CORBA BASED DISTRIBUTED REAL-TIME SIMULATION OF A CLAMING MACHINE WITH 3D VISUALIZATION

by

Zonghui Zheng

A thesis

presented to the University of Manitoba

in fulfillment of the

thesis requirement for the degree of

Master of Science

In

Mechanical & Industrial Engineering

Winnipeg, Manitoba, Canada 2001

© Zonghui Zheng

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-80101-2

Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES
*****
COPYRIGHT PERMISSION

CORBA BASED DISTRIBUTED REAL-TIME SIMULATION OF A CLAMING MACHINE WITH 3D VISUALIZATION

BY

ZONGHUI ZHENG

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

of

MASTER OF SCIENCE

ZONGHUI ZHENG © 2001

Permission has been granted to the Library of the University of Manitoba to lend or sell copies of this thesis/practicum, to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film, and to University Microfilms Inc. to publish an abstract of this thesis/practicum.

This reproduction or copy of this thesis has been made available by authority of the copyright owner solely for the purpose of private study and research, and may only be reproduced and copied as permitted by copyright laws or with express written authorization from the copyright owner.

ABSTRACT

Manitoba Hydro uses a P&H-T250 crane and an M.P. McCaffrey Inc. cable driven claming device to remove trash from the front wall of hydro dams. Due to the complexity of the task and lack of perception of the underwater working status, inexperienced operators running this machine may encounter difficulties and potentially cause damage to both the environment and the machine. To solve this problem, the concept of coordinated-motion control with graphic human-machine interfacing is proposed. Before any control strategy can be implemented, it is desirable that the system be modeled mathematically. Then, any control algorithm can be tested on the model first. It is also preferred that based on this model, new operators can be trained on a simulator before they have the opportunity to run the real machine.

This research has made the following contributions. First, a mathematical model of the claming machine was developed. Based on this model, a real-time simulation with 3D graphics and interactive features was created. Common Object Request Broker Architecture (CORBA) technology was then employed to distribute the complex dynamics calculations on a server while relieving the client computer to concentrate on graphics rendering, collision detection and control signal collection. Critical Mass Lab's simulation toolkit was also used on the client side to make the physics-based simulation more realistic and interactive. The proposed control algorithm can be developed on this platform, and novice operators can be trained on this interactive simulator.

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Nariman Sepehri for his guidance and help with this project. I would also like to thank my friends for providing the necessary distraction - the importance of which is often underestimated. I want to send particular thanks to Chung Ying Amy Chan for proofreading.

But most of all, I'd like to thank my family for being everything you wish your family to be - the perfect gift nobody else can replace.

Table of Contents

List of Figures

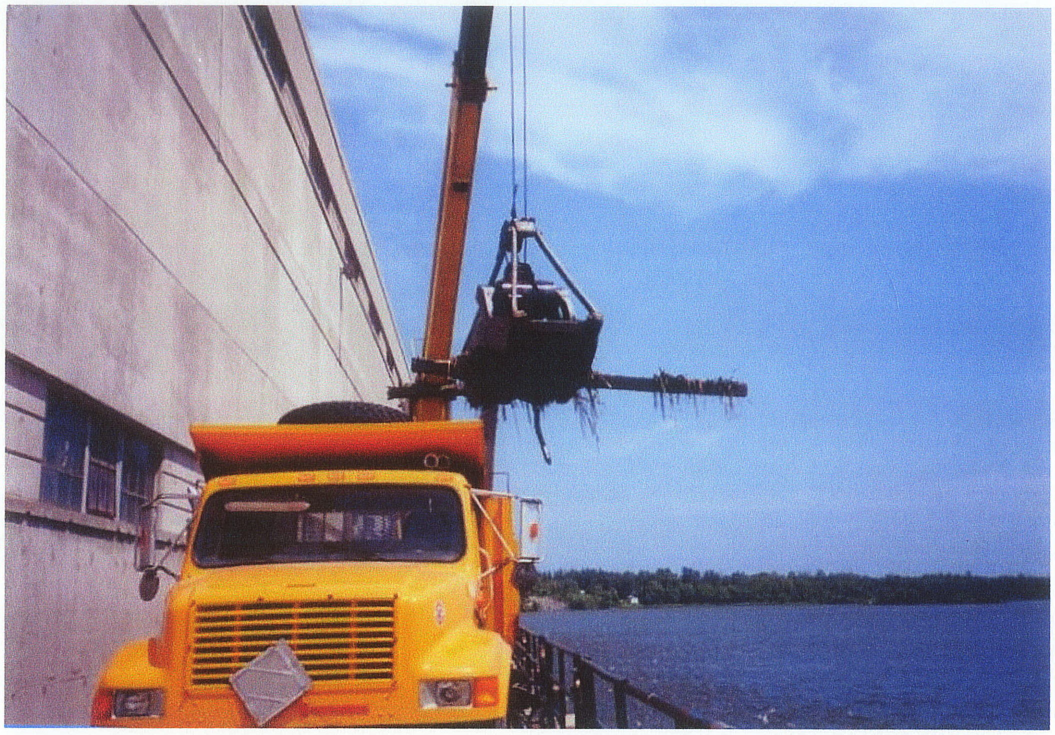
List of Tables

Chapter 1

Introduction

1.1 Motivation

Manitoba Hydro uses P&H-T250 cranes equipped with M.P. McCaffrey Inc. cable driven claming devices are used to remove trash from the front of hydro dams. The trash rack is about 5 meters wide and 20 meters high. Trash may weigh up to 25 tons. They consist of large logs and are usually jammed between the racks and/or accumulate in front of the gate, 50 feet below the water level. Currently, control of the crane and the claming device create problems. The operator manipulates the two cables, which are connected to the claming device separately. The operator lowers the claw-type claming device until it hits the bottom of the inlet gate. He/she then closes the claming device hoping to pick up something. To remove trash from the inlet gate racks, the operator first brings the claming device as close as possible to the inlet gate, then lowers the claming device while closely watching the tension in the hoists. A change in the tension may indicate collision with an object. In order to bring trash up and load it into a truck, the operator relies on his/her experience and manipulates the two cables individually, making them work cooperatively; otherwise, the cables might get jammed on the drum or one of the motors driving the cables will be overloaded. Moreover, since the operator can not see under the water, he/she can only observe the tensions on the two cables to control the machine. The working space is also quite limited. Figure 1.1 shows the crane, the claming device and their working environment.



(a)

Fig. 1.1: (a) The crane and claming device; (b) working environment.

Robotics and associated technologies have recently been employed as a partial substitute for humans in unstructured and hazardous environments. For the trash-removing tasks, a suitable computer graphics interface to enhance the operator's perception and an algorithm of coordinated motion control are desirable; these will relieve most of the stress of the operators in performing the work. It is therefore desired to develop a simulation program as a platform for further studies of control algorithms of the crane and the claming device. It is essential also that new operators be trained on a virtual machine in order to avoid possible damage due to human error. Such a simulation program must fulfill the following requirements:

(1) It must be built upon an accurate mathematical model. Since subsequent control algorithms will be developed on the virtual machine, the mathematical model of the crane and claming device should be able to produce results close to real-world responses.

(2) It should be user friendly. Control of the virtual machine should give the trainee the feeling of controlling the real machine. They will experience not only a realistic response with the virtual machine, but also a detailed virtual environment similar to the real one. The simulation software should also be interactive with the operator in training.

(3) It must be able to run in "real-time". Real-time reaction is very critical to this type of simulation program. A significant time lag will impair the training effect, since the operators will experience highly unrealistic scenarios.

(4) It should be cost effective. Although currently the price of mainframe computers or high-end workstations has been greatly reduced, they are still expensive. Employing such a simulation program to train operators has to be of low cost.

(5) The graphics interface should be elaborately designed so that it is ready to be used to facilitate the users' completion of the operations.

To meet all these requirements, especially the limited CPU capability restriction on desktop PCs, a way to utilize existing technology to solve the problem has to be found. Over the past decade, it has become clear that distributed object computing (DOC) can help to alleviate many software complexities and difficulties (Attoui, 1991). DOC represents the confluence of two major areas of software technology:

Distributed computing systems – Techniques for developing distributed systems focus on integrating multiple computers to act as a scalable computational resource.

Object-oriented (OO) design and programming – Techniques for developing the OO systems focus on reducing complexity by creating reusable frameworks and components that reify successful design patterns and software architectures (Booch, 1994; Ishikawa *et al.* 1992).

DOC is the discipline that uses OO techniques to distribute reusable services and applications efficiently, flexibly, and robustly over multiple, often heterogeneous, computing and networking elements. Recently the timing characteristics of data and function components of an object have been added to OO concept to make DOC more appropriate for real-time systems (Kim, 1997; Takashio and Tokoro, 1992). At the heart of contemporary distributed object computing is DOC middleware. DOC middleware is an object-oriented software that resides between applications and the underlying operating systems, protocol stacks, and hardware to enable or simplify the manner in which these components are connected and interoperate (Cobb and Shaw, 2000). Distribution middleware builds upon the lower-level infrastructure middle-ware to automate common network programming tasks, such as parameter marshaling/demarshaling, socket and request demultiplexing, and fault detection/recovery. Common examples of distribution middleware include the Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA) (OMG, 1995), Microsoft's Distributed COM (DCOM), and JavaSoft's Remote Method Invocation (RMI). In this thesis CORBA is chosen because of the nature of our simulation program and CORBA's general acceptance in the industry.

1.2 Objectives and Scope of This Work

In this thesis, a simulation program for the crane and claming device is developed. The system is a 'Window on World (WoW)' system that uses standard desktop monitors to display a 2D image of a 3D scene (Isdale, 1998). Users can control the virtual crane with a 3-axis joystick, and control the claming device with two single-axis joysticks. The control of the claming device is the same as on the real machine. 3D animation shows the real-time response of the virtual machine on the screen. The Client/server structure supports multi-users sharing the resource of a single server.

For the dynamics calculations, mathematical models of the crane and claming device are developed. Fulfilling the Object Orientated Programming (OOP) concept (Booch, 1994), the dynamics calculation algorithm for the crane, the claming device and the hydraulic driven units is wrapped into individual objects (classes) so that the code can be reused. OpenGL is adopted as the 3D graphics application programming interface (API). Since OpenGL is a renderer rather than a modeler, all 3D models are created in AutoCAD and then imported into our program. Display lists are utilized to speed up the rendering process. A reusable view class and a texture importing class are developed. With Critical Mass Lab's Dynamics Toolkit 2.0 and Collision Toolkit 1.0, real-time swing motion is created. Finally, CORBA technology is implemented to divide the simulation program into Service Providers (SPs) and Service Receivers (SRs). The interfaces for the SPs and the SRs are defined with Interface Definition Language (IDL). On the SP side, a special object called Object Factory is implemented, which produces a dynamics calculation object corresponding to each SR connected. The newly created dynamics calculation object continuously sends SRs the required data to construct the scene. The SR itself acts as a server calling methods in remote SP with control signals. The communications between an SP and SRs are all oneway calls; therefore, even when the connection is jammed or slow, the program can still run smoothly and the time lag effect is minimized. Concurrence models for both SP and SR are also carefully selected in order to obtain the best performance.

The simulation program is built on a framework that can be used for other physics-based visual simulations. The dynamics calculation module, including the mathematical models for the crane, the claming device, and hydraulic components, are all coded in an OOP way that they can be reused with little effort. The document/view structure with OpenGL can also be easily used to generate animations in Windows operating systems. The way we employ CORBA technology to distribute complex time-consuming computation over two or more computers and obtain higher data processing rates is also applicable to other similar problems.

The organization of this thesis is as follows. In Chapter 2 the mathematical models for the claming device, the crane and the hydraulic components are derived.

Chapter 3 explains the SR's structure and the animation generation with OpenGL. The detailed implementation of CORBA technology to achieve distributed computation is presented in Chapter 4. Chapter 5 presents results of the simulation and SP and SR performance analysis and conclusions are drawn in Chapter 6.

Chapter 2

Modeling of the System

The machine consists of a crane and a claming device. Both of them are driven by hydraulic actuators. In this chapter, the linkage dynamics for the claming device is first described, followed by the derivation of the crane's dynamics equations. The governing equations for the hydraulic actuators are also described.

2.1 Modeling of the Claming Device

2.1.1 Mathematical Model

An M.P. McCaffrey Inc. cable driven claming device is shown in Figure 2.1. Part of the equations for the mechanism of the claming device were derived by B. Surridge (1996).

The claming device is modeled as a 3-link slider mechanism. Link 1, with a length of 1.3 meters, is a member connecting the top of the grapple to the bucket. Link 2 is the bucket with a length of 1.2 meters. Link 3 is the pulley around which cable 2 curls. The forces applied to the model are the result of tensions originating from cable 1, and cable 2, and the gravitational force on each member. Cable 1, which is responsible for raising and lower the grapple, has a corresponding force that is referred to here as T1. Cable 2, which is partly responsible for opening and closing the grapple, has a corresponding force that is referred to here as T2. The angle that link 1 and link 2 make

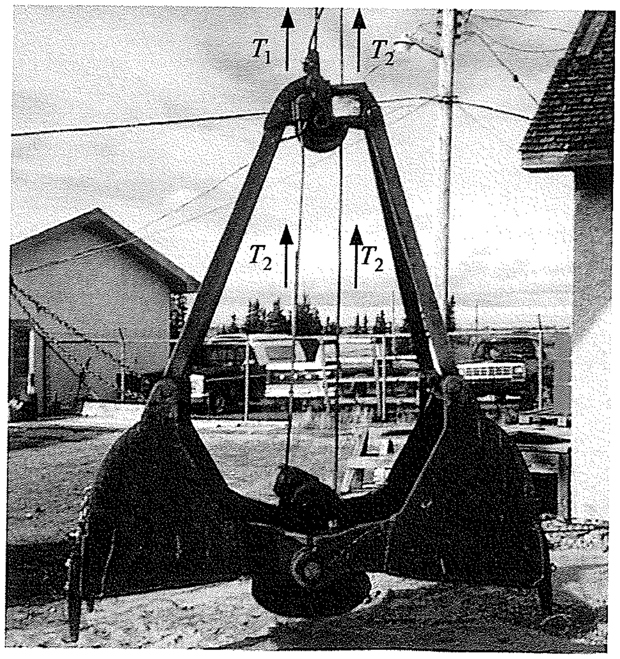


Fig. 2.1: Photo of the claming device.

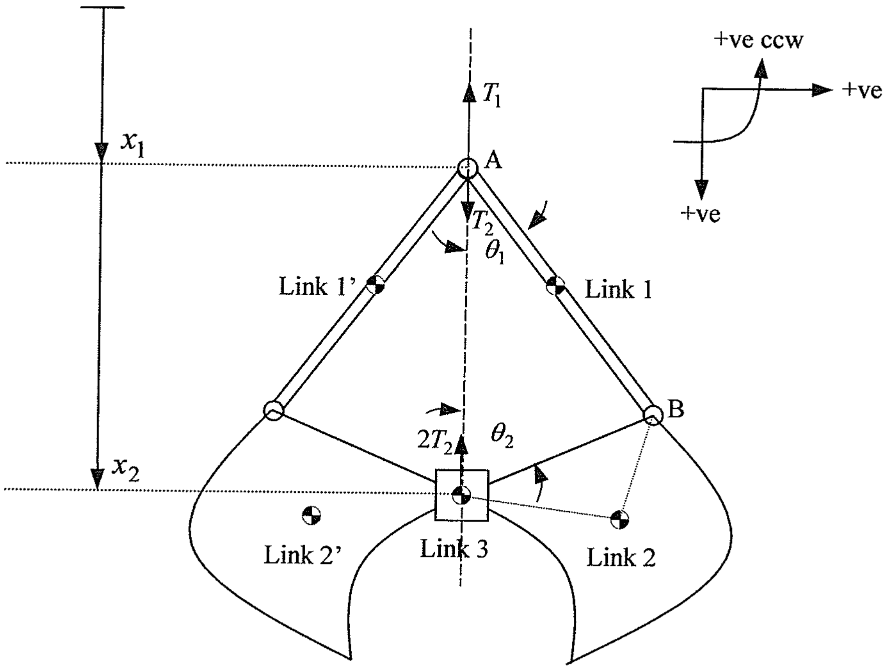


Fig. 2.2: Notations used on claming device model.

Table 2.1: Physical parameters of the claming device.

	Link 1	Link 2	Link 3
Mass (kg)	25	350	25
Size (m)	1.37	1.14	-----

with the vertical plane are referred to here as θ_1 and θ_2, respectively. The position of the top and bottom of the grapple are noted as x_1 and x_2, respectively. Notations used for the claming device model are shown in Figure 2.2. Note that the cables are removed as their effective forces are included.

Figure 2.3 shows the free body diagrams of link 1, 2 and 3.

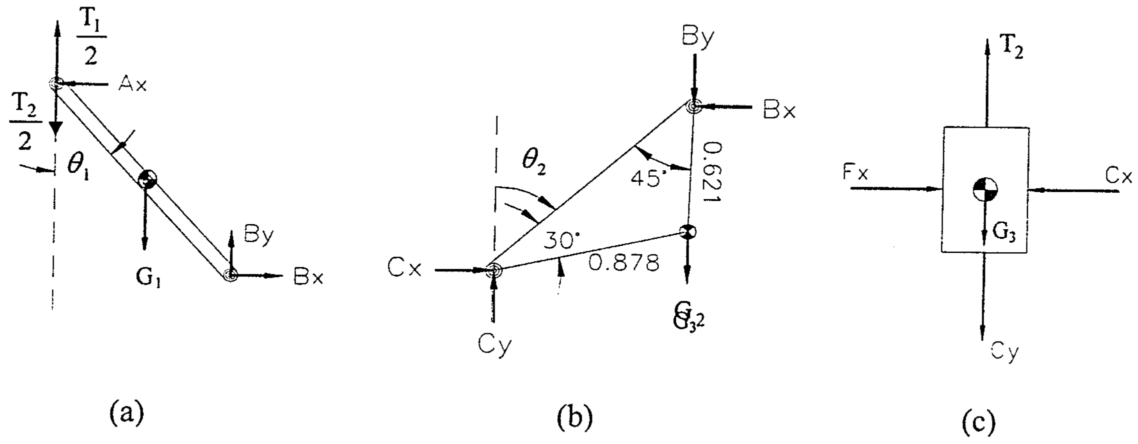


Fig. 2.3: Free Body Diagrams for (a) Link 1; (b) Link 2; (c) Link 3.

Dynamics equations are derived for each link respectively:

Link 1, for example:

$$-A_x + B_x = m_1 a_x \tag{2.1}$$

$$-B_x - T_1 + m_1 g = m_1 a_y \tag{2.2}$$

$$A_x \frac{l_1}{2}\cos(\theta_1) + B_x \frac{l_1}{2}\cos(\theta_1) + B_y \frac{l_1}{2}\sin(\theta_1) - (\frac{T_1 - T_2}{2})\frac{l_1}{2}\sin(\theta_1) = I_1 \alpha_1 \tag{2.3}$$

where A_x, B_x, B_y, and T_1 are forces acting on the bode, m_1 is the mass of link 1, a_y and a_x are link 1's center of gravity accelerations along y and x directions, α_1 is the angular acceleration of link 1, I_1 is the moment of inertia of link 1, θ_1 is the angle that link 1 makes with the vertical reference, and g is the acceleration due to gravity (9.81 m/s^2). For link 2 and 3, the dynamics equations are derived in the same manner.

The kinematic equation that is used for each member involves the normal and tangential accelerations. Figure 2.4 illustrates the normal and tangential accelerations for link 1. Thus the acceleration of Link 1 at the center of gravity is $a^{G_1} = a^A + a^{G_1/A}$.

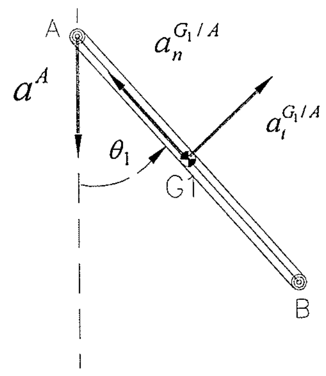


Fig. 2.4: Kinematics of Link 1.

This kinematic equation shows that the acceleration of the center of gravity for link 1, a^{G_1}, is equal to the acceleration of point A, which is equal to a^A plus the acceleration of the center of gravity with respect to point A, $a^{G_1/A}$. $a_n^{G_1/A}$ stands for the normal acceleration of center of gravity with respect to A, and $a_t^{G_1/A}$ stands for tangential acceleration of center of gravity with respect to A. Therefore, we have the following equations:

$$a_x^{G_1} = -\omega_1 \frac{l_1}{2}\sin(\theta_1) + \alpha_1 \frac{l_1}{2}\cos(\theta_1) \tag{2.4}$$

$$a_y^{G_1} = a^A - \omega_1^2 \frac{l_1}{2}\cos(\theta_1) - \alpha_1 \frac{l_1}{2}\sin(\theta_1) \tag{2.5}$$

where ω_1 is angular velocity of link 1 and a^A is the acceleration of position A. Similar kinematic equations for links 2 and 3 can be derived.

The acceleration diagram is depicted in Figure 2.5.

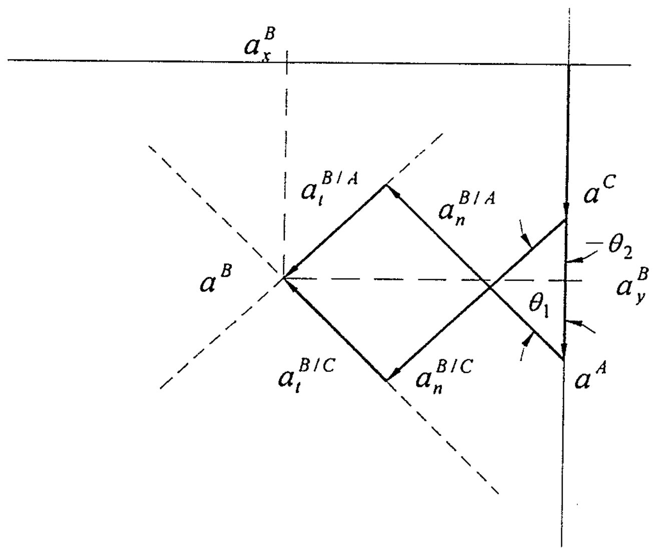


Fig. 2.5: Acceleration Diagram.

$$a_y^B = a^A - a_n^{B/A}\cos(\theta_1) + a_t^{B/A}\cos(\frac{\pi}{2} - \theta_1)$$

$$a_y^B = a^C + a_n^{B/C}\cos(\theta_2) + a_t^{B/C}\cos(\frac{\pi}{2} + \theta_2)$$

$$a^A - \omega_1 l_1 \cos(\theta_1) - \alpha_1 l_1 \sin(\theta_1) = a^C + \omega_2 l_2 \cos(\theta_2) + \alpha_2 l_2 \sin(\theta_2) \tag{2.6}$$

Equation 2.7 is found by a similar manner.

$$-\omega_1^2 l_1 \sin(\theta_1) + \alpha_1 l_1 \cos(\theta_1) = \omega_2^2 l_2 \sin(\theta_2) - \alpha_2 l_2 \cos(\theta_2) \tag{2.7}$$

So far we have a total of 17 equations, including nine dynamics equations, six kinematic equations and two extra equations, for the 3-link-slider mechanism. Eliminating four redundant equations, there are 13 equations with 13 unknowns. Hence all unknowns can be solved.

The following table shows the known and unknown variables in the equations:

Table 2.2: Known and unknown variables for claming device.

Unknown variables	A_x	$a_x^{G_1}$	B_y	F_x	$a_y^{G_2}$	α_2	a^C
	B_x	$a_y^{G_1}$	α_1	$a_x^{G_2}$	C_y	a^A	
Known variables	T_1	T_2	θ_1	θ_2	ω_1	ω_2	

θ_1, θ_2, ω_1 and ω_2 can be calculated based on the claming device's current positions and velocities of points A and C. From the dynamics equations derived, one can arrange the known and unknown variables into matrix equations. All unknowns can then be solved.

The movement of the claming device is limited because of geometric restrictions. The claming device has three working states, referred as 'Free', 'Full' and 'Distributed'. When in the Free state, the claming device bodies rotate freely within the range of their two extremes. The dynamic equations derived above are valid only for this working state. As it reaches its extreme, the claming device either works in the Full or Distributed state. While in the Full state, all three of its links are moving together as one single body. Only one force from the connecting cables applies to either point A or C. While in the Distributed state, the 3 links are also moving together without relative movements. Forces from two cables are applying on link 1 and 3 individually, which result in different accelerations on each link. Notice that the two forces applied to points A and C can only be positive because in this case cables cannot transfer negative forces to the claming device.

When the claming device is fully closed or opened, the acceleration of all three links is

$$a = -(T - (m_1 + m_2 + m_3 + m_l)g)/(m_1 + m_2 + m_3 + m_l) \qquad (2.8)$$

T is the force applied to point A or C (see Figure 2.2); m_1, m_2, and m_3 are the masses of links 1, 2, and 3, and m_l is the mass of the load of the claming device.

When the claming device is in the Distributed state, the acceleration of each link is calculated as if they are in the Free state.

