].Type = LightType.Point;
            dev.Lights[0].Diffuse = System.Drawing.Color.FromArgb
(255,255,255);
            dev.Lights[0].Range = 500f;
            dev.Lights[0].Enabled = true;

            dev.Lights[1].Type = LightType.Directional;
            dev.Lights[1].Diffuse = System.Drawing.Color.FromArgb(255, 255,
255);
            dev.Lights[1].Enabled = true;

            // Reset vertex buffers
            scene.ResetVertexBuffers();

            // Set the Projection Matrix
            Aspect = (float)DispPort.Size.Width /
(float)DispPort.Size.Height;

            if ((Camera.RenderAs & RenderType.Perspective) ==
RenderType.Perspective)
            {
                device.RenderState.CullMode = Cull.Clockwise;
                device.Transform.Projection =
Matrix.PerspectiveFovLH((float)Math.PI / 4, Aspect, 0.2f, 300.0f);
            }
            else
            {
                device.RenderState.CullMode = Cull.Clockwise;
                device.Transform.Projection = Matrix.OrthoLH(Aspect *
Camera.OrthRange, Camera.OrthRange, 0.0f, 10000.0f);
            }

            if ((Camera.RenderAs & RenderType.Left) == RenderType.Left)
            {
                scene.SetGridTransform (Matrix.RotationX ((float) Math.PI /
2));
            }
            else if ((Camera.RenderAs & RenderType.Front) ==
RenderType.Front)
```

```
            {
                scene.SetGridTransform(Matrix.RotationZ((float)Math.PI / 2));
            }
        }
```

```
        public void Reset()
        {
            presentParams.BackBufferHeight = (int) (DispPort.Height * 1);
            presentParams.BackBufferWidth = (int) (DispPort.Width * 1);
            device.Reset(presentParams);
        }

        public void Render()
        {

            if (device == null)
                return;

            if (!Program.Render)
                return;

            //Clear the backbuffer to a white color
            device.Clear(ClearFlags.Target | ClearFlags.ZBuffer,
System.Drawing.Color.FromArgb(255, 255, 255), 1.0f, 0);
            //Begin the scene
            device.BeginScene();


            // Update the camera
            device.Transform.View = Matrix.LookAtLH(Camera.Position,
Camera.LookAt, Camera.Up);

            // Set Lights
            device.Lights[0].Position = Camera.Position;
            device.Lights[0].Update();

            device.Lights[1].Direction = Camera.Position - Camera.LookAt;
            device.Lights[1].Update();


            scene.Render ();

            //End the scene
            device.EndScene();

            try
            {
                device.Present();
            }
            catch { }

        }
```

```
        public void MouseRay_Update(float mouseX, float mouseY)
        {
            this.mouseX = mouseX;
```

```
            this.mouseY = mouseY;
        }
```

GeometryObject

The GeometryObject class provides a wrapper for loading and rendering meshes.

Private Variables:

Parent	Parent scene of the object.
matDefault	Default rendering material.
matHighlight	The material for rendering when highlighted.
Transform	Transformation matrix for object.

Public Variables:

Geom	Mesh containing vertex cloud.

```
        private Scene Parent;

        private Material matDefault = new Material(); // Material
        private Material matHighlight = new Material();

        private Matrix Transform = new Matrix();

        public Mesh Geom = null;
```

Public Functions:

float CheckIntersection (float,float)	Checks to see if a ray projected through the passed co-ordinates intersects with the mesh.
void Initalise(device, string, Scene)	Initialises the object, loading a mesh.
void Render (device, bool)	Render to the device specified, render as highlighted(?).
void SetMaterial (Material)	Sets the default material to that specified.
void SetHighlightMaterial(Material)	Sets the highlight material to that specified.

void SetTransform (Matrix)	Sets the transformation matrix of the object.

```
        public void SetMaterial (Material mat){
            material_Default = mat;
        }
```

```
        public void SetHighlightMaterial(Material mat)
        {
            material_Highlight = mat;
        }
```

```
        public void SetTransform (Matrix m)
        {
            Transform = m;
        }
```

```
        public void Initalise(Device dev, string meshName, Scene Parent)
        {
            this.Parent = Parent;

            ExtendedMaterial[] materials = null;


Directory.SetCurrentDirectory(System.Windows.Forms.Application.StartupPath +
@"\..\..\Meshes\");

            Geom = Mesh.FromFile(meshName, MeshFlags.SystemMemory , dev, out
materials);

            material_Default = materials[0].Material3D;

            material_Default.Ambient = material_Default.Diffuse;
            material_Default.Specular = System.Drawing.Color.White;


            // Default the highlight colour to yellow;

            Geom.ComputeNormals();

            material_Highlight.Diffuse = System.Drawing.Color.Yellow;
            material_Highlight.Ambient = material_Highlight.Diffuse;
            material_Highlight.Specular = System.Drawing.Color.White;

        }
```

```
        public void Render (Device dev, bool highlight)
        {
            dev.Transform.World = Transform;
            if (!highlight)
            {
                dev.Material = material_Default;
            }
            else
            {
                dev.Material = material_Highlight;
```

```
            }

            Geom.DrawSubset (0);
        }
```

```
        public float CheckIntersection(float mouseX, float mouseY)
        {
            float rtn = float.MaxValue;
            IntersectInformation closestIntersection;
            Vector3 mouseRay_Near = new Vector3 (mouseX,mouseY,0);
            Vector3 mouseRay_Far = new Vector3 (mouseX,mouseY,1);


            mouseRay_Near.Unproject(Parent.Parent.device.Viewport,
Parent.Parent.device.Transform.Projection,
Parent.Parent.device.Transform.View, Transform);
            mouseRay_Far.Unproject(Parent.Parent.device.Viewport,
Parent.Parent.device.Transform.Projection,
Parent.Parent.device.Transform.View, Transform);

            bool intersects = Geom.Intersect(mouseRay_Near, mouseRay_Far, out
closestIntersection);

            if (intersects)
            {
                rtn = closestIntersection.Dist;
            }

            return rtn;
        }
```

Scene

The scene class provides an interface for rendering objects within a scene; line graphics are created and placed in vertex buffers. Solid shaded bodies are represented as GeometryObject's.

Local Variables:

grid_XLines	Gridlines from the origin in either direction; x-axis.
grid_ZLines	Gridlines from the origin in either direction; z-axis.
grid_XSize	Units between each line; x-axis.
grid_ZSize	Units between each line; z-axis.
grid_Transform	Transformation matrix for grid.
vertexBuffer_GridPatch	Vertices describing the grid.
vertexBuffer_ServoAxis	Vertices describing a servo axis.
vertexBuffer_Origin	Vertices describing the origin symbol.
vertexBuffer_Line	Vertices describing a line.
vertexBuffer_Triangle	Vertices describing a triangle.
vertexBuffer_Locator	Vertices describing the locator symbol.
RobotLinks	Array of links to render.
ArrowPos	Arrow in positive direction.
ArrowNeg	Arrow in negative direction.
ArrorRot	Rotational Arrow.

Public Variables:

Parent	Parent display port.
dev	Device to render to.
IdentyMatrix	4x4 Identity Matrix.

```
        private const int grid_XLines = 10;
        private const int grid_ZLines = 10;
```

```
        private const float grid_XSize = 10;
        private const float grid_ZSize = 10;

        private Matrix grid_Transform = new Matrix();

        private VertexBuffer vertexBuffer_GridPatch = null;
        private VertexBuffer vertexBuffer_ServoAxis = null;
        private VertexBuffer vertexBuffer_Origin = null;
        private VertexBuffer vertexBuffer_Line = null;
        private VertexBuffer vertexBuffer_Triangle = null;
        private VertexBuffer vertexBuffer_Locator = null;

        private GeometryObject[] RobotLinks = new
GeometryObject[RobotProps.num_Links];

        private GeometryObject ArrowPos = new GeometryObject();
        private GeometryObject ArrowNeg = new GeometryObject();
        private GeometryObject ArrowRot = new GeometryObject();

        public DirectXDisplayPort Parent;
        public Device dev = null;

        public static Matrix IdentyMatrix;
```

Public Functions:

`Void Initalise(Device, DisplayPort)`	Parent display port.
`Void ResetVertexBuffers()`	Device to render to.
`Void SetGridTransform(Matrix)`	4x4 Identity Matrix.
`Void Render()`	

Private Functions:

`void Render_Robot()`	Renders the robot in solid shading.
`void Render_RobotBones ()`	Renders kinematic diagram of the robot.
`void Render_ServoTorqueArrows()`	Renders arrows defining torque directions.
`void Render_EndEffecterForce()`	Renders arrows showing force at end Effecter.
`void Render_Grid()`	Renders the grid.
`void Render_ServoAllAxis()`	Renders axes of motors.
`void Render_InternalForces()`	Renders forces from gravity.

`Void Render_ServoTorquePoints ( int)`	Renders points used in torque calculation.
`void Render_ServoOrigins ()`	Renders origins of kineatic links.

```
        public void Initalise(Device dev, DirectXDisplayPort Parent)
        {
            this.dev = dev;
            this.Parent = Parent;

            IdentyMatrix = Matrix.Scaling(1, 1, 1);

            ResetVertexBuffers();

            SetGridTransform(IdentyMatrix);

            int i;

            #region Robot

            // Create Objects
            for (i = 0; i < RobotProps.num_Links; i++)
            {
                RobotLinks[i] = new GeometryObject();
            }

            // Load meshes

            for (i = 0; i < RobotProps.num_Links; i++)
            {
                RobotLinks[i].Initalise(dev,
RobotProps.linkTable[i].meshName, this);
                Console.Write("Loading mesh {0:F}", i);
                Console.WriteLine(RobotProps.linkTable[i].meshName);
            }


            #endregion

            // Load arrow

            ArrowPos.Initalise(dev, "ArrowPositive.x", this);
            ArrowNeg.Initalise(dev, "ArrowNegative.x", this);

            ArrowRot.Initalise(dev, "ArrowRot3.x", this);

            ArrowNeg.Geom.ComputeNormals();
            ArrowPos.Geom.ComputeNormals();
            ArrowRot.Geom.ComputeNormals();

        }
```

```
        public void ResetVertexBuffers()
        {
            if (dev == null) return; // Exit if we have no device
```

```
            int i = 0;

            #region Define Line

            vertexBuffer_Line = new
VertexBuffer(typeof(CustomVertex.PositionColored), 2, dev, 0,
CustomVertex.PositionColored.Format, Pool.Default);
            CustomVertex.PositionColored[] verts_Line =
(CustomVertex.PositionColored[])vertexBuffer_Line.Lock(0, 0);
            verts_Line[0].Position = Vector.Vec3(0, 0, 0);
            verts_Line[0].Color = System.Drawing.Color.Green.ToArgb();
            verts_Line[1].Position = Vector.Vec3(1, 1, 1);
            verts_Line[1].Color = System.Drawing.Color.Green.ToArgb();
            vertexBuffer_Line.Unlock();

            #endregion

            #region Define Triangle

            vertexBuffer_Triangle = new
VertexBuffer(typeof(CustomVertex.PositionColored), 3, dev, 0,
CustomVertex.PositionColored.Format, Pool.Default);
            CustomVertex.PositionColored[] verts_Triangle =
(CustomVertex.PositionColored[])vertexBuffer_Triangle.Lock(0, 0);
            verts_Triangle[0].Position = Vector.Vec3(0, 0, 0);
            verts_Triangle[0].Color = System.Drawing.Color.White.ToArgb();
            verts_Triangle[1].Position = Vector.Vec3(0, 1, 1);
            verts_Triangle[1].Color = System.Drawing.Color.White.ToArgb();
            verts_Triangle[2].Position = Vector.Vec3(1, 0, 1);
            verts_Triangle[2].Color = System.Drawing.Color.White.ToArgb();
            vertexBuffer_Triangle.Unlock();

