Real Time Three-Dimensional Robotics Simulation.

Masters Thesis

by

Dan Lingman©
850658618

May, 1996

Department of Mathematics
Lakehead University

0-612-33409-0

Acknowledgements

This project would not have been possible without the guidance and encouragement of my thesis supervisors - Dr. X. Li and Dr. M. Meng. Thanks also go to Dr. W.S. Lu for acting as an external examiner and to Dr. L.D. Black for acting as an internal examiner. Thanks most of all to my wife, May.

Abstract

This thesis is concerned with the design and implementation of a real-time robotics simulator with three-dimensional graphics. The simulator allows for internal or external control of a number of robotic manipulators with revolute or prismatic joints. These manipulators may interact with each other and other objects in the simulated environment. All objects in the simulation will have physical properties such as mass and fragility, and can exert forces upon each other. An external program is planned to allow easy construction of models from an assortment of pre-designed pieces.

This thesis describes four areas of the simulation in detail. The first of these is the modelling scheme used to represent objects in the simulation. The second is the actual simulation design. The third is a discussion on the use of external programs to control the simulation. Last is a summary of the programming environment and how it relates to the simulation.

Table of Contents

Table of Contents

List of Figures

1. Introduction

Testing a control algorithm or studying the dynamic properties of a robotic system requires either experimental implementation or computer simulation. Experimental implementation is not always available, as in the use of robotics in space applications. When experimental implementation is possible, it might be too expensive to allow its use.

Computer simulation with animation offers an alternative that gives the feel of a physical experiment, with the benefit of low cost, easy reconfiguration, and a quick turnaround time for development of new control algorithms.

This simulation is a discrete event simulation system. By this I mean one in which a phenomenon of interest changes value or state at discrete moments of time rather than continuously with time. Discrete event simulation was chosen over continuous simulation for this project for a number of reasons. The first is that animation is inherently a discrete process. Animation is a sequence of still pictures, or frames, displayed at discrete moments of time. Continuous simulation could be used, but discrete simulation maps in a natural way to the process of rendering and displaying frames of animation. The second reason is the object-oriented nature of the simulation. Messages moving from one object to another are discrete events themselves, and fit nicely into a discrete event simulation.

An object-oriented approach was used in the design and implementation of this simulation. Object-oriented means that data in the computer is represented as a collection of objects that communicate with each other by sending messages to one another [BUDD91]. This approach was chosen over traditional structured programming for a number of reasons. First, modelling the physical components as objects within the simulation allows for easy understanding of their relationships. For example, a joint is a physical object that can move manipulator components upon receiving a signal to do so. In the simulation, the Joint object can accept a command to change angle or position.

When this message arrives, it will move the appropriate structure to simulate the same joint motion that would have taken place in the physical components.

The second reason comes from the hardware we used. The simulation hardware environment is a NeXT workstation. The NeXT was designed to work with objects. Its operating system and development environment are aimed at creating object-oriented programs.

The simulator that was constructed can model any manipulator that can be viewed as a collection of convex polyhedra and either revolute or prismatic joints. It allows for pre-programmed or interactive control of all joints within the simulation. The manipulators can interact with each other and other objects in the environment. Simulations of moderate complexity (under 200 polyhedra) can be run with frame rates of approximately 10 frames per second on the NeXT TurboStation. Simulations of greater complexity will still run, but with a lower frame rate on the NeXT TurboStation.

1.1. Thesis Organisation

Chapter 2 discusses the details of how the simulator models the physical objects that are to be simulated. The objects are made up of rigid collections of convex polyhedra, connected by joints. These joints can be either prismatic or revolute. Collision detection is performed by process of elimination using bounding spheres, then polyhedra to polyhedra intersection. Kinematics is discussed as a means of determining joint positions and tool paths.

Chapter 3 covers the process of defining the models. The simulator is designed to use simple ASCII formatted files for input. A possible translator from AutoCAD is discussed. The groundwork is laid for a custom design package for robotic system design using off the shelf parts.

Chapter 4 is an introduction to the simulator and how it can be used. Detailed instructions are given for each aspect of preparing for and executing a simulation run. The simulator allows for detailed time triggered joint motion and dynamic camera control. It also can be used to generate a sequence of scene description snapshots. These snapshots can be photorealistically rendered using RenderMan to produce an animation of the simulation run for later viewing.

Chapter 5 covers the use of external programs to control the simulator, or to prepare command sequences for it to execute. The different types of control programs (pre-programmed, interactive, and feedback based) are discussed and compared.

Chapter 6 gives an in-depth look at the internal working of the simulator. There are fourteen basic object types that make up the simulation. What each of these does, and how they communicate with each other using messages are discussed. The timed event processing that makes this a discrete event simulation is detailed, along with the sequence of events that occurs once the "Play" button has been pressed to start a simulation run.

Chapter 7 discusses the programming environment and why the NeXT was chosen as a base for the simulator. Development tools such as Project Manager and Interface Builder are given. A brief outline of the advantages of Objective-C over other languages is presented, along with sample code. RenderMan is a powerful three dimensional rendering tool, and its availability was the main reason for choosing the NeXT. NeXT also has a good collection of inter-application communication tools, which allow for easy integration of external control programs. Finally, the operating system and hardware specifications are given.

Chapter 8 is the conclusion of the entire project. The problems and results of the project are discussed. Future directions and possible enhancements are given. One of these, the Robot Construction Kit, would greatly enhance the usefulness of the simulation by allowing the user to simulate robotic systems that use readily available components.

2. Related Works

There are many other robotics simulators available, both commercially and publicly accessible. This section gives a brief overview of some of these simulators. It is by no means an exhaustive list.

Commercial Simulators

ADAMS: Mechanical Dynamics Inc. ADAMS is a general purpose dynamics simulator for UNIX. You can use this package to simulate any type of mechanism, including but not limited to robotics systems. Given a model of the system, ADAMS builds a set of equations and solves it through time. It can handle static, quasi-static, dynamic and kinematics simulations. It uses its own windowing system, and has a strong interface. It is difficult to learn, and at times requires FORTRAN programming to take full advantage of the features.

Workspace: Robotics Simulations Ltd. Workspace is an industrial robot simulator that runs on the IBM PC. It is designed to allow for the off-line programming of many different industrial and educational robots. It has a library of standard robots, and allows for interactive design of new robots. While a simulation is running, the user is able to view forces and torques generated as well as a graphical view of the system being simulated. Motion commands can be fairly complex, and the operating parameters of all mechanisms is fully specifiable.

Publicly Available Simulators

EROS: JPL, NASA. EROS (Erann's Robot Simulator) is designed to simulate mobile robots on the Macintosh. It uses a construction kit approach to building robots to be simulated. The user builds both the robot and the environment, and then programs the robot to perform tasks in this environment. It was inspired by a truck simulator by Hanks and Firby, but is designed to operate at a lower level than TruckWorld. This allows for a more realistic simulation.

Simderella: Simderella is a multipart simulator for UNIX. There is a display module, a simulation module, and a control program. The modules communicate using UNIX sockets, which allows a distributed simulation. The modular approach allows for easy upgrading and conversion. If a kinematics controller is desired, that can be used. A neural net based controller with feedback from the simulator could be substituted and the results compared. The program can also be used to simultaneously send control signals to the simulated robot and a real robot.

Other Simulators

MAGIK: NASA Johnson Space Center Automation Robotics and Simulation Division (JSC AR&SD). The Manipulator Analysis Graphic Interactive Kinematic is the primary tool used by AR&SD, Mission Operations Directorate and the International Space Station for manipulator task analysis. It allows the users to conduct kinematic analysis for robotic operations, in both pre-programmed and user controlled modes. It can handle multiple manipulators, multiple viewpoints. Simulated cameras can be inserted into the simulation space and manipulated to give a realistic point of view to the simulation. The Canadian Space Agency has chosen to use MAGIK as the base for their Operations Kinematics Simulator and to train astronauts for mobile servicing of space vehicles.

3. Three-Dimensional Modelling and Motion

A robotics system is a controlled interaction between moving objects. How the objects are represented is an important facet of the simulation. In this system, a hierarchical system is used to represent the objects being moved. Each object is broken down in the extreme into a collection of joints and convex polyhedra. Bounding spheres are used for both main items and each subitem to allow for rapid collision detection.

The motion of the objects in the simulation is time based. At regular intervals, the system updates the position of each object. As a joint may be scheduled to move over a long time period, the current time is determined, and motion proportional to the elapsed time is performed.

Each time an object is moved, it must be checked for collision with other objects. The hierarchical nature of the objects being moved makes a series of bounding spheres a natural way to quickly eliminate most of the objects involved. Any that are still suspected of collision are checked in detail.

Programming motion on a joint by joint basis is tedious and difficult at best. Because of this, the system has been designed to interface with other programs. These external programs can either be used to generate a file containing pre-programmed joint motions, or to interactively control the simulation by feeding it joint motions on the fly. In either case, the external program can get feedback from the system to determine where objects are at a given time.

3.1. Representation of Objects

There are two kinds of objects referred to in this project. The first of these is programming objects which are discussed in section 6.1. The second is abstract things that are intended to represent physical objects. We discuss the second class of objects in this section.

Objects in the simulation are made up of an ordered collection of items and joints. Each item is a rigid entity made up of subitems that are fixed in position relative to each other. Items are connected to one another by joints.

Subitems are convex polyhedra. They are represented as a collection of vertices and faces. To save space each vertex is stored only once and each face is made up of an ordered list of vertex indices. The vertices that make up a face are ordered in such a way that the outside can easily be determined.

Each subitem maintains a central point and normal vectors for each face. Both are used for collision detection. A radius of a minimum bounding sphere, centred around the central point is stored. Physical properties of the subobject, such as tensile strength, colour, and mass are included. Since the subitem is a convex polyhedron, the central point can be chosen to be the centre of mass for the subitem to simplify calculations. (If the subitem were not a convex polyhedron, the centre of mass might not be on the inside of the subitem.)

Subitems are defined relative to the origin. When loaded in, all vertices can be multiplied by a transform matrix to define the initial position and orientation of the subitem. Since this is only done at load time, the transform matrix does not need to be stored.

An item maintains a linked list of subitems. The subitems are considered to be rigidly positioned with respect to each other within the item. It also has an over all bounding sphere defined by a central point and a radius. The central point is picked to represent the centre of mass. To speed calculations, the entire mass of the object is stored here as well. A transformation matrix is stored for the entire object. This transformation matrix is used whenever the object needs to be redrawn, and for collision detection. An item also stores force vectors for each of the major axis of motion and rotation. These are used to resolve collisions.

A joint is made up of an axis defined by two vertices, a position which is relative to its original state, and pointers to all items that are attached to the joint. If the joint is prismatic, the axis defines the movement vector that the attached parts will slide along. If the joint is revolute, the axis is the axis of rotation. One of the pointers is the base for the joint. It is considered to be a fixed object when the joint moves. The remainder of the pointers indicate which items and joints will have to be updated when the joint changes position. An example of this is a wrist joint in a person. The arm would be the base, since moving the wrist does not affect the arm. The position of the hand, fingers, and knuckles will be changed if the wrist joint moves.

The entire simulation is made up of two arrays, one of joints, and the other of items. Together, these define all the physical objects that can interact in this simulation.

3.2. Types of Motion

There are two main types of motion, revolute and prismatic. Each can be described as a simple co-ordinate transform matrix.

Translation is performed differently (as an addition of vectors) from rotation (as multiplication of a vector and a matrix). If points are expressed as homogeneous coordinates, both transformations can be treated as multiplications. For a point to be converted to homogenous coordinates, a fourth coordinate is added. Instead of (X,Y,Z), we now have (X,Y,Z,W). Two sets of homogeneous coordinates (X,Y,Z,W) and (X',Y',Z',W') represent the same point if and only if one is a multiple of the other. Also, at least one of the coordinates must not be zero: (0,0,0,0) is not allowed. Because we are now using four coordinates to represent a point, we must use four by four matrices for the transformations [FOLE90].

Prismatic motion is translational motion. It has no natural equivalent. It can be best described as two members sliding over one another. There is a simple matrix describing the motion:

$$\begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where dx, dy and dz give the change in position.