If the claming device is entering the Distributed state from the Free state, the velocity of each link is calculated assuming the collision among the links is completely inelastic, which is

$$v = (m_1 v_1 + m_2 v_2 + m_3 v_3)/(m_1 + m_2 + m_3 + m_l) \qquad (2.9)$$

Because the claming device is symmetric along its vertical axis, when considering only the claming device, there are no horizontal velocities. This is only a one-time-change that happens when it enters the Distributed state from the Free state. Once the claming device is in the Distributed state, the dynamics calculation uses equations for the Free state.

2.1.2 Algorithm of Switching among Working States

As mentioned above, the claming device has three working states and corresponding dynamic equations: Free, Distributed and Full. Possible state changes are listed in Table 2.3 as follows.

Table 2.3: Possible working state changes.

Previous Working States	**Next Possible Working States**
Free	Free
	Distributed
Distributed	Free
	Distributed
	Full
Full	Full
	Distributed

In addition to a flag indicating working states, two other flags are used to reflect detailed working status of the claming device: one is flag 'Geo-Status', the other is flag 'Load-Status'. The effect of these flags and their values are listed in Table 2.4.

Table 2.4 Additional flags used in describing the working states.

Flag Name	Effect	Flag Status
Geo-Status	Describing the geometric status of the claming device, either it reaches its maximum or minimum limit and cannot move further.	Open
		Close
		In-range
Load-Status	Indicating if the claming device is carrying any load.	With-load
		Without-load

Once the device is carrying load, even when it is operating within its geometry extremes, it is still considered as working in either the Distributed state or the Full state depending on its previous working state. This is because there should not be any relative movements among the claming device's links when it is firmly carrying some load. This situation is the same as using pliers to grip a solid object. Therefore, depending on the combinations of working states and flags, the claming device changes its working states when certain conditions are satisfied. This is shown in Table 2.5.

The change of the Geo-Status flag is done by examining the claming device's geometric conditions. The change of the Load-Status flag is based on contact detection and force monitoring. The flags are set as 'With-load' and "Distributed" when the claming device bodies are in contact with the object and the previous working state is Free. When it is in the Distributed state, the static force between its left bucket and right bucket (link 2) is monitored. Once the force is lower than a certain value, we assume the friction between the bucket and the load is too small to hold the object in place, so the object is released and the claming device enters the Free state from the Distributed state. Figure 2.6 shows the flow chart for changing the claming device's working state and dynamics equations accordingly.

By changing the working states dynamically, the claming device can be modeled mathematically. The behavior of the claming device is described by the dynamic equations at any given time. Note that the dynamic equations are highly non-linear because of collision of the claming device's bodies and the sudden changes of its working status. For this reason we have to use a very small time step in our integrating subroutine.

Our simulation takes 10000 cycles to determine the dynamics of the claming device for one second, which is quite computationally heavy.

Table 2.5: Conditions for the claming device to change working states.

Previous Status			Conditions to Change	Next Working State
Previous Working State	Flag Geo-Status	Flag Load-Status		
Full	Open	WithoutLoad	$T_1 > 0$	Distributed
			above condition violated	Full
	Close	WithoutLoad/WithoutLoad	$T_2 > 0$	Distributed
			above condition violated	Full
	In-Range	WithLoad	$T_1 >$	Distributed
			$T_1 = 0$	Full
Distributed	Open	WithoutLoad	$T_2 = 0$	Full
			$a^A - a^C > 0$	Free
			above 2 conditions violated	Distributed
	Close	WithLoad/WithoutLoad	$T_1 = 0$	Full
			$a^A - a^C < 0$	Free
			above 2 conditions violated	Distributed
	In-Range	WithLoad	Flag Load-Status changes	Free
			$T_1 = 0$	Full
			above 2 conditions violated	Distributed
Free	Open	WithoutLoad	No condition	Distributed
	In-Range	WithoutLoad	No condition	Free
		WithLoad	No condition	Distributed
	Close	WithoutLoad	No condition	Distributed

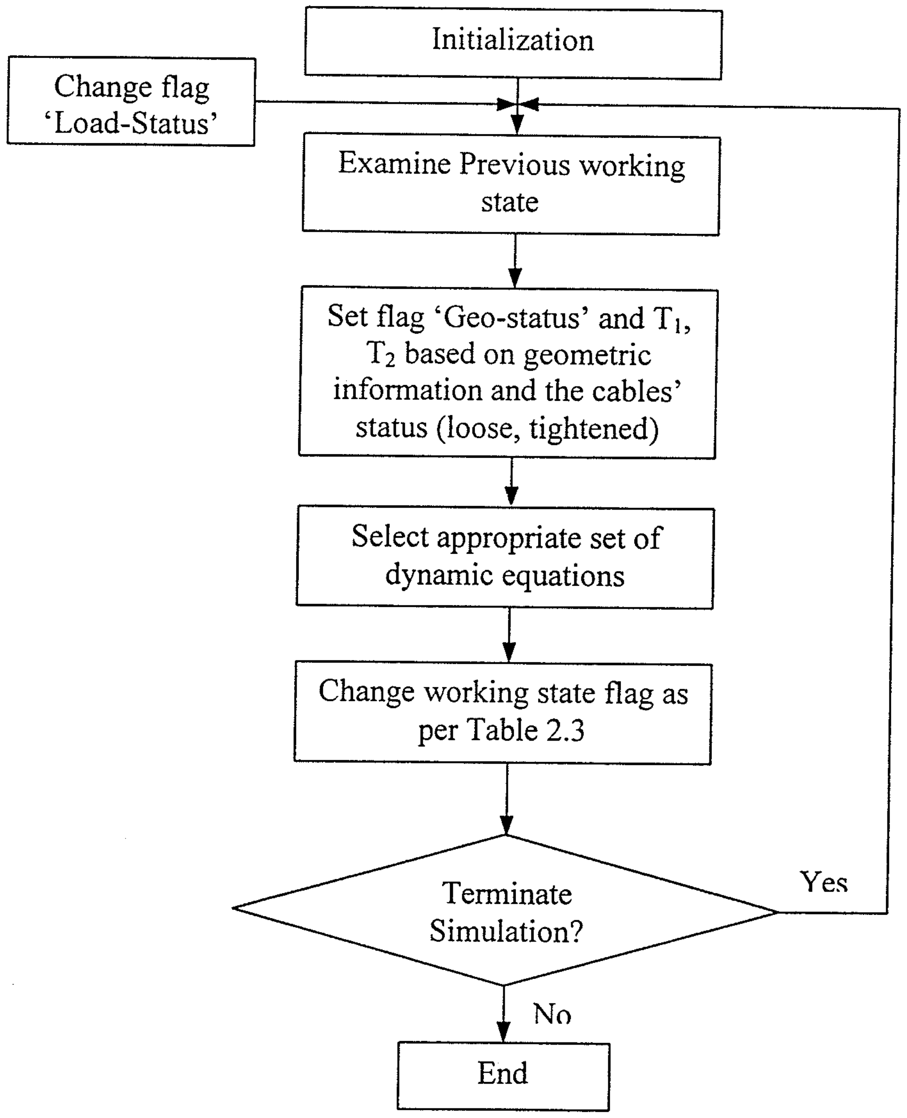
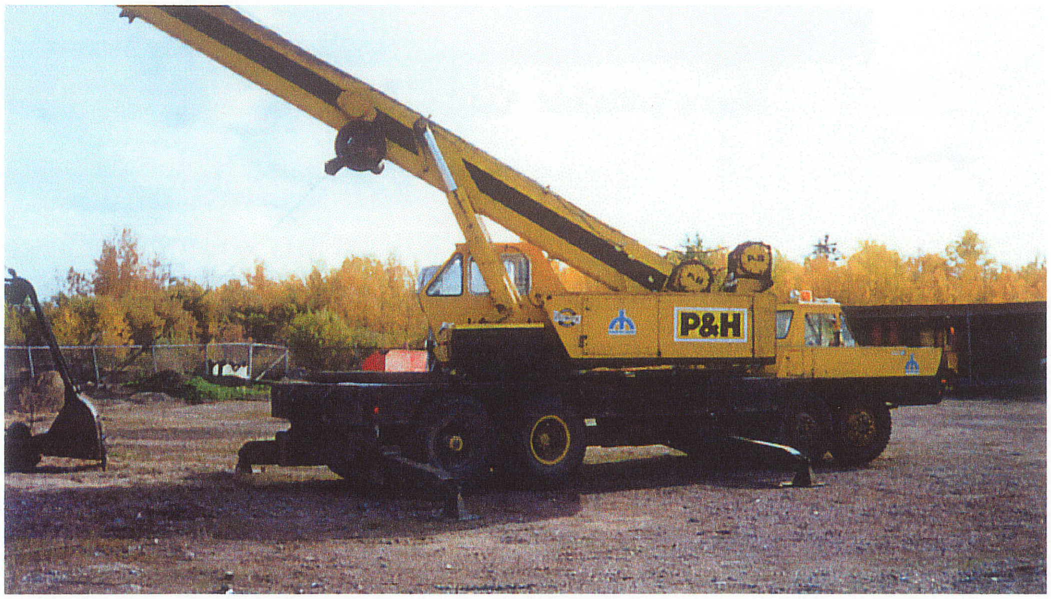

Initialization
Change flag 'Load-Status'
Examine Previous working state
Set flag 'Geo-status' and T_1, T_2 based on geometric information and the cables' status (loose, tightened)
Select appropriate set of dynamic equations
Change working state flag as per Table 2.3
Terminate Simulation?
Yes
No
End

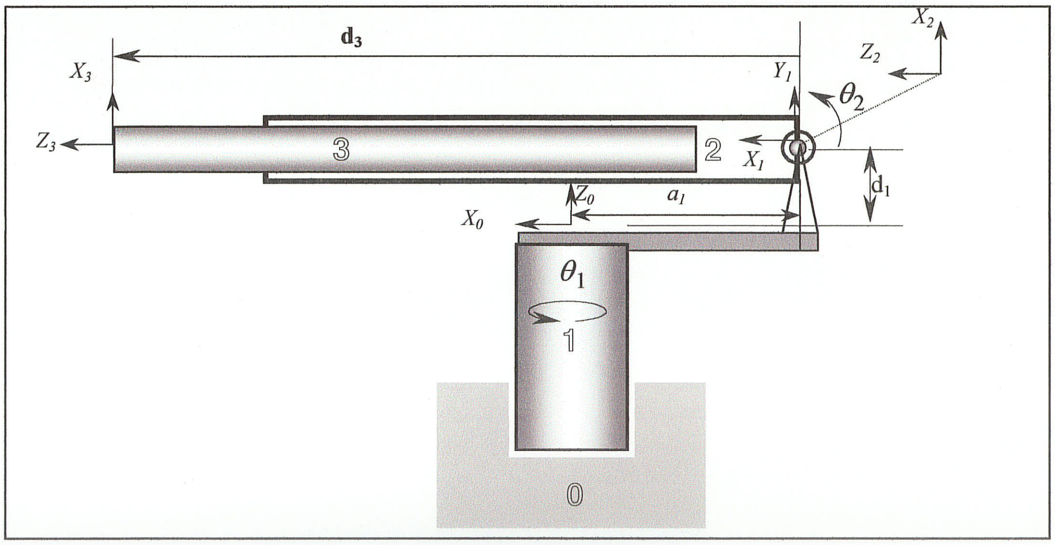
Fig. 2.6: Flow chart for changing the claming device's working state and dynamics equations.

2.2 Modeling of the Crane

The mobile crane used in this thesis is a P&H-T250 crane (see Figure 2.7a). It is considered as a hydraulic powered robot arm with three degrees-of-freedom. The upper structure of the machine is rotated on a carriage by a hydraulic motor through a reduction gear. This motion is referred here as "swing". The other two main links, whose motions are called "boom" and "telescope", are movable around their joints by hydraulic



(a)



(b)

Fig. 2.7: (a) P&H-T250 crane; (b) joint coordinates.

cylinders. There are another two hydraulic motors located at the back of the carriage that drive two winches to release or withdraw the cables connected to the claming device. By changing the length of the two cables separately, the tasks of opening, closing, lifting and lowering of the claming device are performed. The output power of these two motors is different. The one with higher output power is called the main winch motor. Cable driven by this motor is connected to point A of the claming device. This motor is responsible for handling most of the load. The other motor drives the second cable, which is connected with point B of the claming device through a pulley. This motor is called the auxiliary winch motor. The boom, telescope and swing together with the main and auxiliary winch motion are responsible for placing the claming device at the desired location in the working environment.

2.2.1 Linkage Dynamics

The dynamics model of the crane consists of the model of the linkage and the model of the actuators driving the manipulator joints and winches. The linkages and actuator dynamics are described in this section. Table 2.6 shows the physical parameters of the crane.

Table 2.6: Physical parameters of the crane.

	Swing Cab	Telescopic Arm
Bounding Dimension (m) (Length $\times$ width $\times$ height)	$4.0 \times 2.4 \times 1.4$	$9.7 \times 0.4 \times 0.6$
Mass (kg)	1621	5400
Range of motion	$0 \sim 360^{\circ}$	Boom: $0 \sim 50^{\circ}$ Extension range: 9.7~24.4 m

The kinematics of the crane are derived using the descriptive conventions established by Denavit and Hartenburg (Schilling, 1990). Link coordinate frames attached are shown in Figure 2.7b. The parameters for the model are shown in Table 2.7, where Denavit and Hartenburg parameters, θ_n, d_n, α_n and a_n are defined.

Table 2.7: Denavit-Hartenburg parameters for crane model.

Link	Variable	θ_n	d_n	a_n	α_n	*home*
1	θ_1	θ_1	d_1	$-a_1$	90°	0
2	θ_2	θ_2	0	0	90°	90°
3	d_3	0	D_3	0	0	d_3

Using parameters defined in Table 2.7, the transformation matrix of each link is obtained as follows:

$$A_1 = \begin{bmatrix} \cos\theta_1 & 0 & \sin\theta_1 & -a_1\cos\theta_1 \\ \sin\theta_1 & 0 & -\cos\theta_1 & -a_1\sin\theta_1 \\ 0 & 1 & 0 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
$$A_2 = \begin{bmatrix} \cos\theta_2 & 0 & \sin\theta_2 & 0 \\ \sin\theta_2 & 0 & -\cos\theta_2 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (2.10)$$
$$A_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $\boldsymbol{A_i}$ is the transformation matrix that relates the coordinate frame of link i to the coordinate frame of link $i-1$. Transformations T_i $(i = 1,2,3)$ of each link coordinate frame with respect to the base coordinate system, $\{X_{0,}, Y_{0,} Z_0\}$, are obtained by taking the products of the $\boldsymbol{A_i}$ transformations.

$$T_i = \prod_1^i A_i \qquad (i = 1,2,3) \tag{2.11}$$

The dynamics of the manipulator are then defined by the following equations:

$$\tau_i = \sum_{j=1}^{3} D_{ij}\ddot{\theta}_j + \sum_{j=1}^{3}\sum_{k=1}^{3} D_{ijk}\dot{\theta}_j\dot{\theta}_k + D_i \qquad (i = 1,2,3) \tag{2.12}$$

where

$$\begin{cases} D_{ij} = \displaystyle\sum_{p=\max(i,j)}^{3} Trace\left(\frac{\partial \boldsymbol{T}_p}{\partial \theta_j}\boldsymbol{J}_p\frac{\partial \boldsymbol{T}_p^T}{\partial \theta_i}\right) \\ D_{ijk} = \displaystyle\sum_{p=\max(i,j,k)}^{3} Trace\left(\frac{\partial^2 \boldsymbol{T}_p}{\partial \theta_j \partial \theta_k}\boldsymbol{J}_p\frac{\partial \boldsymbol{T}_p^T}{\partial \theta_i}\right) \\ D_i = \displaystyle\sum_{p=i}^{3} - m_p\, \boldsymbol{g}^T \frac{\partial \boldsymbol{T}_p}{\partial \theta_i}\, {}^p\bar{\mathbf{r}}_p \end{cases} \tag{2.13}$$

and

$$\boldsymbol{J}_p = \begin{bmatrix} \dfrac{-I_{xx_p} + I_{yy_p} + I_{zz_p}}{2} & I_{xy_p} & I_{xz_p} & m_p\bar{x}_p \\ I_{xy_p} & \dfrac{I_{xx_p} - I_{yy_p} + I_{zz_p}}{2} & I_{yz_p} & m_p\bar{y}_p \\ I_{xz_p} & I_{yz_p} & \dfrac{I_{xx_p} + I_{yy_p} - I_{zz_p}}{2} & m_p\bar{z}_p \\ m_p\bar{x}_p & m_p\bar{y}_p & m_p\bar{z}_p & m_p \end{bmatrix} \tag{2.14}$$

τ_i is the joint torque/force at joint i, $\boldsymbol{J}_p$ is the pseudo-inertia matrix of link p, $\boldsymbol{g} = [g_x \quad g_y \quad g_z \quad 0]^T$ is the gravitational acceleration vector in the base coordinate system, and ${}^p\bar{\mathbf{r}}_p = [\bar{x}_p \quad \bar{y}_p \quad \bar{z}_p \quad 0]^T$ is the vector of the coordinates of the center of gravity of link p with respect to the same link coordinate system. D_{ij} represents the coupling inertia between joints i and j. D_{ijk} represents the Coriolis forces at joint i due to velocities at joints j and k. Finally, D_i represents the gravity loading at joint i (Paul, 1981).

Assuming the joint torque/forces τ_i, joint positions θ_i and joint velocities $\dot{\theta}_i \; (i = 1,2,3)$ are known, the three equations obtained from (2.12) are solved simultaneously for the three joint accelerations, $\ddot{\theta}_1$, $\ddot{\theta}_2$ and $\ddot{\theta}_3$. These accelerations are integrated to get the joint velocities and positions in simulation programs. Note that in this case, $\ddot{\theta}_3$ is replaced with $\ddot{d}_3$ because joint 3 is a prismatic joint.

The relationship between the torque τ at joint 2 and the effective actuation force *F*, which is generated by the hydraulic cylinder, is obtained by applying the principle of virtual work.

$$\tau \cdot \mathrm{d}\theta = F \cdot \mathrm{d}X \tag{2.15}$$

where $d\theta$ and dX denote the incremental changes in joint displacement at joint 2 and piston displacement, respectively. Therefore, the joint torque τ can be calculated as:

$$\tau = F \frac{\mathrm{d}X}{\mathrm{d}\theta} \tag{2.16}$$

The joint displacement θ and the piston displacement X are related by the geometrical configuration. The mechanism of joint 2 is depicted in Figure 2.8. The dashed black lines

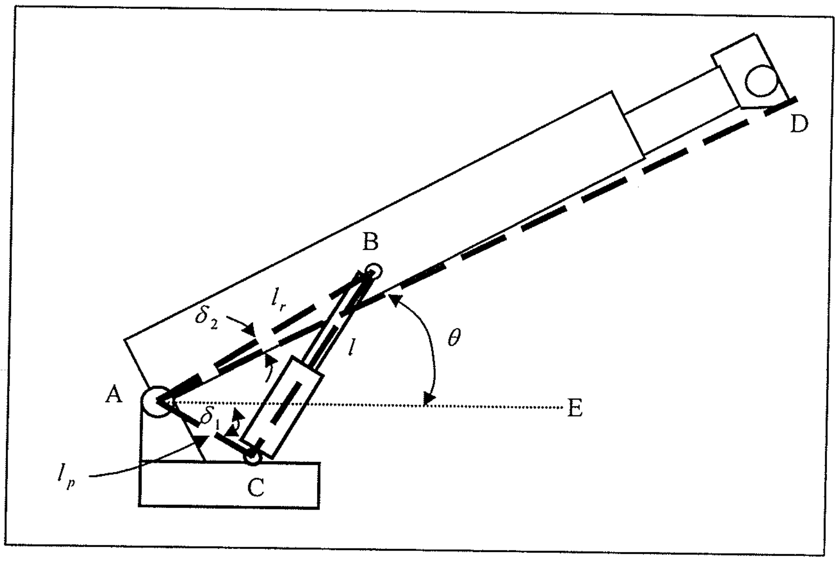


Fig. 2.8: Mechanism of joint 2.

denote virtual links. δ_1 is the angle between AC and AE; δ_2 is the angle between AB and AD. Both δ_1 and δ_2 are constant. l, l_r and l_p are the length of link BC, AB and AC, respectively. l_r and l_p are fixed.

In triangle ABC, we have:

$$l^2 = l_p^2 + l_r^2 + 2l_p l_r \cos(\theta + \delta_1 + \delta_2) \tag{2.17}$$

Taking the derivative of (2.17) yields:

$$2l\frac{dl}{dt} = -2l_p l_r \sin(\theta + \delta_1 + \delta_2)\frac{d\theta}{dt} \tag{2.18}$$

$$\frac{dl}{dt} = \dot{X} = \frac{-l_p l_r \sin(\theta + \delta_1 + \delta_2)}{\sqrt{l_p^2 + l_r^2 + 2l_p l_r \cos(\theta + \delta_1 + \delta_2)}}\dot{\theta} \tag{2.19}$$

Therefore, by submitting into equation (2.16) we have:

$$\tau = F \frac{-l_p l_r \sin(\theta + \delta_1 + \delta_2)}{\sqrt{l_p^2 + l_r^2 + 2l_p l_r \cos(\theta + \delta_1 + \delta_2)}} \tag{2.20}$$

2.2.2 Actuator Dynamics

As illustrated in Figure 2.9, the hydraulic system of the P&H-T250 crane consists of pumps and valves. An engine provides power to all pumps. As seen in Figure 2.9, there are five individual pumps: one for the main winch, one for the auxiliary winch, one for the swing, one for the boom hoist, and one for the telescope. Here we assume there is no power supply limit, which means the engine is powerful enough to drive all 5 pumps at the same time, and there is no flow constraint for each actuator. Therefore, there is no coupling effect involved. Among the actuators, three of them are driven by hydraulic motors. The others are driven by hydraulic cylinders.

The hydraulic actuator model in this thesis consists of only the actuator itself and its servo-valve (Merritt, 1967). Figure 2.10 shows a typical hydraulic cylinder with an open center control valve. For this kind of actuator, the governing nonlinear equations that describe the fluid flow distribution in the valve can be written in their simplest forms as follows:

$$Q_i = kwx\sqrt{P_s - P_i} \tag{2.21}$$

$$Q_o = kwx\sqrt{P_o - P_e} \tag{2.22}$$

$$Q_e = Q - Q_i = kw(1-x)\sqrt{P_s - P_e} \tag{2.23}$$

where Q_e is the remaining pump flow back to the tank having an exit pressure P_e, P_i and P_o are the input and output line pressure. k is the orifice coefficient and w is the area gradient. The orifice areas a_i, a_e and a_o were assumed to be linearly proportional to the normalized spool displacement, x.

The fluid compressibility equation is

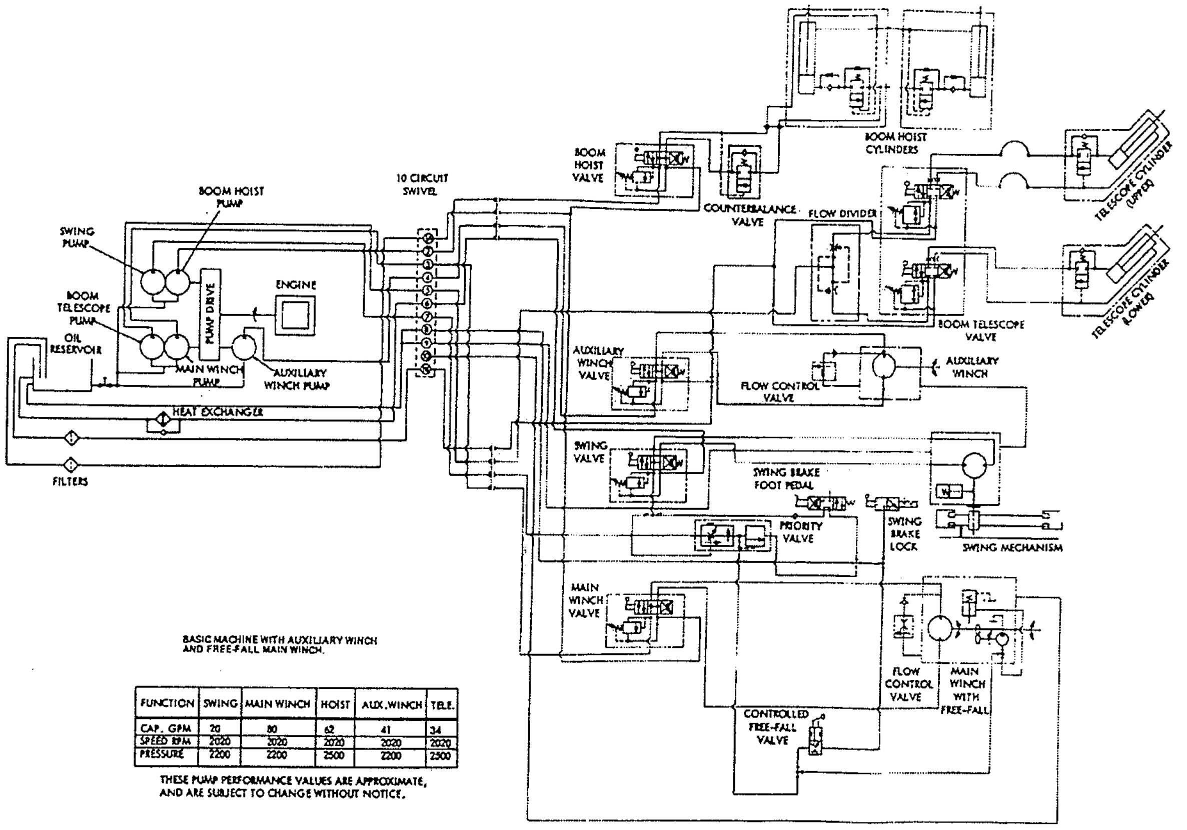
$$\dot{P} = \frac{1}{C}(Q - Q'), \qquad C = \frac{V}{\beta} \tag{2.24}$$

where P is the pressure in a control volume, C is the hydraulic compliance of the flexible hoses connecting the valve to the actuator, β is the bulk modulus of the oil and V is the volume of oil.