            #endregion

            #region Define Origin

            vertexBuffer_Origin = new
VertexBuffer(typeof(CustomVertex.PositionColored), 6, dev, 0,
CustomVertex.PositionColored.Format, Pool.Default);

            CustomVertex.PositionColored[] verts_Origin =
(CustomVertex.PositionColored[])vertexBuffer_Origin.Lock(0, 0);


            verts_Origin[0].Position = Vector.Vec3(0, 0, 0);
            verts_Origin[0].Color = System.Drawing.Color.Red.ToArgb();
            verts_Origin[1].Position = Vector.Vec3(5, 0, 0);
            verts_Origin[1].Color = System.Drawing.Color.Red.ToArgb();

            verts_Origin[2].Position = Vector.Vec3(0, 0, 0);
            verts_Origin[2].Color = System.Drawing.Color.Green.ToArgb();
            verts_Origin[3].Position = Vector.Vec3(0, 5, 0);
            verts_Origin[3].Color = System.Drawing.Color.Green.ToArgb();

            verts_Origin[4].Position = Vector.Vec3(0, 0, 0);
            verts_Origin[4].Color = System.Drawing.Color.Blue.ToArgb();
            verts_Origin[5].Position = Vector.Vec3(0, 0, 5);
```

```
            verts_Origin[5].Color = System.Drawing.Color.Blue.ToArgb();


            vertexBuffer_Origin.Unlock();

            #endregion

            #region Define Grid


            // Create the grid patch vertex buffer
            vertexBuffer_GridPatch = new
VertexBuffer(typeof(CustomVertex.PositionColored), 4, dev, 0,
CustomVertex.PositionColored.Format, Pool.Default);
            CustomVertex.PositionColored[] verts_Grid =
(CustomVertex.PositionColored[])vertexBuffer_GridPatch.Lock(0, 0);

            verts_Grid[0].Position = Vector.Vec3(-1, 0, -1);
            verts_Grid[0].Color = System.Drawing.Color.DarkGray.ToArgb();
            verts_Grid[1].Position = Vector.Vec3(-1, 0, 0);
            verts_Grid[1].Color = System.Drawing.Color.DarkGray.ToArgb();
            verts_Grid[2].Position = Vector.Vec3(-1, 0, -1);
            verts_Grid[2].Color = System.Drawing.Color.DarkGray.ToArgb();
            verts_Grid[3].Position = Vector.Vec3(0, 0, -1);
            verts_Grid[3].Color = System.Drawing.Color.DarkGray.ToArgb();

            vertexBuffer_GridPatch.Unlock();

            #endregion

            #region Define Axie Co-Ords

            vertexBuffer_ServoAxis = new
VertexBuffer(typeof(CustomVertex.PositionColored), RobotProps.num_Servos * 2,
dev, 0, CustomVertex.PositionColored.Format, Pool.Default);

            CustomVertex.PositionColored[] verts_ServoAxis =
(CustomVertex.PositionColored[])vertexBuffer_ServoAxis.Lock(0, 0);

            for (i = 0; i < RobotProps.num_Servos * 2; i += 2)
            {
                verts_ServoAxis[i + 0].Position = RobotProps.servoTable[i /
2].BaseAxis.GetPoint(40);
                verts_ServoAxis[i + 0].Color =
System.Drawing.Color.Red.ToArgb();
                verts_ServoAxis[i + 1].Position = RobotProps.servoTable[i /
2].BaseAxis.GetPoint(-40);
                verts_ServoAxis[i + 1].Color =
System.Drawing.Color.Red.ToArgb();
            }
            vertexBuffer_ServoAxis.Unlock();
            #endregion

            #region Define Locator Co-ords
```

```
            vertexBuffer_Locator = new
VertexBuffer(typeof(CustomVertex.PositionColored), 6, dev, 0,
CustomVertex.PositionColored.Format, Pool.Default);

            CustomVertex.PositionColored[] verts_Locator =
(CustomVertex.PositionColored[])vertexBuffer_Locator.Lock(0, 0);

            for (i = 0; i < RobotProps.num_Servos * 2; i += 2)
            {
                verts_Locator[0].Position = Vector.Vec3(-1, 0, 0);
                verts_Locator[0].Color = System.Drawing.Color.Blue.ToArgb();

                verts_Locator[1].Position = Vector.Vec3(1, 0, 0);
                verts_Locator[1].Color = System.Drawing.Color.Blue.ToArgb();

                verts_Locator[2].Position = Vector.Vec3(0, -1, 0);
                verts_Locator[2].Color = System.Drawing.Color.Blue.ToArgb();

                verts_Locator[3].Position = Vector.Vec3(0, 1, 0);
                verts_Locator[3].Color = System.Drawing.Color.Blue.ToArgb();

                verts_Locator[4].Position = Vector.Vec3(0, 0, -1);
                verts_Locator[4].Color = System.Drawing.Color.Blue.ToArgb();

                verts_Locator[5].Position = Vector.Vec3(0, 0, 1);
                verts_Locator[5].Color = System.Drawing.Color.Blue.ToArgb();
            }
            vertexBuffer_Locator.Unlock();

            #endregion

        }
```

```
        public void SetGridTransform(Matrix TransformMat)
        {
            grid_Transform = TransformMat;
        }
```

```
        public void Render()
        {

            Material tmpMaterial = new Material();



            // Render components which use lighting
            #region Lit

            dev.RenderState.Lighting = true;

            // Robot Links
            if (RenderOptions.Robot_Physical) Render_Robot();

            if (RenderOptions.EndEffecter_ForceArrows !=
RenderOptions.ObjectRenderFlags.None) Render_EndEffecterForce();
```

```
            if (RenderOptions.Servo_TorqueDirection !=
RenderOptions.ObjectRenderFlags.None) Render_ServoTorqueArrows();


            #endregion

            // Render components which are pre lit
            #region Non-Lit
            dev.RenderState.Lighting = false;


            dev.VertexFormat = CustomVertex.PositionColored.Format;

            if (RenderOptions.World_Grid) Render_Grid();

            dev.RenderState.ZBufferEnable = false;

            if (RenderOptions.World_Origin)
            {
                dev.Transform.World = Matrix.Scaling(1.5f, 1.5f, 1.5f);
                dev.SetStreamSource(0, vertexBuffer_Origin, 0);
                dev.DrawPrimitives(PrimitiveType.LineList, 0, 3);
            }

            if (RenderOptions.EndEffecter_Origin)
            {
                dev.Transform.World =
RobotProps.endEffecter.Transform.ToD3DMatrix();
                dev.SetStreamSource(0, vertexBuffer_Origin, 0);
                dev.DrawPrimitives(PrimitiveType.LineList, 0, 3);
            }

            if (RenderOptions.Robot_Kinimatic) Render_RobotBones();

            if (RenderOptions.Servo_Origins) Render_ServoOrigins();

            dev.RenderState.ZBufferEnable = true;
            if (RenderOptions.Servo_Axis !=
RenderOptions.ObjectRenderFlags.None) Render_ServoAllAxis();
            dev.RenderState.ZBufferEnable = false;

            if (RenderOptions.Servo_Internal !=
RenderOptions.ObjectRenderFlags.None) Render_InternalForces();

            if (RenderOptions.Servo_TorqueCalcPoints)
            {
                if (RenderOptions.Robot_Physical)
                {
                    Render_ServoTorquePoints(RobotProps.mouseRay_Over);
                }
                else
                {
                    Render_ServoTorquePoints(1);
                }

            }
            dev.RenderState.ZBufferEnable = true;
```

```
            #endregion
        }
```

```
        private void Render_Robot()
        {
            int mouseRay_Over = -1;

            if (Parent.mouseRay_Check)
            {
                float mouseRay_MinDepth = 100000;
                float mouseRay_Depth;

                for (int i = 1; i < RobotProps.num_Links; i++)
                {


RobotLinks[i].SetTransform(Matrix4.ToD3DMatrix(RobotProps.linkTable[i].Global
_TransformMat));
                    mouseRay_Depth =
RobotLinks[i].CheckIntersection(Parent.mouseX, Parent.mouseY);
                    if (mouseRay_Depth < mouseRay_MinDepth)
                    {
                        mouseRay_MinDepth = mouseRay_Depth;
                        mouseRay_Over = i;
                    }
                }
                RobotProps.mouseRay_Over = mouseRay_Over - 1;
            }
           // Parent.mouseOver = mouseRay_Over - 1;   // Remove offset

            for (int i = 0; i < RobotProps.num_Links; i++)
            {

RobotLinks[i].SetTransform(Matrix4.ToD3DMatrix(RobotProps.linkTable[i].Global
_TransformMat));
                if (i != RobotProps.mouseRay_Over + 1)
                {
                    RobotLinks[i].Render(dev, false);
                }
                else
                {
                    RobotLinks[i].Render(dev, true);
                }
            }

        }
```

```
        private void Render_ServoTorqueArrows()
        {
            Matrix tmpMatrix = new Matrix();
            int servoId;
            int TickCount = (Environment.TickCount % 5000);
            Quaternion Rotation = new Quaternion ();
```

```
            float fAngle = (float)TickCount * (2.0f * (float)Math.PI) /
5000.0f;


            for (int i = 0; i < RobotProps.num_Links; i ++){

                servoId = RobotProps.linkTable[i].ServoID;

                if ((servoId != -1) && ((RenderOptions.Servo_TorqueDirection
== RenderOptions.ObjectRenderFlags.All) || (servoId ==
RobotProps.mouseRay_Over && RobotProps.linkTable[i].TorqueArrows > 0)))
                {

                    if (Math.Sign(RobotProps.servoTable[servoId].TorqueVec.Z)
> 0)
                    {
                        Rotation.RotateYawPitchRoll(0, 0, fAngle);
                    }
                    else
                    {
                        Rotation.RotateYawPitchRoll(0, (float)Math.PI,
fAngle);
                    }

                    if (RobotProps.servoTable[servoId].TorqueVec.Z != 0)
                    {
                        tmpMatrix.Transform(Vector.Vec3(0, 0, 0),
Quaternion.Identity, Vector.Vec3(RobotProps.linkTable[i].TorqueArrowScale,
RobotProps.linkTable[i].TorqueArrowScale,
RobotProps.linkTable[i].TorqueArrowScale), Vector.Vec3(0, 0, 0), Rotation,
Vector.Vec3(0, 0, RobotProps.linkTable[i].TorqueArrow1Offset));
                        tmpMatrix = Matrix.Multiply(tmpMatrix,
RobotProps.linkTable[RobotProps.servoTable[servoId].LinkID].Global_TransformM
at.ToD3DMatrix());

                        ArrowRot.SetTransform(tmpMatrix);
                        ArrowRot.Render(dev, false);

                        if (RobotProps.linkTable[i].TorqueArrows == 2)
                        {

                            tmpMatrix.Transform(Vector.Vec3(0, 0, 0),
Quaternion.Identity, Vector.Vec3(RobotProps.linkTable[i].TorqueArrowScale,
RobotProps.linkTable[i].TorqueArrowScale,
RobotProps.linkTable[i].TorqueArrowScale), Vector.Vec3(0, 0, 0), Rotation,
Vector.Vec3(0, 0, RobotProps.linkTable[i].TorqueArrow2Offset));
                            tmpMatrix = Matrix.Multiply(tmpMatrix,
RobotProps.linkTable[RobotProps.servoTable[servoId].LinkID].Global_TransformM
at.ToD3DMatrix());
                            ArrowRot.SetTransform(tmpMatrix);
                            ArrowRot.Render(dev, false);
                        }
                    }
                }
            }
        }
```