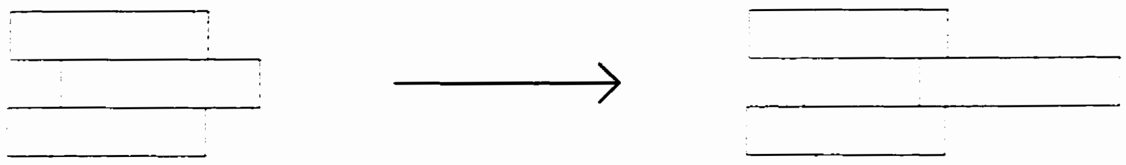

Figure 3.2-1: Example of translational motion

Revolute motion is similar to joint motion in nature. It is the revolution of a member around a pivotal axis. There are three main cases, each getting more specific, and each requiring a more complicated matrix to express it.

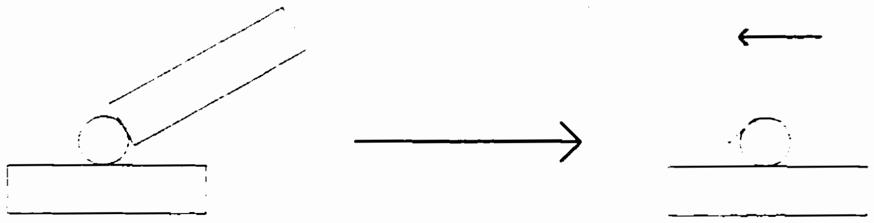

Figure 3.2-2: Example of revolute motion

The first of these is rotation around one of the three primary axis. For the X-axis case, the resulting matrix is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where ϕ is the angle of rotation. The matrices are similar if either the Y-axis or Z-axis is the axis of rotation.

The second is rotation is around a vector that passes through the origin. In this case, the procedure has four steps. First, a matrix A is determined that would bring the non-origin point to one of the major axis. This matrix is made up of the product of the two matrices needed to align the object with an axis. (The first of these, A1, will move the point into one of the planes, and the second, A2, will move the point along a plane to an axis.) Once the rotational vector has been moved into the line with an axis, the desired rotational matrix, B, can be computed. The next step is to compute the matrix C needed to move the rotational vector back into position. This is done by using a procedure similar to that for calculating the A matrix. The final step is to multiply all three matrices, A, B, and C together to form a new matrix D. This matrix has the direct transform values for rotation by ø degrees around the given vector.

The third case is for rotation around an arbitrary axis. This is again a four step procedure. The first step is to calculate a matrix A that translates one end of the axis to the origin. This is done using the procedure for prismatic joint motion. Next is to calculate the rotational matrix B around the translated axis. Since the axis now passes through the origin, we can calculate B using the above method. Thirdly, we calculate a matrix C that will reverse the translation of A using the prismatic method. The last step is to multiply the three matrices together to give a resulting transform matrix D, which will correctly rotate around an arbitrary axis [FOLE90].

In this project, matrices are available from RenderMan for translation and rotation around an origin crossing axis [UPST90]. To simplify the programming, these were used as needed, and RenderMan routines for fast matrix multiplication were used. These matrix operations are all heavily optimised to take advantage of the DSP chip in the NeXT. [NEXT92]

3.3. Collision Detection

When a joint moves, a number of items will move. Each joint has a list of items that must be moved when the joint changes position. These items must be checked against all other items to determine if collision has taken place. Since the items that are moving can be considered to be fixed with respect to each other, there is no need to check to see if a collision occurred between any two moving items. As an example, consider a person moving one arm at the shoulder but keeping all the other joints in the arm in the same position. The hand may collide with the body, but it will never collide with any portion of the arm that is moving.

Rapid collision detection is a major consideration. In a simulation that may have dozens of items and hundreds of subitems, it is impractical to check each subitem against each subitem. A better method has to be used if performance is not to drop to the level of being useless.

The first level of checking is a bounding sphere intersection check between items. This can be done quickly, and if the spheres do not intersect, then an entire item (made up of a number of subitems) can be eliminated as a possible collision victim. Figure 3.3-1 shows a planar example of this.

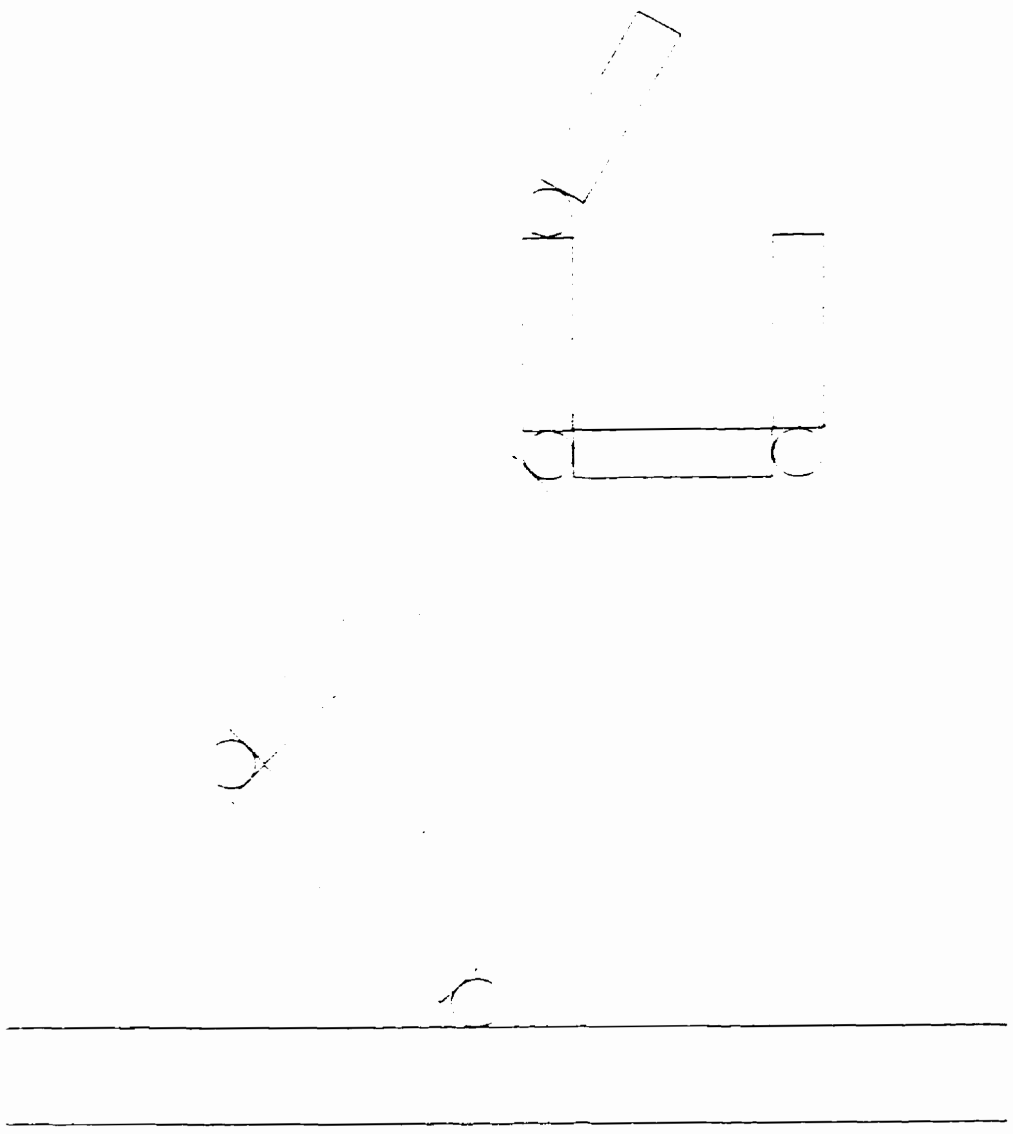

Figure 3.3-1 Planar robot with bounding circles

Any items that are found to have bounding sphere overlap are then checked further. Each subitem has its own bounding sphere. These are checked against the other items sphere to quickly eliminate any subitems that could not possibly have collided with any of the other item's subitems. This results in two lists, one for each item involved, of subitems with bounding spheres that overlap the other items bounding sphere.

These two lists are then compared to each other. Bounding spheres for each pair of subitems are checked for overlap. If this occurs, then the actual subitems must be checked for collision using geometric methods.

Since each subitem is a convex polyhedron, if the objects have collided then at least two of the faces are intersecting. One of the objects is chosen, and each face is compared with all the faces of the other object. An intersection line is computed for the two planes, and checked to see if this line falls within the polygons making up the two faces. If it does, then we have a collision, and the result bubbles back up through the various levels. If not, the comparisons continue.

Once a collision has been detected, something must be done about it. There are two main categories of objects - those that are free to move, and those that are not. If the object is an object that cannot be moved, the motion of the first object is reversed, and the attempted motion is placed back into the queue for a later attempt.

If the object can be moved, then it has the joint motion applied to it as well. After it is moved, it is checked for collision with any other object. If this occurs, motion is reversed for both the moved object, and the object moving it. The motion that caused the problem is then requeued for later.

It was originally intended that forces acting on each object would be stored, and periodically resolved. This proved to be to computationally intensive for this simulation, but is still under consideration for a future enhancement of the program.

3.4. Kinematics

Kinematics as applied to robotics is the modelling of a robotic manipulator using a Cartesian co-ordinate system instead of a joint positioning system. It is far easier to describe a task that a robot must complete using Cartesian co-ordinates. Examples are positions that a welding tool must be moved to, or a path that must be followed by a paint sprayer. These are related but slightly different tasks.

3.4.1 Determining Joint Positions

Given that a tool must be moved to a specific co-ordinate (X,Y,Z), what positions should the joints of the robot arm be at for this to occur? There may be a number of solutions to this, but we will focus on determining one of the solutions.

Figure 3.4.1-1 gives an example of a planar robot with two joints and two arm segments. The toolface C is to be located at X_C, Y_C. The problem is to solve for Q_1 and Q_2, the joint angles. L_1 and L_2 are the lengths of each arm segment, calculated from the centre of the joints.

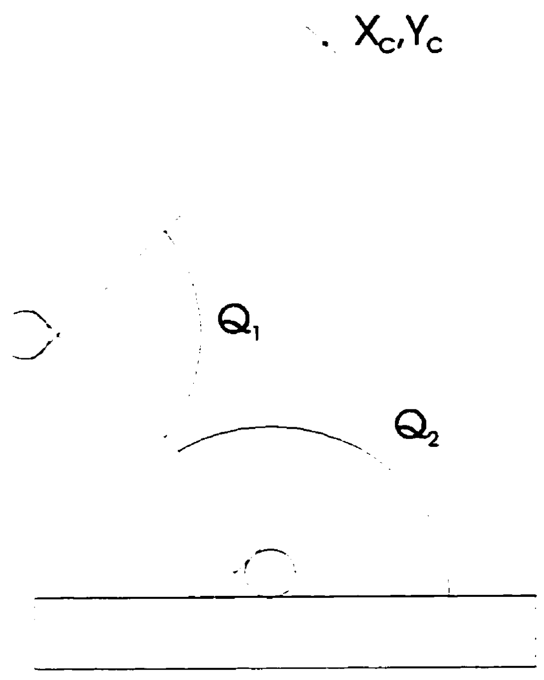

Figure 3.4.1-1 Planar robot with joints highlighted

$$X_C = L_1\cos(Q_1) + L_2\cos(Q_1 + Q_2)$$
$$Y_C = L_1\sin(Q_1) + L_2\sin(Q_1 + Q_2)$$

Squaring and adding the above equations gives us:

$$X_C^2 + Y_C^2 = L_1^2 + L_2^2 + 2L_1L_2\cos(Q_2)$$

$$\cos(Q_2) = (X_C^2 + Y_C^2 - L_1^2 - L_2^2)/(2L_1L_2)$$
$$Q_2 = \cos^{-1}((X_C^2 + Y_C^2 - L_1^2 - L_2^2)/(2L_1L_2))$$

$$X_C = L_1\cos(Q_1) + L_2(\cos(Q_1)\cos(Q_2) - \sin(Q_1)\sin(Q_2))$$
$$X_C = (L_1 + L_2\cos(Q_2))\cos(Q_1) - L_2\sin(Q_2)\sin(Q_1)$$

$$Y_C = L_1\sin(Q_1) + L_2(\sin(Q_1)\cos(Q_2) + \cos(Q_1)\sin(Q_2))$$

$$Y_C = L_2\sin(Q_2)\cos(Q_1) + (L_1 + L_2\cos(Q_2))\sin(Q_1)$$

$$\begin{bmatrix} X_C \\ Y_C \end{bmatrix} = \begin{bmatrix} L_1 + L_2\cos(Q_2) & -L_2\sin(Q_2) \\ L_2\sin Q_2 & L_1 + L_2\cos(Q_2) \end{bmatrix} \begin{bmatrix} \cos(Q_1) \\ \sin(Q_1) \end{bmatrix}$$

Solving the matrix relation for $\cos(Q_1)$ and $\sin(Q_1)$ we get:

$$\begin{bmatrix} \cos(Q_1) \\ \sin(Q_1) \end{bmatrix} = \frac{1}{D} \begin{bmatrix} -(L_1 + L_2\cos(Q_2)) & L_2\sin(Q_2) \\ -L_2\sin(Q_2) & -(L_1 + L_2\cos(Q_2)) \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \end{bmatrix}$$

where D is:

$$D = L_1^2 + L_2^2 + 2L_1L_2\cos(Q_2).$$

From the above, Q_1 is given by $Q_1 = ATAN2(\sin(Q_1), \cos(Q_1))$ [MEGA93]

Three dimensional cases, and cases with a large number of joints rapidly get more complicated, and are beyond the scope of this thesis.