The force balance equation for a cylinder is

$$m\ddot{X} = P_i A_i - P_o A_o - F - F_c \tag{2.25}$$

where m is the mass of the cylinder rod, P_i and P_o are the pressures in the two cylinder chambers, A_i and A_o are the piston areas on the two sides of the actuator, d is the viscous damping of the cylinder and F_c is the Coulomb friction of the cylinder.



BASIC MACHINE WITH AUXILIARY WINCH AND FREE-FALL MAIN WINCH.

FUNCTION	SWING	MAIN WINCH	HOIST	AUX. WINCH	TELE.
CAP. GPM	20	80	62	41	34
SPEED RPM	2020	2020	2020	2020	2020
PRESSURE	2200	2200	2500	2200	2500

THESE PUMP PERFORMANCE VALUES ARE APPROXIMATE, AND ARE SUBJECT TO CHANGE WITHOUT NOTICE.

Fig. 2.9: P&H-T250 crane's hydraulic system scheme.

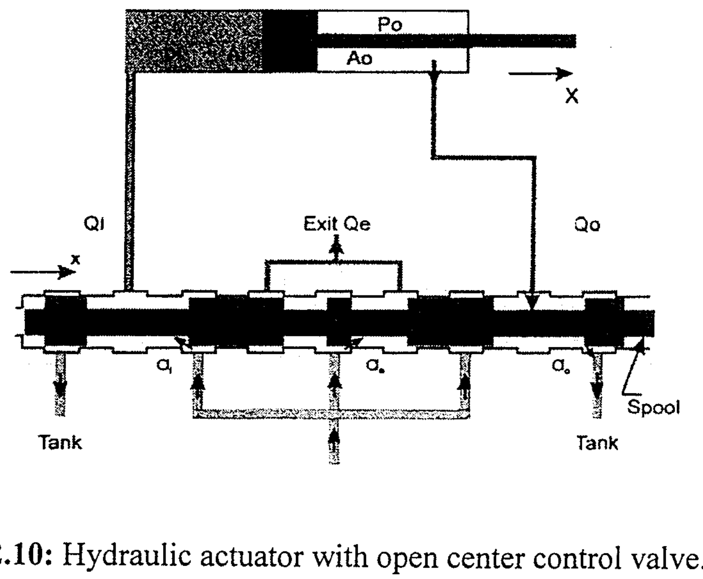


Fig. 2.10: Hydraulic actuator with open center control valve.

For the hydraulic motor used for the winch and the swing, the governing equations are:

$$\tau = D_m(P_i - P_o) \tag{2.26}$$

where D_m is the volumetric displacement of the hydraulic motor and τ is the torque generated. The torque is transmitted through a gear train to rotate the winch and the upper body of the crane. Power losses due to leakage are ignored in this thesis.

2.3 Integration of Crane and Claming Device

So far the claming device and the crane were modeled individually. However, the claming device and the crane are related to each other. To solve this problem, based on some assumptions, we can simplify the dynamics complexity incurred by the coupling.

The crane has an anti-swing mechanism that prevents the claming device from swinging too much. As illustrated in Figure 2.11, the anti-swing mechanism includes a spring roll and a steel chain connecting the claming device body. Through the steel chain, the spring roll exerts a force on the claming device and brings it back to its neutral position. With this mechanism, the swing motion of the claming device and the possible rotation along the cables are greatly reduced. Furthermore, the operator can hardly control both the crane and the claming device simultaneously. Control of the claming device itself needs the operator to use his/her both hands. So when the crane is moving, the status of the claming device always remains the same.

Based on the facts mentioned above, we assume that the claming device is part of the crane arm if the length of the cables does not change. While the crane arm is moving,

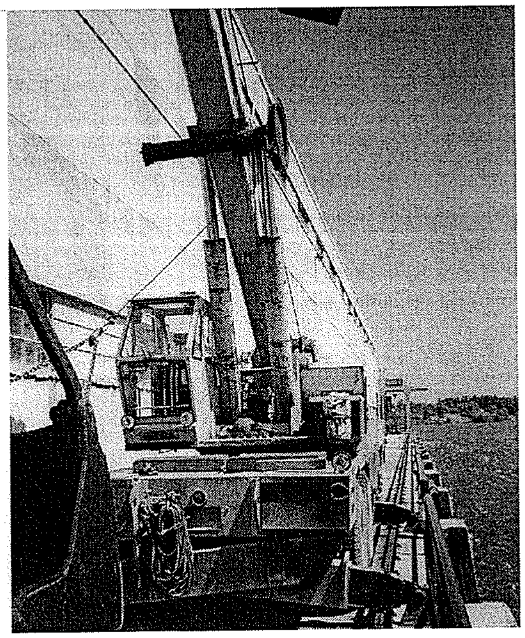


Fig. 2.11: Anti-swing mechanism on the crane.

vertically below the crane arm tip. According to the position of the claming device relative to the crane arm, the inertia of link 3 of the crane model is recalculated. The updated inertia for link 3 is plugged in the pseudo-inertia matrix in equation (2.14). This means that the accuracy of the dynamics model is sacrificed in favor of simplicity of calculation.

Chapter 3

User Interface Design

In this chapter, the development of the simulation framework is explained in depth. Based on the Microsoft Foundation Class (MFC), the framework adopts OpenGL as the graphics Application Programming Interface (API) and provides interactive 3D animation. First, the overall program hierarchy is discussed. Then, the method used to create the scene is introduced. Reusable classes that wrap certain OpenGL functions to hide the complexity of maneuvering primitives and texture mapping, are also described. Finally, the method of signal collection and collision detection with Critical Mass Lab's Simulation Toolkit is illustrated. The framework is not specific to this simulation. Virtually any interactive simulation can be built on this structure.

3.1 Overview of the Simulation Program

The goal of this thesis is to develop a simulation program with a suitable graphics user interface. Users control the virtual machine with a set of joysticks in the same way as they do on a real machine. The users are placed in the loop of a real-time simulation, immersed in a world both autonomous of and responding to their actions. Figure 3.1 shows a picture of the whole simulator.

For shorter development time, this program is built based on the Microsoft Foundation Class (MFC) Library and adopts the classic Document/View structure (Microsoft, 1998). MFC is an "application framework" for programming in Microsoft Windows. MFC provides much of the code necessary for managing windows, menus,

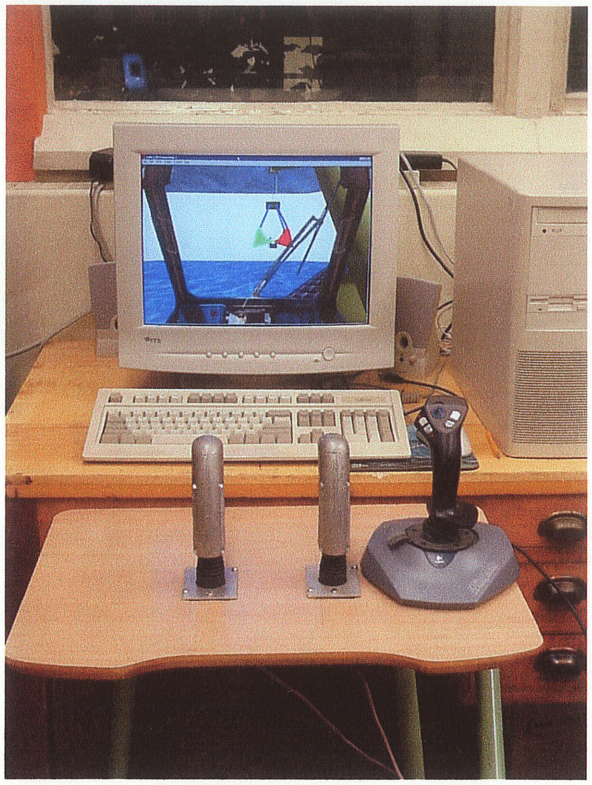


Fig. 3.1: Picture of the real-time simulator.

and dialog boxes, performing basic input/output, storing collections of data objects, and so on. Given the nature of C++ class programming, it is easy to extend or override the basic functionality that MFC supplies. The most important classes that were used in this thesis are CColorView and COpenGLView, which are inherited directly from the CView class provided by MFC. All control signals and user commands are captured and processed in the 'view' class.

In this thesis, two separate modules were developed. One module calculates the real-time dynamics; the second one displays the behavior of the virtual machine interacting with a virtual environment. Since the Microsoft Windows operating system is the most commonly used system in the market, the simulation is mainly developed for this platform. Part of the code remains portable by using platform independent technology. For the graphics, OpenGL was chosen. For the distributed computation, Common Object Request Broker Architecture (CORBA) technology is employed.

Figure 3.2 shows the diagram of the system. Arrows represent the data flow direction. The dynamics calculation module simply gets the control signals generated by the view module and computes the future states of the machine. The view module is responsible for interacting with the user. Control signals and user commands are collected via the joysticks and/or keyboard and are directed to the dynamics calculation module. The view module also continuously updates the graphics of the working environment and the machine on the screen, and thus produces the animation.

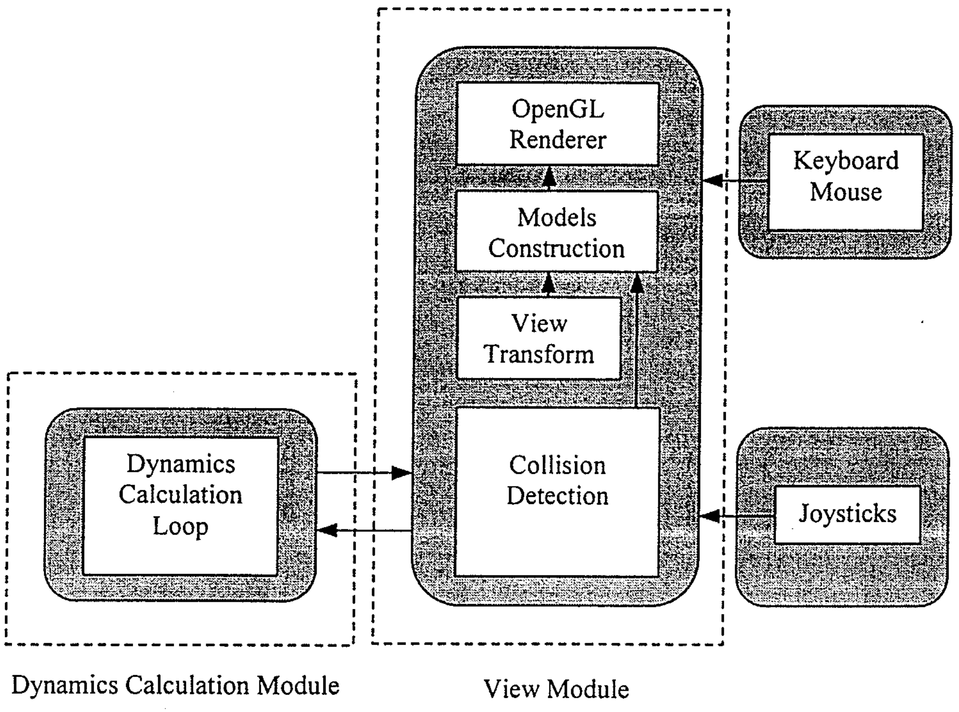


Fig. 3.2: System diagram.

3.2 Drawing with OpenGL

3.2.1 Introduction to OpenGL

Silicon Graphics Inc. has developed OpenGL (short for "Open Graphics Library") as a successor to the IRIS Graphics Library (IRIS GL). IRIS GL is a hardware independent graphics interface that has been implemented on numerous graphics devices of varying sophistication. OpenGL has a 3D rendering philosophy similar to IRIS GL, but it has removed outdated functionality and replaced it with more general functionality making it a distinctly new interface (Kilgard 1995). In 1992, OpenGL was proposed as a standard for 3D graphics and the industry consortium known as the OpenGL Architectural Review Board (ARB) was formed. Currently the use of OpenGL is free of charge. Applications developers do not need to license OpenGL. Hardware vendors that create binaries to ship with their hardware are the only developers that require a license.

OpenGL provides a layer of abstraction between a graphics hardware and an application program. To the programmer, it is visible as an Application Programming Interface (API) consisting of about 120 distinct commands. To maintain good performance rates the OpenGL API allows complete access to the graphics operations at the lowest possible level that still provides device independence. As a result it does not provide a means for describing or modeling complex geometric objects (Segal, 1993). OpenGL commands specify how a certain result should be produced rather than what exactly the result should look like, i.e., OpenGL is procedural rather than descriptive. OpenGL uses immediate mode rendering; when the graphics system is used to create a scene or object, each function and command has an immediate effect on the frame buffer and the result of each action is immediately visible on the screen. The designers of

OpenGL present the graphic system as a state machine. The routines that OpenGL supplies provide a means for the programmer to manipulate OpenGL's state machine to generate the desired graphics output. The programmer puts the machine in different states/modes that remain in effect until he/she changes them. The programmer can at any time query the system for the current value of any of the state variables. The OpenGL API is defined in terms of the C programming language, but bindings for several other languages exist. Currently the Architectural Review Board controls C, C++, Fortran, Pascal, and Ada binding specifications. However, several unofficial bindings exists for other languages, e.g., Java, Tcl/Tk, and Python (Neider et al. 1993).

OpenGL and Direct3D are both 3D graphic APIs. Direct3D is for Windows only while OpenGL is cross-platform. For games, both are equally fast and feature-rich. For professional 3D graphics, OpenGL is used almost exclusively. As predicted in PC Magazine (Ozer, 1998), currently standard desktop machines are being generally equipped with 3D accelerator cards, with performance levels on these machines reaching the level of performance on previous high-end workstations. Therefore, OpenGL is becoming more popular.

Typically an OpenGL program starts by creating a window, and a corresponding framebuffer, in which the screen picture will be drawn. Next, the programmer can display basic geometric objects like points, lines, or polygons, and he or she can assign colors and materials to these. More complex objects can be put into a display list, which can be used later as a single object. OpenGL does all the display processing; it uses objects' attributes and 3D world attributes (such as light sources or orientation) to produce a 2D image in the framebuffer. Since it is only a renderer and not a modeler, 3D models are typically first modeled using modeling software such as AutoCAD or 3Dmax; and are then imported into the OpenGL application.

3.2.2 Setting up an OpenGL Framework for Windows

Because this simulation is built upon the MFC and adopts the document/view structure, it is very natural to take advantage of the CView Class and extend its functions to make an OpenGL framework for Windows. Therefore the reusable COpenGLView class and, later, the CColorView class are derived from the base CView class. Six basic steps are required to use OpenGL in a Windows program:

- Getting a device context (DC) for the rendering location
- Selecting and setting a pixel format for the device context
- Creating a rendering context (RC) associated with the device context
- Drawing with OpenGL commands
- Releasing the rendering context
- Releasing the device context

Detailed information about device context and rendering context can be found in Microsoft's Foundation Class technical documentation (Microsoft, 1998). The COpenGLView class implements these steps in appropriate member functions and provides transparent OpenGL encapsulation for programmers. Figure 3.3 shows the schematic framework of using OpenGL in a CView derived class with MFC. The MFC framework calls the OnCreate member function when an application requests to create the window. This function then calls InitializeOpenGL that does most of the initialization

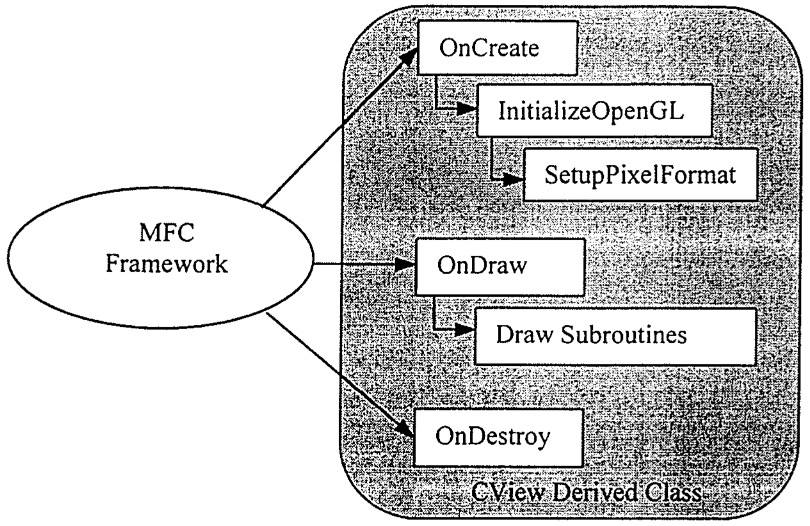


Fig. 3.3 Schematic framework of using OpenGL in a CView derived class.

work, including choosing the pixel format, obtaining the Device Context (DC), creating an Rendering Context (RC) from the DC, and finally making the RC current. Once the RC is created properly, the application can use OpenGL commands to draw into a specific area in a window. Finally, when the user terminates the application, MFC framework calls OnDestroy, which frees up the occupied resource by deleting both RC and DC.

The `SetupPixelFormat()` function chooses a pixel format for the OpenGL renderer. Pixel formats are the translation layer between OpenGL calls and the rendering operation that Windows performs. The capabilities of an OpenGL window depend on the pixel format selected. The available pixel formats depend on the implementation of OpenGL that is running, the current video mode in which Windows is running, and the video hardware installed. The developing environment for this simulation is as follows:

- Operating system: Windows 98

- Color mode: 32bit (true color)
- Video Card: Creative 3D Blaster Annihilator with NVDIA's GeForce 256 chipset

When the simulation program's View object is being constructed, the above pixel will be put as an argument to the win32 function `ChoosePixelFormat()`. The `ChoosePixelFormat()` function finds an appropriate pixel format supported by a device context closest to the given pixel format. Then `SetPixelFormat()` is called with the found pixel format to set it as the specified device context's current pixel format. In this way, the simulation can always get the best possible pixel format available from the specific platform on which the program is running.

3.2.3 Creating the Animation

Animation is a series of pictures being drawn continuously. There is more than one way to set Windows to draw repeatedly. The first method is to use a timer within program. For example, every 30 milliseconds the timer sends a message calling the drawing routine to do the rendering. This method is not effective in this study. Firstly, the Windows timer has a resolution about 55 milliseconds, i.e., the maximum refresh rate is 20 frames per second, which is insufficient for smooth animation. Secondly, the Windows timer message has the lowest priority to be processed. In the application investigated in this thesis, the user keeps sending Windows keyboard or joystick events, thus the processing of the timer event will be delayed further. Windows does, however, have a higher resolution timer called 'multimedia timer'. It is accurate up to 1 millisecond without any time lag. Therefore, a multimedia timer was used to collect control signals in this thesis. Here the graphics rendering and control signal collecting are not combined together.

This simulation employs a simple yet efficient method to generate animation. At the end of the scene rendering, the window is invalidated and a message is sent to repaint it. This message will be put into the message queue and processed when it is time to repaint. The code to achieve this is:

```
if ( m_bAnimationRunning )
{
      //draw subroutines
      InvalidateRect(0, FALSE);
      GetParent()->PostMessage(WM_PAINT);
}
```

The above code does the following tasks: if the animation is current running, after issuing all the drawing commands, it posts a message to tell Windows that the entire client area is invalid and needs to be redrawn. The message remains in the message queue until it is processed. The advantage of this approach is that it allows Windows to do other background processing while continuously updating the animation. In this simulation, it allows other tasks like control signals collection to be performed simultaneously. This is also an adaptive way to ensure that the animation won't cause problems on lower speed computers. The drawback of this method is that it simply invalidates the whole client area to force Windows to redraw the entire scene, which is sometimes inefficient. Rendering can be faster with this method if the specific area of the scene that needs to be redrawn can be determined and specified in the `InvalidateRect()` function.

3.2.4 Scene Construction

A scene may consist of many objects. Every 3D model will eventually become 2D images in a buffer and be shown on the screen. In other words, because the screen is a raster device, any 3D model must be transformed into a set of points with colors in memory. OpenGL performs this job. It accepts commands specifying how a certain result should be drawn on the screen. Specified in OpenGL's model coordinate system, a vertex goes through a number of steps before it turns up as a pixel on the screen. Figure 3.4 shows the steps that a vertex goes through on its way to the screen buffer. The final result of the vertex is a combination of its color, material, lighting effects, texture if any, and results of blending. In this sense, OpenGL is more of a renderer than a modeler. A complex model can consist of tens of thousands of vertices, lines or polygons. Primitives like points, lines and polygons can be drawn using OpenGL. These basic shapes are to be combined to build the required 3D image.

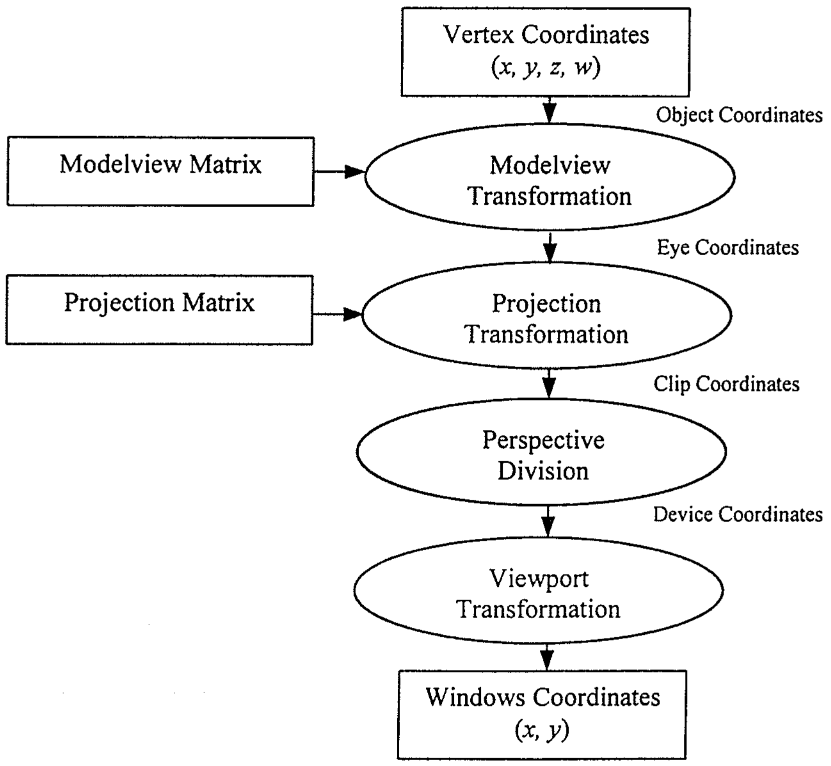
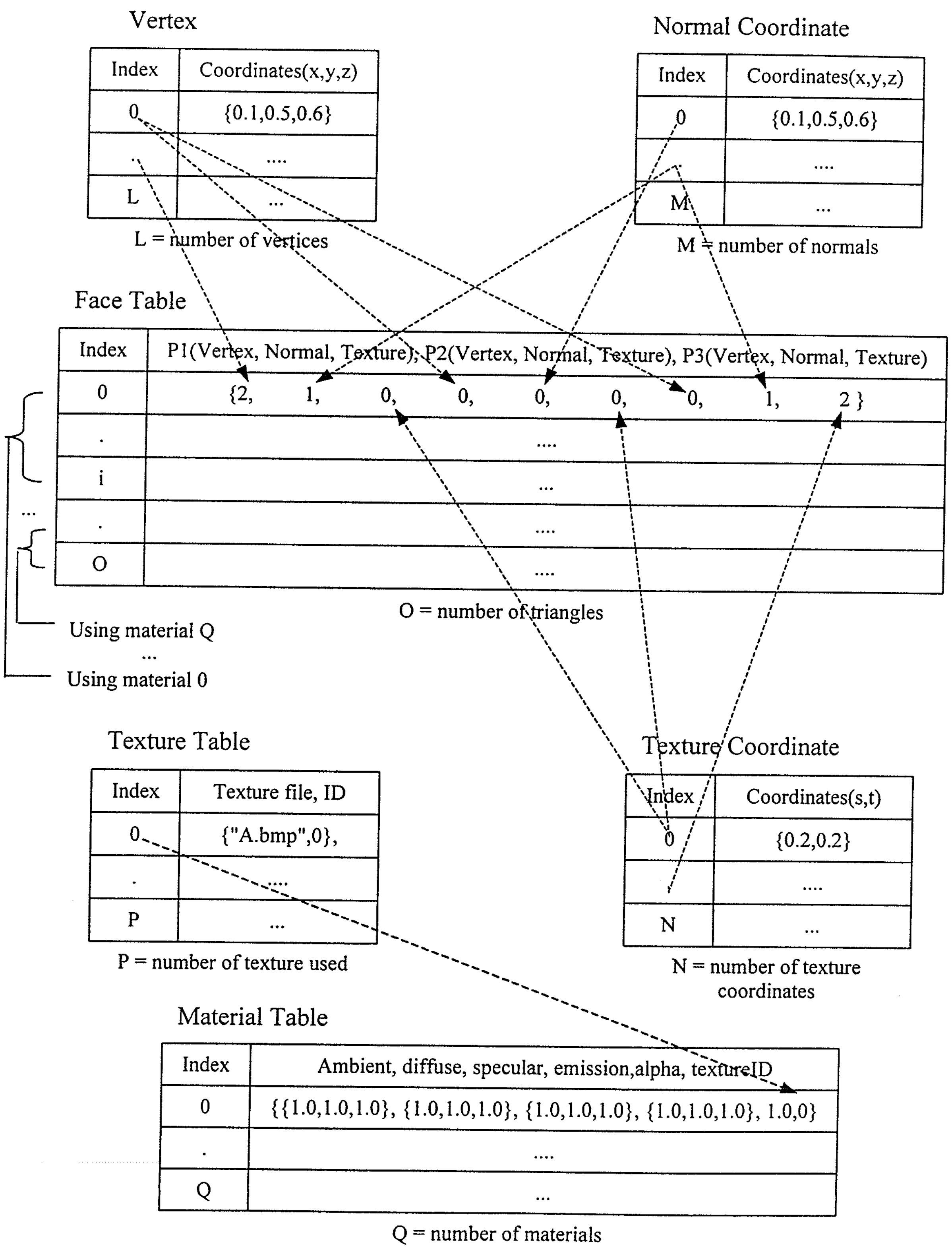


Fig. 3.4: Transformation of a vertex into window coordinates.