```
        private void Render_EndEffecterForce()
        {
            Vector3 Force;
            Matrix tmpMatrix = new Matrix();
            float scalingFactor = 10;

            if (RenderOptions.EndEffecter_ForceArrows ==
RenderOptions.ObjectRenderFlags.All)
            {
                Force = RobotProps.endEffecter.Force;
            }
            else
            {
                if (RobotProps.mouseRay_Over >= 0)
                {
                    Force =
RobotProps.servoTable[RobotProps.mouseRay_Over].ForceVec;
                }
                else
                {
                    Force = Vector.Vec3(0, 0, 0);
                }

            }

            // Y Axis
            tmpMatrix = Matrix.Scaling(Force.Y / scalingFactor, Force.Y /
scalingFactor, Force.Y / scalingFactor);
            tmpMatrix = Matrix.Multiply(tmpMatrix,
(Matrix.Translation(Vector.Add(Vector.Vec3(0, 5 * Math.Sign(Force.Y), 0),
RobotProps.endEffecter.Point))));

            if (Force.Y > 0)
            {
                ArrowPos.SetTransform(tmpMatrix);
                ArrowPos.Render(dev, false);
            }
            else
            {
                ArrowNeg.SetTransform(tmpMatrix);
                ArrowNeg.Render(dev, false);
            }

            // X Axis
            tmpMatrix = Matrix.Scaling(Force.X / scalingFactor, Force.X /
scalingFactor, Force.X / scalingFactor);
            tmpMatrix = Matrix.Multiply(tmpMatrix,
Matrix.RotationZ((float)Math.PI / 2));
            tmpMatrix = Matrix.Multiply(tmpMatrix,
(Matrix.Translation(Vector.Add(Vector.Vec3(-5 * Math.Sign(Force.X), 5 *
Math.Sign(Force.Y), 0), RobotProps.endEffecter.Point))));

            if (Force.X > 0)
            {
                ArrowPos.SetTransform(tmpMatrix);
                ArrowPos.Render(dev, false);
            }
```

```
            else
            {
                ArrowNeg.SetTransform(tmpMatrix);
                ArrowNeg.Render(dev, false);
            }

            // Z Axis
            tmpMatrix = Matrix.Scaling(Force.Z / scalingFactor, Force.Z /
scalingFactor, Force.Z / scalingFactor);
            tmpMatrix = Matrix.Multiply(tmpMatrix,
Matrix.RotationX((float)Math.PI / 2));
            tmpMatrix = Matrix.Multiply(tmpMatrix,
(Matrix.Translation(Vector.Add(Vector.Vec3(0, 5 * Math.Sign(Force.Y), 5 *
Math.Sign(Force.Z)), RobotProps.endEffecter.Point))));

            if (Force.Z > 0)
            {
                ArrowPos.SetTransform(tmpMatrix);
                ArrowPos.Render(dev, false);
            }
            else
            {
                ArrowNeg.SetTransform(tmpMatrix);
                ArrowNeg.Render(dev, false);
            }
        }
```

```
        private void Render_Grid()
        {
            for (float grid_X = (-grid_XLines); grid_X <= (grid_XLines + 1);
grid_X += 1)
            {
                for (float grid_Z = (-grid_ZLines); grid_Z <= (grid_ZLines +
1); grid_Z += 1)
                {

                    dev.Transform.World =
Matrix.Multiply(Matrix.Multiply(Matrix.Translation(grid_X, 0, grid_Z),
Matrix.Scaling(grid_XSize, 1, grid_ZSize)), grid_Transform);

                    dev.SetStreamSource(0, vertexBuffer_GridPatch, 0);
                    dev.DrawPrimitives(PrimitiveType.LineList, 0, 2);
                }
            }

            dev.Transform.World =
Matrix.Multiply(Matrix.Multiply(Matrix.Scaling(-grid_XSize * (grid_XLines * 2
+ 2), 1, -grid_ZSize * (grid_ZLines * 2 + 2)), Matrix.Translation(-grid_XSize
* (grid_XLines + 1), 0, -grid_ZSize * (grid_ZLines + 1))), grid_Transform);

            dev.SetStreamSource(0, vertexBuffer_GridPatch, 0);
            dev.VertexFormat = CustomVertex.PositionColored.Format;
            dev.DrawPrimitives(PrimitiveType.LineList, 0, 2);
        }
```

```
        private void Render_ServoAllAxis()
```

```
        {
            CustomVertex.PositionColored[] vb;
            int servoId;

            dev.Transform.World = Matrix.Identity;

            for (int i = 0; i < RobotProps.num_Links; i++)
            {

                servoId = RobotProps.linkTable[i].ServoID;

                if ((servoId != -1) && ((RenderOptions.Servo_Axis ==
RenderOptions.ObjectRenderFlags.All) || servoId == RobotProps.mouseRay_Over))
                {
                    vb =
(CustomVertex.PositionColored[])vertexBuffer_Line.Lock(0, 0);
                    vb[0].Position = Globals.RobotProps.servoTable
[servoId].Axis.GetPoint (25);
                    vb[0].Color = System.Drawing.Color.Blue.ToArgb();
                    vb[1].Position =
Globals.RobotProps.servoTable[servoId].Axis.GetPoint(-25);
                    vb[1].Color = System.Drawing.Color.Blue.ToArgb();
                    vertexBuffer_Line.Unlock();
                    dev.SetStreamSource(0, vertexBuffer_Line, 0);
                    dev.DrawPrimitives(PrimitiveType.LineList, 0, 1);
                }
            }
        }
```

```
        private void Render_InternalForces()
        {
            CustomVertex.PositionColored[] vb;
            int servoId;

            dev.Transform.World = Matrix.Identity;

            for (int i = 0; i < RobotProps.num_Links; i++)
            {
                servoId = RobotProps.linkTable[i].ServoID;

                if ((servoId != -1) && ((RenderOptions.Servo_Internal ==
RenderOptions.ObjectRenderFlags.All) || servoId == RobotProps.mouseRay_Over))
                {
                    dev.Transform.World = Matrix.Multiply(Matrix.Scaling(1,
1, 1),
Matrix.Multiply(Matrix.Translation(RobotProps.linkTable[i].CenterOfMass_Arm_L
ocal), RobotProps.linkTable[i].Global_TransformMat.ToD3DMatrix()));

                    dev.SetStreamSource(0, vertexBuffer_Origin, 0);
                    dev.DrawPrimitives(PrimitiveType.LineList, 0, 3);

                    vb =
(CustomVertex.PositionColored[])vertexBuffer_Line.Lock(0, 0);

                    vb[0].Position = Vector.Vec3 (0,0,0);
                    vb[0].Color = System.Drawing.Color.Pink.ToArgb();
```

```
                    vb[1].Position =
Vector.ScalarProduct(RobotProps.linkTable[i].Local_Gravity, .05f);
                    vb[1].Color = System.Drawing.Color.Pink.ToArgb();

                    vertexBuffer_Line.Unlock();
                    dev.SetStreamSource(0, vertexBuffer_Line, 0);
                    dev.DrawPrimitives(PrimitiveType.LineList, 0, 1);


                }
            }

        }
```

```
        private void Render_ServoTorquePoints(int servoId)
        {
            CustomVertex.PositionColored[] vb;
            Vector3 TorqueNormalCenter;
            Vector3 TorqueNormal = new Vector3();

            if (servoId < 0 || servoId >= RobotProps.num_Servos) return; //
Exit if servo dosn't exist

            dev.Transform.World = Matrix.Identity;
            // Render Torque normal

            TorqueNormalCenter = Vector3.Add(RobotProps.endEffecter.Point,
RobotProps.servoTable[servoId].TorquePoint);
            TorqueNormalCenter.Multiply(.5f);

            TorqueNormal = RobotProps.servoTable[servoId].TorqueNormal;
            TorqueNormal.Multiply(20f);

            vb = (CustomVertex.PositionColored[])vertexBuffer_Line.Lock(0,
0);
            vb[0].Position = TorqueNormalCenter;
            vb[0].Color = System.Drawing.Color.White.ToArgb();
            vb[1].Position = Vector3.Add(TorqueNormalCenter, TorqueNormal);
            vb[1].Color = System.Drawing.Color.White.ToArgb();
            vertexBuffer_Line.Unlock();
            dev.SetStreamSource(0, vertexBuffer_Line, 0);
            dev.DrawPrimitives(PrimitiveType.LineList, 0, 1);

            // Draw Projections

            vb = (CustomVertex.PositionColored[])vertexBuffer_Line.Lock(0,
0);
            vb[0].Position = RobotProps.servoTable[servoId].Axis.Point;
            vb[0].Color = System.Drawing.Color.Red.ToArgb();
            vb[1].Position = RobotProps.servoTable[servoId].TorquePoint;
            vb[1].Color = System.Drawing.Color.Red.ToArgb();
            vertexBuffer_Line.Unlock();
            dev.SetStreamSource(0, vertexBuffer_Line, 0);
            dev.DrawPrimitives(PrimitiveType.LineList, 0, 1);
```

```
            vb = (CustomVertex.PositionColored[])vertexBuffer_Line.Lock(0,
0);
            vb[0].Position = RobotProps.endEffecter.Point;
            vb[1].Position = RobotProps.servoTable[servoId].TorquePoint;
            vertexBuffer_Line.Unlock();
            dev.SetStreamSource(0, vertexBuffer_Line, 0);
            dev.DrawPrimitives(PrimitiveType.LineList, 0, 1);


            // Axis projection point
            dev.Transform.World = Matrix.Multiply(Matrix.Scaling(1, 1, 1),
Matrix.Translation(RobotProps.servoTable[servoId].Axis.Point));
            dev.SetStreamSource(0, vertexBuffer_Locator, 0);
            dev.DrawPrimitives(PrimitiveType.LineList, 0, 3);

            // Torque Point
            dev.Transform.World = Matrix.Multiply(Matrix.Scaling(1, 1, 1),
Matrix.Translation(RobotProps.servoTable[servoId].TorquePoint));
            dev.SetStreamSource(0, vertexBuffer_Locator, 0);
            dev.DrawPrimitives(PrimitiveType.LineList, 0, 3);

            // End Effecter
            dev.Transform.World =
Matrix.Translation(RobotProps.endEffecter.Point);
            dev.SetStreamSource(0, vertexBuffer_Locator, 0);
            dev.DrawPrimitives(PrimitiveType.LineList, 0, 3);
        }
```

```
        private void Render_ServoOrigins()
        {
            int i;
            dev.SetStreamSource(0, vertexBuffer_Origin, 0);

            for (i = 0; i < RobotProps.num_Links; i++)//
            {
                if (RobotProps.linkTable[i].ServoID != -1)
                {
                    dev.Transform.World =
Matrix.Multiply(Matrix.Invert(RobotProps.linkTable[i].Local_TransformMat.ToD3
DMatrix()), RobotProps.linkTable[i].Global_TransformMat.ToD3DMatrix());

                    dev.DrawPrimitives(PrimitiveType.LineList, 0, 3);
                }
            }

        }
```

```
        private void Render_RobotBones()
        {
            int i;
            CustomVertex.PositionColored[] vb;

            vb = (CustomVertex.PositionColored[])vertexBuffer_Line.Lock(0,
0);
            vb[0].Color = System.Drawing.Color.Black.ToArgb();
            vb[1].Color = System.Drawing.Color.Black.ToArgb();
```

```
            vertexBuffer_Line.Unlock();

            for (i = 0; i < RobotProps.num_Links - 1; i++)
            {
                dev.Transform.World = IdentyMatrix;
                vb =
(CustomVertex.PositionColored[])vertexBuffer_Line.Lock(0, 0);
                vb[0].Position =
RobotProps.linkTable[i].Global_TransformMat.GetTranslation();
                vb[1].Position =
RobotProps.linkTable[i+1].Global_TransformMat.GetTranslation();
                vertexBuffer_Line.Unlock();
                dev.SetStreamSource(0, vertexBuffer_Line, 0);
                dev.DrawPrimitives(PrimitiveType.LineList, 0, 1);

            }
        }
```

Namespace App.Math_Classes

Line

The line class defines a line, which consists of two vectors. One of which specifies direction, and the other a point which the line passes through.

Public Variables:

Point	A point the line passes through.
Vec	The direction of the line.