3.4.2. Path Generation

For an application such as painting, a path must be defined for the robot to track. Control commands may occur at given points along this path. (These might turn the paint spray on or off). There are two main methods of determining the path that should be followed. The path may be determined algorithmically or by using a teaching device.

In an algorithmic path definition, a function is defined that gives the motion of the arm. This function is usually defined as a sequence of line segments and curves that are connected together to define the overall path. This type of path is usually generated by a computer to deal with the current situation.

A teaching device is a human manipulated object that is used to simulate the task that the robot is performing. In a painting situation, the operator would actually move a spray gun to perform the task, and the resulting motion and actions would be recorded for playback by the robotic system at a later time. This allows an easy way for a complex task to be modelled. It also has the advantage that although it may appear to be using a Cartesian system, the actual joint co-ordinates can be recorded as the task is performed. In this way, the task has been broken down into a number of very small joint motions allowing easy simulation.

If a true path was generated, it is relatively simple to build a sequence of joint movements. At discrete time intervals, the position of the tool is determined. This is translated into a set of joint positions, which are compared with the current positions. The difference is calculated giving the desired motion for this time interval. Figure 3.4.2-1 gives an illustration of a path in the plane, and how a robot arm is oriented at several points along the path.

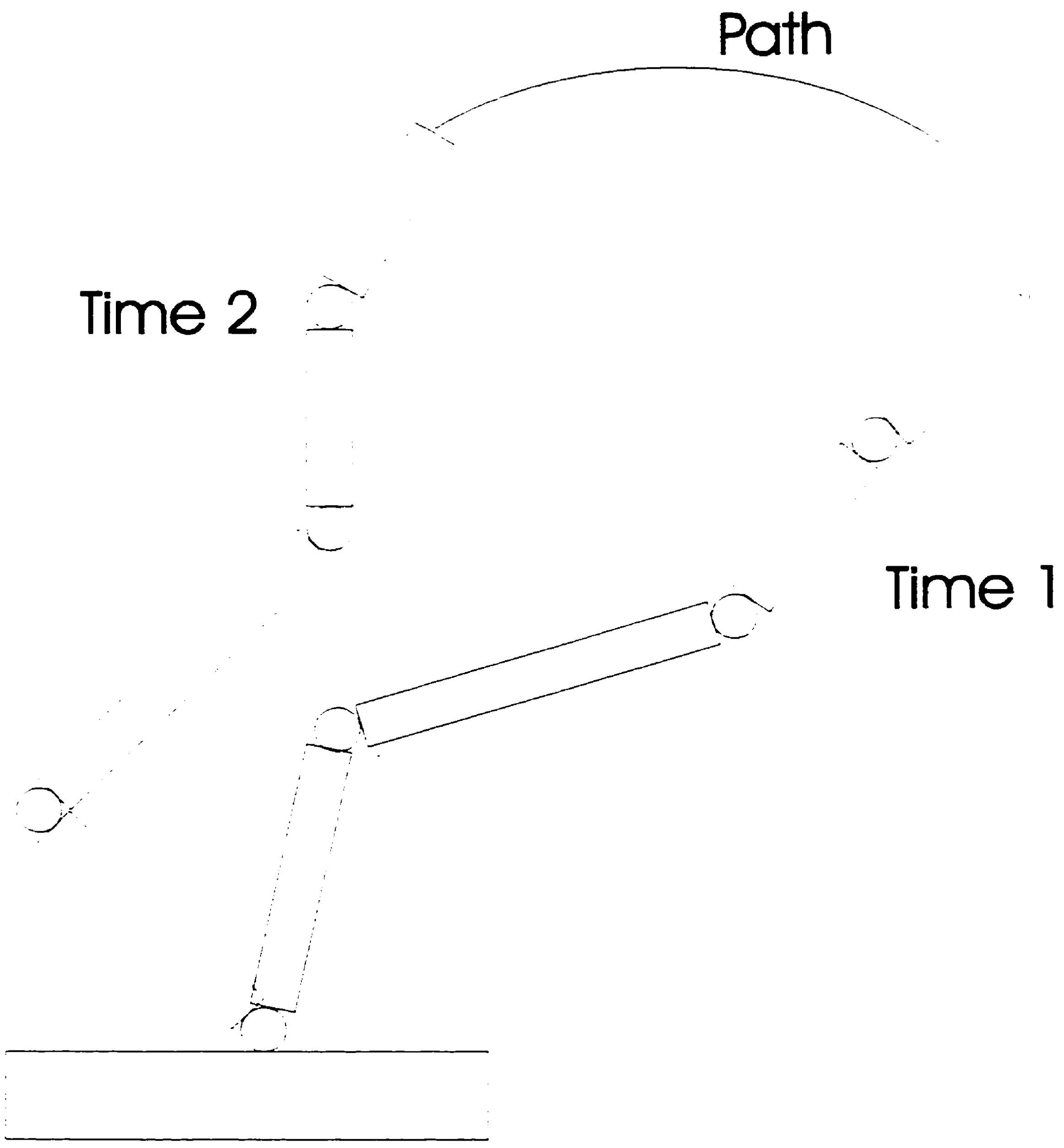

Figure 3.4.2-1 - Planar robot following a path

3.5. Summary

In this chapter, we discussed a variety of reasons for using collections of convex polyhedra as a base for modelling objects. The first was that objects can be grouped together in any way to give an approximation of any shape, convex or concave. This allows us to simulate any rigid manipulator segment. From this, we can directly get to an arbitrarily given configuration of manipulators. The second was that collision detection is much easier and faster with convex objects. Optimised computational geometry algorithms exist if it can be guaranteed that all polyhedra are convex [PREP85]. The third reason was that RenderMan is much faster at rendering collections of convex polyhedra than at rendering single complicated and possibly concave objects. RenderMan also gave the advantage of hiding most of the messy details of how motion worked. Its primitives for translation and rotation of objects before rendering meant that most of the tedious math did not have to be coded in.

A choice was made to have the simulator use joint positioning rather than toolface positioning. The first reason was that this simulator was to be completely general. By allowing the positions of the joints to be the main factor, robots with an arbitrary amount of complexity, or multiple arms could be modelled.

The simulation allows the user to test models of arbitrary complexity. Sequences of concurrent joint motion commands can be given, and the net results viewed. The simulation can be paused at any time during a run and new commands can be inserted to try out different motions. This allows for maximum flexibility and complete control over the simulation.

4. Specifying Robotic Manipulators and Other Objects

In this situation, the time taken to enter in a representation of the robot is a small amount of the overall project time. This project allows for general simulation, rather than restricting the user to a specific robotic system. Because of this, it is likely that a wide range of robotics systems will be simulated. The need for a quick way of defining a system, and reusing it later is important. Most projects will probably be completed with off the shelf components. Having a toolkit that can hold common components makes designing a system much easier.

It was this need that led to the development of the Robot Construction Kit (R.C.K.) as a quick way of plugging together pre-defined components. Engineering design software such as AutoCAD is in wide use. Because of this, a conversion utility was devised that will allow importation from this popular CAD program. Conversion is not automatic, but requires a minimum of user intervention. AutoCAD's DXF format was chosen to allow importation from a wide range of design tools, such as AutoCAD, 3D Studio, and MicroStation.

4.1. Main Simulator File Formats

There are two main files used in the simulation. These are the event file, and the robot description file. Both of these are ASCII text files, and can easily be edited.

The event file is straightforward. It is merely a list of joint motions along with the time these motions are to take place. The format is as follows:

```
Number_of_Events
Event_Number Joint_Number Action
Start_Min Start_Sec Start_CentiSec
Finish_Min Finish_Sec Finish_CentiSec
```

The number of events is an integer, and it indicates how many entries there are in this file. When it is read in, the program knows how many events to allocate room for, and it is also used for a consistency check when end of file is reached. The event number is an integer, and is used to indicate the position of this event into the list. When the entire event has been read in from disk, it is copied into the block with this number. The joint number is an integer, and indicates which joint is being affected by this event. Action is a floating point, and gives the amount of change that the joint should take. This is in degrees for revolute motion. Start and finish times refer to the time the event is supposed to begin and end. Motion will be scaled so that it completes on schedule. All time entries are integers. Min refers to minutes, sec to seconds, and centisec to 100ths of a second. There is currently a limit of 1000 event entries at any given time. This is hard coded into the program, but could be changed at a later date.

The syntactic form of the robot description file is somewhat more complicated, as it has to deal with a three-level hierarchy (Item, Subitem and vertices), each of which may have an arbitrary number of elements. It can be viewed as a multiway tree which has been traversed in preorder format. Preorder traversal visits the root of a tree, then all children of the tree processing each child completely before moving to the next child. Here is the format for it:

Number of Items	(Integer)
Number of Joints	(Integer)
(Robot item #1)	
Mobile	(1 (Yes) or 0 (No))
Red Green Blue	(Floats between 0.0 and 1.0)
Initial Transform Matrix	(4 x 4 matrix of floats - this gives the starting transform for the object.)
Bound Radius	(Float - Radius of item bound sphere)
Bound X	(Float - X Co-ordinate of bound sphere)
Bound Y	(Float - Y Co-ordinate of bound sphere)
Bound Z	(Float - Z Co-ordinate of bound sphere)
(Robot sub-item #1)	
Number of points	(Integer)
Bound Radius	(Float - Radius of sub-item bound sphere)
Bound X	(Float - X Co-ordinate of bound sphere)
Bound Y	(Float - Y Co-ordinate of bound sphere)
Bound Z	(Float - Z Co-ordinate of bound sphere)
(Point #1)	
X Y Z	(Floats - co-ordinates of point)
(Point #2...)	
Number of Polygons	(Integer - number of faces of sub-item)
#1 #2...	(Integers - number of vertices in each polygon)
(Polygon #1)	

#1 #2...	(Integers - list of vertices that make up this polygon)
(Polygon #2...)	
(Robot sub-item #2...)	
(Robot item #2...)	
(Joint #1)	
Joint_Type	(Integer - 1 = Revolute, 2 = Prismatic)
End1_X	(Float - X Co-ordinate of end one of joint)
End1_Y	(Float - Y Co-ordinate of end one of joint)
End1_Z	(Float - Z Co-ordinate of end one of joint)
End2_X	(Float - X Co-ordinate of end two of joint)
End2_Y	(Float - Y Co-ordinate of end two of joint)
End2_Z	(Float - Z Co-ordinate of end two of joint)
Minimum	(Float - Minimum value of joint)
Maximum	(Float - Maximum value of joint) (If Min and Max both equal -1.0, then the joint can rotate freely)
Current	(Float - Current joint position)
Base Item	(Integer - This is what the joint is attached to)
Number of Attached Items	(Integer - how many items have to be moved if the joint position changes)
Attached Items	(List of Integers - These are the actual items that have to be moved when the joint position changes)
(Joint #2...)	