To construct the scene, all 3D models are first created in AutoCAD and then imported into the simulation program in form of display lists. A display list is a group of OpenGL commands that have been stored for execution. In that list, all of the vertices, lighting, calculations, textures, and matrix operations stored are calculated when the list is created. Only the results of the calculations are stored in the display list. When the display list is called, the results can be used directly. Because this program draws the object parts repeatedly, using a display list can increase the rendering speed significantly.

Objects created in AutoCAD are saved in 3DMax's 3ds format. Then, the data is extracted from the 3ds file. Tables are created to hold the details of the vertex coordinates, normal coordinates, texture coordinates and different materials. The data structure is shown in Figure 3.5. When the program creates the display list, it first initializes all the textures. All bitmap files in the texture table are loaded sequentially. OpenGL textures are generated from the bitmaps and bound with a texture name for later use. Next, the program reads data from the face table, where indices of vertex, normal and texture coordinates are stored. Base on the data, a set of GL commands is executed to generate the display list. The flow chart for generating the display list is shown below in Figure 3.6.

Geometric shapes fall mainly into two categories: those that consist of a combination of elementary geometric surfaces and those that cannot be expressed with primitives. Since it is difficult to model the cables directly with primitives provided in OpenGL (such as line segments), another tool is used here to represent the deforming curves of the cables.



Vertex

Index	Coordinates(x,y,z)
0	{0.1,0.5,0.6}
.	....
L	...

L = number of vertices

Normal Coordinate

Index	Coordinates(x,y,z)
0	{0.1,0.5,0.6}
.	....
M	...

M = number of normals

Face Table

Index	P1(Vertex, Normal, Texture), P2(Vertex, Normal, Texture), P3(Vertex, Normal, Texture)
0	{2, 1, 0, 0, 0, 0, 0, 1, 2 }
.	....
i	...
.	....
O	....

O = number of triangles

Texture Table

Index	Texture file, ID
0	{"A.bmp",0},
.	....
P	...

P = number of texture used

Texture Coordinate

Index	Coordinates(s,t)
0	{0.2,0.2}
.	....
N	...

N = number of texture coordinates

Material Table

Index	Ambient, diffuse, specular, emission,alpha, textureID
0	{{1.0,1.0,1.0}, {1.0,1.0,1.0}, {1.0,1.0,1.0}, {1.0,1.0,1.0}, 1.0,0}
.	....
Q	...

Q = number of materials

Fig. 3.5: Data structure for constructing triangle mesh.

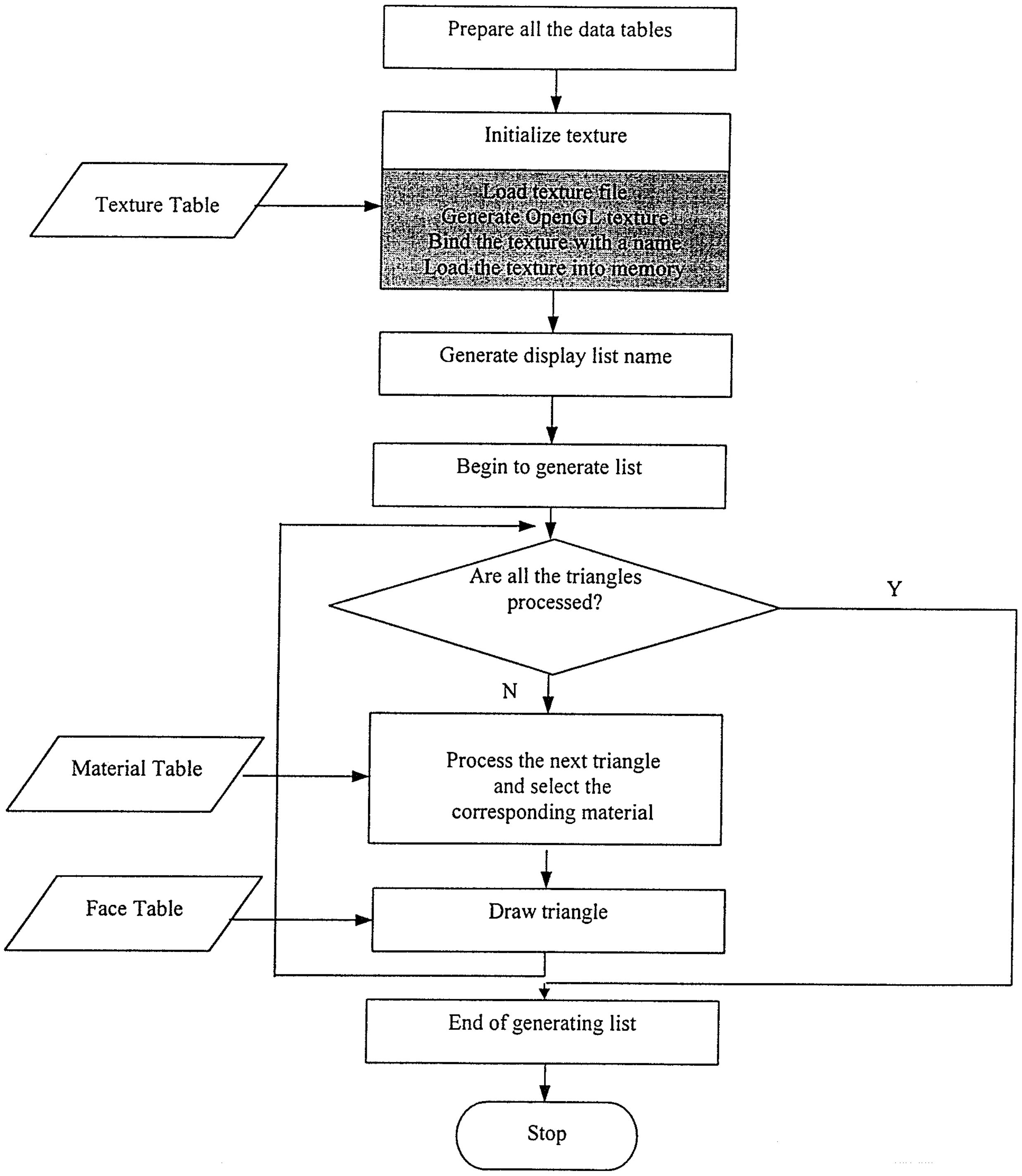

Prepare all the data tables
Initialize texture
Load texture file
Generate OpenGL texture
Bind the texture with a name
Load the texture into memory
Texture Table
Generate display list name
Begin to generate list
Are all the triangles processed?
Y
N
Material Table
Process the next triangle and select the corresponding material
Face Table
Draw triangle
End of generating list
Stop

Fig. 3.6: Algorithm for generating display list.

The tool is called Non-Uniform Rational B-splines (NURB) (Farin, 1990). Although the program keeps tracking to which extent they are loosened or tightened, it doesn't make much sense to precisely show the shape of the cables based on these values. Modeling of loose cables can be a complicated topic in and of itself. Here the loose cables are only graphically shown.

As illustrated in Figure 3.7(a), there are four cable segments that are represented by the NURB curves: the lower parts of the two upper cables and two lower cables. The NURB curve of the upper cables are modeled as curves of order 4 (Farin, 1990). As seen in Figure 3.7(b), five control points are defined for each of them, namely c_1 to c_5. The knot vector is {0.,0.,0.,0.,0.5,1.,1.,1.,1.}, so the curve coincides with control points c_1 and c_5. When the program detects that the cables are getting loosened, based on how much they loosen, control points c_2 and c_4 move from their original positions with an offset. The offset is proportional to how much the cables are loosened. In this way, although the lengths of loose cables are not precisely shown in the simulation, the users can still see how much the cables are getting loosened. For the lower cables, it is almost the same. The only difference is that the lower cables are modeled as curves of order 3 with 3 control points.

In order to construct a realistic scene, there are other tasks that have to be completed. These involve modeling of the scene, creating light sources, selecting proper lighting model, defining material properties, blending two or more objects to obtain special effects, and so on.

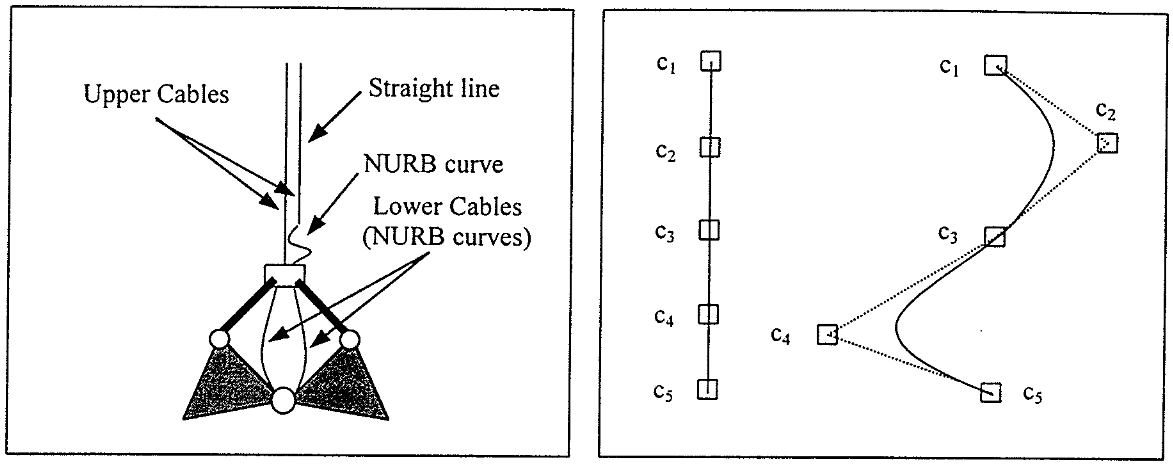


Fig. 3.7: (a) Cables layout; (b) control points for upper cable curve.

3.2.5 Scene Viewing

Once the scene is constructed, it is very convenient to view the scene from any desired angle. Multiple views are employed to build a better human-computer graphics interface. Five views have been generated for this simulation. They are called Full View, Top View, Window View, Focus View and Total View. User can switch from one to the other very easily with only a button click. The purpose of the view design is to facilitate the user with better knowledge of the working status of the claming device and crane so that they can be familiarized with the machine operation more quickly. The five views are described below:

<1> *Full View*. The crane and the claming device together with the working environment are shown in this view. The user can view the operating machine from any angle.

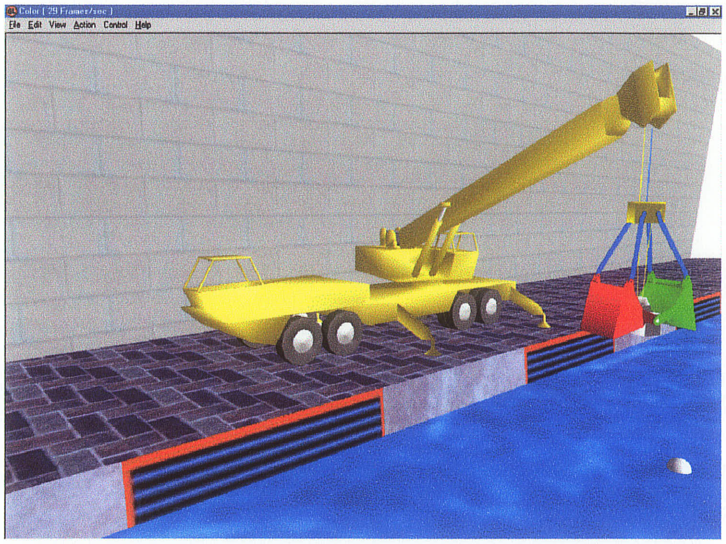


Fig. 3.8: Full View.

<2> *Top View.* With orthographic projection, grid lines are drawn on the surface of working area without deformation. This view will help the operator to place the claming device in the desired position.

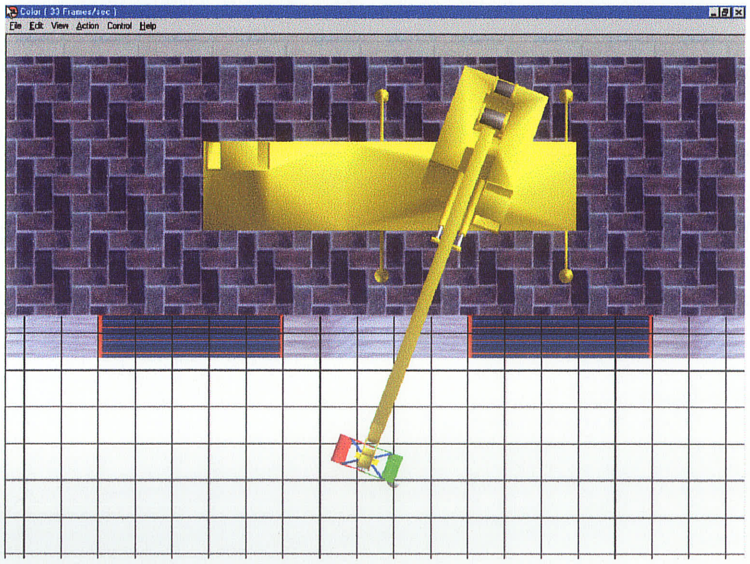


Fig. 3.9: Top View.

<3> *Window View*. This view simulates what the operator would see when he/she sits in the cabin and looks though the window. This is useful if the simulation is used for training operators.

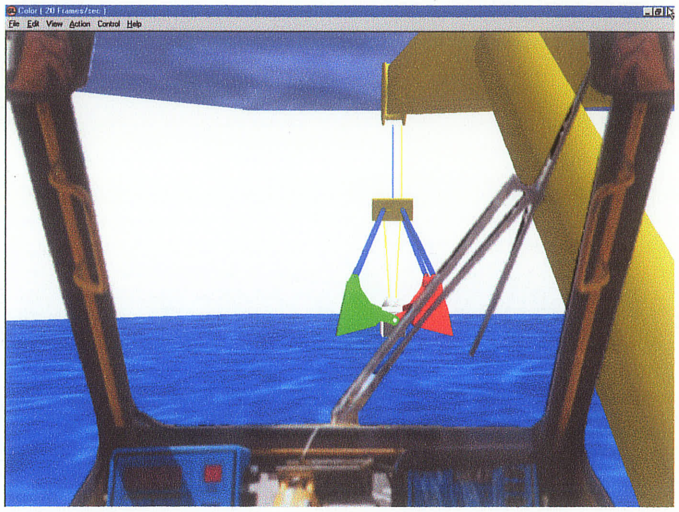


Fig. 3.10: Window View.

<4> *Focus View*. Details of the claming device from a close position can be seen in this view. Because the viewpoint is always following the claming device, information about its working status is still perceivable even when it is working under water.

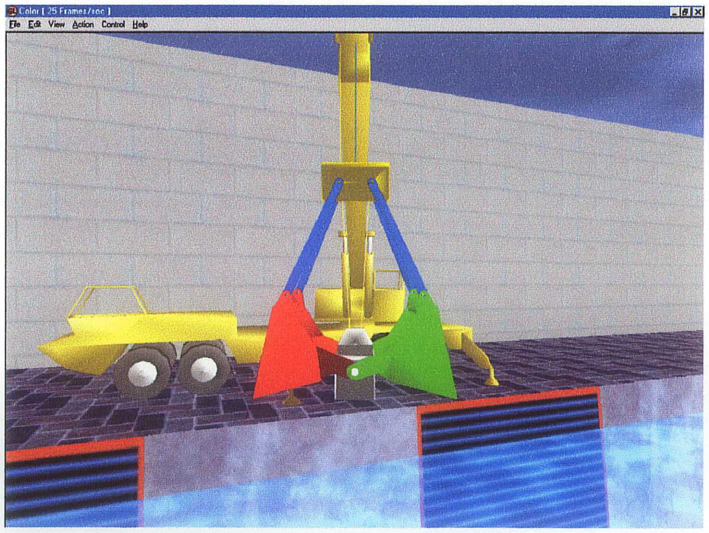


Fig. 3.11: Focus View.

<5> *Total View*. This view contains all four other views and shows them together on the screen. User may choose this view to have overall information about the claming machine when performing tasks.

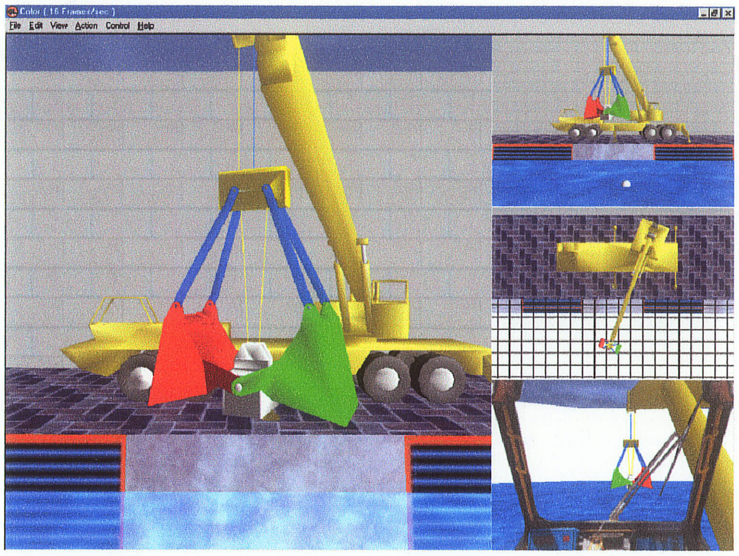


Fig. 3.12: Total View.

Different views are achieved by changing the viewpoint, projection method, and manipulating the model-view matrix. For different views, the drawing routine is optimized to speed up the rendering. Based on the relationship between the viewpoint and the scene, parts of the graphic image that are hidden when the viewpoint moves beyond certain range are detected. The simulation program stops issuing drawing commands when such images become invisible. Since rendering of complex objects with a huge number of triangles consumes a large share of CPU time, using this method greatly increases the refreshing rate of the animation. The enhancement is most obvious when the Total View is chosen, where all four views are rendered on the screen and the scene has to be rendered four times.

3.3 Control Signal Collection

As illustrated in Figure 3.13, the user interacts with the virtual machine with certain input devices. Input devices are used to produce control signals to the crane as well as the claming device. On the actual machine, the operator uses 5 joysticks to control its functions. Each joystick corresponds to one of the motions of the swing, the boom, the telescope, cable one and cable two of the claming device. The control of the virtual claming device is done with one multi-axis joystick and two single axis joysticks. This is analogous to the real operation of the claming device. The joystick used for controlling the virtual crane is a WingMan Extreme Digital 3D model from Logitech Inc. The handle of the joystick provides four axis of rotations, seven programmable buttons, and an 8-way hat switch.. Non-occupied axis and buttons might be used for the claming device or other usage. The single-axis joysticks are Model 220 with trigger style pistol grip from P-Q Controls Inc. The output of the joysticks ranges from –5 volts to +5 volts. A CIO-DAS1602/16D/A-A/D board converts the analog outputs of the handles into digital signals. The layout and functions of the joysticks are shown in Figure 3.13.

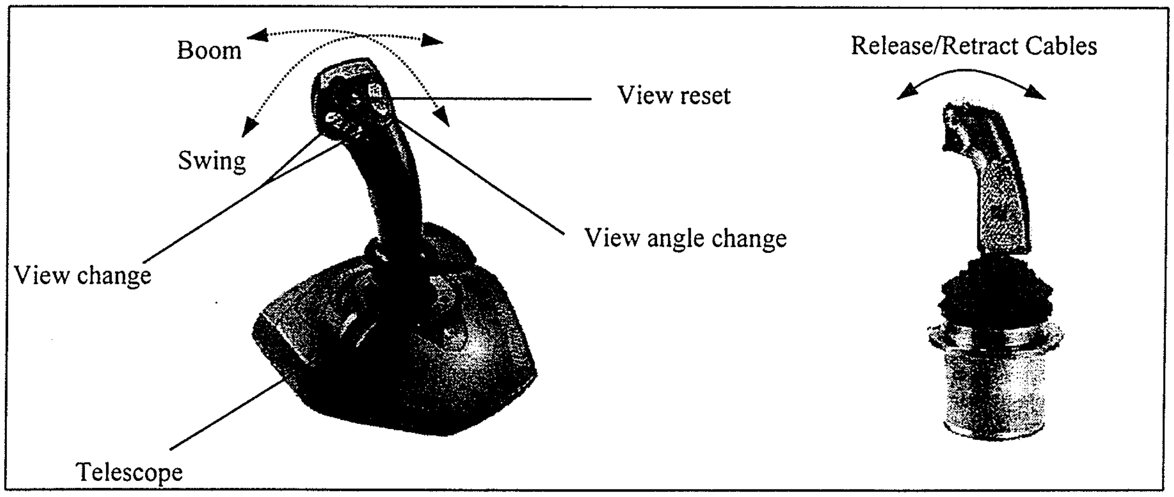


Fig. 3.13: Joysticks used for crane and claming device control.

All control signals are collected by the PC as discrete signals. The sampling rate should be sufficiently small and constant to ensure stable performance of the simulation. The normal timer service provided by the MFC is not appropriate in this case because of poor resolution (minimum 50 ms on a PC running Windows 98) and lack of consistency. The normal timer message has a low priority so the processing of this kind of message could be delayed. Multimedia timer services allow applications to schedule timer events with the greatest resolution possible for the hardware platform.

Timer events are started using the `timeSetEvent()` function. This function returns a timer identifier that can be used to stop or identify timer events. A periodic timer event occurs every time a specified number of milliseconds has elapsed. The interval between periodic events is called the *event delay*. Here the event delay is manually set to be 50 milliseconds. The relationship between the resolution of a timer event and the length of the event delay is important in timer events. With a resolution of 5 milliseconds and an event delay of 50 milliseconds, the timer services notify the callback function after the interval ranging from 45 to 55 milliseconds. So the sampling frequency is approximately 20Hz. When processing the timer event, the program queries the state of each button and handles, and then converts the states into specific values. Based on these values, the program switches the views, changes the viewpoints, and controls the virtual crane and the virtual claming device.

3.4 Further Improvement of the Graphics

In the mathematical model of the crane and claming device, the swinging motion of the claming device is generally ignored because of the increased complexity of the model despite a minor impact on the simulation results. However, without showing this behavior, the claming device is like a rigid body attached to the crane. This seems very unreal. Critical Mass Lab's simulation toolkits consist of a Dynamics toolkit and a Collision toolkit (MathEngine, 2001). With these two toolkits, the swinging motion can be graphically simulated, as well as more natural behavior of other objects in the environment. Moreover, since collision detection plays a vital role in an interactive simulation, the addition of collision detection capability can lead to much more complex interactions between the machine and working environment. For systems where objects are rendered as polygonal meshes, collision detection is performed by detecting intersections between polygons (Garcia-Alonso et al.; 1994). Simulations of situations such as picking up a log or some other object can only be implemented with collision detection.

3.4.1 Critical Mass Lab's Simulation Toolkit

The Dynamics Toolkit (Mdt) is designed to provide a set of functions for simulating rigid body behaviors. The Collision Toolkit (Mct) is designed to detect contacts among geometrical bodies in a certain space, and the Simulation Toolkit (Mst) acts as a bridge between Mdt and Mct, automates the process and manages memory usage. In this thesis, both the dynamics and collision toolkits are used.