```
public Vector3 Point;
public Vector3 Vec;
```

Public Functions:

Constructor ()	Creates a line with no direction which passes through the origin.
Constructor (Vector3)	Creates a line with the specified direction, passing through the origin.
void CreateLine(Vector3, Vector3)	Creates a line which passes through the point specified with the specified direction.
Vector3 GetPoint(float)	Gets a point which is the specified distance from the defined point.
Vector3 ClosestPoint(Vector3)	Returns the point which is on the line that has the minimum distance to a specified point.

```
public Line()
{
    Point = new Vector3(0, 0, 0);
    Vec = new Vector3(0, 0, 0);

}
```

```
public Line(Vector3 Vec)
{
    Point = new Vector3(0, 0, 0);
    Vec = new Vector3(Vec.X ,Vec.Y , Vec.Z);
```

```
        }
```

```
        public void CreateLine(Vector3 p, Vector3 v)
        {
            Point.X = p.X;
            Point.Y = p.Y;
            Point.Z = p.Z;

            Vec.X = v.X;
            Vec.Y = v.Y;
            Vec.Z = v.Z;
        }
```

```
        public Vector3 GetPoint(float t)
        {
            Vector3 rtn = new Vector3();

            rtn.X = Point.X + t * Vec.X;
            rtn.Y = Point.Y + t * Vec.Y;
            rtn.Z = Point.Z + t * Vec.Z;

            return rtn;

        }
```

```
        public Vector3 ClosestPoint(Vector3 p)
        {
            Vector3 rtn = new Vector3();
            float t = 0;

            if ((Vec.X != 0) || (Vec.Y != 0) || (Vec.Z != 0))
            {

                t = -(Vec.X * Point.X - Vec.X * p.X + Vec.Y * Point.Y - Vec.Y
* p.Y + Vec.Z * Point.Z - Vec.Z * p.Z) / (Vec.X * Vec.X + Vec.Y * Vec.Y +
Vec.Z * Vec.Z);
            }

            rtn = this.GetPoint(t);

            return rtn;

        }
```

Matrix4

The Matrix4 class provides an implementation of a four by four matrix

Public Variables:

Data	Contains the elements of the matrix in a 4x4 array.

```
public float[,] Data = new float[4, 4];
```

Public Static Functions:

Void Add(Matrix4,Matrix4,Matrix4)	Adds two matrices and returns the result in a third.
Void Multiply(Matrix4,Matrix4,Matrix4)	Multiples two matrices placing the result in another.
Vector3 Multiply(Matrix4,Vector3)	Multiplies a matrix and vector, returning the resulting vector.
Vector3 TranslatePoint (Matrix4, Vector3)	Translates the point by translation data in a standard transformation matrix.
Vector3 RotateVector (Matrix4, Vector3)	Rotates a vector using rotational data from a standard transformation matrix.
Vector3 LocaliseVectorRotation (Vector3, Matrix)	Rotates a vector from global space to a local co-ordinate system.
Vector3 LocaliseVectorRotation(Vector3, Matrix4, Matrix4)	Rotates a vector from one co-ordinate space to another.
Line TransformLine(Matrix4, Line)	Transforms a line into local co-ordinates from global space.
Matrix ToD3DMatrix(Matrix4)	Converts a Matrix4 element to Direct3D matrix.
Matrix4 FromD3DMatrix(Matrix)	Returns a Matrix4 instance from a Direct3D matrix.

```
public static void Multiply(Matrix4 m1, Matrix4 m2, Matrix4 target)
{

    target.Data[0, 0] = m1.Data[0, 0] * m2.Data[0, 0] + m1.Data[1, 0]
* m2.Data[0, 1] + m1.Data[2, 0] * m2.Data[0, 2] + m1.Data[3, 0] * m2.Data[0,
3];
```

```
            target.Data[1, 0] = m1.Data[0, 0] * m2.Data[1, 0] + m1.Data[1, 0]
* m2.Data[1, 1] + m1.Data[2, 0] * m2.Data[1, 2] + m1.Data[3, 0] * m2.Data[1,
3];
            target.Data[2, 0] = m1.Data[0, 0] * m2.Data[2, 0] + m1.Data[1, 0]
* m2.Data[2, 1] + m1.Data[2, 0] * m2.Data[2, 2] + m1.Data[3, 0] * m2.Data[2,
3];
            target.Data[3, 0] = m1.Data[0, 0] * m2.Data[3, 0] + m1.Data[1, 0]
* m2.Data[3, 1] + m1.Data[2, 0] * m2.Data[3, 2] + m1.Data[3, 0] * m2.Data[3,
3];

            target.Data[0, 1] = m1.Data[0, 1] * m2.Data[0, 0] + m1.Data[1, 1]
* m2.Data[0, 1] + m1.Data[2, 1] * m2.Data[0, 2] + m1.Data[3, 1] * m2.Data[0,
3];
            target.Data[1, 1] = m1.Data[0, 1] * m2.Data[1, 0] + m1.Data[1, 1]
* m2.Data[1, 1] + m1.Data[2, 1] * m2.Data[1, 2] + m1.Data[3, 1] * m2.Data[1,
3];
            target.Data[2, 1] = m1.Data[0, 1] * m2.Data[2, 0] + m1.Data[1, 1]
* m2.Data[2, 1] + m1.Data[2, 1] * m2.Data[2, 2] + m1.Data[3, 1] * m2.Data[2,
3];
            target.Data[3, 1] = m1.Data[0, 1] * m2.Data[3, 0] + m1.Data[1, 1]
* m2.Data[3, 1] + m1.Data[2, 1] * m2.Data[3, 2] + m1.Data[3, 1] * m2.Data[3,
3];

            target.Data[0, 2] = m1.Data[0, 2] * m2.Data[0, 0] + m1.Data[1, 2]
* m2.Data[0, 1] + m1.Data[2, 2] * m2.Data[0, 2] + m1.Data[3, 2] * m2.Data[0,
3];
            target.Data[1, 2] = m1.Data[0, 2] * m2.Data[1, 0] + m1.Data[1, 2]
* m2.Data[1, 1] + m1.Data[2, 2] * m2.Data[1, 2] + m1.Data[3, 2] * m2.Data[1,
3];
            target.Data[2, 2] = m1.Data[0, 2] * m2.Data[2, 0] + m1.Data[1, 2]
* m2.Data[2, 1] + m1.Data[2, 2] * m2.Data[2, 2] + m1.Data[3, 2] * m2.Data[2,
3];
            target.Data[3, 2] = m1.Data[0, 2] * m2.Data[3, 0] + m1.Data[1, 2]
* m2.Data[3, 1] + m1.Data[2, 2] * m2.Data[3, 2] + m1.Data[3, 2] * m2.Data[3,
3];

            target.Data[0, 3] = m1.Data[0, 3] * m2.Data[0, 0] + m1.Data[1, 3]
* m2.Data[0, 1] + m1.Data[2, 3] * m2.Data[0, 2] + m1.Data[3, 3] * m2.Data[0,
3];
            target.Data[1, 3] = m1.Data[0, 3] * m2.Data[1, 0] + m1.Data[1, 3]
* m2.Data[1, 1] + m1.Data[2, 3] * m2.Data[1, 2] + m1.Data[3, 3] * m2.Data[1,
3];
            target.Data[2, 3] = m1.Data[0, 3] * m2.Data[2, 0] + m1.Data[1, 3]
* m2.Data[2, 1] + m1.Data[2, 3] * m2.Data[2, 2] + m1.Data[3, 3] * m2.Data[2,
3];
            target.Data[3, 3] = m1.Data[0, 3] * m2.Data[3, 0] + m1.Data[1, 3]
* m2.Data[3, 1] + m1.Data[2, 3] * m2.Data[3, 2] + m1.Data[3, 3] * m2.Data[3,
3];

        }
```

```
        public static void Add(Matrix4 m1, Matrix4 m2, Matrix4 target)
        {
            target.Data[0, 0] = m1.Data[0, 0] + m2.Data[0, 0];
            target.Data[1, 0] = m1.Data[1, 0] + m2.Data[1, 0];
            target.Data[2, 0] = m1.Data[2, 0] + m2.Data[2, 0];
```

```
            target.Data[3, 0] = m1.Data[3, 0] + m2.Data[3, 0];

            target.Data[0, 1] = m1.Data[0, 1] + m2.Data[0, 1];
            target.Data[1, 1] = m1.Data[1, 1] + m2.Data[1, 1];
            target.Data[2, 1] = m1.Data[2, 1] + m2.Data[2, 1];
            target.Data[3, 1] = m1.Data[3, 1] + m2.Data[3, 1];

            target.Data[0, 2] = m1.Data[0, 2] + m2.Data[0, 2];
            target.Data[1, 2] = m1.Data[1, 2] + m2.Data[1, 2];
            target.Data[2, 2] = m1.Data[2, 2] + m2.Data[2, 2];
            target.Data[3, 2] = m1.Data[3, 2] + m2.Data[3, 2];

            target.Data[0, 3] = m1.Data[0, 3] + m2.Data[0, 3];
            target.Data[1, 3] = m1.Data[1, 3] + m2.Data[1, 3];
            target.Data[2, 3] = m1.Data[2, 3] + m2.Data[2, 3];
            target.Data[3, 3] = m1.Data[3, 3] + m2.Data[3, 3];
        }
```

```
        public static Vector4 Multiply(Matrix4 m, Vector4 v){
            Vector4 rtn;
            rtn.W = m.Data[0, 0] * v.W + m.Data[1, 0] * v.X + m.Data[2, 0] *
v.Y + m.Data[3, 0] * v.Z;
            rtn.X = m.Data[0, 1] * v.W + m.Data[1, 1] * v.X + m.Data[2, 1] *
v.Y + m.Data[3, 1] * v.Z;
            rtn.Y = m.Data[0, 2] * v.W + m.Data[1, 2] * v.X + m.Data[2, 2] *
v.Y + m.Data[3, 2] * v.Z;
            rtn.Z = m.Data[0, 3] * v.W + m.Data[1, 3] * v.X + m.Data[2, 3] *
v.Y + m.Data[3, 3] * v.Z;
            return rtn;
        }
```

```
        public static Vector3 TranslatePoint(Matrix4 m,Vector3 p)
        {
            Vector3 rtn = new Vector3();

            rtn.X = m.Data[3, 0] + p.X;
            rtn.Y = m.Data[3, 1] + p.Y;
            rtn.Z = m.Data[3, 2] + p.Z;

            return rtn;
        }
```

```
        public static Vector3 RotateVector(Matrix4 m, Vector3 v)
        {

            Vector3 rtn = new Vector3();

            rtn.X = m.Data[0, 0] * v.X + m.Data[1, 0] * v.Y + m.Data[2, 0] *
v.Z;
            rtn.Y = m.Data[0, 1] * v.X + m.Data[1, 1] * v.Y + m.Data[2, 1] *
v.Z;
            rtn.Z = m.Data[0, 2] * v.X + m.Data[1, 2] * v.Y + m.Data[2, 2] *
v.Z;

            return rtn;
        }
```

```
        public static Vector3 LocaliseVectorRotation(Vector3 vec, Matrix4
Frame_To)
        {
            Vector3 rtn = new Vector3();