4.2. Creating the Simulator Data Files

The current system uses ASCII files that allow the user to manually enter in co-ordinate form. This is slow, and does not lead to easy re-use of items that have been defined. If a commercially available robot arm is to be simulated, the user will have to estimate the co-ordinates that make up the arm, enter them in, and run the simulator to see if this matches reality. This leads to errors, frustration, and to the user only running simple simulations on the system. There are several approaches to this problem.

While it is possible to enter object definitions manually, this soon becomes a very tedious and error prone task. As well, modern CAD systems are commonly used to represent items. It would be convenient to be able to use these tools to define a robotics system. This led to the design of the AutoCAD conversion program. A conversion from AutoCAD was chosen because AutoCAD is popular and widely compatible with a large number of other CAD tools [GESN93]. Because this program was left for future work, it is discussed in chapter 9.

The Robotics Construction Kit (R.C.K.) was planned to provide an integrated solution for this problem. It would have a palette of popular robotics pieces that the user could combine together using a GUI to build the scene piece by piece. This gives instant feedback, and solves any problems of inconsistent data. Pieces can be scaled, rotated, and attached to one another. Scenes can be saved, and then loaded in to act as a whole like any other piece. This allows the definition of complex objects that can be re-used. When a scene is complete, a menu option would be used to save the scene in the main simulator file format.

4.3. Summary

A simple ASCII file format was chosen over other ways of representing objects and command data. Since no software was available to define the structures, a text editor was the easiest way to generate input files for testing.

Later, the idea of importing data from CAD programs arose, and it made sense to remain with ASCII, since most programs can export their data to a text file. A translator might need to be built, but it is far easier to parse simple text files.

Not everyone has easy access to a CAD package, or wants to design their robots completely from scratch, so something else was needed. The Robot Construction Kit was designed to have a library of commercially available robotic subsystems that could be joined together. This would allow the user to test designs before purchases of expensive equipment were made. The system would also be flexible enough to build user designed subsystems for later reuse out of basic geometric building blocks. Due to time constraints, the Robot Construction Kit has been left as future work.

5. The Robotics Simulator

The robotics simulator allows the user to simulate the actions of a number of robotic arms. The arms can be controlled by a pre-determined set of joint motions, or by external program control. As the arms move around, they can interact with other objects that have been placed into the simulation. The viewpoint of the simulation can be changed at any time during the simulation. A simulation run can be saved as a set of commands to repeat the simulation later, or as a series of snapshots for playback like a movie. Individual screen shots can also be taken from any viewing point.

5.1 Main Screen and Menus

The simulator is made up of a menu area, and a main screen.

The menu is a standard pulldown menu. Its options are:

Load Scene: This allows the user to load in a scene description.

Load Events: This loads in a saved list of joint motion commands.

Save Events: This saves the current list of joint motion commands for later replay.

Display WireFrame: This changes the display mode to WireFrame. In this mode, the simulation runs at a much higher frame rate, but the display quality is lower. This is best for initial testing of simulation parameters

Display Solid: This changes the display mode to solid modelling. This is slower than WireFrame, but allows a more realistic view of the scene. This is commonly used when the simulation is paused to allow a detailed snapshot to be taken.

Take Snapshot: This produces a RIB file for later display or printing via RenderMan.

Start Capturing: This starts the simulation capturing a sequence of movie frames. Each time the display is redrawn, the clock is paused, and a second copy of the display is dumped to a file. This allows the user to capture the entire simulation run for playback with an external program.

Stop Capturing: This stops the program from capturing display frames.

Quit. This is used to quit the application.

The main screen is shown in figure 5.1-1.

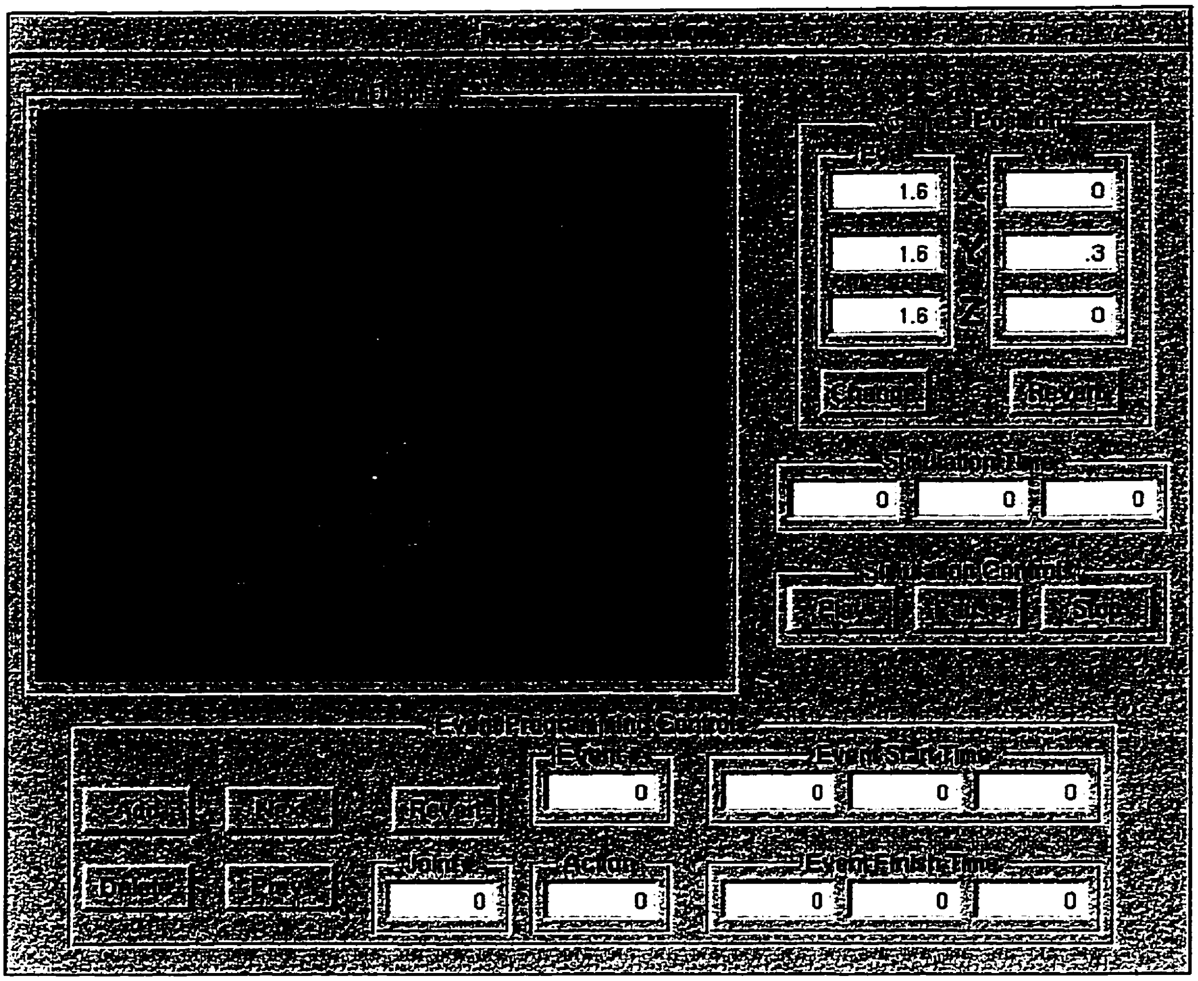

Figure 5.1-1 Main display

The main screen is broken into a number of areas. These are:

The main display area. This is the large box in the upper left hand side where the actual simulation is displayed. Both wire frame and solid modelling (shown above) are supported.

The camera positioning area. This is in the upper right hand corner of the main screen. From there, the viewpoint and viewing direction may be adjusted. Zooming in or out is possible if either the eye or view co-ordinates are scaled.

The simulation control panel is just below the camera positioning area. The buttons in here are similar in operation to those on a compact disc player.

The simulation clock is below the simulation control panel. As the simulation progresses, it advances. This is used to determine when a joint command should be programmed to start.

At the bottom of the screen is the joint event entry area. When the simulation is being run in a pre-programmed mode, the joint motion commands can be edited using this area.

5.2. Using the Simulator

When the simulation is started up, the first thing the user will do is to load in a simulation file. When this is done, the initial positions of the objects in the scene are displayed. Events can either be loaded from a file, or entered in using the event entry section of the screen.

Next is camera positioning. This can be adjusted to allow the user to view the portion of the scene that is of interest. This can also be changed during the simulation, although pausing the simulation first is suggested to allow for accurate viewing.

Once all of this is completed, the simulation is ready to run. There are three main buttons that operate the simulation. These are labelled Play, Pause, and Stop. The buttons work in a fashion that is familiar to anyone who has used a compact disc player.

The Play button starts the simulation running. Events are executed resulting in joint motion. An on-screen clock keeps pace with the events as they execute.

The Pause button freezes the action. It allows the user to reposition the camera, produce a snapshot of the current scene, or enter additional events into the event list. Pressing Play will start the motion again.

The Stop button stops the simulation, and resets everything back to the initial conditions. The scene is viewed as if it had just been loaded, and all events are restored in the event list. If events have been entered satisfactorily, the user can save them to a file for later playback. At this point, the user can reload the scene or event lists, edit the event list, re-run the simulation, or quit the application.

5.2.1. Event Entry

Event entry is done in two ways. The first of these is through external program control, which will be discussed in a later section. The second, which is talked about here, is by using the Event Programming Control Panel. This takes up the bottom portion of the main simulator screen (See Figure 5.2.1-1). The control panel allows manual entry of joint level commands into an event list.

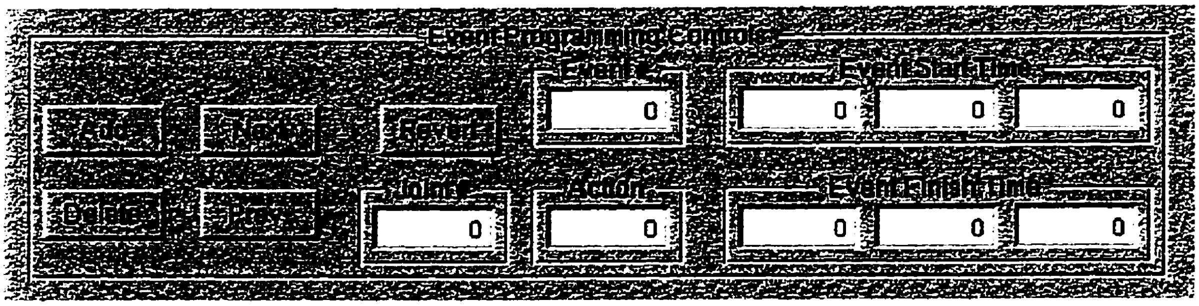

Figure 5.2.1-1 - Event entry area

An event list can be loaded from disk, or saved to disk by using the event submenu. Selecting "Open..." will produce a file browser that will let the user select an event file to be loaded. Selecting "Save..." will allow the user to specify a directory and file to save the current event list to.

There are nine text entry areas and five buttons making up this control panel. The text entry areas are broken into three main sections: Action, Start Time, and Finish Time.

The Action area contains the Joint Number, Event Number, and Action. The Joint Number holds an integer value, which refers to the absolute joint number being moved in the simulation. Joint numbering starts at zero, and is highly dependent on the input data files. The Event Number is not normally entered by the user. It is generated automatically by the addition of new events, or by moving forward or backward in the existing list. The user can enter a value here, which will then display the event records that match that event number. Action is a floating point number that is interpreted based on the action to be taken. If the joint in question is a revolute joint, then it refers to the amount of rotation to be applied to the joint in degrees. If the joint is a prismatic joint, then this gives the distance of translation in standard units.