Dynamics Toolkit:

All *bodies* are treated as rigid bodies in the Mdt. Rigid bodies represent objects that have finite extent and never deform at all. They are defined with fundamental physical properties as well as extra physical properties such as surface conditions.

Physical properties are mass and inertia tensors. Surface conditions are friction and restitution coefficients. Rigid bodies have kinematic attributes describing their position and movement. Net applied forces and torque are also attributes of rigid bodies in the Dynamics Toolkit.

Collision Toolkit:

The function of the Mct is quite straightforward. It determines whether any two bodies are in contact. The Mdt may then uses this contact information to make the two bodies behave appropriately.

3.4.2 Generation of Swinging Motion

Since we are only interested in the swinging motion of the claming device, it is not necessary to model all the objects in the scene. To generate the swinging motion, only two Mdt bodies need to be created. As seen in Figure 3.14, Body$_0$ is attached to the crane arm tip, and Body$_1$ represents the claming device. A spring joint is used to connect these two bodies. The spring joint is set with two hard limits, which means the spring cannot be stretched or compressed, so that it acts like a steel cable. The position of Body$_0$ and the length of the spring joint are updated continuously, according to the positions of the crane arm tip and the claming device, respectively, at every time step. The Mdt is responsible for updating the position of Body$_1$ based on the internal dynamics calculation. Body$_1$, which represents the claming device, can rotate freely with respect to Body$_0$. In this way, the claming device is seemingly hanging below and swinging under the crane arm tip, and a real swinging motion is generated.

Drawing of the swinging cables and the claming device is implemented in a subroutine. The subroutine draws the cables and the claming device along the current coordinate system's Y axis. The upper points of cables are located at the origin of the coordinate. When drawing the swinging claming device, the program simply rotates the current coordinate and then calls the drawing subroutine. Based on the relative position between Body$_0$ and the Body$_1$, the rotation matrices are calculated. As seen in Figure 3.14, coordinate O_1' represents the object coordinate before rotation; coordinate O_1 represents the object coordinate after rotation.

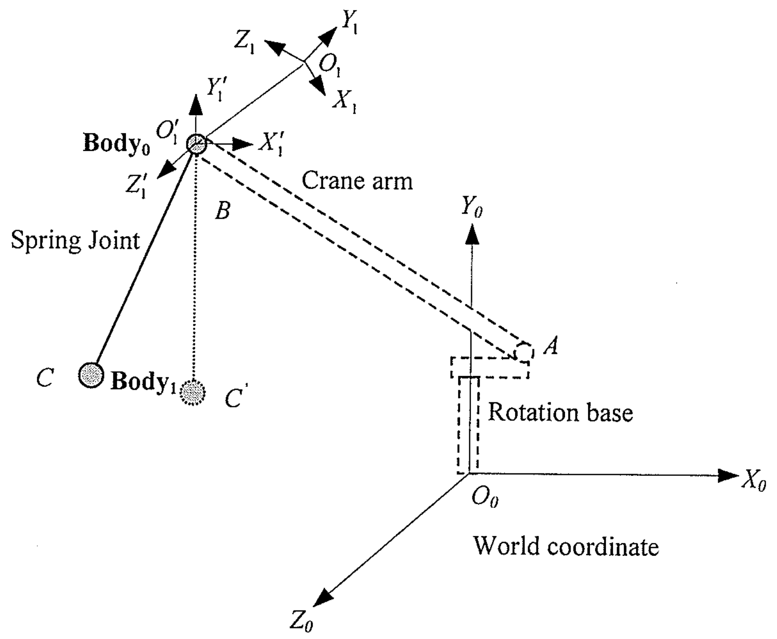


Fig. 3.14: Models for swing motion generation.

The orientation of the claming device is defined by two normalized vectors, $\hat{U}$ and $\hat{V}$. $\hat{U}$ aligns with vector $\overrightarrow{CB}$. $\hat{V}$ is defined as perpendicular to the plane ABC and pointing outwards. Because the initial orientation of the claming device, which is represented by the vector $\hat{U}'$ and $\hat{V}'$, is known, ($\hat{U}'$ aligns with vector $\overrightarrow{C'B}$. $\hat{V}'$ is perpendicular to plane ABC' and pointing outwards) The desired rotation matrix is easily obtained by combining two rotation matrices: $\boldsymbol{M}=\boldsymbol{M}_1\boldsymbol{M}_2$, where $\boldsymbol{M}_1$ is the rotation matrix that rotates $\hat{U}'$ to $\hat{U}$, and $\boldsymbol{M}_2$ is the rotation matrix that rotates $\hat{V}'$ to $\hat{V}$. Note that here the sequence of multiplication does not affect the final result because the two vectors $\hat{U}$, $\hat{V}$ and $\hat{U}'$, $\hat{V}'$ are perpendicular to each other. The MathEngine toolkits provide a few methods to facilitate the calculation of the rotation matrices $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$. The function `MeQuaternionForRotation` returns quaternion A, which rotates vector A to vector B along a vector perpendicular to both of them. Another function, `MeQuaternionToTM(),` converts the quaternion to rotation matrix.

3.4.3 Pick-and-Place Simulation

The virtual claming device should be able to perform pick-and-place tasks when the simulation program runs. This is accomplished via contact detection using Mct. Only some of the object pairs are of interest to perform contact detection. Currently, contact information is required only for pairs like the claming device and the load, or the load and the working environment (the hydro dam). Collision models for the load can be in any shape, for example a cylinder representing a piece of wood, or a particle system representing a pile of debris. An object with a ball shape is put into the simulation as the load for convenience, which can be replaced by some complex models later. The collision models' composition for the claming device and working environment are shown in Figure 3.15. The claming device's collision model includes two identical parts, left half clam and right half clam. Each of them is composed of four *ConvexMesh* type of primitives denoted by A, B, C and D. Both parts associate with the same dynamic body, Body_1. The collision model for the working environment is composed of three *Box* type and one *Plane* type of primitives. The working environment collision models are set as static models because of the fact that they never move to save some computation cycles. The claming device's collision model is much simpler than its graphics model. The Contact detection process would be costly if the graphics model, which is composed of thousands of triangles full of details, is used directly as the collision model. The simulation shows that the simplified collision model coincides well with its graphic representation.

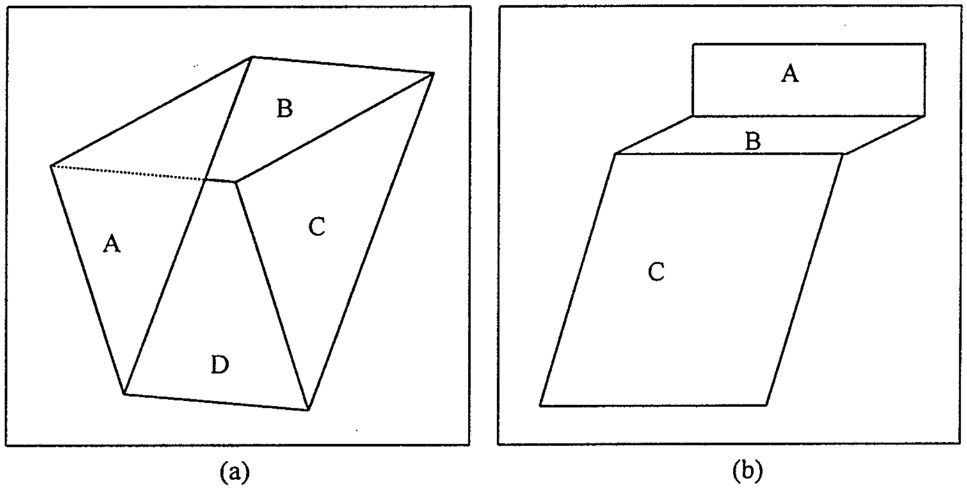


Fig. 3.15: Collision models for: (a) left part of claming device; (b) working environment.

The coordinates of the collision model have to match that of the 3D graphics model and the associated dynamic body. This involves a three-way synchronization among the dynamics models for dynamics calculation, the graphics model rendered on the display, and the collision model used for determining contacts. As illustrated in Figure 3.16, the synchronization is done by data sharing among different modules. After the Runga Kutta integration subroutine, the dynamics calculation subroutine yields the geometry data of the crane's status, i.e. θ_1, θ_2, d_3, and that of the claming device, i.e. l

and α. l is the length of the tightened cable defining the position of the claming device, and α describes the opening angle of the bucket. From the crane's geometric data, the position of the crane arm tip is determined. The Mdt body dynamics calculation module takes this information to update the position of Body$_0$. The output of this module is the updated position and orientation of both the claming device and the load. Finally the geometry data of all objects in the scene is shared by the contact detection module and the rendering module. Both modules need to know precisely the position and the orientation of all the objects with which they interact.

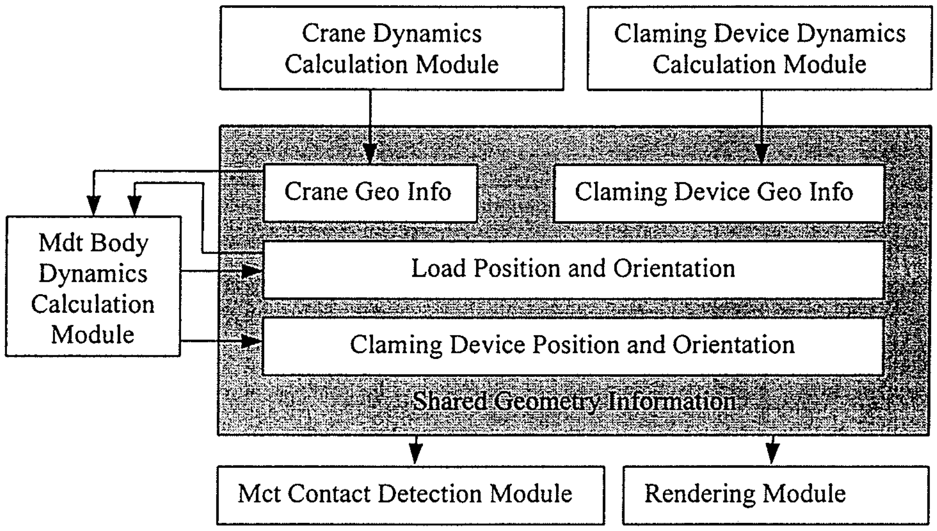

Crane Dynamics Calculation Module
Claming Device Dynamics Calculation Module
Crane Geo Info
Claming Device Geo Info
Mdt Body Dynamics Calculation Module
Load Position and Orientation
Claming Device Position and Orientation
Shared Geometry Information
Mct Contact Detection Module
Rendering Module

Fig. 3.16: Synchronization by data sharing among different modules.

Chapter 4

Distributed Computation with CORBA

Performance for a real-time simulation is critical. If the computation of dynamics cannot be done within a specified time or the refresh rate is too low, users may experience unrealistic visual effects. The dynamics calculation of the crane and the claming device is very computationally heavy because of the small integration time step required. Thus, the simulation needs to run on powerful machines. With the popularity of computer networking, however, it is possible to share the burden with multiple computers. This simulation can take advantage of the network and distributes the computation to more than one computer. Therefore, employing network computation lowers the requirement of the hardware platform on which the simulation runs.

Recent advances in commercial software have motivated the development of an appropriate scheme for this real-time simulation. One of them is the advent of Common Object Request Broker Architecture (CORBA) standards for distributed objects. The CORBA standards have emerged as a promising basis for enabling the development of heterogeneous distributed object-oriented systems. The CORBA architecture provides a high-level location-transparent language-independent software bus through which a client object can call the operations of another object (remote or local) without any knowledge of either the location of the server object or the way the server object is implemented (Kim, 1998). Built on top of CORBA technology, a simulation is divided into service provider and service receiver. As a service provider, one computer can serve many other computers that need the service. A well-designed structure offers both flexibility and robustness against network time lags.

4.1 System Analysis

The system structure and data flow are shown in Figure 4.1.

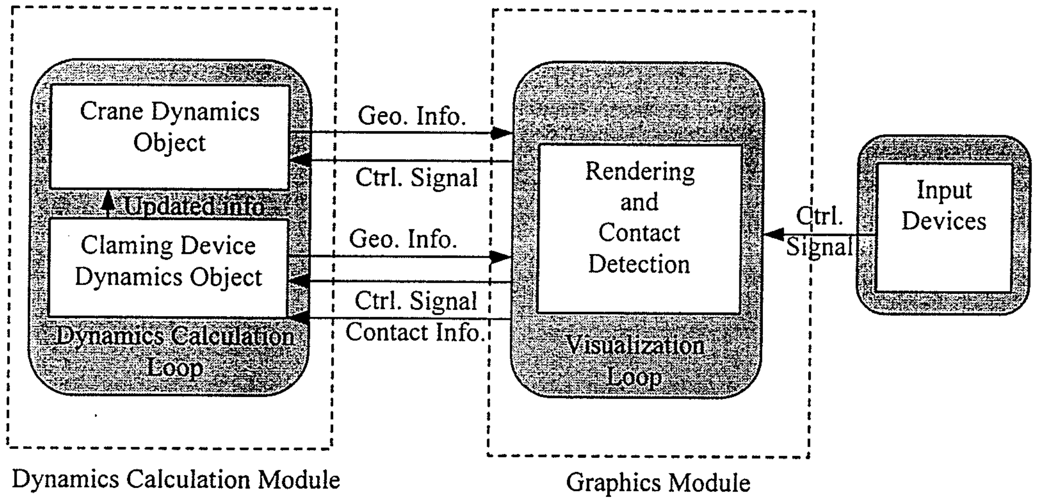


Fig. 4.1: System structure and data flow.

There are two major objects in the dynamics calculation module: the crane object and the claming device object. The dependency relationship between these two objects is that, at each time step, the crane object needs information about the claming device's working state in order to update its dynamic parameters. The data exchange between the graphics module and dynamics module is also shown in Figure 4.1. The graphics module receives the user's inputs via joysticks and keyboard, and then it sends them to the dynamics calculation module. Contact information from the contact detection subroutine in the graphics module is also sent together with the control signals. In return, the dynamics calculation module sends the geometric information back to the graphics module, which is used to construct the scene.

An integration subroutine is put into the dynamics calculation loop and numerically integrates the differential equations. Because of the high stiffness, the integration time step must be very small. Thus, the dynamics calculations are relatively time-consuming. If the dynamics calculation and image rendering are put into a single thread and executed sequentially, based on our tests, the results are not acceptable on a desktop with a Pentium III 550 MHz CPU, which represents a middle class computer in today's market. One possible solution is to use multiple threads of execution to achieve concurrency. Figure 4.2 shows the proposed program layout.

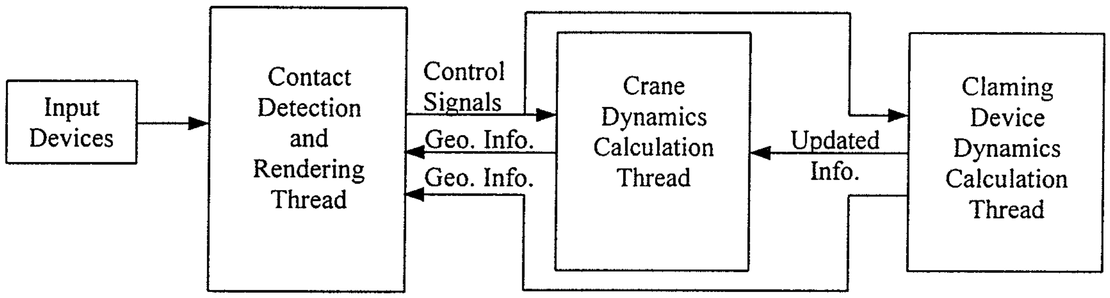

Input Devices
Contact Detection and Rendering Thread
Control Signals
Geo. Info.
Geo. Info.
Crane Dynamics Calculation Thread
Updated Info.
Claming Device Dynamics Calculation Thread

Fig. 4.2: Simulation using multi-threads.

When implemented on the same computer, the animation refresh rate increased from 2 or 3 frames per second (FPS) to an average of 15 FPS. But the computation time for 50 milliseconds of dynamics took more than 100 milliseconds on average. In other words, it cannot meet the real-time requirement. This configuration will greatly benefit from a computer with multiple CPUs. The same program will run faster on more powerful machines, but it is more cost effective to have it run on common desktops. This is feasible with the dramatically improved efficiency and processing power offered by the concept of distributed computing.

Distributed object computing extends an object-oriented programming system by allowing objects to be distributed across a heterogeneous network, so that each of these distributed object components interoperate as a unified one. These objects may be distributed on different computers throughout a network, living within their own address space outside of an application, and yet appear as though they were local to an application. The communication between the distributed objects can be implemented using standard Transfer Control Protocol/Internet Protocol (TCP/IP) and User Datagram Protocol (UDP). This involves low level socket programming that is buried deeply in the source code and hard for other programmers to understand. In the case presented in this thesis, the interface of each object is designed with less data flow use in mind. Since the amount of data passing between the different parts is relatively small, we decided to distribute the interacting objects at a higher level.

Three of the most popular distributed object paradigms are Microsoft's Distributed Component Object Model (DCOM), the Object Management Group's (OMG) CORBA and JavaSoft's Java/Remote Method Invocation (Java/RMI) (Box, 1997, Wollrath *et al.*, 1996). Since Java/RMI can only be implemented using Java, and because of its dynamic interpretive nature, it will probably never have the same performance as a compiled C/C++ code. For an application with a critical time requirement such as the one in this thesis, Java was not considered. CORBA and DCOM have many similarities. They are both platform and language independent, and supporting static and dynamic object invocation. The reasons for choosing CORBA are as follows:

- Binary compatibility is not needed. The only language used in this thesis is C++, which has excellent CORBA support.
- A CORBA program is more portable on different operating systems. On the other hand, DCOM relies on Win32 specific features such as the system registry. Using

CORBA may increase the usability of the simulation program if later the server is ported to a Unix system.

- CORBA is free to use on both Windows and Unix platforms, while DCOM will incur a cost on Unix.

Using CORBA, however, does have some drawbacks. Current CORBA implementations incur extra overheads from data copying, inefficient server demultiplexing techniques, long chains of intra-ORB function calls, and non-optimized buffering algorithms used for network reads and writes. Some investigations have been done to alleviate this problem (Gokhale and Schmidt, 1998). In this thesis we tend to minimize the data flow in the remote function calls, therefore reducing some of the overhead.

4.2 Common Object Request Broker Architecture (CORBA)

4.2.1. Introduction

"The CORBA specification, written and maintained by the Object Management Group (OMG), supplies a balanced set of flexible abstractions and concrete services needed to realize practical solutions for the problems associated with distributed heterogeneous computing" (Henning, 1999)

The OMG was formed in 1989 to address the problems associated with developing a portable, heterogeneous application distribution system. The OMG produced a set of specifications, called the Object Management Architecture (OMA), which has at its core the CORBA specification. The OMA specifies how distributed objects are handled in a platform independent way and how these objects are able to interact with one another. The OMA is split into two related models: the Object Model and the Reference Model.

The Object Model specifies how the interfaces of the distributed objects may be described in a platform-independent way. It describes an object as an encapsulated entity with an immutable distinct identity whose services are accessed only through well-defined interfaces. Clients use an object's services by issuing *requests* to the object (Henning, 1999). The implementation of these services is not important to the calling object and, along with the location of the object, is not directly accessible.

The Reference Model describes how the distributed object interaction is achieved across heterogeneous networks. It provides interface categories that are general groupings for object interfaces. An Object Request Broker (ORB) conceptually links all of these interface categories. The ORB transparently facilitates the communications between the objects, and activates them (if necessary) when they are requested. The Reference Model defines several categories of interfaces. They are all linked together using the ORB communications infrastructure.

4.2.2 General Request Flow

Figure 4.3 shows the abstract data flow model for a client application making a request of a server application. Request passes from client to server as follows:

(1) The client has a choice of two options when making a request. The request can be passed to the ORB through either the *static stubs* or the *Dynamic Invocation Interface* (DII). The static stubs are compiled into the object's interface implementation whereas the DII allows object interfaces to change during runtime. The DII also allows for an

addition of new objects during runtime. For the majority of cases similar to this simulation program, the static stubs are sufficient, since the object interfaces are known at compile time.

(2) The client ORB dispatches the request to the server ORB using the networking infrastructure.

(3) The server ORB, on receiving a request, dispatches the request to the object adapter that is responsible for creating the target object.

(4) The client side object adapter then contacts the servant on the server side, which implements the target object. The server also has the choice of a static or dynamic invocation mechanism when contacting the servant.

(5) After the servant has executed the request, the return values are passed back to the caller object in the client.

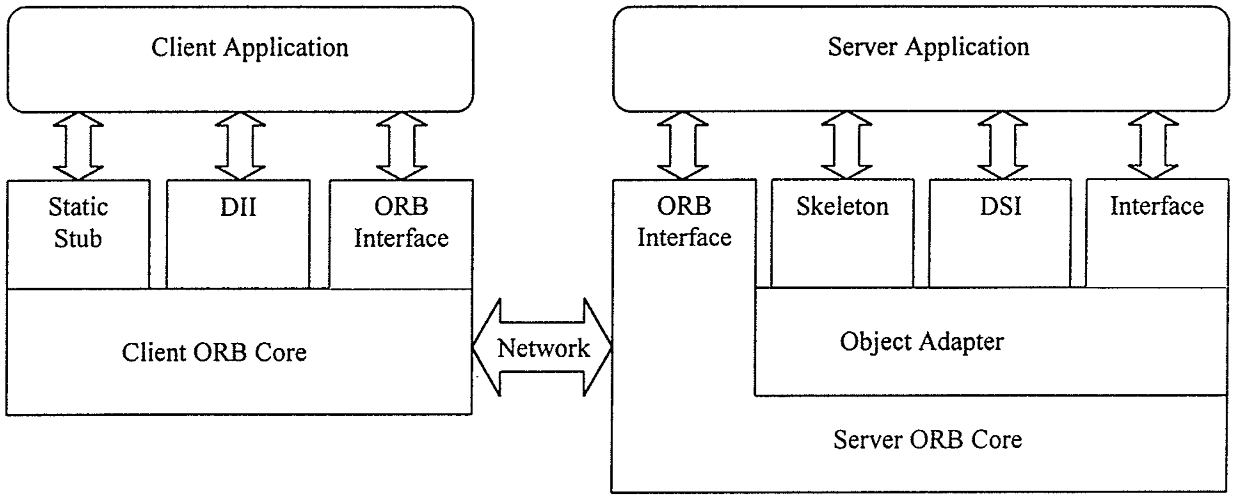


Fig. 4.3: Data flow in CORBA model.

CORBA also implements several different types of request:

- *Synchronous*: Dispatching a synchronous request causes the client to block until a return value is received. This form of request is identical to a *remote procedure call.*

- *Deferred Synchronous*: In this case, the client makes the request, continues processing and later polls for the response.
- *Oneway*: This is the best effort type of request where the client is not assured that the request will be received by the server. The client simply sends oneway requests to the server, and then continues executing the subsequent codes.

4.2.3 Operation Invocation and Dispatch Facilities

CORBA applications operate by receiving/invoking requests on CORBA objects. Within this context, the OMG specifies two general approaches:

• *Static invocation and dispatch*: Here, the IDL is translated into language specific 'stubs' and 'skeletons', which are then compiled into the application programs. This gives the application static knowledge of the programming language data types and functions mapped from the IDL definitions of the remote objects. The stub is a client side object that allows the request to be made to the remote object via a normal function call. In C++, the stub is a member function of a class called a 'proxy' that represents the remote target object in the local application. Similarly, the *skeleton* is a server side object that processes the request and dispatches it to the appropriate servant function.

• *Dynamic invocation and dispatch*: Here, the construction and dispatch of CORBA requests is handled at run-time rather than at compile-time. Information about the interfaces and types of the remote objects is obtained either from a human operator or from an Interface Repository (IR), which is a CORBA service that provides run-time access to IDL definitions.

4.2.4 Inter-ORB Protocols

The communication protocol between ORBs from different venders was standardized with the General Inter-ORB Protocol (GIOP). GIOP specifies transfer syntax and a standard set of message formats to allow independently developed ORBs to communicate over any connection-oriented network. The Internet Inter-ORB Protocol (IIOP) is a GIOP implementation over TCP/IP and must be supported by all ORBs that claim CORBA compliance. Additionally, ORB interoperability requires the use of a standard object reference format. While object references are opaque to the applications that use them, they contain information that all ORBs must be able to understand in order to communicate with the desired object. The standard reference format is called Interoperable Object Reference (IOR) and remains flexible enough to support any GIOP implementation. The IOR identifies one or more supported protocols and, for each protocol, encapsulates the data required to contact the server using that particular protocol.

4.3 CORBA Implementation towards Real-time Simulation

4.3.1 IDL Modeling

Definitions of the interfaces to objects can be defined in two ways. Interfaces can be defined statically in an interface definition language, called the OMG Interface Definition Language (OMG-IDL). This language defines the types of objects by specifying their interfaces. An interface consists of a set of named operations and the parameters specific to those operations. Alternatively, interfaces can be added to an Interface Repository service. This service represents the components of an interface as objects, permitting runtime access to these components. In this thesis, all objects are defined using IDL.