            Matrix ToMat = Frame_To.ToD3DMatrix();
            ToMat = Matrix.TransposeMatrix(ToMat);
            rtn =
Math_Classes.Matrix4.RotateVector(Math_Classes.Matrix4.FromD3Dmatrix(ToMat),
vec);

            return rtn;

        }
```

```
        public static Vector3 LocaliseVectorRotation(Vector3 vec, Matrix4
Frame_From, Matrix4 Frame_To)
        {
            Vector3 rtn = new Vector3();

            Matrix FromMat = Frame_From.ToD3Dmatrix();
            Matrix ToMat = Frame_To.ToD3Dmatrix();

            Matrix TransformationMat = Matrix.Multiply (FromMat,
Matrix.TransposeMatrix (ToMat));

            rtn =
Math_Classes.Matrix4.RotateVector(Math_Classes.Matrix4.FromD3Dmatrix(Transfor
mationMat), vec);

            return rtn;

        }
```

```
        public static Line TransformLine(Matrix4 m, Line l)
        {
            Line rtn = new Line();

            rtn.Point = TranslatePoint(m, l.Point);
            rtn.Vec = RotateVector(m, l.Vec);

            return rtn;
        }
public static Matrix ToD3Dmatrix(Matrix4 m)
        {
            Matrix rtn = new Matrix();

            rtn.M11 = m.Data [0, 0];
            rtn.M12 = m.Data[0, 1];
            rtn.M13 = m.Data[0, 2];
            rtn.M14 = m.Data[0, 3];

            rtn.M21 = m.Data[1, 0];
            rtn.M22 = m.Data[1, 1];
            rtn.M23 = m.Data[1, 2];
```

```
            rtn.M24 = m.Data[1, 3];

            rtn.M31 = m.Data[2, 0];
            rtn.M32 = m.Data[2, 1];
            rtn.M33 = m.Data[2, 2];
            rtn.M34 = m.Data[2, 3];

            rtn.M41 = m.Data[3, 0];
            rtn.M42 = m.Data[3, 1];
            rtn.M43 = m.Data[3, 2];
            rtn.M44 = m.Data[3, 3];

            return rtn;
        }
```

```
        public static Matrix4 FromD3Dmatrix(Matrix m)
        {
            Matrix4 rtn = new Matrix4();
            rtn.Data[0, 0] = m.M11;
            rtn.Data[0, 1] = m.M12;
            rtn.Data[0, 2] = m.M13;
            rtn.Data[0, 3] = m.M14;

            rtn.Data[1, 0] = m.M21;
            rtn.Data[1, 1] = m.M22;
            rtn.Data[1, 2] = m.M23;
            rtn.Data[1, 3] = m.M24;

            rtn.Data[2, 0] = m.M31;
            rtn.Data[2, 1] = m.M32;
            rtn.Data[2, 2] = m.M33;
            rtn.Data[2, 3] = m.M34;

            rtn.Data[3, 0] = m.M41;
            rtn.Data[3, 1] = m.M42;
            rtn.Data[3, 2] = m.M43;
            rtn.Data[3, 3] = m.M44;

            return rtn;
        }
```

Public Functions:

void FromColumnVec (Vector4, Vector4, Vector4 ,Vector4)	Fills the matrix from four 4-D vectors, these vectors are treated as the columns when inserted into the matrix.
void FromRowVec (Vector4, Vector4, Vector4 ,Vector4)	Fills the matrix from four 4-D vectors, these vectors are treated as the rows when inserted into the matrix.

void FromDHParam(float, float, float, float)	Fills the matrix according to D-H parameters.
void FromMatrix(Matrix4)	Fills the matrix as a copy.
void ToMatrix(Matrix4)	Copies this matrix into another.
Matrix ToD3DMatrix ()	Returns a Direct3D Matrix.
Vector3 GetTranslation ()	Returns the translation data from the matrix.
void WriteToConsole()	Displays the matrix in text format on the console.

```
        public void FromColumnVec (Vector4 c1, Vector4 c2, Vector4 c3,Vector4
c4)
        {
            Data[0,0] = c1.W;
            Data[0,1] = c1.X;
            Data[0,2] = c1.Y;
            Data[0,3] = c1.Z;

            Data[1,0] = c2.W;
            Data[1,1] = c2.X;
            Data[1,2] = c2.Y;
            Data[1,3] = c2.Z;

            Data[2,0] = c3.W;
            Data[2,1] = c3.X;
            Data[2,2] = c3.Y;
            Data[2,3] = c3.Z;

            Data[3,0] = c4.W;
            Data[3,1] = c4.X;
            Data[3,2] = c4.Y;
            Data[3,3] = c4.Z;
        }
```

```
        public void FromRowVec (Vector4 r1, Vector4 r2, Vector4 r3, Vector4
r4)
        {
            Data[0,0] = r1.W;
            Data[1,0] = r1.X;
            Data[2,0] = r1.Y;
            Data[3,0] = r1.Z;

            Data[0,1] = r2.W;
            Data[1,1] = r2.X;
            Data[2,1] = r2.Y;
            Data[3,1] = r2.Z;

            Data[0,2] = r3.W;
            Data[1,2] = r3.X;
            Data[2,2] = r3.Y;
            Data[3,2] = r3.Z;
```

```
            Data[0,3] = r4.W;
            Data[1,3] = r4.X;
            Data[2,3] = r4.Y;
            Data[3,3] = r4.Z;
        }
```

```
        public void FromDHParam(float theta, float alpha, float a, float d)
        {
            this.FromRowVec(Vector.Vec4(Math.Cos(theta),    -Math.Sin(theta)
* Math.Cos(alpha),    Math.Sin(theta) * Math.Sin(alpha),      d *
Math.Cos(theta)),
                            Vector.Vec4(Math.Sin(theta),    Math.Cos(theta) *
Math.Cos(alpha),     -Math.Cos(theta) * Math.Sin (alpha),    d *
Math.Sin(theta)),
                            Vector.Vec4(0,                  Math.Sin(alpha),
Math.Cos(alpha),                        a),
                            Vector.Vec4(0,                  0,
0,                                      1));

        }
```

```
        public void FromMatrix(Matrix4 m)
        {
            int i, k;

            for (i = 0; i < 4; i++)
            {
                for (k = 0; k < 4; k++)
                {
                    Data[i, k] = m.Data[i, k];
                }
            }
        }
```

```
        public void ToMatrix(Matrix4 m)
        {
            int i, k;

            for (i = 0; i < 4; i++)
            {
                for (k = 0; k < 4; k++)
                {
                    m.Data[i, k] = Data[i, k];
                }
            }
        }
```

```
        public Matrix ToD3DMatrix (){
            return ToD3DMatrix(this);
        }
```

```
        public Vector3 GetTranslation ()
        {
```

```
            Vector3 rtn = new Vector3 ();

            rtn.X = this.Data[3,0];
            rtn.Y = this.Data[3,1];
            rtn.Z = this.Data[3,2];

            return rtn;

        }
```

```
        public void WriteToConsole()
        {
            Console.Write ("\n\n{0:F}     {1:F}     {2:F}     {3:F}\n",
Data[0,0], Data[1,0],Data[2,0],Data[3,0]);
            Console.Write ("{0:F}     {1:F}     {2:F}     {3:F}\n",
Data[0,1], Data[1,1],Data[2,1],Data[3,1]);
            Console.Write ("{0:F}     {1:F}     {2:F}     {3:F}\n",
Data[0,2], Data[1,2],Data[2,2],Data[3,2]);
            Console.Write ("{0:F}     {1:F}     {2:F}     {3:F}\n",
Data[0,3], Data[1,3],Data[2,3],Data[3,3]);

        }
```

Solver6DoF

The solver class provides a wrapper for implementing a compiled MatLab function, which implements the Moore-Penrose pseudo inverse.

Public Static Functions:

Vector6 Solve (float[], float[])	Solves a 6x6 matrix of co-efficients, which are equal to a six dimentional vector.

```
public static Vector6 Solve(float[] Coefficients, float[] Constants)
        {
            Solve6DoF2.Solver Solver = new Solve6DoF2.Solver (); // The
Matlab solver

            // Assign near zero values to zero to prevent overly large
determanints
            double Cuttoff = Globals.RobotProps.cutoff_Solver6DoF;

            for (int Column = 0; Column < 6; Column ++){
                for (int Row = 0; Row < 6; Row++)
                {
                    if (Coefficients[Row * 6 + Column] < Cuttoff)
                    {
                        Coefficients[Row * 6 + Column] = 0;
                    }
                }
            }


            // Matlab data types

            MWNumericArray Coefficients_MW = new MWNumericArray(6, 6,
Coefficients);
            MWNumericArray Constants_MW = new MWNumericArray(6, 1,
Constants);
            MWNumericArray Solution_MW = null;
            Double [,] NativeArray = null;



            Solution_MW = (MWNumericArray)Solver.Solve6DOF(Constants_MW,
Coefficients_MW); // Solve
            NativeArray =
(Double[,])Solution_MW.ToArray(MWArrayComponent.Real); // Convert from matlab
back to C# data type


            // Put solution into a Vector6
            Vector6 Solution = new Vector6(NativeArray[0, 0], NativeArray[1,
0], NativeArray[2, 0], NativeArray[3, 0], NativeArray[4, 0], NativeArray[5,
0]);
```

```
            Solution.WriteToConsole();
            return Solution;
        }
```

Vector

The vector class contains helper functions for manipulating three and four dimensional vectors.

Public Static Functions:

Vector3 Vec3 (float, float, float)	Returns a new three dimentional vector with the specified values.
Vector3 Vec3 (double, double, double)	"
Vector4 Vec4 (float, float, float, float)	Returns a new four dimentional vector with the specified values.
Vector4 Vec4 (double, double, double, double)	"
Vector3 Subtract (Vector3, Vector3)	Subtracts one vector from the other.
Vector4 Subtract (Vector3, Vector3)	"
Vector3 Add (Vector3, Vector3)	Adds two vectors.
float Distance (Vector3, Vector3)	Returns the distance between two points stored as vectors.
Vector3 CrossProduct (Vector3, Vector3)	Returns the cross product of two vectors.
Vector3 ScalarProduct (Vector3, float)	Performs a scalar multiplication.
Vector3 Normalise (Vector3)	Returns a vector of unit length with direction equal to the input vector.
Vector3 NormalTo (Vector3, Vector3, Vector3)	Returns a vector normal to the plane specified by three points.
Vector3 Copy (Vector3)	Copies values from one vector to a new one.

void WriteToConsole(Vector3)	Displays the vector on the console window.
void WriteToConsole(Vector4)	"

```
public static Vector3 Vec3(float x, float y, float z)
{
    Vector3 Rtn;
    Rtn.X = x;
    Rtn.Y = y;
    Rtn.Z = z;
    return Rtn;
}
```

```
public static Vector3 Vec3(double x, double y, double z)
{
    Vector3 Rtn;
    Rtn.X = (float)x;
    Rtn.Y = (float)y;
    Rtn.Z = (float)z;
    return Rtn;
}
```

```
public static Vector4 Vec4(float w, float x, float y, float z)
{
    Vector4 Rtn;

    Rtn.W = w;
    Rtn.X = x;
    Rtn.Y = y;
    Rtn.Z = z;

    return Rtn;
}
```

```
public static Vector4 Vec4(double w, double x, double y, double z)
{
    Vector4 Rtn;