The Start Time and Finish Time areas are almost identical. Each holds three integer values. These are minutes, seconds, and hundredths of a second. Start Time refers to the beginning time for the event, and Finish Time to the ending time of the event. Combined with the amount of the Action variable, they determine the rate of motion for this joint. If this exceeds the parameters for the joint, a warning will appear, and the finish time will be scaled to the minimum time for the motion to be completed.

The first of the buttons is labelled "Add". It checks the currently entered event record for validity. If everything is valid, it adds the record to the master event list, then moves to a blank record at the end of the list to wait for more input.

The second button is labelled "Delete". This will delete the currently displayed event record from the event list. All the event records that were after this one will have their event numbers decreased by one. The next event record (or the previous one is this had been the last) will be displayed on the screen.

The third and fourth buttons are labelled "Next" and "Prev.". They will move to the next or previous record in the event list after saving the current event.

The last button is labelled "Revert". It will overwrite any changes that have been made to this event record with the values that are currently stored in the event record.

5.2.2. Camera Positioning

Camera positioning is done using the Camera Position Control Panel, situated in the top right hand corner of the main simulator screen (See Figure 5.2.2-1). This control panel allows the user to modify the location and viewpoint of the camera at any time. It is suggested that the simulation be paused before the camera is moved. This will allow the user to select the best possible viewing position.

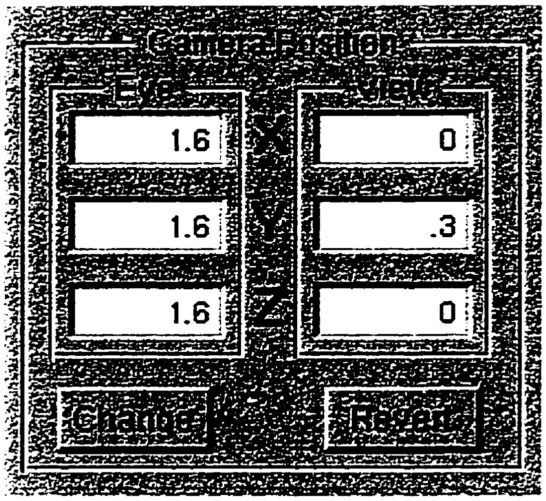

Figure 5.2.2-1 - Camera position area

The Camera Position Control Panel consists of six text entry boxes and two buttons. Any floating point number may be entered into the text entry areas. The first set of these buttons is labelled "Eye", and gives the co-ordinates of the actual viewing position. The second set is labelled "View", and gives the direction of viewing. They combine to give a camera direction vector. How large the image is depends only on the Eye position relative to the objects being viewed. The View position is only used to give a viewing vector.

The first button, labelled "Change", copies the values from the Eye and View positions into main memory, and updates the simulation viewing window. The second button, labelled "Revert", overwrites the text entry areas with the values already stored. It can be used in case of a data entry error.

Hitting the return key in any of the text entry areas moves the cursor to the next area. When the Z position of the Eye has been entered the cursor will move to the X position of View. If return is pressed when the cursor is in the Z position of the View, it has the same effect as if the Change button had been pushed. This allows rapid entry of camera position values.

5.2.3 Running a Simulation

The simulation control area is used to run the actual simulation. There are three buttons in this area; Play, Pause and Stop. These work in a similar fashion to those on a compact disc player. This model was chosen due to its familiarity. Figure 5.2.3-1 shows the simulation control area.

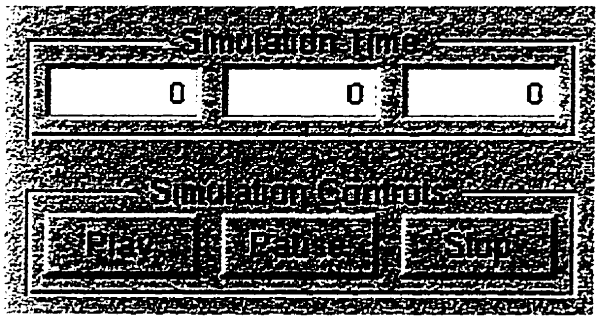

Figure 5.2.3-1 - Simulation control area

Pressing the Play button will do one of two things. If the simulation has been paused, execution of the simulation will take off from where it had been paused. If this is the first time the simulation has been run since being loaded in, or if the Stop button had been used to halt execution of the simulation then this will start a new simulation run. The clock will be reset to zero. All objects in the simulation will return to their initial positions. Once things have been reset, the joint commands will start to execute.

Pressing the Pause button will freeze the simulation. At this point, the clock will halt, and all motion will stop. This is commonly used to get a single frame snapshot of the simulation, add new events to the event list, or to adjust the camera position. (Note that the camera can be adjusted while the simulation is running, but it is usually simpler to determine the correct position when things are not moving.)

Pressing the Stop button will also freeze the simulation, but will also indicate that this run is over. The scene does not reset in case the user wants to capture the current display. Like pressing the stop button on a compact disc player, this will lose the current position in the simulation, and the simulation must be run from the beginning again.

5.2.4 Saving a Simulation Run

The Save and Load Event menu items allow the user to keep a set of joint commands for later playback. When the Save Events menu option is selected, the user is prompted for a filename and directory using a standard NeXTStep save dialog to save the current event list under. When Load Events is selected, the user is given a file browser to select an event file to be loaded. This will replace the current event list.

5.2.5. Capturing Frames in High Resolution

Individual frames of the simulation can be saved for later display. The frames are saved in RenderMan's RIB format. This allows the viewing angle to be manipulated using an external program, or for the scene to be re-rendered at high resolution. This permits high quality printed output of individual scenes from any viewing position.

The best time to take a snapshot is when the simulation is paused. This allows the user to position the camera to obtain the desired viewpoint. When the camera is properly positioned, the Take Snapshot menu option is selected. This will bring up a standard NeXTStep save dialog that allows the user to chose a directory and filename to save this screen shot in.

The RenderMan manual gives detailed instructions on how to display a RIB file. The utility RIBViewer provides a simple interface for displaying a single RIB file.

5.2.6. Generating a Sequence of Frames

It is possible to save a complete simulation run as a sequence of RIB files that can be rapidly displayed by an external program. Once you are satisfied with how a simulation run looks, stop the simulation. Select Sequence of Frames from the main menu. The file browser will allow you to choose a directory and a base filename for the RIB files to be stored under. The next time that you press the Play button, as the simulation runs, as each frame is displayed, a copy will also be made to a new RIB file. When you press the Stop button to end the run, a message will be displayed to confirm to you that the entire simulation run has been saved.

5.3. Real Time Simulation

This simulation is running in real time. By this I mean that one second of real clock time corresponds to one second of simulated time. If the robot being simulated would take thirty seconds to complete an action, thirty seconds will elapse before that action has finished. The alternative is a simulator that either takes less time or more time to run than the system that is being simulated. Extreme examples would be a simulation of the solar system, or the motion of electrons around an atom. There are advantages and disadvantages to having a simulator that runs in real time. These are discussed below.

One disadvantage of a real time simulator is that it is impractical to simulate long tasks. A user wishing to view the ending sequence of a series of commands must wait for the previous commands to run. This can be partially bypassed by saving the simulation state at points along the run, and using the current state of the robot as a starting position. With a system that has a variable time control, it is possible to fast forward through certain sections of the command sequence, and slow things down for detailed examination at other times.

The limitation of system complexity due to finite computer speed is the main disadvantage. A system that can simulate 200 polygons at 10 frames per second will be reduced to a jerky display when given 20,000 polygons. This simulator attempts to run as fast as possible on whatever hardware it is given. It will adapt the discrete time steps to adjust to the system load based on real elapsed time. Given faster and faster computers to run on, each step will be shorter, and the display cycle will show an increasingly smoother display. With a non-real time simulation, the time steps can be rigidly controlled, and set to whatever time scale is desired [BURG89].

The primary advantage of a real time simulation is that the user gets immediate feedback, and interaction is possible. It is possible to link the simulation to a real robot, and combine simulated and real position data and commands on the main display. The user could use a teaching wand to control the simulation directly, and see the effects of their actions as they give new commands to the robot. New joint motions can be entered in, and their effect can be seen immediately.

In all of these cases, the key is immediate control and feedback. The simulation becomes far more meaningful if the user can do something and see the effect. A non-real time system might give far more accurate data, but a real time system is more intuitive to use.

5.4 Observations

The simulator software allows for a number of different observations to be made. The user can obtain individual snapshots in time, or a movie like sequence of frames for later playback. The effect of different manipulator configurations and different control sequences is far easier to see than in a real robotics lab. Experimental conditions such as space or underwater are possible, where in a real lab it may be either impossible, or extremely expensive to run a real robot.

The ability to freeze time allows for detailed examination of how the simulation is progressing. Combined with the ability to edit control sequences that are currently running, this gives the user a degree of control over the simulation that would be impossible with a real robotics system. Each point in time can be viewed from any number of viewing positions.

An individual snapshot gives a precise view of the simulation at a given point in time, but full understanding comes with watching the simulation progress. By saving the simulation run as a sequence of RIB files, it is possible to build a three dimensional "movie". Because the data is still stored in a three dimensional format, it is possible to view the scene from different points. The playback of the simulation can be repeated to illustrate points about the motion of manipulators.

It is simple to adjust the manipulator configuration and control sequences. In a real laboratory, equipment might not be available or might be difficult to reconfigure. This simulation allows different designs to be tested in an economical and efficient manner. The experimenter can quickly determine if a sequence of commands will perform the desired task without fear of damaging the manipulator or the objects being manipulated. If the commands do not execute as planned, it is simple to adjust them to achieve the desired results.

Having access to a zero gravity laboratory is something that is beyond the reach of most robotics centres. Underwater conditions may be difficult to achieve as well in the real world. Much of the robotics equipment that is currently available is not designed to work in an underwater environment, so the effects of water drag can not be seen in a lab. The simulator allows "access" to both of these environments, with no added cost and no concerns about machine durability under adverse conditions.

5.5 Summary

The layout and elements of the simulator screen and menus came about gradually. Interface Builder allowed the screen elements to be moved around, added, or removed as the program grew. The original design had only the main display area and the simulator control panel with just a play and a stop button. From the similarity to a Compact Disc player, a pause button was added. Event entry was next to be added, and the format changed several times, until it arrived at the current layout. This format was chosen to allow precise control of all joints in the simulation.

The use of RenderMan as a modelling base led to experimentation to determine lighting conditions and an ideal camera position. Lights placed at each corner allow for uniform lighting of the simulation, but the camera position was difficult to decide on. Eventually, the camera position was left up to the user via the camera control section. This had the added benefit of allowing the user to pause the simulation, and reposition the camera to allow a better view of what is going on during the simulation run.

A real-time simulation offers immediate feedback and allows interactivity between the user and the simulation. It gives a very intuitive feel for what is happening in the system as variables change. A non-real-time system can be used to model events with timescales that are not usefully modellable in real time. System speed is not a limiting factor for non-real-time simulations because they can spend any amount of time that is needed to simulate the effect of one time step.

The simulator allows the user to make a number of useful observations about the manipulators and objects they are simulating. Conditions and equipment that are out of reach of the user can be easily simulated, expanding the range of possible research.

6. External Control of the Simulator

The simulator as it stands forces the user to decide on each joint movement, and enter each one individually. For a complicated system of manipulators, or a complex set of movements, this is difficult at best. External control programs allow for easier control of the simulation. They allow advances in robot programming techniques, as well as updated situations to be easily programmed. There are a number of high quality robotic programming languages available. Examples would be the Cambridge University Robot Language (CURL), the University of Chicago's RAP System and Unimation's VAL 11. Using the ability of the simulator to accept external control, we can utilise these programming languages to make the simulator a more useful tool.