The terms server and client have only relative meanings. A server is a passive entity that offers a service and waits for requests from clients to perform that service. A client is an active entity that obtains service from servers. Here, we call the application where the computation of the dynamics takes place a Service Provider (SP) and the application that performs mainly rendering tasks a Service Receiver (SR). Both the SP and the SR can act as a client or a server.

The interfaces for the SP and SR respectively are defined in IDL as follows:

For the SP:

```
module CLAM
{
      typedef float ControlSignal[5];

      interface CraneClam
      {
            oneway void GetControlInput(in ControlSignal
            voltages);

          void unsubscribe();
      }; // interface CraneClam

      interface CraneClamAdmin
      {
          CraneClam subscribe(in CraneClamClient clientRef);

      }; // interface CraneClamAdmin

}; // module CLAM
```

For the SR:

```
module CLAM
{
      typedef float DisplayData[8];

      interface CraneClamClient
      {
            oneway void getDisplayData(in DisplayData Data);
        }; // interface CraneClamClient
}; // module CLAM
```

The SP contains two interfaces: one is called `CraneClam`; the other is called `CraneClamAdmin`. `CraneClam` is the interface of the dynamics calculation object. The operation `GetControlInput` takes a parameter of type `ControlSignal` as its single argument. No output parameters are provided. The `ControlSignal` data type is actually an array of five floating point numbers. The elements in `ControlSignal` are all voltage signals to the proportional values of [0] crane's swing motor, [1] crane's boom cylinders, [2] crane's telescope cylinders, [3] claming device's main hoist motor, and [4] claming device's auxiliary hoist motor. The other operation defined in this interface is called unsubscribe, which takes no input argument and is simply used to disconnect the existing connection between the SP and the SR. `CraneClamAdmin` is an interface of a factory object. A factory object provides access to one or more additional objects. In this application, all SRs that want to utilize the service of the `CraneClam` object have to ask the factory object `CraneClamAdmin` to produce one for them. The operation defined in `CraneClamAdmin`, `subscribe()`, takes a reference of `CraneClamClient` and returns a reference of `CraneClam`. The CORBA reference will be explained later in this chapter.

The SR contains only one interface, `CLAM`. `getDisplayData` is the only operation defined in this interface. It takes in an array of floating point numbers. The meaning of each element in the array is listed in Table 4.1.

Table 4.1: Definition of elements in type DisplayData.

Index of type DisplayData	**Meaning of the element**
0	Swing rotation angle in degree
1	Boom angle in degree
2	Telescope length in meter
3	Distance between claming device and crane arm tip in meter
4	Opening angle of the claming device in degree
5	Length of cable 1 in meter
6	Length of cable 2 in meter
7	Mass of the load in kg. 0 indicates no load.

The interaction among the interfaces is shown in Figure 4.4.

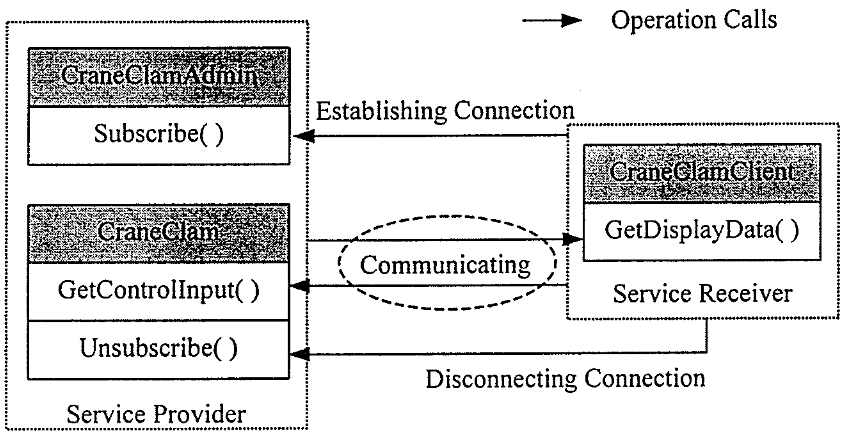


Fig. 4.4: Interactions between interfaces.

Operation `GetControlInput` in interface `CraneClam` and operation `getDisplayData` in interface `CraneClamClient` are defined as *oneway* calls. Operations declared as oneway calls are meant to provide an unreliable send-and-forget delivery mechanism, similar to UDP datagrams. A oneway operation may be lost and never be delivered to the server. For the ORB used in this application, ORBacus from OOC Inc. (OOC, 2000) provides oneway calls that are guaranteed not to block the caller. This non-blocking feature is very desirable in this case despite the fact that oneway calls can be lost if the caller sends them quicker than the server can accept them. In this application, when the client calls the server (either SP calls SR or SR calls SP), the calling rate is 20 calls per second. The benefits of using oneway calls in this application are obvious. First of all, it reduces the dependence between the SP and SRs. The simulation can still run even when network congestion happens. The client program won't block simply because the network is not able to deliver its call to the server. In case of heavily congested network traffic, the position information won't be updated.

4.3.2 Objects Implementation

The interfaces defined with IDL are used to generate the client stubs and the object implementation skeletons. Stubs and skeletons are produced by the IDL compiler. On the client side, in order to access the operations provided by the object, the client program simply includes the source file generated by the IDL compiler. When an object reference enters the address space of a client, the ORB instantiates a proxy object and passes an object reference to that proxy to the client application code. The client can then invoke the operations on the proxy via the reference. With language mapping from IDL to C++, the reference is simply a C++ pointer to the proxy instance. The implementation of the operation in the proxy then takes all the actions that are required to locate the correct server, marshals the invocation onto the wire, and sends it to the server. The ORB instantiates a proxy whenever a new reference enters a client's address space, so clients never create proxies directly. Once the proxy is instantiated, the client can invoke operations on it, and the ORB locates the server and establishes network connections transparently on behalf of the client. Once the client is finished with the remote object, it has to inform the ORB. This enables the ORB to reclaim resources associated with a proxy, such as memory and network connections.

On the server side, the IDL compiler generates separate skeleton header and source files. The skeleton classes defined in those files provide an up-call interface for the ORB into the server application code. Each skeleton class provides a pure virtual function for each IDL operation. To dispatch an incoming request, the server-side run time invokes the corresponding virtual function on an instance of the skeleton class. Since skeleton classes contain pure virtual functions, they cannot be instantiated directly. These pure virtual functions are to be implemented in a servant class derived from its skeleton. The header file of the servant class for the `CraneClamClient` object is as follows:

```
Class CraneClamClient_impl:
      public POA_CLAM::CraneClamClient,
      public PortableServer::RefCountServantBase
{
      CArrayTS* array_;
public:
      CraneClamClient_impl(CArrayTS*);
      virtual ~CraneClamClient_impl();
      void getDisplayData(const DisplayData
            display_temp);
};//class CraneClamClient_impl
```

The functionality of this class is quite simple. It contains a private member, an array class called `CArrayTS`, which provides thread-safe data storage for an array. Thread-safe data storage is achieved by a lock implemented in its method to access the data stored. Once a thread has access to the data, other threads that need to access the data have to wait until the occupying thread releases the lock. Thread-safe data access is important. The server located on the SR side is a multi-threaded application. A remote SP might call the `getDisplayData()` operation to pass in the data for drawing, and the drawing thread retrieves the data. Potential conflict is avoided by implementing thread-safe data access in this case.

The servant class for the `CraneClamAdmin` object does not do much. Once its `subscribe()` function is called, it checks to see if the passed-in `CraneClamClient` reference is valid. If the reference is valid, it creates a `CraneClam` object in the heap and returns the reference of this object back to the caller. The relationship between these two objects and their life cycles is shown in Figure 4.5.

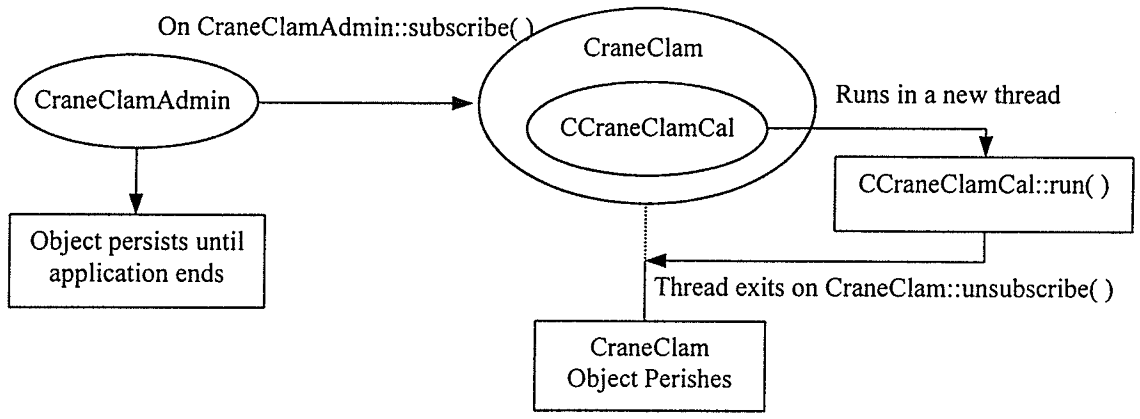

On CraneClamAdmin::subscribe( )
CraneClam
CraneClamAdmin
CCraneClamCal
Runs in a new thread
CCraneClamCal::run( )
Object persists until application ends
Thread exits on CraneClam::unsubscribe( )
CraneClam Object Perishes

Fig. 4.5: Diagram of objects' life cycles.

The servant class for the `CraneClam` contains a `CCraneClamCal` object. This object holds instances of the crane, the claming device, the motor, the cylinder, the valves and the pumps classes. Actual calculations of the crane and the claming device's dynamics takes place in this object's `run()` subroutine. Once the `CraneClamAdmin` creates the `CraneClam` object, an object of `CCraneClamCal` is instantiated. The SR calls the `CraneClamAdmin`'s `subscribe( )` function with its `CraneClamClient` reference. This reference is passed on to the `CraneClam` object and further to the `CCraneClamCal` object. `CCraneClamCal` object uses this reference to invoke the `getDisplayData()` operation and sends the results back from dynamics calculation.

The algorithm implemented in the `CCraneClamCal::run()` subroutine for solving dynamics is shown in Figure 4.6. Except for the initializations, the whole block of code is placed in a `while` loop. The `while` checks the state of `m_done` flag, which is changed only in `CraneClam::unsubscribe()` operation. The SR invokes this operation to set the flag. After the flag is set, the thread terminates its execution and exits. The `CraneClam` object deactivates itself after it detects the termination of the thread. The program sends out the results of the dynamics calculation every 50 milliseconds. This is guaranteed with the use of time sequence control. Within the loop there are variables to mark down the loop starting time and ending time. By checking the difference, the program knows how much time is elapsed during the dynamics calculation. If the time spent in calculation is smaller than the desired 50 milliseconds, the thread will wait until it reaches 50 milliseconds then it continues to run. Time spans of greater than 50 milliseconds are what we try to avoid and seldom happen in our tests. The dynamics calculation thread contains two inner loops for calculating the dynamics of the crane and the claming device. The inner loops run for a certain number of cycles, depending on their steps, to yield results for the next 50 milliseconds. A part of the results of the claming device is used to update parameters for the crane.

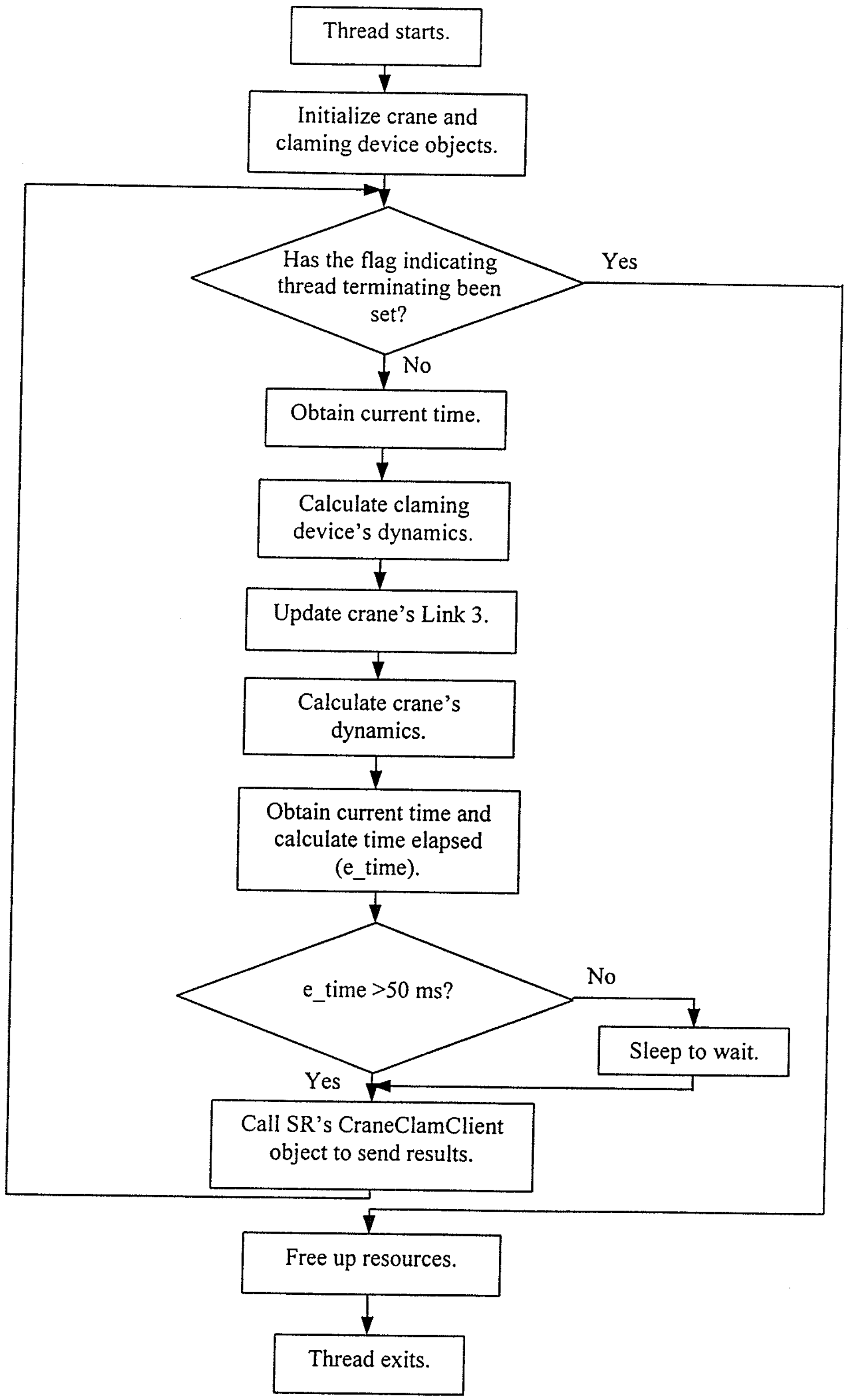

Thread starts.
Initialize crane and claming device objects.
Has the flag indicating thread terminating been set?
Yes
No
Obtain current time.
Calculate claming device's dynamics.
Update crane's Link 3.
Calculate crane's dynamics.
Obtain current time and calculate time elapsed (e_time).
e_time >50 ms?
No
Sleep to wait.
Yes
Call SR's CraneClamClient object to send results.
Free up resources.
Thread exits.

Fig. 4.6: Flow chart of solving crane and claming device's dynamics.

4.3.3 Concurrency Model

Threads can significantly improve an application's structure and can also help make application development more intuitive by delegating specific tasks to threads, which otherwise would have required sophisticated mechanisms to integrate them into a single execution flow. Multi-threading also allows complete use of CPU resources especially in multi processor systems. Applications built upon of CORBA can adopt their own threading strategies, as do the underlying ORBs (Object Request Broker). While an ORB can be used to invoke methods on objects in the same application context, the most common case is to invoke methods on an ORB in a different application context whether on the same system or on a remote system. By doing so CORBA applications already exploit one level of parallelism (Schmidt *et al.*, 2000b). Adding a second level of parallel execution to this by using threads on the client or the server side creates a whole new set of problems. Choosing a good or bad concurrency model can make a big difference on the application's performance. The ORB that was chosen in this application is ORBacus (OOC, 2000), which provides a variety of concurrency models. Each concurrency model provides a unique set of properties with advantages and disadvantages. Based on this knowledge, the adoption of one model for this application is explained in detail.

Since ORBacus allows different concurrency models to be established for the client and server activities of an application, it is possible for us to choose appropriate concurrency models for the SP and SR, respectively. The client-side concurrency models are 'Blocking', 'Reactive' and 'Threaded'. The server-side concurrency models are 'Reactive', 'Threaded', 'Thread-per-Client', 'Thread-per-Request' and 'Thread Pool'.

The blocking model applies only to the client side. This means that the ORB is blocked while sending requests. If the requests sent by the client are oneway calls, sending these requests will not block the ORB. The ORB detects if sending the oneway request can cause blocking at run-time. If it causes blocking, the ORB puts the oneway request into a request buffer first and sends it either when it will not block or when the next request enters the buffer.

The reactive model for servers accepts incoming requests from several clients. It serializes all incoming requests and works by completely finishing one request before paying attention to the next one. All incoming requests are put into a queue. This model is still single-threaded and the server cannot respond to network events while it is busy servicing a request. From the application's point of view, the reactive model for a client is still blocking. Once the client sends a request, it has to wait until the server replies to the request. However, the ORB never blocks; it simply sends out the request and uses a 'select' loop to wait for the reply. The ORB relies on an event-handler called *Reactor* to work properly. *Reactor* is an instance in ORBacus where special objects can register if they are interested in specific events. These events can be network events, such as an event signaling that the data is ready to be read from a network connection. ORBacus provides three reactors for different platforms.

A client with a threaded concurrency model uses two separate threads for each connection to a server, one for sending requests and another for receiving replies. A threaded server uses separate threads for receiving requests from clients and sending replies. In addition, there is a separate thread dedicated to accepting incoming connection requests, so that a threaded server can serve more than one client at a time. ORBacus's threaded server concurrency model allows only one active thread in the user code. This means that even though many requests can be received simultaneously, the execution of these requests is serialized.

The thread-per-client server concurrency model is very similar to the threaded server concurrency model, except that the ORB allows one active thread-per-client in the user code. In the thread-per-request server concurrency model, the ORB creates a new thread for each request. The thread pool model uses threads from a pool to carry out requests, so that threads are created only once and can be reused for other requests.

Note that all the concurrency models mentioned above refer to the ORB's threading models. Application can adopts their own threading models in addition to the ORB's threading model. In the simulation program, both the SP and the SR employ multiple threads in their code while adopting different concurrency models for their client and server. The SP creates a separate thread performing dynamics computation tasks whenever there is a new SR connected to it. The object running in the newly created thread then communicates with the SR using oneway calls. The SR can terminate the connection at any time. The termination of the connection also kills the thread created for the SR. The number of the SRs connected to the same SP is only limited by the power of the computer on which the SP runs. There are two important issues that have to be addressed:

1. The requests, both from the SR to the SP and from the SP to the SR, are all oneway calls except the subscribe and unsubscribe calls at the beginning and the end. Oneway calls do not block execution even the single-threaded concurrency model is used by the ORB.

2. For the SP and the SR, the execution time of a oneway call is relatively small, i.e. it takes little time for the server to service the client request.

Based on these facts, blocking model is chosen for the client side of SR. The ORB does not create threads for sending requests and receiving replies. Thus, the overhead of creating additional threads and thread context switching is reduced. This threaded model is chosen for the client side of SP because of the concurrency model's send-in-the-background effect. On the server side, the situations of the SP and the SR are different. The SR's server only services one client, which is the CraneClam object created in the SP in response to a SR's subscribe call. Therefore, the SR's ORB will not receive simultaneous requests. But the SP's ORB has chances of receiving incoming requests from all connected SRs as well as sending out calls at the same time. For the SR's server, the reactive model is quite appropriate for its high efficiency. A standard reactor is used in the program. In a Windows application, it is typical to use a Windows reactor to handle both the Windows GUI events and the CORBA network events. However, this is not applicable in this case, so another approach is employed. Although in ORBacus's specification, it is said that the ORB has to be put into the main thread of the application, this turns out to be not necessary. As such, the Windows message loop was placed in the main thread and the ORB event loop in the second one. Therefore the reactor can concentrate on CORBA network event only without knowing the existence of any Windows messages.

Since more than one SR can be connected to the same SP, the server of the SP needs to be able to handle many requests in a very short time. The SP is currently intended to run on a PC, which generally has the power to support 4 clients or more at a time. The possible request rate may be up to over 80 calls per second. The execution time for each request is very short; in our test it is less than 1 millisecond in average. The response delay to each request is so small that it can be ignored. The only noticeable delay for incoming requests may take place when a new SR calls the subscribe operation. Establishing the connection and initializing the new servant usually take more time. Therefore the reactive model and the threaded model are both applicable. Each has its own advantages over the other on different platforms. Within a single CPU environment, the reactive model is faster because of less overhead incurred by multi-threading. The threaded model allocates a separate thread dedicated to accept incoming connection requests to minimize the delay. For each client connected, the threaded model provides two threads for receiving requests and sending replies. Thus under a multi-CPU environment, the threaded model yields a faster response to each request. The default model is set as reactive in the application but can be changed with command line arguments. The schematic diagram of the concurrency model is shown in Figure 4.7. Figure 4.7 shows a reactive concurrency model server for the SP. In the figure we can see that all requests are executed in the SP's main thread; therefore, the execution of the requests must be serialized. Thread synchronization for the servants is not necessary. There is a one-to-one relationship between the clients of the SRs and servants in the SP.

4.3.4 Locating Objects with Object Reference URLs

Although the application can obtain the desired object reference from a service call 'Naming Service', the CORBA system has the problem of the "chicken and egg", as do all networking systems. In order for the system to operate, it needs to know the location of the server and most likely the first object reference. The question is how this first reference is resolved in the network.

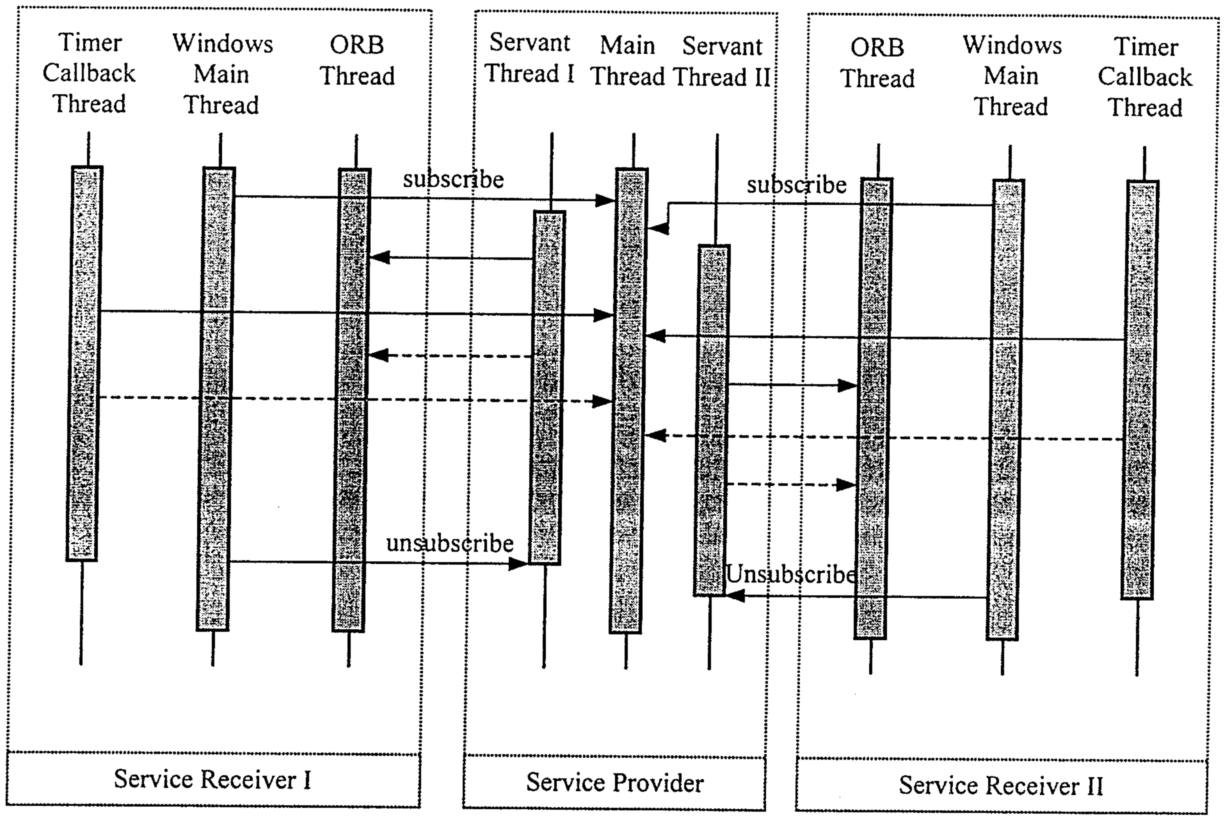


Fig. 4.7: Threads used in Service Provider and Service Receiver.