    Rtn.W = (float) w;
    Rtn.X = (float) x;
    Rtn.Y = (float) y;
    Rtn.Z = (float) z;


    return Rtn;
}
```

```
public static Vector3 Subtract(Vector3 v1, Vector3 v2)
{
    Vector3 rtn = new Vector3();
    rtn.X = v1.X - v2.X;
    rtn.Y = v1.Y - v2.Y;
    rtn.Z = v1.Z - v2.Z;
    return rtn;
```

```
        }
```

```
        public static Vector4 Subtract(Vector4 v1, Vector4 v2)
        {
            Vector4 rtn = new Vector4();
            rtn.W = v1.W - v2.W;
            rtn.X = v1.X - v2.X;
            rtn.Y = v1.Y - v2.Y;
            rtn.Z = v1.Z - v2.Z;
            return rtn;

        }
```

```
        public static Vector3 Add(Vector3 v1, Vector3 v2)
        {
            Vector3 rtn = new Vector3();

            rtn.X = v1.X + v2.X;
            rtn.Y = v1.Y + v2.Y;
            rtn.Z = v1.Z + v2.Z;

            return rtn;
        }
```

```
        public static float Distance(Vector3 p1, Vector3 p2)
        {
            float rtn;
            Vector3 tmp = new Vector3();

            tmp = Vector.Subtract(p1, p2);
            rtn = (float)Math.Sqrt(tmp.X * tmp.X + tmp.Y * tmp.Y + tmp.Z *
tmp.Z);

            return rtn;
        }
```

```
        public static Vector3 CrossProduct(Vector3 v1, Vector3 v2)
        {
            Vector3 rtn = new Vector3();

            rtn.X = (v1.Y * v2.Z - v2.Y * v1.Z);
            rtn.Y = (v1.X * v2.Z - v2.X * v1.Z);
            rtn.Z = (v1.X * v2.Y - v2.X * v1.Y);

            return rtn;
        }
```

```
        public static Vector3 ScalarProduct(Vector3 v, float s)
        {
            Vector3 rtn = new Vector3();

            rtn.X = v.X * s;
            rtn.Y = v.Y * s;
            rtn.Z = v.Z * s;
```

```
            return rtn;
        }
```

```
        public static Vector3 Normalise(Vector3 v1)
        {
            Vector3 rtn ;
            float mag;

            rtn = Vector.Copy(v1);

            mag = Distance(rtn, Vec3(0, 0, 0));
            if (mag != 0)
            {
                rtn.X /= mag;
                rtn.Y /= mag;
                rtn.Z /= mag;
            }

            return rtn;
        }
```

```
        public static Vector3 NormalTo(Vector3 p1, Vector3 p2, Vector3 p3)
        {
            Vector3 a = new Vector3();
            Vector3 b = new Vector3();
            Vector3 n = new Vector3();

            a = Subtract(p2, p1);
            b = Subtract(p3, p1);

            n = Vector3.Cross (a, b);

            // Normalise the vector

            n = Normalise(n);
            return n;
        }
```

```
        public static Vector3 Copy(Vector3 v)
        {
            Vector3 rtn = new Vector3();

            rtn.X = v.X;
            rtn.Y = v.Y;
            rtn.Z = v.Z;

            return rtn;
        }
```

```
        public static void WriteToConsole (Vector4 v){
            Console.Write("\n\n{0:F}     {1:F}     {2:F}     {3:F}\n", v.W,
v.X, v.Y , v.Z);
        }
```

```
        public static void WriteToConsole(Vector3 v)
```

```
        {
            Console.Write("{0,-7:F}      {1,-7:F}      {2,-7:F}\n", v.X, v.Y,
v.Z);
        }
```

Vector6

Vector6 provides the implementation of a six dimensional vector:

Public Variables:

A	Element of the vector.
B	"
C	"
D	"
E	"
F	"

```
public float A;
public float B;
public float C;
public float D;
public float E;
public float F;
```

Public Static Functions:

Vector6 Vec6(float, float, float, float, float, float)	Creates a vector from passed values.
Vector6 Vec6(double, double, double, double, double, double)	"
Vector6 Vec6 (Vector3, Vector3)	Creates a vector from two three dimentional vectors.
Vector6 Add (Vector6, Vector6)	Adds two vectors.
Vector6 Subtract (Vector6, Vector6)	Subtracts two vectors.
Vector6 Scalar (Vector6, Vector6)	Multiplies each element in a vector by its corresponding element in a second vector.
Vector6 Scalar (Vector6, float)	Multiplies every element in the vector by a scalar.

float Total (Vector6)	Returns the total of all elements in the vector.

```
        public static Vector6 Vec6(float A, float B, float C, float D, float
E, float F)
        {
            Vector6 rtn = new Vector6(A, B, C, D, E, F);
            return rtn;
        }
```

```
        public static Vector6 Vec6(double A, double B, double C, double D,
double E, double F)
        {
            Vector6 rtn = new Vector6((float)A, (float)B, (float)C, (float)D,
(float)E, (float)F);
            return rtn;
        }
```

```
        public static Vector6 Vec6(Vector3 v1, Vector3 v2)
        {
            Vector6 rtn = new Vector6(v1.X, v1.Y, v1.Z, v2.X, v2.Y, v2.Z);
            return rtn;
        }
```

```
        public static Vector6 Add(Vector6 v1, Vector6 v2)
        {
            Vector6 rtn = new Vector6();

            rtn.A = v1.A + v2.A;
            rtn.B = v1.B + v2.B;
            rtn.C = v1.C + v2.C;
            rtn.D = v1.D + v2.D;
            rtn.E = v1.E + v2.E;
            rtn.F = v1.F + v2.F;

            return rtn;
        }
```

```
        public static Vector6 Subtract(Vector6 v1, Vector6 v2)
        {
            Vector6 rtn = new Vector6();

            rtn.A = v1.A * v2.A;
            rtn.B = v1.B * v2.B;
            rtn.C = v1.C * v2.C;
            rtn.D = v1.D * v2.D;
            rtn.E = v1.E * v2.E;
            rtn.F = v1.F * v2.F;

            return rtn;
        }
```

```
        public static Vector6 Scalar(Vector6 v1, Vector6 v2)
        {
            Vector6 rtn = new Vector6();
```

```
            rtn.A = v1.A * v2.A;
            rtn.B = v1.B * v2.B;
            rtn.C = v1.C * v2.C;
            rtn.D = v1.D * v2.D;
            rtn.E = v1.E * v2.E;
            rtn.F = v1.F * v2.F;

            return rtn;
        }
```

```
        public static Vector6 Scalar(Vector6 v1, float Scalar)
        {
            Vector6 rtn = new Vector6();

            rtn.A = v1.A * Scalar;
            rtn.B = v1.B * Scalar;
            rtn.C = v1.C * Scalar;
            rtn.D = v1.D * Scalar;
            rtn.E = v1.E * Scalar;
            rtn.F = v1.F * Scalar;

            return rtn;
        }
```

```
        public static float Total(Vector6 v)
        {
            return v.A + v.B + v.C + v.D + v.E + v.F;
        }
```

Public Functions:

Constructor ()	Creates vector with no direction.
Constructor (float, float, float,float, float, float)	Creates a vector with the specified direction.
Contructor (double, double, double, double, double, double)	“
void Add (Vector6)	Adds a vector.
void Subtract (Vector6)	Subtracts a vector.
void Scalar (Vector6)	Multiplies each element by its corresponding element in a second vector.
void Scalar (float)	Multiplies every element in the vector by a scalar.

void Add (int, float)	Adds a value to the specified element.
Void Scalar (int, float)	Multiplies the specified element a scalar.
float GetElement (int)	Returns the value of the specified element.
void SetElement(int, float)	Sets the element to the specified value.
void SwapElements(int, int)	Swaps two elements within the vector.
void WriteToConsole()	Writes the vector to the console window.

```
        public Vector6()
        {
            this.A = 0;
            this.B = 0;
            this.C = 0;
            this.D = 0;
            this.E = 0;
            this.F = 0;
        }
```

```
        public Vector6(float A, float B, float C, float D, float E, float F)
        {
            this.A = A;
            this.B = B;
            this.C = C;
            this.D = D;
            this.E = E;
            this.F = F;
        }
```

```
        public Vector6(double A, double B, double C, double D, double E,
double F)
        {
            this.A = (float)A;
            this.B = (float)B;
            this.C = (float)C;
            this.D = (float)D;
            this.E = (float)E;
            this.F = (float)F;
        }
```

```
        public void Add(Vector6 v)
        {

            this.A = this.A + v.A;
            this.B = this.B + v.B;
            this.C = this.C + v.C;
            this.D = this.D + v.D;
            this.E = this.E + v.E;
            this.F = this.F + v.F;

        }
```

```
        public void Subtract(Vector6 v)
        {

            this.A = this.A * v.A;
            this.B = this.B * v.B;
            this.C = this.C * v.C;
            this.D = this.D * v.D;
            this.E = this.E * v.E;
            this.F = this.F * v.F;

        }
```

```
        public void Scalar(Vector6 v)
        {

            this.A = this.A * v.A;
            this.B = this.B * v.B;
            this.C = this.C * v.C;
            this.D = this.D * v.D;
            this.E = this.E * v.E;
            this.F = this.F * v.F;

        }
```

```
        public void Scalar(float Scalar)
        {

            this.A = this.A * Scalar;
            this.B = this.B * Scalar;
            this.C = this.C * Scalar;
            this.D = this.D * Scalar;
            this.E = this.E * Scalar;
            this.F = this.F * Scalar;

        }
```

```
        public float Total()
        {
            return A + B + C + D + E + F;
        }
```

```
        public void Add(int Element, float Value)
        {
            switch (Element)
            {
                case 0:
                    A = A + Value;
                    break;
                case 1:
                    B = B + Value;
                    break;
                case 2:
                    C = C + Value;
                    break;
                case 3:
```

```
                D = D + Value;
                break;
            case 4:
                E = E + Value;
                break;
            case 5:
                F = F + Value;
                break;
        }
    }
```

```
    public void Scalar(int Element, float Value)
    {
        switch (Element)
        {
            case 0:
                A = A * Value;
                break;
            case 1:
                B = B * Value;
                break;
            case 2:
                C = C * Value;
                break;
            case 3:
                D = D * Value;
                break;
            case 4:
                E = E * Value;
                break;
            case 5:
                F = F * Value;
                break;
        }
    }
```

```
    public float GetElement(int Element)
    {
        switch (Element)
        {
            case 0:
                return A;
            case 1:
                return B;
            case 2:
                return C;
            case 3:
                return D;
            case 4:
                return E;
            case 5:
                return F;
        }
        return 0;
    }
```

```
        public void SetElement(int Element, float Value)
        {
            switch (Element)
            {
                case 0:
                    A = Value;
                    break;
                case 1:
                    B = Value;
                    break;
                case 2:
                    C = Value;
                    break;
                case 3:
                    D = Value;
                    break;
                case 4:
                    E = Value;
                    break;
                case 5:
                    F = Value;
                    break;
            }
        }
```

```
        public void SwapElements(int Element1, int Element2)
        {
            float tmp = this.GetElement(Element1);
            this.SetElement(Element1, this.GetElement(Element2));
            this.SetElement(Element2, tmp);
        }
```

```
        public void WriteToConsole()
        {

            Console.Write("{0,-7:F}     {1,-7:F}     {2,-7:F}     {3,-7:F}
{4,-7:F}     {5,-7:F}\n", A, B, C, D, E, F);

        }
```

Namespace App.Physical_Components

ArmModel

The ArmModel class, models the functionality of the simulated haptic device.

Local Variables:

ClassVerbose	Does the class output anything to the console(?).
Num_TransFrames	Number of transformation frames.

Public Variables:

Grip	EndEffecter of the device.
GroudFrame	The base transformation matrix for the device.
Controller	Comms controller for the arm.