6.1. Types of External Control Programs

There are two main classes of external control programs. The first of these is predetermined control. In this case an external control program is used to build a sequence of joint movements that will complete the task needed. This is then read in by the simulation, and a run can be made. The second class is real-time control. By real-time we mean that the joint movements are being determined at the time that the simulation is being run. These commands could be coming from direct user commands, an external program, input from an external device, or a combination of any of these three. It allows for two directional data flow, and a direct link must be made between the simulation and the controller. This is useful for hands on simulation, or feedback controlled simulations.

Predetermined, or pre-programmed control is mainly used for repetitive jobs, such as assembly line work. It gives the user a guarantee that each run will have the same results. The joint motion can either be manually calculated or be determined by a kinematics program that will produce a command file. In either case, an external file is created, then loaded into the simulator for a run. Based on the results of this, the command file may be modified to get slightly different results.

Real-time programs can be broken down into two main categories. These are user controlled, and feedback controlled.

User control means that the user of the software has an interface that is allowing him to decide where to position the arm using interactive controls while the simulation is being run. The results of this could be stored to create a command file, for later pre-programmed operation. This mode is used commonly to "teach" industrial robots how to perform routine tasks, such as painting.

Feedback control is more complicated. In this case, there is an actual robot that is interacting with the program as it runs. The control program has a programmed set of goals, and is trying to simulate them, and control a real robot at the same time. With this type of program, actual position data from the robot would be used to correct the simulation representation whenever the simulation deviates from the real robot. This could also be used to enhance user control of a simulation.

6.2. Pre-programmed Kinematics

Pre-programmed kinematics allow for an algorithmic breakdown of a task into a sequence of joint motions needed to carry out the task. This can result in an input file that can be run automatically by the system, and then used for control of a real robotics system at a later date.

A path is created for the toolface to follow. This may be made up of a sequence of straight or curved lines, and may have designated speeds for the toolface for certain sections of the path.

The next step is a breakdown of the path into a set of closely sampled points along the path. This gives a set of discrete toolface positions at designated time periods.

Each of these toolface positions is then used to determine a set of joint positions that will allow that position to be reached. When choosing joint positions there can be multiple solutions to the equations. Ideally, the best solution requires minimum joint motion.

One of the problems with this type of programming is that care needs to be taken not to exceed joint parameters. Kinematics allows motions which are dynamically impractical. In some cases, a rapid flip of multiple joints might be the only way to follow a path. While the simulator permits this, if this happened in the real world we would run into problems with acceleration limits and inertia.

6.3. Interactive Control of the Simulation

Interactive control is when the actions of the user have an immediate effect on the simulation. This could be through entering a new destination point for the toolface, or by having the simulation track the users motion in some way.

This can be done in several ways. The first is through a teaching system, which is commonly used in industrial applications. Another way is to use a custom controller such as a Spaceball to move the toolface around. A control program is needed to determine the correct joint positions to follow the toolface. A third way is to have a completely software driven solution, with sliders representing the desired joint positions. The user could move the sliders, and this would cause the simulation to update the positions of the arm. In all cases, the results can be saved for later pre-programmed runs.

In all cases a program would be in place that would take the commands, either from a teaching unit, or from a control program, and convert them into a sequence of joint motion commands. These would be inserted into the event queue as if they had been entered in manually, and would affect the currently running simulation. Because of this, once a sequence has been recorded, it can be written out to a file for alter playback, or manually modified.

There are a few possible problems with interactive control. The first is that a fair bit of processor time must be used to deal with the user input. A rapid sampling rate is needed, and this will tend to steal cycles from the simulation. This can be dealt with by having an external system act as a pre-processor for the user input. The second is that it can be prone to errors. Since all user motion is being tracked, inadvertent motion, pauses and so forth, will show up in the recorded sequence of joint motions. As well, there will be a large amount of data, which will be dependent on the rate of sampling. This will result in a data set that will be difficult to edit to remove errors. The user will have to rerun the simulation, again attempting to duplicate a sequence of tasks.

6.4. Feedback Based Control

In a feedback based control system, a real robotics system is coupled with the simulated system. Data is transmitted from the actual robot to update the simulation. Data from the simulation can be used to move the real robot, resulting in a feedback loop of correction and counter correction.

This could be used to provide remote monitoring and control of a robotics system. It can also be used to simulate the addition of a new piece of hardware to an existing system at a low cost. The entire system, both real and imaginary, can be entered into the simulation. This would allow the user to try out a new piece of robotics equipment to see how compatible it would be with existing system components. Designs for add on pieces can be formulated, and flaws can be found before costly prototyping.

Joint positioning can be obtained through a number of different means, such as optical tracking, or motor positioning. In any case, the set of joints positions would be passed to the simulation to allow it to update the on screen display.

A translator would be needed to convert between the joint positioning values of the simulation and whatever method is used by the robotics system. Hardware would be needed to send control commands to the robotics system from the translator. Depending on how complicated the control and translation units are, it may be best to move them onto an separate system from the simulator, to avoid degradation of performance.

There will be synchronisation problems between the simulation and the actual robotics system. There are lags in processing of commands, and joint motion will not be instantaneous. For example, when the simulation sends a command to the robotics system to move to an angle of 30 degrees, this will take time. Whatever program is giving feedback to the simulation must be written in such a way as to compensate for this lag. If this is not done, feedback from the arm may cause the simulation to believe that a collision has taken place, and it will assume that the motion was not performed.

As the simulation progresses, periodic corrections to the simulation image of the real system must be made. This will deal with round off errors, imprecision in position readings, and the time lag problem. The simplest way to do this is to pause the robot, determine the true positions of the joints, and have the simulator apply these joint positions from the starting values.

6.5. Summary

The simulator's method of specifying individual joints motions allows for precise control of the robot arms. It also makes it very easy to interface the simulator to external control programs. This allows a great deal of flexibility in controlling the simulation dynamically.

Real-time controllers, such as a program using a Spaceball to move the toolface around, or a graphic interface will use processor cycles that could be better used to give a smoother simulation. For this reason, it would be more efficient to have the control programs running on a separate machine.

Off-line programs can be interleaved with the simulator via the pause and play buttons on the main control panel. This would allow the user to see the progression of the off-line program, and correct it if needed as errors occur.

7. Simulator Internals

The simulator can be seen as a collection of loosely coupled objects that get things done by sending messages to one another.

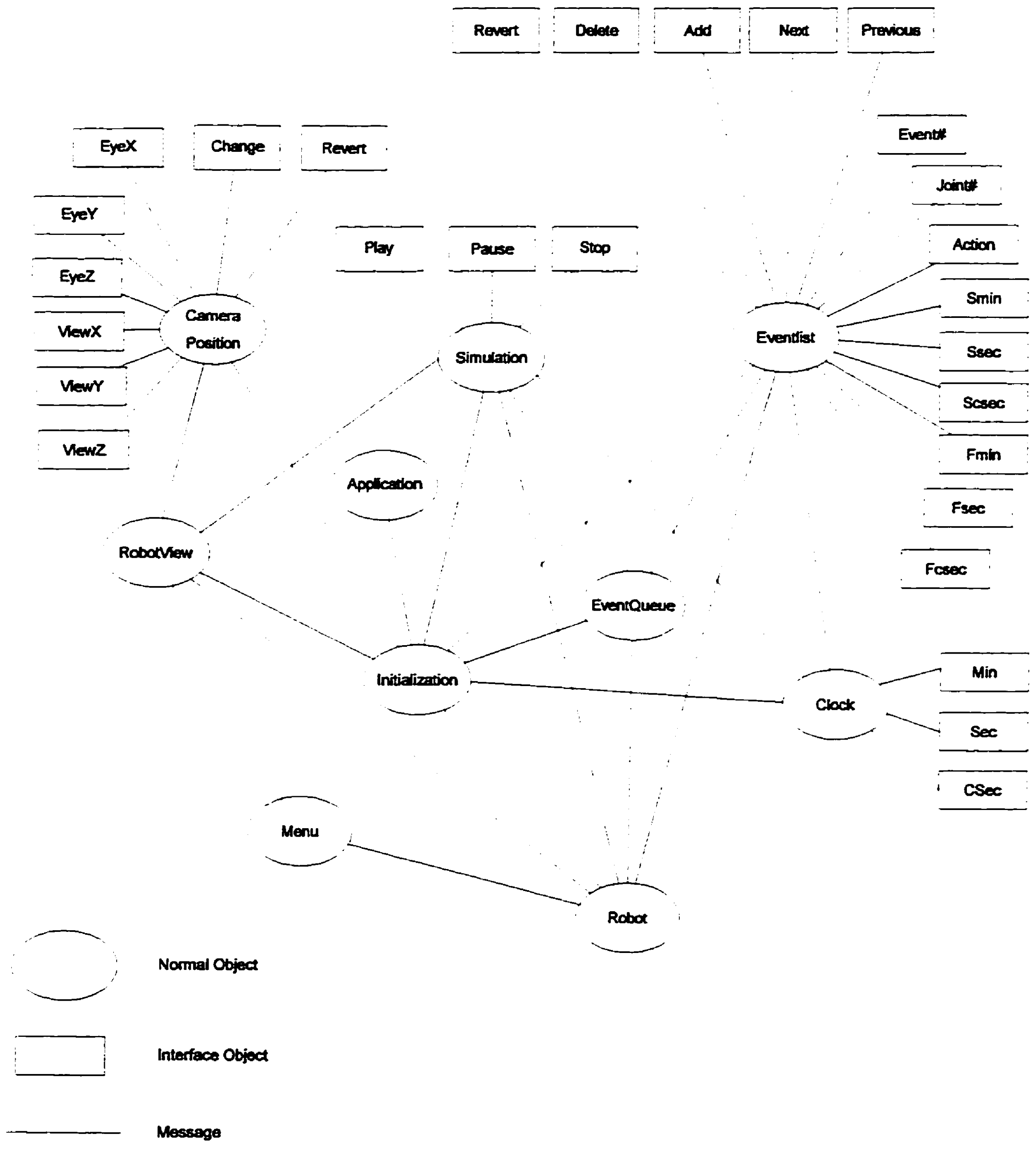

Figure 7-1. Object relationships.

7.1. Objects in the Simulation

There are a large number of objects in the simulation. What each object does, as well as how it operates, will be discussed in this section.

The objects described are: Camera Position, Robot View, Menu, Application, Initialisation, Simulation, Robot, Event Record, Event List, Event Queue, Robot Item, Robot Sub-Item, Joint, and Clock.

The Robot, Robot Item, Robot Sub-Item, and Joint objects are contained in the bubble labelled Robot in figure 7-1. This allows for an implementation independent overview of the simulation. It is possible to remove these items, and replace them with others that would directly control a real arm, and get real feed back from that arm. The main simulation code is the same.

The Application Object is created when the program begins. It is given a delegate, in this case the Initialisation object, whom it sends an init message to. There is no user-defined data stored in this object.

This is the standard way of initialising applications under NeXTStep. It can also be used to control what happens when the operating system decides that an application should be passed a document to process. An example is double clicking a document to open it. In this case, the Application object would pass parameters to its delegate to tell it a document must be opened.

The Camera Position object allows the user to control the virtual camera that is viewing the simulation. It stores the current X, Y, and Z co-ordinates for where the camera is located. It also stores the location that is being looked at in X, Y, Z format.

This object is tightly coupled with the section of the interface that controls it. It is designed to grab the co-ordinates that were entered into the screen, and pass them along to the RobotView object to be used when a screen update is performed.

The Event Record object holds information about a joint movement command. It keeps track of the joint to be moved, the start and end times of the motion, and the amount the joint is to be moved.

This object is essentially a container, and has the usual methods for accessing it's data elements. It also lets the user see if the event takes place before a given time. This is used by the Event Queue object to check if this event is scheduled for execution yet.

The Event List is where event records are stored for processing. It has controls that allow the user to browse through it, deleting or adding entries as needed. It can sort itself on a request from the simulation, in which case it will be sorted by starting time of the event records stored in it. It can also store or retrieve a set of events to disk.

The Event List can store one thousand event records in it. This is an arbitrary limit, and could easily be changed to support a larger number of events, or a dynamically changing list size.

When an Event Record is added to the Event List, it is checked for consistency against the currently loaded Robot. It will not allow the user to enter in joint motion commands for non-existent joints.