Most current CORBA systems overcome this problem in one of two ways. One way is to save the object reference in a file. The application that needs to access the object also has access to this file. Whenever the object reference changes, the file needs to be updated and retrieved by the application. The other way is called Resolve Initial References. This involves the ORB itself storing details about CORBA services and passing them on to the application as required. Both methods are effective, but they are not very convenient in this case, because the SP is not supposed to be fixed on a specific server. The approach chosen to this problem is to generate a relatively constant object reference, which can be constructed on the SR side without relying on any SP-generated file. The first object reference remains unchanged until the SP runs on a different machine or a different IP address of the machine where the SP is running changes. The advantage of this approach is its simplicity and flexibility. To achieve this goal, an understanding of the object reference is needed.

Interoperable Object Reference (IOR) contains a number of standardized components that are the same for all ORBs as well as proprietary information that is ORB-specific. To permit source code compatibility across different ORBs, clients and servers are not allowed to see the representation of an object reference. Instead, they must treat an object reference as a black box that can be manipulated only through a standardized interface. An IOR contains the following basic information:

- *Repository ID*: This is a string identifying the derived type of the IOR at the time the IOR is created.
- *Endpoint Inf*o: This field contains all the information needed by the client ORB to establish a network connection with the server ORB. The field contains information about what network protocol to use and the physical addressing information appropriate to the network protocol that is chosen. The endpoint field may contain the actual address of the endpoint server, or it may point to some other implementation repository that contains details about the location of the server.
- *Object ke*y: This field contains information that is proprietary to a particular ORB vendor. The client side simply sends this field as a block of data even though sometimes the client ORB may not be able to decode it.

Using IIOP, an object reference can be thought of as encapsulating several pieces of information including the hostname, the port number and the object key. Each time a server is executed, the root Portable Object Adapter (POA) manager selects a new port number on which to listen for incoming requests. Each object created by a server is assigned a unique key. The order in which the server creates its objects may also affect the keys assigned to those objects.

Object references can be converted to or from a string. The stringified object reference has more than one format. In addition to the first cumbersome 'IOR:' format, two other formats with URL-like syntax were introduced by the Interoperable Naming Service (INS). One is called 'corbaloc:URLs', the other is called 'corbaname:URLs'. In this thesis, corbaloc:URLs is chosen to construct the object reference for the first object in the SP.

The corbaloc:URL for the IIOP protocol has the following structure: corbaloc:[iiop]:[version@]host[:port]/object-ID

The components of the URL are as follows:

- iiop - This is the default protocol for corbaloc:URLs, and therefore is optional.
- version - The IIOP version number in major.minor format. The default is 1.0.
- host - The hostname of the server.
- port - The port on which the server is listening. The default is 2089.
- object-ID - A stringified object ID.

Normally, object keys contain the information necessary to uniquely identify a POA and a servant within the POA. However, the object ID used above does not contain information that identifies both the POA and the servant. To solve this problem, ORBacus defines the interfaces BootManager and BootLocator. The `BootManager::add_binding()` operation binds an object ID to an object reference.

The `BootManager::remove_binding()` operation is used to remove a binding. A `BootLocator` object can be registered with the `BootManager` using the `set_locator()` operation and is used to dynamically locate a reference for a given object ID. Based on the above knowledge, if the SR has the information about the host name, the port number and the object ID, it can construct the stringified object reference in the corbaloc:URLs format and then convert this string into the real object reference. Note that this method has not been standardized by the OMG.

The host name in corbaloc:URLs format can be the canonical host name or numeric IP address. In this application the IP address of the host is used. When the SP starts to run, it will show the host's IP address and port number on the screen. Each time a server is executed, the Root POA manager may select a new port number on which to listen to incoming requests. To prevent the port number from changing every time, in this application the OAport option is set to allow the Root POA manager to use the specified port number. The specific port number is 1015, which is not a reserved port for normal usage. Error messages may be shown in case of the occupation of this port. The first object in the SP is activated using POA with the PERSISTENT life span policy and the USER_ID object identification policy. This ensures that the object references are created with a user assigned ID and always have the same keys.

Chapter 5

Demonstrative Results

In this chapter, various experiments have been conducted to test the performance and reliability of the dynamics models of the claming device and the crane. A typical pick-and-place task has been simulated using the joysticks. The service provider's (SP) performance when connected to different numbers of service receivers (SR) is discussed. The effect of using different threading strategies on the SP and the SR is also illustrated.

5.1 Dynamic Simulation Results

5.1.1 Simulation of Claming Device Model

This section shows the claming device's response to certain step inputs. The control signals are tuned to allow the claming device to operate in all possible working states. Figure 5.1 shows the control signals to the claming device's servo-valves. Signal 1 is applied to tighten or release cable 1, and Signal 2 is applied on cable 2.

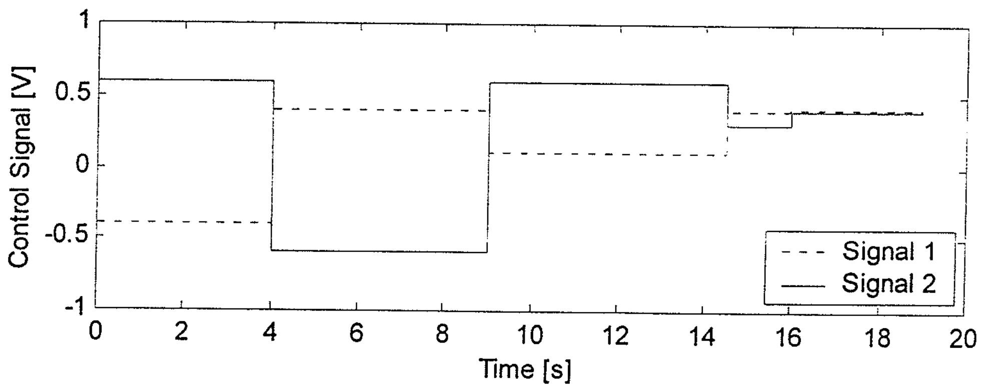


Fig. 5.1: Control signal to the servo valves.

Figure 5.2, 5.3 and 5.4 illustrate the tension applied on cable 1 and cable 2, the angle of θ_2, and the velocity of points A and C (please refer to Figure 2.2).

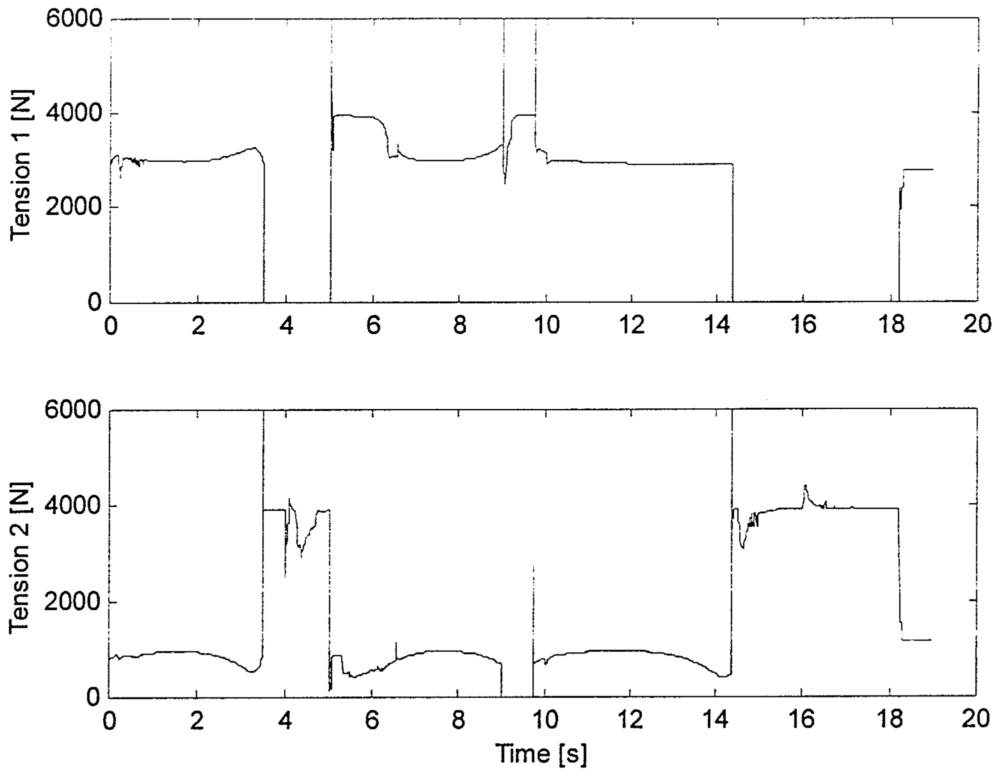


Fig. 5.2: Tensions applied on Cables.

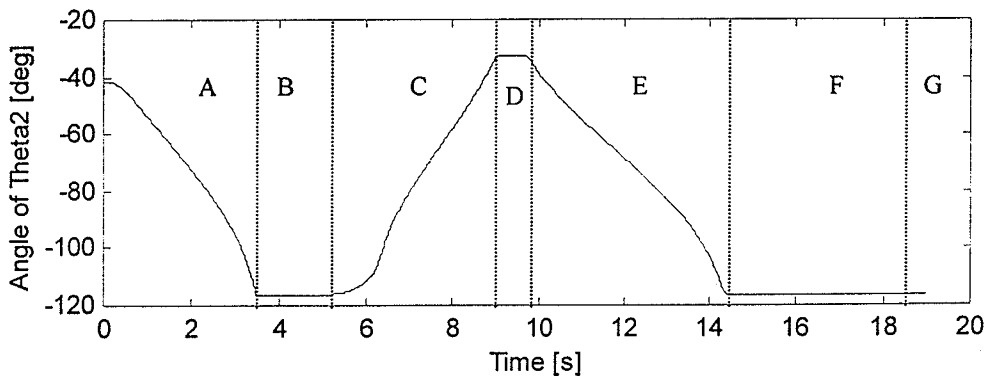


Fig. 5.3: Angle of θ_2.

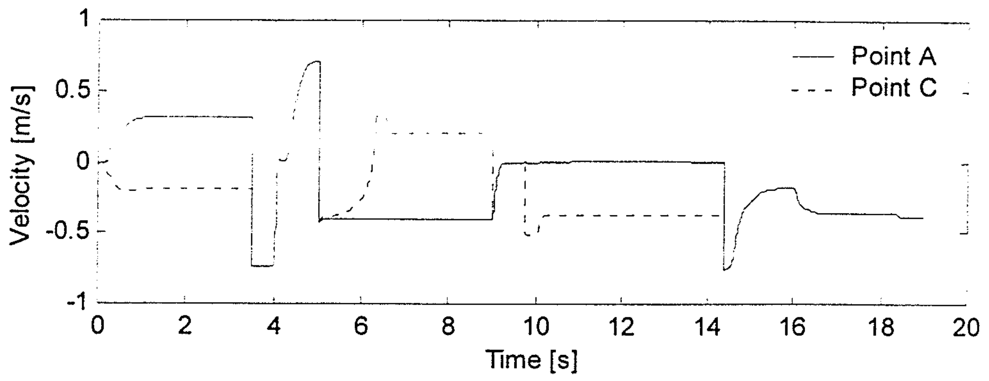


Fig. 5.4: Vertical speed of the claming device's bodies.

From Figure 5.2 to Figure 5.4, we can see the claming device can work in different working states. In Figure 5.4, the curve is divided into 7 phases along the time axis, each of them indicating a different working state. Phase A, C and E are the 'Free' states; B and F is 'Fully Closed'; D is 'Fully Opened'; and G is the "Distributed' state. Initially control signal 1 is negative and control signal 2 is positive. These control signals manipulate the hydraulic motors. One of the motors pulls up the claming device at point A while the other releases it at point C. Therefore the claming device starts to close (Phase A), and the absolute angle of θ_2 becomes larger and larger. (For the notation of θ_2, refer to Figure 2.2.) The claming device becomes fully closed and remains on this working state after 3.5s (Phase B). The control signals change at the fourth second and invert the signs. Consequently, the claming device starts to open until it is fully opened (Phase C). In phase D, the claming device is fully opened. Later it is closed again (Phase E and F), but this time the claming device reaches a steady stage of fully closed while distributing the load to both two cables (Phase G). Every time the claming device changes its working state, it has to pass the state of 'Distributed'. Usually this is a transient state, which ends very quickly. But this state can also be stable. For example, in this experiment, at the final stage this working state is maintained for a prolonged time (Phase G).

5.1.2 Simulation of Crane Model

This section demonstrates the crane's response to a step input signal. The same signal is applied to all three servo-valves for the swing, boom and telescope, respectively. Figure 5.5 shows the control signal. Figure 5.6 to 5.12 illustrate the responses of the links.

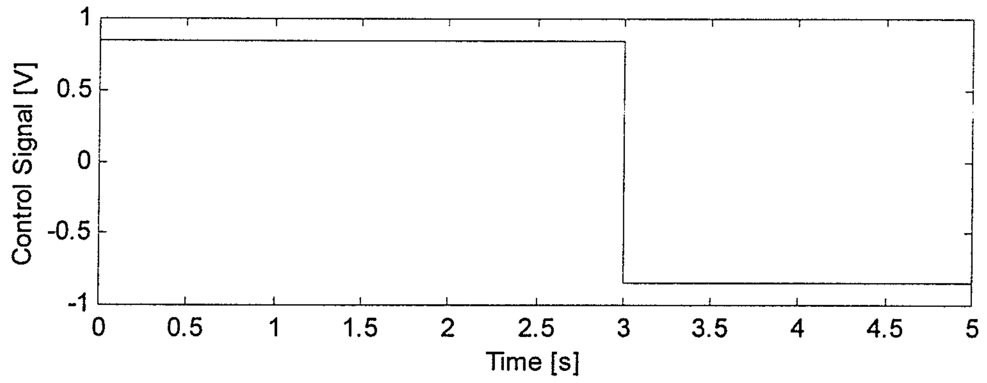


Fig. 5.5: Control signal to the crane model.

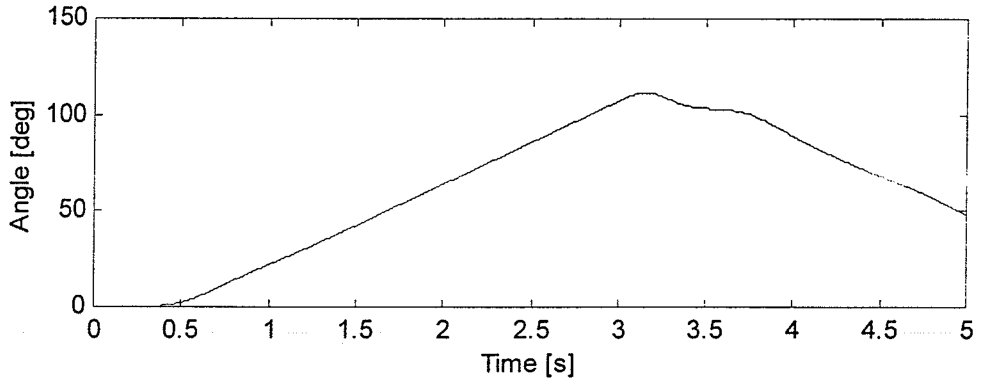


Fig. 5.6: Swing rotation angle.

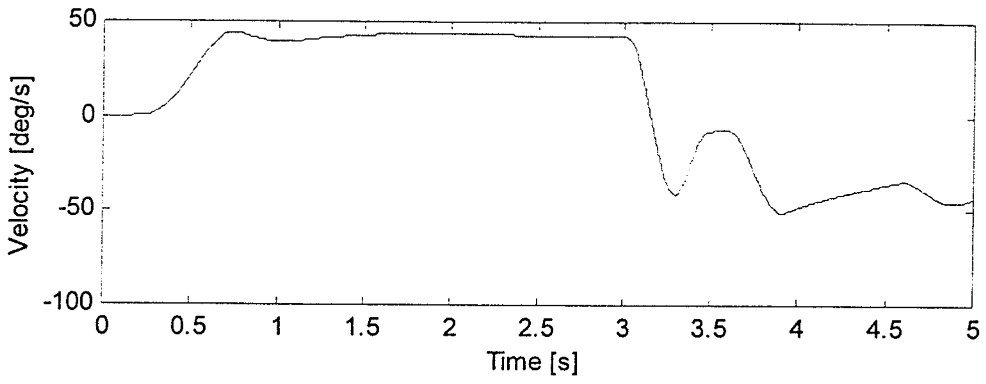


Fig. 5.7: Swing angular velocity.

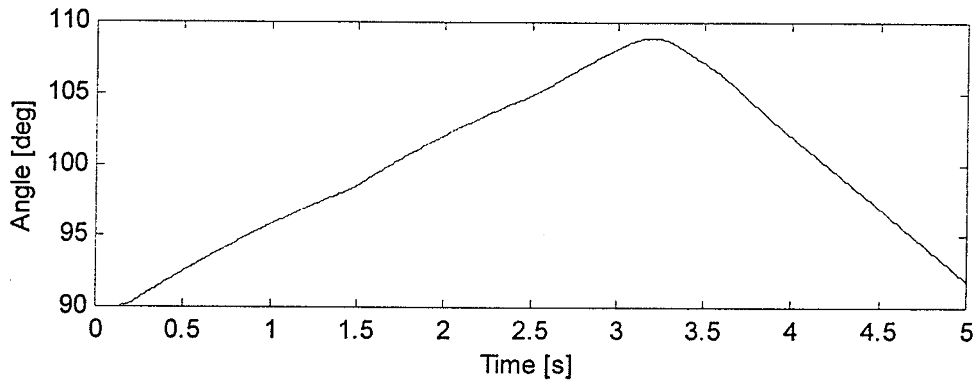


Fig. 5.8: Boom rotation angle.

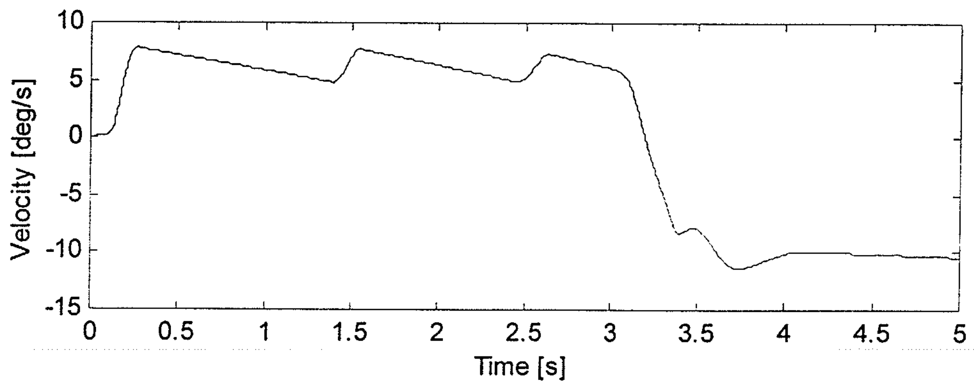


Fig. 5.9: Boom angular velocity.

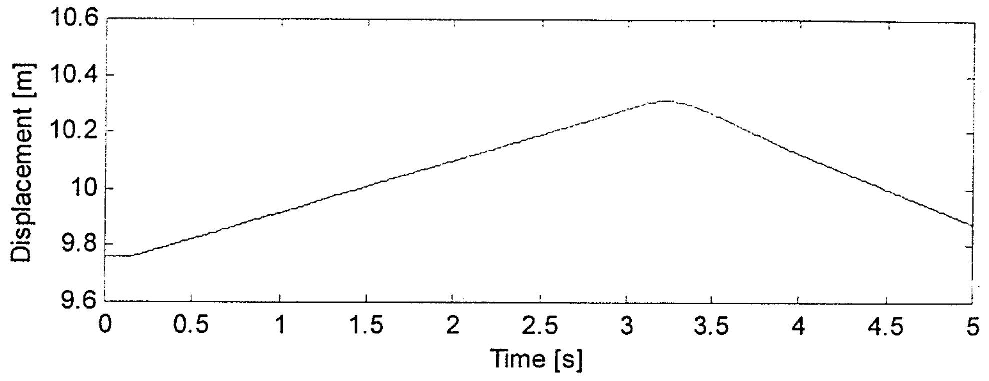


Fig. 5.10: Telescopic arm length.

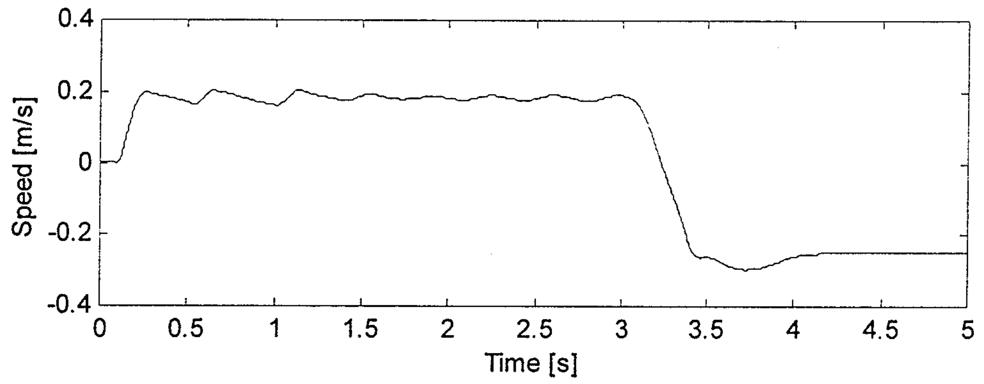


Fig. 5.11: Telescopic arm moving speed.

In the above figures, we can see that some oscilations occur in the swing, boom and telescope moving velocities. This is primarily due to the sudden change of the input signals. The parameters of the hydraulic system were taken from a typical 215B caterpillar excavator machine. The parameters used in the simulation are assumed based on previous knowledge. The parameters, especially those of the hydraulic systems like the valve and cylinders, have obvious and profound effects of the whole system's response.

5.1.3 Machine Performing Pick-and-Place Task

In this experiment, the user of the simulation controls the crane and the claming device, in an online manner, to perform a typical pick-and-place task. An object is located on the water. The crane starts from its home position and the operator manipulates the arm so that the claming device can be placed above the object. Next, the operator lowers the claming device, picks up the object in the bucket, brings it to the appropriate position and releases it. Control signals are generated with the joysticks and control handles.

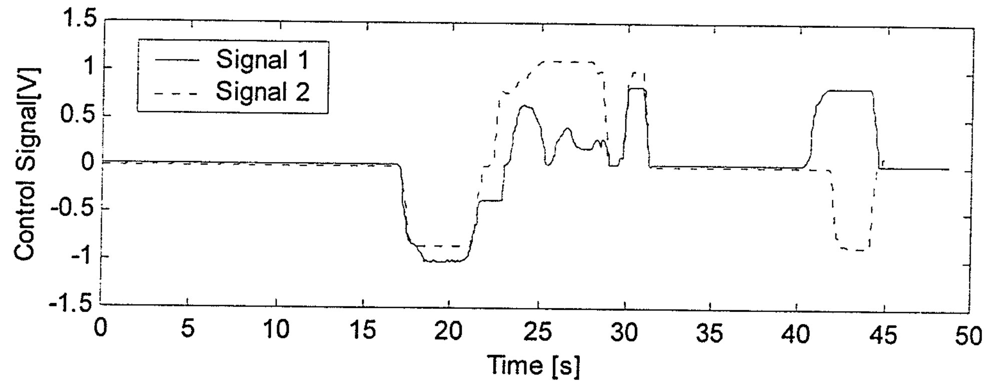


Fig. 5.12: Control signals to claming device.

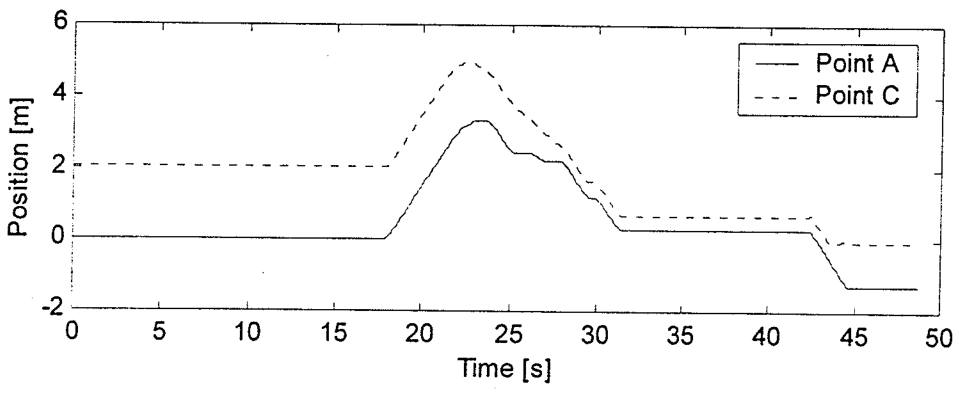


Fig. 5.13: Position of points A and C on claming machine with respect to the water level.