```
        private bool ClassVerbose = true;

        private const int num_TransFrames = 8;

        public EndEffecter Grip = new EndEffecter();

        public Matrix4 GroundFrame = new Matrix4();

        public Threads.RobotController Controller = new
Threads.RobotController();
```

Public Functions:

Constructor ()	Initialises the model.
void FowardKinimatics ()	Performs forward kinematic equations.
void Transformations ()	Updates transformation matrices for the chain, depending on motor rotation.
void Forces ()	Calculates forces within the chain.

```
        public ArmModel()
        {
            Link.LoadLinks ();

            Globals.RobotProps.endEffecter = Grip;


            this.FowardKinimatics();
        }
```

```
        public void FowardKinimatics()
        {

            this.Transformations();
            Link.UpdateLinks();
            this.Forces();
        }
```

```
        public void Transformations()
        {
            bool LocalVerbose = false;

            Matrix4 CompleteTrans = new Matrix4();
            Matrix4 tmp = new Matrix4();

            float di;
            float ai;
            float alpha;
            float theta_Offset;


            #region "Define Links"


            // Ground Frame   (rotated 90 about x axis to align Z with first
servo)


            GroundFrame.FromRowVec(Vector.Vec4(1, 0, 0, 0),
                                   Vector.Vec4(0, 1, 0, 10),
                                   Vector.Vec4(0, 0, 1, 0),
                                   Vector.Vec4(0, 0, 0, 1));


            // DH Transform Tables
            for (int i = 0; i < Globals.RobotProps.num_Links; i++)
            {
                float ServoAngle = 0;

                if (Globals.RobotProps.linkTable[i].ServoID >= 0)
                {
                    ServoAngle =
Globals.RobotProps.servoTable[Globals.RobotProps.linkTable[i].ServoID].Angle;
                }

                di = Globals.RobotProps.linkTable[i].di;
```

```
                ai = Globals.RobotProps.linkTable[i].ai;
                alpha = Globals.RobotProps.linkTable[i].alpha;
                theta_Offset = Globals.RobotProps.linkTable[i].thetaOffset;


Globals.RobotProps.linkTable[i].Local_TransformMat.FromDHParam
                (ServoAngle + theta_Offset, alpha, ai, di);

            }


            #endregion

            // Multiply Chain Out

            #region "Multiply Chain"

            // BaseFrame
            GroundFrame.ToMatrix(tmp);

            for (int i = 0; i < Globals.RobotProps.num_Links; i++)
            {
                Matrix4.Multiply(tmp,
Globals.RobotProps.linkTable[i].Local_TransformMat, CompleteTrans);
                CompleteTrans.ToMatrix(tmp);

CompleteTrans.ToMatrix(Globals.RobotProps.linkTable[i].Global_TransformMat);


                if (ClassVerbose && LocalVerbose)
CompleteTrans.WriteToConsole();
            }


            if (ClassVerbose && LocalVerbose) CompleteTrans.WriteToConsole();
            if (ClassVerbose && LocalVerbose) Console.Write("\n\n");

            CompleteTrans.ToMatrix(Grip.Transform);

            #endregion

            Grip.Point = Matrix4.TranslatePoint(Grip.Transform,
Vector.Vec3(0, 0, 0));

        }
```

```
        public void Forces()
        {

            bool LocalVerbose = false;

            int servo_Num;

            Grip.Force = Vector.Vec3(0, 0, 0);

            servo_Num = 0;
```

```
            for (servo_Num = 0; servo_Num < Globals.RobotProps.num_Servos;
servo_Num++)
            {


Globals.RobotProps.servoTable[servo_Num].UpdateForceVec(Grip);
                Grip.Force = Vector.Add(Grip.Force,
Globals.RobotProps.servoTable[servo_Num].ForceVec);


Globals.RobotProps.servoTable[servo_Num].EndEffecterTorqueComponent =

Math_Classes.Matrix4.LocaliseVectorRotation(Globals.RobotProps.servoTable[ser
vo_Num].BaseAxis .Vec,
Globals.RobotProps.linkTable[Globals.RobotProps.servoTable[servo_Num].LinkID]
.Global_TransformMat, Globals.RobotProps.endEffecter.Transform);

                if (ClassVerbose && LocalVerbose)
                {
                    Console.Write("\nServo {0:N}\n", servo_Num + 1);

Vector.WriteToConsole(Globals.RobotProps.servoTable[servo_Num].ForceVec);
                    Vector.WriteToConsole
(Globals.RobotProps.servoTable[servo_Num].TorqueVec);

Vector.WriteToConsole(Globals.RobotProps.linkTable[Globals.RobotProps.servoTa
ble[servo_Num].LinkID+1].TorqueOut);

Vector.WriteToConsole(Globals.RobotProps.servoTable[servo_Num].EndEffecterTor
queComponent);
                }
            }


            if (ClassVerbose && LocalVerbose)
            {
                Console.Write("\nTotal Force On End Effecter\n");
                Vector.WriteToConsole(Grip.Force);

Vector.WriteToConsole(Globals.RobotProps.linkTable[Globals.RobotProps.num_Lin
ks - 1].TorqueOut);
            }
        }
```

Servo

Local Variables:

Global_TransformMat	Transformation matrix for the servo.

Public Variables:

TorqueNormal	Unit vector in direction of force caused by a positive torque.
ForceVec	Force vector caused by torque.
Torque Vec	Vector describing torque about each axis.
EndEffecterForceComponent	Force contribution to End Effecter.
EndEffecterTorqueComponent	Torque contribution to total End Effecter torque.
AxisProjectionPoint	Point on axis closest to End Effecter.
TorquePoint	AxisProjectionPoint updated conditionally.
BaseAxis	Axis of rotation for servo in local space.
Axis	Axis of rotation in world space.
Link ID	App.Physical_Components.Link Reference.
Current	Amps motor is drawing.
TorqueTotal	Total torque about axis.
TorqueGravity	Torque created from mass of links about axis.
TorqueConstant	Mechanical constant for motor newtons per amp.
Angle	Angle of the motor.

```
private Matrix4 Global_TransformMat;

    public Vector3 TorqueNormal = new Vector3();
    public Vector3 ForceVec = new Vector3();
    public Vector3 TorqueVec = new Vector3();
    public Vector3 EndEffecterForceComponent;
    public Vector3 EndEffecterTorqueComponent;
    public Vector3 AxisProjectionPoint = new Vector3();
    public Vector3 TorquePoint = new Vector3();
```

```
        public Line BaseAxis = new Line();
        public Line Axis = new Line ();

        public int LinkID;

        public float Current;
        public float TorqueTotal;
        public float TorqueGravity;
        public float TorqueConstant;
        public float Angle;
```

Functions:

Global_TransformMat Transformation matrix for the motor.

```
public Servo(int LinkID)
        {
            this.LinkID = LinkID;
            Global_TransformMat =
Globals.RobotProps.linkTable[LinkID].Global_TransformMat;

            BaseAxis.CreateLine (Vector.Vec3 (0,0,0), Vector.Vec3 (0,0,1));

            Angle = 0;
            TorqueConstant = 100;
        }
```

```
public void SetCurrent(float c)
        {
            Current = c;
            TorqueVec.Z = c * TorqueConstant;
        }
```

```
public void ModifyCurrent (float c)
        {
            Current += c;
            TorqueVec.Z = Current * TorqueConstant;
        }
```

```
public void SetTorque(float t)
        {
            TorqueVec.Z = t;
            Current = t / TorqueConstant;
        }
```

```
public void UpdateForceVec(EndEffecter EndEffect)
        {
            // Rotate the axis to align with frame
            Axis = Matrix4.TransformLine(Global_TransformMat, BaseAxis);
```

```
            TorqueGravity =
TorqueFromForce(Globals.RobotProps.linkTable[this.LinkID].CenterOfMass_Arm_Lo
cal, Globals.RobotProps.linkTable[this.LinkID].Local_Gravity).Z;
            TorqueTotal = TorqueVec.Z + TorqueGravity;

            ForceVec = ForceFromTorque(EndEffect.Point, true);
        }
```

```
private Vector3 ForceFromTorque(Vector3 Point, bool updateGlobal)
        {
            Vector3 rtn = new Vector3();
            Vector3 AxisProjectionPoint = new Vector3();
            Vector3 TorquePoint = new Vector3();
            Vector3 TorqueNormal = new Vector3();

            float Radius;

            // Find point on the axis which is closest to the end point
            AxisProjectionPoint = Axis.ClosestPoint(Point);

            if (Vector.Distance(AxisProjectionPoint, Point) <
Globals.RobotProps.cutoff_OnAxis) // If the point is on the axis force is 0
            {
                EndEffecterForceComponent = Vector.Vec3(0, 0, 0);
                rtn = Vector.Vec3(0, 0, 0);
                return rtn;
            }
           // Find torque components
            TorqueNormal = Vector.NormalTo(Axis.GetPoint(0),
Axis.GetPoint(1), Point);

           // Scale torque vector based on radius and torque
Radius = Vector.Distance(AxisProjectionPoint ,Point);

            if (Radius >= Globals.RobotProps.cutoff_TorqueRadius)
            {
                EndEffecterForceComponent =
Vector.ScalarProduct(TorqueNormal, 1 / Radius);
                rtn = Vector.ScalarProduct(TorqueNormal, TorqueVec.Z /
Radius);
                rtn.X *= -1f;
            }
            else
            {
                rtn = Vector.Vec3(0, 0, 0);
            }

            if (updateGlobal)
            {
                this.AxisProjectionPoint = AxisProjectionPoint;
                this.TorqueNormal = TorqueNormal;
                this.TorquePoint = AxisProjectionPoint;
            }

            return rtn;
      }
```

```
private Vector3 TorqueFromForce(Vector3 Point, Vector3 Force)
        {
            Math_Classes.Line RotatedAxis = new Line (Axis.Vec);
            Vector3 rtn = new Vector3();


            rtn = Vector3.Cross(Point, Force);

            return rtn;
        }
```

```
public static void SolveTorques (Vector3 DesiredForce, Vector3 DesiredTorque)
{

            Servo[] Servos = Globals.RobotProps.servoTable;

            float[] Coefficient = new float[36];
            float[] Constants = new float[6];
            Vector6 Solution;


            for (int servo_Num = 0; servo_Num <
Globals.RobotProps.num_Servos; servo_Num++)
            {
                Coefficient[servo_Num + 0 * 6] =
Globals.RobotProps.servoTable[servo_Num].EndEffecterForceComponent.X;
                Coefficient[servo_Num + 1 * 6] =
Globals.RobotProps.servoTable[servo_Num].EndEffecterForceComponent.Y;
                Coefficient[servo_Num + 2 * 6] =
Globals.RobotProps.servoTable[servo_Num].EndEffecterForceComponent.Z;

                Coefficient[servo_Num + 3 * 6] =
Globals.RobotProps.servoTable[servo_Num].EndEffecterTorqueComponent.X;
                Coefficient[servo_Num + 4 * 6] =
Globals.RobotProps.servoTable[servo_Num].EndEffecterTorqueComponent.Y;
                Coefficient[servo_Num + 5 * 6] =
Globals.RobotProps.servoTable[servo_Num].EndEffecterTorqueComponent.Z;
            }

            Constants[0] = DesiredForce.X;
            Constants[1] = DesiredForce.Y;
            Constants[2] = DesiredForce.Z;
            Constants[3] = DesiredTorque.X;
            Constants[4] = DesiredTorque.Y;
            Constants[5] = DesiredTorque.Z;


            Solution = Solver6DoF.Solve(Coefficient, Constants);

            for (int servo_Num = 0; servo_Num <
Globals.RobotProps.num_Servos; servo_Num++)
            {
                if (float.IsNaN(Solution.GetElement(servo_Num)) ||
float.IsInfinity (Solution.GetElement(servo_Num)))
                {
```

```
                    Globals.RobotProps.servoTable[servo_Num].SetTorque(0);
                }
                else
                {

Globals.RobotProps.servoTable[servo_Num].SetTorque(Solution.GetElement(servo_
Num));
                }
            }
        }
```

Link

The link class provides a simulation of a physical link

Public Variables:

Global_ TransformMat	Transformation matrix of the link after multiplication through chain.
Local_TransformMat	Local transformation matrix.
Local_Gravity	Force applied by gravity, in local co-ordinates.
di	D-H parameter d^i.
ai	D-H parameter a^{i-1}.
alpha	D-H parameter α.
thetaOffset	Partial D-H parameter θ: servo angle has effect aswell.
ServoID	Identifier for which servo is within the link.
meshName	Name of the mesh file representing the link.
TorqueArrows	Number of arrows to render for torque.
TorqueArrowScale	Size arrows representing direction of torque should be.
TorqueArrow1Offset	Distance along the axis for the first arrow.
TorqueArrow2Offset	Distance along the axis for the second arrow.
CenterOfMass_Link	The centre of mass of the link in local co-ords.
CenterOfMass_Arm	The centre of mass for the rest of the kinematic chain, in global space.
CenterOfMass_Arm_Local	The centre of mass for the rest of the kinematic chain, in local space.
TorqueOut	The total torque this link provides.

Mass	Mass of the link.
MassOnLink	Mass this link is supporting.