Once the user presses the Play button, a timer event periodically polls the Event List for events that have a start point after the current time. All events that are found that meet this criteria are moved from the Event List onto the Event Queue.

The Event Queue is where event records that are being executed are stored. This includes both the joint motion event records and a display event. It has the standard methods for a queue - add an event record to the queue and remove an event record from the queue.

As the simulation progresses, a timer event will send a signal to the Event Queue object. The first event on the queue is removed for processing. If it is a display event, then the Robot object is sent a signal requesting that the display be refreshed. The display event is then placed back onto the end of the queue. This allows for periodic refreshing of the display, while ensuring that all joint movement events are at least partially processed during a time slice.

If the event that was removed from the queue was a joint movement event, the clock is consulted to determine how much time has passed since the last time this joint movement was processed. The amount of joint movement is scaled to match this time that has passed, and the resulting partial joint movement is passed to the robot item to be processed. If the end time for the joint movement event has not arrived yet, the starting time is updated to the current time, and the joint movement event is requeued. This allows for processing of joint motions as rapidly as possible, with the maximum refresh rate possible.

The Initialisation Object is the delegate for the Application Object. It receives only one message, init, from the Application Object. Its role is to send initialisation messages to most of the other objects. The objects it initialises are: Camera Position, RobotView, Simulation, Robot, Event List, Event Queue, and Clock.

An Initialisation Object is a common type of object in most NeXTStep applications. It provides a simple way to initialise the program. When a data file for the application is used to start the program, the Application object will pass the relevant information to the Initialisation object acting as its delegate. This allows the application to open data files automatically.

The Joint Object handles all of the motion commands for the simulation. It consists of two points defining an axis of motion, maximum and minimum values, a current position value, the base Robot Item, and a list of attached Robot Items. The base robot item is the Robot item that will not move when the joint position changes.

When a command is received asking the joint object to change it's position, the transformation matrix needed for the amount of change is calculated using RenderMan routines. This is applied to all of the Robot Items that are attached to this joint, and all the joint objects that are involved have their endpoints updated.

The joint object then asks each of the Robot Items that moved if they hit anything. If a collision occurs, the joint attempts to apply the motion to the hit object. If this results in a collision as well, all of the motion is reversed, and the joint informs the timer event that the motion failed. This will result in the event being re-queued in the hopes that other pending motions will clear the obstruction.

The Robot Object contains all the data stored in the simulation. It has pointers to arrays of Joints and Robot Items, and the number of Joints and Robot Items. When events are added to the event list, the Robot item is queried to determine if the joint in question exists. It receives load messages from the Menu object to read a robot simulation from a file.

Another major task that the Robot Object performs is to update the RenderMan world shape when requested. To do this, it clears the current world shape, and asks each Robot Item to add its current shape to the world shape. This request is propagated down the hierarchy until the entire structure has been updated.

The Robot Object also acts as a broker for joint motion commands. It receives these from a simulation timer event, and passes them on to the appropriate Joint Object.

The Robot Item object represents a rigid collection of Robot Sub-Items. It contains the number of sub-items, an array of pointers to the sub-items, a master transformation matrix for placing the sub-items, a bounding sphere radius, and a bounding sphere centre point.

When a Robot Item moves, all that happens is that the master transformation matrix is updated. The bounding sphere for the entire Robot Item is used to speed up collision detection, since the sub-items of the Robot Item being checked for collision can be rejected if they do not fall within this bounding sphere.

The Robot Item also receives requests from the Robot Object to add its shape to the world shape. It does this by setting the global transformation to its master transformation matrix, and then asking each of its sub-items to add their polyhedra to the world shape.

The Robot Sub-Item object is used to model a convex polyhedron. This is the lowest level that is represented in the hierarchy. It is made up of the number of points, an array of points, the bounding sphere radius, the bounding sphere centre point, and a pointer to a block of memory containing the physical properties of the Sub-Item. This currently only holds the colour of the block, but could be used to hold mass, fragility etc.

When collision detection is being performed, the Robot Sub-Item will check its bounding sphere against other spheres, and if necessary, do a polyhedra intersection check against other Robot Sub-Items.

A Sub-Item will also receive requests to add their shape to the world shape. Since the proper transformations have already been added, they call a RenderMan routine to add a convex polyhedron to the world shape. The data structure that the Sub-Item uses was chosen to allow direct use by RenderMan without translation.

The Robot View Object provides the main interface to RenderMan. It can receive messages from the Menu Object to toggle between wire frame and solid modelling, or from the Camera Position Object to change the viewpoint. It also gets requests from a timer event provided by the Simulation Object to update the display. When this happens, it prepares RenderMan to display an image, and asks the Robot Object to define the RenderMan world shape. Once this is completed, Robot View calls RenderMan to display this world shape.

The Menu Object handles the menu interface for the program. Through this, the display mode (WireFrame or Solid modelling) can be chosen, files loaded or saved, or the application exited.

The Simulation Object receives messages from the main control panel (Play, Stop and Pause). When it receives a Play message, it checks to see if the simulation is currently running. If so, it ignores the message. If the simulation is stopped or paused, the Simulation object launches three timer events.

The first of these periodically sends a Display Time message to the Clock Object. This allows the running time to be constantly updated.

The second checks the event list for pending events, (which are sorted by start time) and moves any events that are ready to be executed to the event queue.

The third polls the event queue for events that are running. If it finds a display event, it sends a message to Robot View asking it to update the world shape. If a normal joint motion is found, it checks the clock to determine the distance the joint should have moved in that time, and then sends a move joint command to that joint. If there is any remaining time for that event, it is placed back onto the queue.

The Clock Object is responsible for maintaining the current time. It periodically gets display messages from the Simulation Object. When this happens, it calculates the elapsed time, and passes this to three text fields for display. It can also be polled for the current time by the event list and the event queue. It can be paused or reset by requests from the Simulation Object.

7.2. Events and Messages

Messages are a key part of object oriented programming. In this simulation, messages are passed between the objects that relate to the real world. A message is an request from one object to another object to perform some action, or to return a value. Messages in the simulation are used primarily to move items and to check for collisions.

When a movement event is removed from the main event queue, a message is sent to the joint involved to update its location. The joint will determine what transformation will perform this motion, and pass that along to all of the items that are attached to it. Each item in turn will apply that transformation to update its internal location. This allows them to be drawn correctly.

In the real world, sending a signal to a joint motor would result in the motor moving to a different position and anything that is attached to that joint would move along with it. Collisions can be dealt with sensors that determine if the motor moved the correct distance.

External programs can be used to generate messages to, or receive messages from the objects in the simulation. This can be used to allow a real robotic arm to be controlled by the simulation, and to pass feedback to the simulation about collisions with items not in its database. In these cases, movement messages would not only be sent to the simulated joint, but to the real one as well. Care would have to be taken to make sure that the resolution of both are the same. Messages coming back from the real arm would be identical to those generated by the items when they detected a collision.

There are a number of "real" events that are handled by the program. "Real" events are standard computer events, such as mouse clicks, key presses, or timer interrupts, as opposed to simulation events.

All interface events are handled automatically by Interface Builder. The interface runs in a separate thread, so rapid response to user generated events is possible.

Timer events are scheduled whenever the "Play" button has been pressed. The three timer events are really clock based interrupts that are designed to call a specified function after a certain interval has occurred. This is done with standard signal handling techniques, along with NeXTStep support for timed interrupts.

Once the "Play" button has begun, a number of things take place. Three timer events are launched. One of these updates the on-screen clock. The second moves events from the event list to the event queue when they are ready to be executed. The third removes events from the event queue, determines the proportion of the event that should execute, and sends the amount of motion to a joint for execution. If the event that was removed was a command to update the screen, a message is sent to the display asking it to refresh itself.

The on-screen clock is for display purposes only. As the simulation runs, this is updated to give the user a feel for when things are happening.

The event list is periodically polled to see if any events on it are ready to begin execution. If their start time has passed, they are copied into the event queue, which will interleave them in a way aimed at giving a smooth animation.

The event queue is also being periodically polled. Whichever event is at the head of the queue is removed, and the time that has elapsed since the start of the event is calculated. The proportion of elapsed time to total time is used to determine which fraction of the motion should be carried out. The amount of motion is multiplied by this fraction, and passed on to the indicated joint for execution. If there is any remaining time for the event, the start time is moved up to the current time, and the event is re-queued for later execution.

This allows the simulation to adjust the motion of its joints so that it appears to be smooth. If there are a lot of events happening simultaneously, then each joint will move a further distance, and if there are few events, they will move a smaller distance. This gives a net effect as if the items in the simulation are moving at a fixed speed, with a variable frame rate. On a faster machine, this provides for smoother animation. On a slower machine, or when solid modelling is used, each frame will take longer to prepare, so a lower frame rate is needed.

7.3. Summary

An object-oriented approach was chosen for several reasons. Due to the nature of Interface Builder, objects would already be present to construct the interface. It made sense to continue using objects for the rest of the simulation to give a consistent feel to the program. Secondly, objects are a very intuitive way to view a simulation. This can best be seen in this program by the way that the Robot object asks each Joint object to move to the new angle, and asks each Robot Item object if it has hit anything.

The decision to use a polling system and a queue to handle the animation was an easy one to make. This gave a fairly consistent frame rate, and allowed for correctly scaled motion as the complexity of the simulation increased, or the processor power is increased.

8. Programming Environment

The programming environment for this project greatly affected the way that it evolved. Not having to worry about details of three dimensional rendering cut the complexity of it down to a manageable size. The interface development tools transparently take care of details that are important to the program itself, but not to the actual simulation. These tools, along with the programming language support and the multi-tasking ability of UNIX [KERN84] combined to make this project a success.

8.1. Project Manager

Project Manager is basically a high-powered make-utility combined with a file manager to keep track of source code and auxiliary files. It allows the user quick access to all files associated with the project. From the main window, there are a number of options.

Run does a compile (if needed) of the current project and then launches it. This places project builder into background.

Debug does a compile (if needed) of the current project, then opens a terminal window with gdb launched in it. This allows the user to run a debugging session on the project. Gdb for the NeXT ties into the editor to provide a special control panel for the debugger which allows the user to perform many functions at a button push.

Attributes allow the user to define a number of project attributes. These include the target type (such as application), icon, main interface file, where the application should be installed, and the main language that the project is designed to use.

Files provides a browser that provides easy access to all files associated with the project. Selecting one of these files allows it to be opened up by its creating package, such as Interface Builder or Edit.

Build brings the project up to date, recompiling and re-linking as needed.

8.2. Interface Builder

Interface Builder is a powerful tool for generating user and object interfaces. The user can define the type and location of interface objects, connect them to each other (and to non-interface objects), and define actions to be taken when objects are activated. It includes a test mode where the interface can be tested without the underlying program code being in place. User interfaces can be build by selecting interface items from a customisable palette, and dragging them into a window. There is a wide range of items available, such as text input boxes, buttons, and scrolling lists.

Once interface objects have been placed, the users connects them to other objects by control-dragging a line from one object to another. Depending on the type of objects involved, the user can set up named relationships between objects, or define actions that one object can take on another. For example, a button could be linked to a calculator object, causing a method in the calculator object to be called when the button is pushed. Another example is linking the calculator object to a text field object. This would give the calculator object a reference to the text field object, allowing it to ask the text field object to change the value it is displaying.

In addition to the basic objects provided by the system, the object browser mode allows the user to define and link objects other than interface objects. These items can be quickly defined in terms of existing objects, as well as having additional outlets and actions. Outlets define relationships between this object and other objects, while actions are methods that are callable by other objects.

All these objects are stored in what is called a nib file. When the nib file is loaded, the objects are initialised, and any references to each other are resolved at that time. There is usually at least two nib files in use for each application - one for handling the information and help messages, and one for the main start-up for the application.

8.3. Objective-C

Objective-C is an extension to the C programming language. It adds the ability to define, create, and destroy objects. It also allows for messages to be sent to objects. It does this in a straightforward way that is much simpler to read and understand than C++.