```
        public Math_Classes.Matrix4 Global_TransformMat = new
Math_Classes.Matrix4();
        public Math_Classes.Matrix4 Local_TransformMat = new
Math_Classes.Matrix4();

        public Microsoft.DirectX.Vector3 Local_Gravity = new
Microsoft.DirectX.Vector3 ();

        public float di;
        public float ai;
        public float alpha;
        public float thetaOffset;
        public int ServoID;
        public string meshName;
        public int TorqueArrows;
        public float TorqueArrowScale;
        public float TorqueArrow1Offset;
        public float TorqueArrow2Offset;

        public Vector3 CenterOfMass_Link = new Vector3();
        public Vector3 CenterOfMass_Arm;
        public Vector3 CenterOfMass_Arm_Local;

        public Vector3 TorqueOut;

        public float Mass;
        public float MassOnLink
```

Public Static Functions:

UpdateLinks	Updates internal forces for all links in the chain.
LoadLinks	Loads the parameters of links from 'Link Tabel.txt'.
ReadLine (System.IO.StreamReader)	Returns a line read from the data file, skips comments.

```
        public static  void UpdateLinks()
        {
            float MassFactor;

            Vector3 tmpVec;
            for (int link_Num = 0; link_Num < RobotProps.num_Links;
link_Num++)
            {
                // Calculate Gravity And Center of mass
```

```
                RobotProps.linkTable[link_Num].Local_Gravity =
Microsoft.DirectX.Vector3.Scale(RobotProps.gravity,
RobotProps.linkTable[link_Num].MassOnLink);
                RobotProps.linkTable[link_Num].Local_Gravity =
Math_Classes.Matrix4.LocaliseVectorRotation(RobotProps.linkTable[link_Num].Lo
cal_Gravity, RobotProps.linkTable[link_Num].Global_TransformMat);

                Vector3 tmpCenter = new Vector3 (0,0,0);
                for (int i = link_Num; i < RobotProps.num_Links; i++)
                {
                    MassFactor = RobotProps.linkTable[i].Mass /
RobotProps.linkTable[link_Num].MassOnLink;

                    tmpVec =
Math_Classes.Matrix4.RotateVector(RobotProps.linkTable[i].Global_TransformMat
, RobotProps.linkTable[i].CenterOfMass_Link);

                    tmpVec =
Math_Classes.Matrix4.TranslatePoint(RobotProps.linkTable[i].Global_TransformM
at, tmpVec);
                    tmpVec.Scale(MassFactor);
                    tmpCenter = Vector3.Add(tmpCenter, tmpVec);
                }

                // Localise center of mass
                RobotProps.linkTable[link_Num].CenterOfMass_Arm = tmpCenter;
                RobotProps.linkTable[link_Num].CenterOfMass_Arm_Local =
Vector3.Subtract(tmpCenter,
Math_Classes.Matrix4.TranslatePoint(RobotProps.linkTable[link_Num].Global_Tra
nsformMat, Math_Classes.Vector.Vec3(0, 0, 0)));
                RobotProps.linkTable[link_Num].CenterOfMass_Arm_Local =
Math_Classes.Matrix4.RotateVector(Math_Classes.Matrix4.FromD3DMatrix(Microsof
t.DirectX.Matrix.TransposeMatrix(RobotProps.linkTable[link_Num].Global_Transf
ormMat.ToD3DMatrix())),
RobotProps.linkTable[link_Num].CenterOfMass_Arm_Local);
                MassFactor = 0;

                if (link_Num >= 1)
                {
                    Vector3 TorqueOut =
Globals.RobotProps.linkTable[link_Num].TorqueOut;
                    if (RobotProps.linkTable[link_Num].ServoID != -1)
                    {
                        Globals.RobotProps.linkTable[link_Num].TorqueOut =

Vector3.Add(Globals.RobotProps.servoTable[RobotProps.linkTable[link_Num].Serv
oID].TorqueVec, Globals.RobotProps.linkTable[link_Num - 1].TorqueOut);
                    }
                    else
                    {
                        Globals.RobotProps.linkTable[link_Num].TorqueOut =
                            Vector3.Add(Globals.RobotProps.linkTable[link_Num
- 1].TorqueOut, Math_Classes.Vector.Vec3 (0,0,0));
                    }

                    Globals.RobotProps.linkTable[link_Num].TorqueOut =
```

```
Math_Classes.Matrix4.LocaliseVectorRotation(Globals.RobotProps.linkTable[link
_Num].TorqueOut, Globals.RobotProps.linkTable[link_Num -
1].Global_TransformMat,
Globals.RobotProps.linkTable[link_Num].Global_TransformMat);


                }
            }
        }
```

```
        static public void LoadLinks()
        {
            System.IO.StreamReader sReader =
System.IO.File.OpenText(System.Windows.Forms.Application.StartupPath +
@"\..\..\Link Table.txt");
            String this_Line;
            String[] split_Line;

            int link_Num;
            this_Line = ReadLine (sReader);

            split_Line = this_Line.Split(Char.Parse(","));

            RobotProps.num_Links = int.Parse(split_Line[0]);
            RobotProps.num_Servos = int.Parse(split_Line[1]);

            RobotProps.linkTable = new Link[RobotProps.num_Links];
            RobotProps.servoTable = new
Physical_Components.Servo[RobotProps.num_Servos];


            for (link_Num = 0; link_Num < RobotProps.num_Links; link_Num++)
            {

                // DH Table Properties
                this_Line = ReadLine(sReader);
                split_Line = this_Line.Split(Char.Parse(","));

                RobotProps.linkTable[link_Num] = new Link ();


                RobotProps.linkTable[link_Num].ai =
float.Parse(split_Line[0]);
                RobotProps.linkTable[link_Num].di =
float.Parse(split_Line[1]);
                RobotProps.linkTable[link_Num].alpha =
float.Parse(split_Line[2]) * (float)(Math.PI / 180);
                RobotProps.linkTable[link_Num].thetaOffset =
float.Parse(split_Line[3]) * (float)(Math.PI / 180);
                RobotProps.linkTable[link_Num].ServoID =
int.Parse(split_Line[4]);

                if (RobotProps.linkTable[link_Num].ServoID != -1)
                {
```

```
RobotProps.servoTable[RobotProps.linkTable[link_Num].ServoID] = new
Servo(int.Parse(split_Line[5]));
                }


                // Graphics details
                this_Line = ReadLine(sReader);
                split_Line = this_Line.Split(Char.Parse(","));

                RobotProps.linkTable[link_Num].meshName =
split_Line[0].Trim();
                RobotProps.linkTable[link_Num].TorqueArrows =
int.Parse(split_Line[1]);
                RobotProps.linkTable[link_Num].TorqueArrowScale =
float.Parse(split_Line[2]);
                RobotProps.linkTable[link_Num].TorqueArrow1Offset =
float.Parse(split_Line[3]);
                RobotProps.linkTable[link_Num].TorqueArrow2Offset =
float.Parse(split_Line[4]);

                // Mass Properties

                this_Line = ReadLine(sReader);
                split_Line = this_Line.Split(Char.Parse(","));

                RobotProps.linkTable[link_Num].CenterOfMass_Link.X =
float.Parse(split_Line[0]);
                RobotProps.linkTable[link_Num].CenterOfMass_Link.Y =
float.Parse(split_Line[1]);
                RobotProps.linkTable[link_Num].CenterOfMass_Link.Z =
float.Parse(split_Line[2]);
                RobotProps.linkTable[link_Num].Mass =
float.Parse(split_Line[3]);

            }


            sReader.Close();

            // Non File Dependant Properties



            for (link_Num = 0; link_Num < RobotProps.num_Links ; link_Num ++)
            {
                RobotProps.linkTable[link_Num].TorqueOut =
Math_Classes.Vector.Vec3(0, 0, 0);

                // Calculate total mass values
                RobotProps.linkTable[link_Num].MassOnLink = 0;
                for (int i = link_Num; i < RobotProps.num_Links; i++)
                {
                    RobotProps.linkTable[link_Num].MassOnLink +=
RobotProps.linkTable[i].Mass;
                }
            }
```

```
            UpdateLinks();


        }
```

```
        private static string ReadLine(System.IO.StreamReader sReader)
        {
            String rtn;

            do
            {
                rtn = sReader.ReadLine();
            } while (rtn.Contains("//") || rtn.Length == 0);

            return rtn;
        }
```

EndEffecter

The EndEffecter class provides a data structure for the final point in the kinematic chain

Public Variables:

Transform	Transformation matrix relative to global space.
Point	Location of the End Effecter in global space.
Torque	Moments of torque about the point specified.
Force	Force applied at the End Effector.

```
public Matrix4 Transform = new Matrix4();
public Vector3 Point = new Vector3();
public Vector3 Torque = new Vector3();
public Vector3 Force = new Vector3();
```

Namespace App.Threads

Program

Provides the entry point for the program, stores static variables about program states

Public Static Variables:

Running	Is the program running, terminate threads if false.
Render	Render the scene if true.

```
public static bool Running = new bool();

public static bool Render = new bool();
```

Private Static Functions:

Main	Entry point for application, launch main form.

```
[MTAThread]
static void Main()
{
    Running = true;
    Render = false;
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new frmMain());
}
```

Render

Public Static Variables:

RenderThread	Thread that runs rendering sequence.

```
        static public Thread RenderThread = new Thread(CallRenders);
```

Public Static Functions:

CallRenders	Render to each frame if rendering enabled.

```
        static void CallRenders()
        {
            int TickCount =0 ;
            int Frames = 0;
            while (Program.Running)
            {


                if (Program.Render)
                {
                    Globals.Keyboard.UpdateKeys();
                    Frames ++;
                    if (Environment.TickCount - 1000 > TickCount)
                    {
                        TickCount = Environment.TickCount;
                            if (Globals.RenderOptions.World_FPS)
Console.Write("\n {0:F}", Frames);
                        Frames = 0;
                    }

                    frmMain.D3DPort[0].Render();
                    frmMain.D3DPort[1].Render();
                    frmMain.D3DPort[2].Render();
                    frmMain.D3DPort[3].Render();
                }
            }
```

Public Functions:

Render	Start the rendering sequence.

```
        public Render()
        {
            RenderThread.Start();
        }
```