Objects in Objective C have two parts - the interface and the implementation. The interface is usually placed into a header file (file.h) and the implementation into a source file (file.m). This allows the construction of libraries of binary code, while allowing others to use this library via the header files.

There is a new type, id, added by Objective-C. It is a pointer to an object. a special case of the null pointer, called nil, is defined as (id) 0.

Two special variables of type id are defined for each object. These are self and super. Self is used to refer to the actual object itself, and super is used to access any methods in the parent object that may have been overwritten. When a method returns, self is a common choice for a return value.

A new pre-processor directive is also added. Import is used in place of include to avoid problems with multiply-included files. Import keeps track of what files have been included in the current file, and will skip any that are already present.

The interface file has a number of parts. These are:

Class Definition - This is one line defining the class name and what object it is derived from.
Instance Variable Declarations - This is where the internal variables of the object are defined. These are private to the object, and can only be accessed through the method of the object.

Class Method Declarations - This defines the methods that can be used by the class itself.
Instance Method Declarations - This defines the methods that the object can use.

The implementation file has a number of parts. These are:

Class Definition - This is one line indicating which class this implementation is being defined for.
Class Method Code - This has the code executed by the Class Methods.
Instance Method Code - This has the code executed by the Instance Methods.

Messages are sent to objects to get them to execute methods. In C++ message passing appears similar to a function call, making it difficult to tell them apart. Objective-C uses a Smalltalk like protocol for passing messages to objects.

A message looks like: [receiver name1:variable1]; In this case it asks the object called receiver to execute the method called name1, passing that method the parameter variable1. There can be any number of name-variable pairs. There must be at least one name, and each variable must be separated by a name followed by a colon, or just a colon. Messages can return objects, normal variables, or nothing. They can be nested, which is why it is preferred to return the variable self.

8.4. RenderMan

RenderMan is a three-dimensional scene description program that is built by Pixar. It comes bundled with the NeXT, and is used by the simulation to provide rapid wireframe or solid modelling routines.

It is not really designed for an animation package - where RenderMan shines is as a tool for building photo-realistic images. The support that it does provide is fast enough for animation of fairly simple objects.

As a side benefit, RenderMan can be configured to write its result to a RIB file instead of rendering immediately. This allows screen shots to be captured for later viewing, or for a simulation that is too complicated to run in real time to be captured as a sequence of images that can be played back later.

RenderMan was chosen for a three dimensional display program for a number of reasons. Primarily, it saved a lot of time that would have had to be spent designing, coding and debugging three dimensional graphics routines. The fact that it came bundled with the development workstation was another big plus.

Choosing RenderMan caused some radical design changes in the main data structures that are used by the simulation. RenderMan manuals were not available in the preliminary stages of the design. This led to a design that was based on a completely different way of looking at objects. When RenderMan was introduced, the basic way of holding three dimensional objects, and of moving them had to change.

RenderMan hid most of the messy details of the matrix manipulation needed for translation and rotation in three space. This allowed more focus on the actual simulation design, and on how collision detection and resolution would work.

8.5. Inter-Application Communications

The programming environment on the NeXT provides for easy inter-application communications. Supplied objects (Listener and Speaker) allow for communications between programs, even those running on different systems.

Listeners are objects that are called remotely. The syntax for calling them is identical to that for any other object in the system. The only difference is that the calling object must be a Speaker.

A Speaker is an object that can make calls to remote objects. Combined with the ability of a Listener to receive these messages, inter-application communications are readily available for use by any program.

Support is automatically included for both programs running on the same machine, and on networked systems. The program does not need to worry about routing messages, error correction, or retransmission requests. This is handled automatically by the operating system.

8.6. NeXTStep and the NeXT Hardware Specifications

This project was implemented under NeXTStep 3.0 on a monochrome Turbo NeXTStation. At this date NeXT has discontinued its hardware line and has ported its NeXTStep environment to the Intel line of processors.

Workstation Specifications:

Turbo NeXTStation

- 33 Mhz 68040, 56001 DSP chip
- 16 Megabytes main memory
- 400 Megabyte hard drive
- CD-ROM player
- Monochrome MegaPixel Display (1024x768)
- NeXTStep 3.0 (Developers Edition)

NeXTStep is the operating system on the NeXT. It is a fully featured version of UNIX, running on top of a Mach microkernel. This provides it with multithreading capabilities. NeXTStep is an object oriented operating system, that comes with a wide variety of applications and toolkits. The main mode of operation is graphical, through the WorkSpace Manager (similar to the Macintosh Finder). The graphic display used Display Postscript as a rendering engine. A standard UNIX shell session is available inside of a window under WorkSpace Manager.

Since this is a version of UNIX, multitasking is fully supported, and NeXTStep has strong support for interapplication communications, even between applications running on different machines. The Mach kernel provides support for multithreading within processes. This is used by most programs transparently in that their user interfaces are running in one thread, and the remainder of the program is running under another. This gives a quick response time to user actions, such as mouse clicks.

8.7 Summary

The NeXT turned out to be an excellent choice for the development of the simulator. The system came with strong tools for developing a program with three dimensional graphics.

Interface Builder and Objective-C, combined with the Project Manager and RenderMan made coding and testing easier than with any other toolset I have used before. On a different platform, this project would have taken several times longer to write.

The syntax of Objective-C is much cleaner than C++. It is immediately obvious what is a message, and what is a regular function. Interface Builder helps in this distinction as it allows the user to define how objects are related to each other. It also allows for a seamless interface to be constructed with no programming needed. The interface can be tested rapidly, and things changed quickly and simply if needed.

The real star is RenderMan. It took care of a major amount of the work by providing fast rendering, both of wire frame and solids. Its support for matrix operations took care of most of the mathematical operations. RenderMan is also highly optimised for the NeXT, and takes full advantage of the DSP chip for matrix operations.

9. Conclusions and Future Work

A robotics simulator was an obvious choice for my thesis project. I have always been interested in robotics, although I never had the opportunity to build or use a "real" robotics system. I have also been playing with computer graphics on and off for several years. When given a chance to combine the two of them my decision was already made.

Originally, I viewed this mainly as an exercise in three dimensional graphics. Building a graphics engine would take a fair amount of work, and if the engine was designed correctly, the robotics simulation would practically take care of itself. This was before finding out that my platform would be a NeXT, and come complete with RenderMan to handle the graphics part of things and Interface builder to handle the user interface details. Looking back, I realise that concerns about how to implement a three dimensional engine have been covered many times before. I should have focused on the details of how the simulator would work, and what the look and feel of the interface should be.

There were some problems right from the beginning. The first and most obvious was the fact that I was in a different city from my thesis supervisors. Communication via E-mail and telephone was extremely useful, but I think I would have been able to do a better job under more direct supervision. While the tools on the NeXT are good, I ran into difficulty in getting access to people that were actually using the NeXT for programming work to ask questions. A lot of time was spent digging through the manuals to look up syntax of a call, only to discover that the documentation I needed was available only in third party manuals. Ordering these took time, and slowed things down. Objective-C is a very powerful and intuitive extension to the C programming language. I found it much easier to use than C++. It still took time to become proficient at using it and the other tools such as Interface Builder and RenderMan.

I think the hardest part of the program was the collision detection routines. RenderMan does not have optimised routines to determine if a collision between convex polyhedra has occurred. It would have made my life much easier if it did. My first attempts at this just plain did not work, through an error in how I was using the transformation matrices. Later attempts worked, but were so slow as to be almost useless. Eventually, I extended the format to include a hierarchy of bounding spheres, and elimination of as many sections of the scene as possible. This came about from reading about raytracing techniques which use similar techniques to speed up ray-object intersection checks.

Overall, I view the project as a success. I set out to develop a useful and powerful general purpose simulator, while learning more about robotics, computer graphics, and simulation in general along the way. The simulator that was developed, while not perfect, is powerful enough for most users needs. It is extendible, through either modifications to the program itself, or through the use of external control programs. It can be used to simulate any robotics system without additional programming. It runs with a good frame rate, which allows real-time simulation. It is also general enough that it can be used to simulate any robotics system, which was a major goal. I learned a lot while building this project, and I hope that it helps others to learn more about robotics systems as well. The source and object code of the simulator have been made freely available to enhance the learning process.

Tests were performed on two robots. The first of these was the simple robot seen in the picture of the main simulator display (Figure 5.1-1). This is made up of five polyhedra (the four that make up the main robot, plus one detached cube which is obscured by the robot in this view) and three joints. With this simple configuration, the simulator was able to redraw the screens in either solid or wireframe mode with all joints moving simultaneously at twenty-five frames per second.

Creating a more complex robot by hand would have been both tedious and prone to errors. The simulator does not rule out intersection of sub-items within items. To simulate a complex robot, the subitems that made up the simple robot were duplicated a number of times. Then, when a joint moved, it moved many objects instead of one. The resulting robot had forty polyhedra, and was able to produce solid mode animation at fifteen frames per second. When the display was changed to wireframe mode, twenty-four frames per second were displayed.

It should be noted that these animation rates were being produced on a 33 MHz 68040 based computer. Real-time display of several hundred solid polyhedra at twenty-five frames per second should be possible if the program were ported to a modern system. I feel that these rates of animation are sufficient for use both in teaching and research applications.

There are a number of different projects that would be nice to work on to extend the usefulness of the simulator.

The first of these is to move the simulator from the 68040 NeXTStep platform to an Intel based platform. (DOS/Windows or OS/2). NeXT has discontinued it's hardware line, and the Intel based version of NeXTStep is not widely available. Moving to an Intel platform, under a widely available operating system would greatly increase the number of potential users of the simulator. This would be a major undertaking, since the RenderMan component would need to be rewritten, and the interface converted over from the use of Interface Builder to an equivalent for the PC. Objective-C could still be used, since this is available from a number of vendors.

The second is to support printing of the main display area directly within the application. On the NeXT, this would be a relatively simple matter, since the display is already using PostScript. On a PC based system, this would be a fair bit more involved, but still within a few weeks work.

Multiple camera displays and arm mounted cameras are possible. Ideally, the command set would be expanded to allow the user to switch between cameras during the simulation in the same manner as selecting a joint motion.

Better resolution of collisions is needed. The user should be able to specify sensors on the arms (and other objects) that would indicate when a collision has occurred. This could then be fed out to an external controller for proper handling.

The Robot Construction Kit needs to be moved off of paper and into reality. Most of the design work is completed for this project, and having it available would greatly enhance the usefulness of the simulator.

The AutoCAD translator is completely designed. It would read in an AutoCAD file and parse it to build a list of sub-items. It would display these sub-items, and prompt the user to enter in an item number to join these items together. This would give the user the maximum flexibility in defining how objects relate to each other. It would also be possible to define a sub-item as a joint, and the major axis of this sub-item would determine the axis of the joint. The user would then enter in the item numbers that the joint would connect to. This method will require a fair bit of entry from the user. It may be possible to program an AutoCAD extension using Lisp that would allow the definition of valid scene files from directly within AutoCAD.

References

BUDD91 Budd T. "An Introduction to Object-Oriented Programming", Addison-Wesley, 1991.

BURG89 Burger P. and Gillies D. "Interactive Computer Graphics: Functional, Procedural and Device-Level Methods", Addison-Wesley, 1989

FOLE90 Foley J.D., van Dam A., Fiener S.K., and Hughes J.F. "Computer Graphics: Principles and Practice", Addison-Wesley, 1990.

GESN93 Gesner R., Boersma T., Coleman K., Hill D., Tobey P. "Inside AutoCAD Release 12 for Windows", New Riders, 1993.

KERN84 Kernighan B. W. and Pike R. "The UNIX Programming Environment", Prentice-Hall, 1984.

MEGA93 Megahed S.M. "Principles of Robot Modelling and Simulation", John Wiley & Sons, 1993.

NEXT92 NeXT Computers Inc. "The On-line NeXT References", NeXT Computers Inc., 1992.

PREP85 Preparata F.P. and Shamos M.I. "Computational Geometry: An Introduction", Springer-Verlag, 1985.

UPST90 Upstill S. "The RenderMan Companion", Addison-Wesley, 1990.

IMAGE EVALUATION TEST TARGET (QA-3)