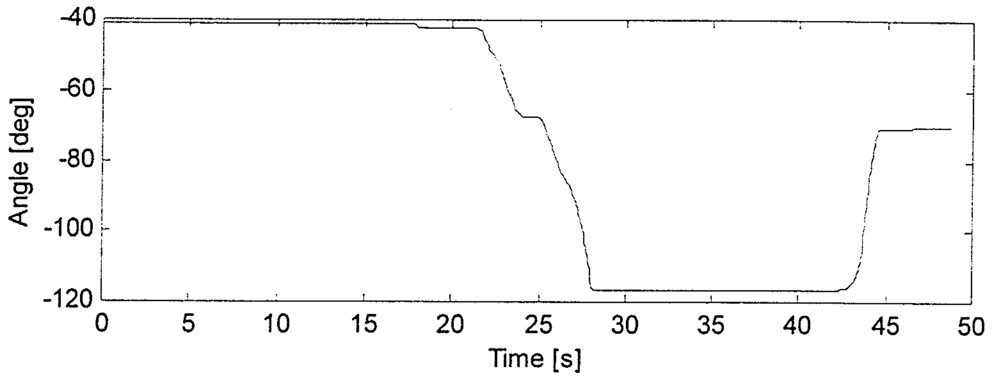


Fig. 5.14: Angle of θ_2.

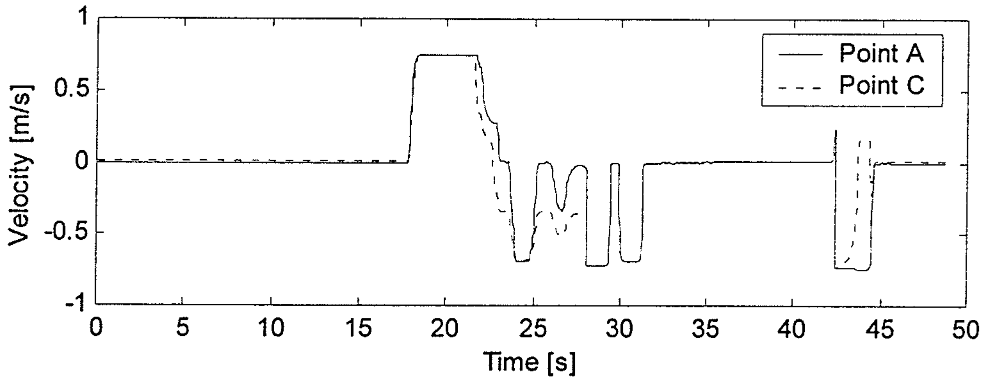


Fig. 5.15: Velocity of points A and C on claming device.

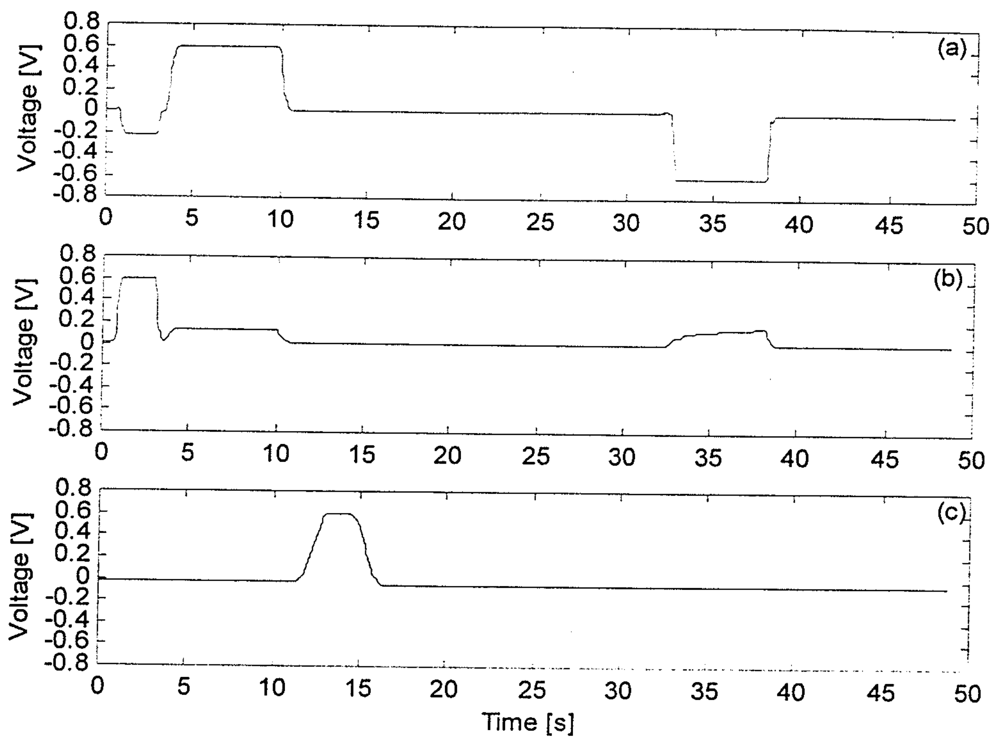


Fig. 5.16: Control signal to the crane actuators: (a) swing; (b) boom; (c) telescope.

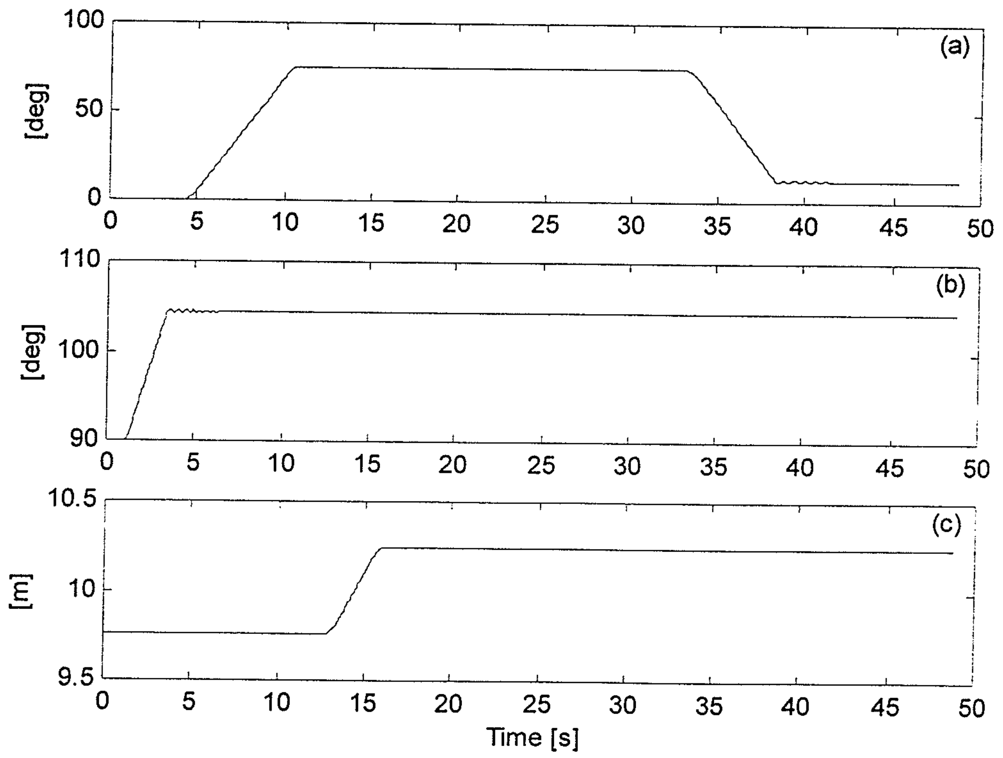


Fig. 5.17 Crane joint position profiles: (a) swing; (b) boom; (c) swing.

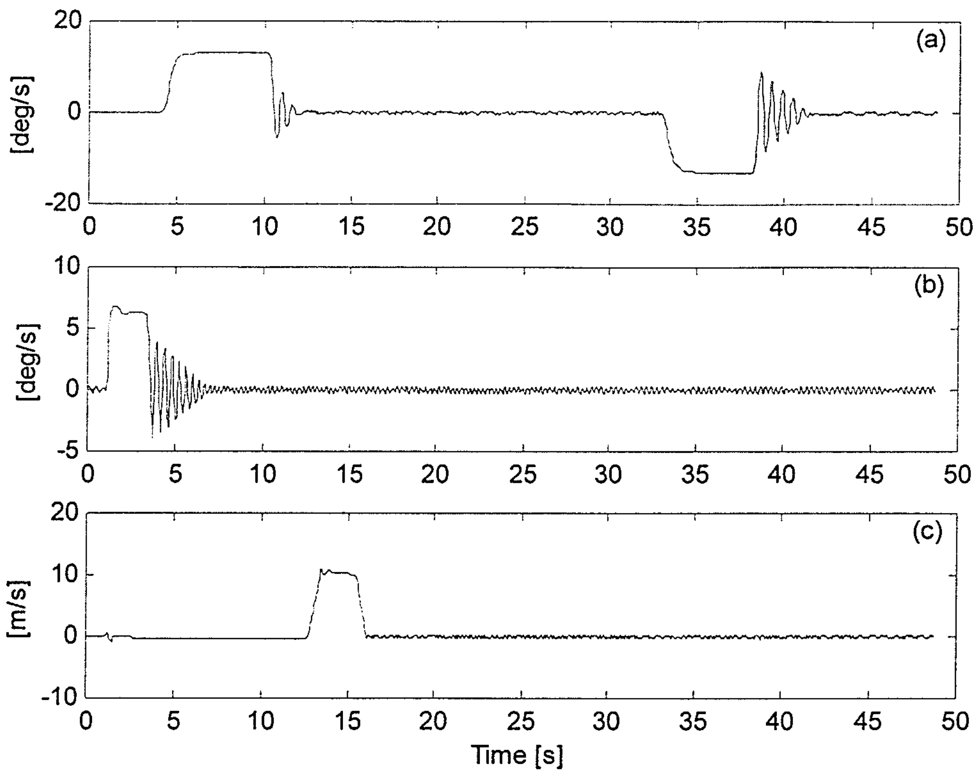


Fig. 5.18 Joint velocity profiles: (a) swing; (b) boom; (c) telescope

5.2 Performance Analysis

The performance criterion for the Service Provider (SP) is the computation time of the dynamics, i.e., the time needed to calculate the machine's response to certain inputs for a specific time span. The time span is set to 50 milliseconds, which is 20 frames per second (FPS). This is the average refresh rate for the SR. If the computation time is more than 50 milliseconds, it means that the simulation cannot achieve the real-time goal. The performance criterion for the Service Receiver (SR) is the refresh rate. A refresh rate of more than 30 FPS is generally desired because it provides smooth animation to the user. Based on the average computation power of today's desktop PC, a compromise of 20 FPS was chosen. Tests for the SP and SRs were conducted under different environments with different configurations. For the specific platform on which we tested the simulation, the maximum number of clients that a server can support was concluded from the results.

5.2.1 Service Provider's Performance

The hardware capability of the platform where the SP runs has the most significant effect on the service provider's performance. In our tests, the SP was running on a PC with Intel Pentium III 866 MHz CPU with 128MB RAM. Two sets of data were logged to measure the SP's performance, the computation time (referred as *C-Time*) needed to calculate the 50ms dynamics simulation and the overall time (referred as *O-Time*) needed to send out the drawing data. Overall time is the same as the computation time if the computation time is equal to or larger than the required 50ms. Otherwise an additional waiting time, during which the computer will wait until the 50ms elapses, is added in. In this case the computation time plus the waiting time is the overall time. Several SRs are running on machines with different settings. Since all SRs are sending oneway calls to the SP at a constant rate using a multimedia timer and only the calling rate may affect the performance of the SP, only the number of SRs has effect on the SP's performance. In addition, with different combinations of concurrency models employed, the SP will yield different results, especially when the number of the SRs increases.

5.2.1.1 SP's Performance vs. Number of SRs

The C-Time and O-Time with different numbers of the SRs connected to the SP are presented in Figure 5.20. As the default setting, *Blocking* client and *Reactive* server were set as the SP's concurrency models. Each test ran for more than 15 seconds. Only the results from the first second to the tenth seconds are shown here.

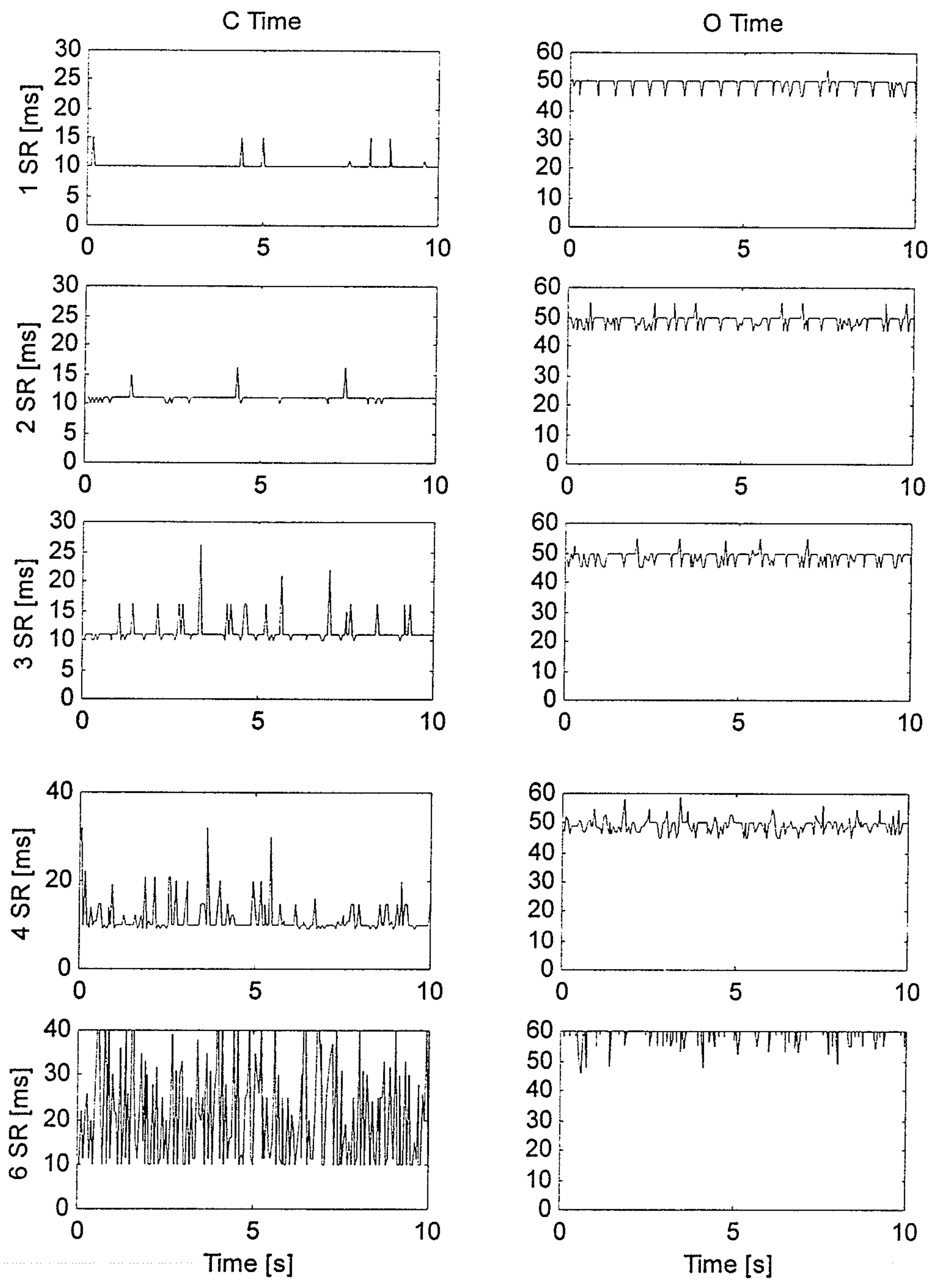


Fig. 5.19: SP's performance vs. SRs connected:
(a) computation time (ms); (b) overall time (ms).

From Figure 5.20 we can see that both the computation time and the overall time increase as more and more SRs are connected to the SP. When six SRs are connected to the SP, the computation time is still in the acceptable range, but the average overall time exceeds the preset 50 milliseconds. The result shows that one SP running on the hardware platform described above can support up to four SRs simultaneously.

5.2.1.2 SP's Performance vs. Concurrency Models

In this test, the SP was still running on the same platform as before, but the concurrency models for the client and server were changed. With 4 SRs connected, the SP with different combination of client and server concurrency models yielded different results.

With reference to Figure 5.21 we can see that SP with the *Threaded* client and the *Reactive* server concurrency model performs the best, while the worst is the SP with the *Blocking* client and the *Thread Pool* server concurrency model. From this test we conclude that the bottleneck of this program is making calls instead of executing incoming requests. For a small number (less than or equal to 4) of SRs, the SP with the blocking client and the reactive server concurrency model provides the best performance. For a larger number of SRs, the SP with the threaded client and the reactive server performs the best. Use of *thread-per-client* or *thread pool* concurrency model for the server should be generally avoided because the overhead involved in thread context switching outweighs the benefits they provide, especially on a single CPU platform. This simulation has never been

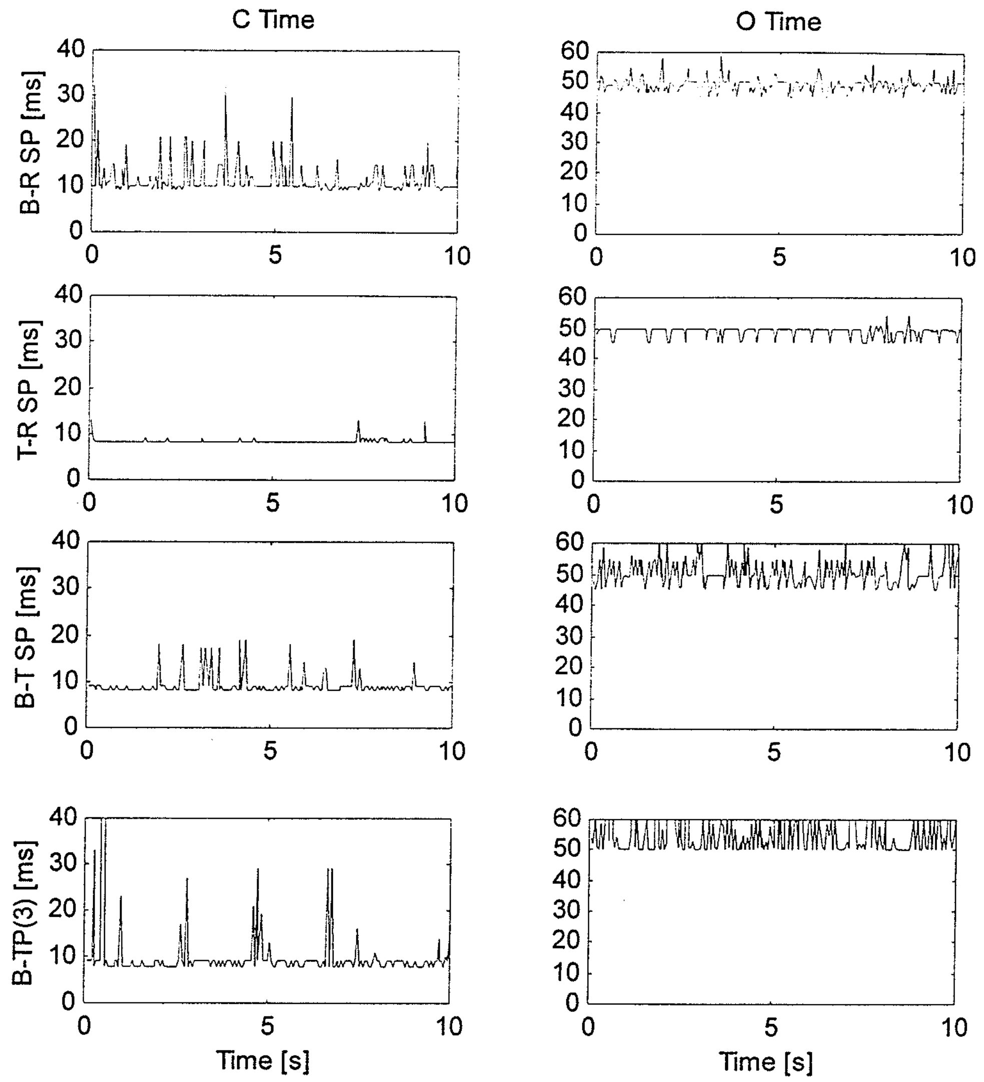


Fig. 5.20: SP's performance with different Concurrency Models:
(a) computation Time; (b) overall Time.

tested on a multi-CPU platform due to lack of testing equipment. Based on the real multi-tasking nature of a multi-CPU platform, the performance of the SP is guaranteed to be much better. Concurrency models can be set with command line options when the SP starts. Therefore, depending on the platform and the number of SRs to be connected, the user of the simulation should select the best concurrency model combinations.

5.2.2 Service Receiver's Performance

As mentioned before, refresh rate is the most important criterion for measuring the performance of a service receiver. According to our tests, the refresh rate is most affected mostly by the hardware capability of the computer where the SR runs, especially the rendering and 3D capability of the video card. In our tests, the SR ran on a PC with a single Pentium III 550 MHz CPU and a GeForce 256 video card with 32MB RAM.

The refresh rate changes when different views are selected and shown on the screen. The more complex the view is, the more triangles the video card has to render, and the longer the rendering takes. With reference to Figure 5.22, refresh rates of the first five seconds for each view under different circumstances are shown. For the definition of each view and screen shots, please refer to section 3.2.4.

Refresh rates of over 30 FPS can be achieved for all views except view 5. View 5 actually shows all the other views together on the screen. In order to get a higher refresh rate, a better video card with OpenGL support is needed. Fluctuation of the refresh rate can be observed once the connection between the SR and the SP established. This is caused by the overhead of thread context switching and remote call sending and receiving. Refresh rates over 30 FPS are generally not necessary. In the final version of the SP program, there is a 30 FPS limit for the refresh rate in order to save computation power for other processes and threads.

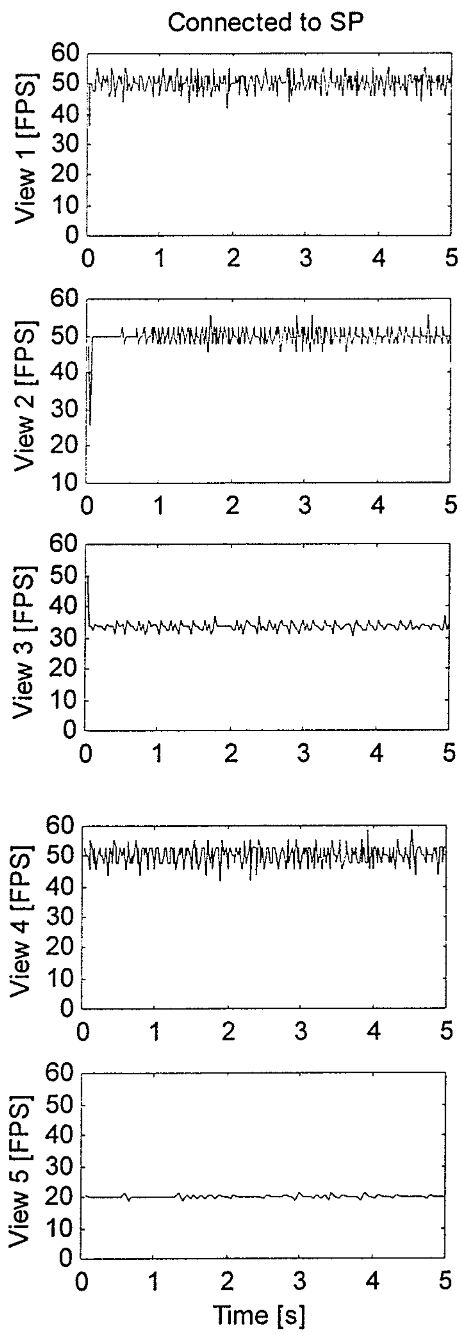


Fig. 5.21: SR's refresh rate for different views.

5.2.3 Performance Comparison between a Stand-alone Simulator and Distributed Simulator

Before the distributed computation technology of CORBA was used, a simulation program that includes the rendering and calculation of dynamics was developed. This program utilizes the same algorithm to calculate the machine's dynamics, the same 3D models and scene to render the views, and the same multimedia timer to sample the control signals. The only difference is that instead of distributing computations over two machines, it used multithreading to achieve the parallel computation. Because of the limited CPU power provided by today's desktop PCs, the performance of this stand-alone simulation was not satisfactory. Table 5.1 compares the performance of two simulations. The stand-alone simulator ran on a machine with a Pentium III 550 MHz CPU and a GeForce 256 video card. The same machine was also used as the service receiver of the CORBA based simulator. The service provider ran on a PC with a Pentium III 866 MHz CPU.

Table 5.1: Performance comparison between two simulators with different structure.

	Stand-alone Simulator	CORBA Based Simulator (4 SRs connected to 1 SP)
Average Time needed for 50 ms Dynamics (ms)	C-Time:94, O-Time:94	C-Time: 15, O-Time:50
Average Refresh Rate for View 1 (FPS)	17	50
Average Refresh Rate for View 2 (FPS)	13	50
Average Refresh Rate for View 3 (FPS)	12	34
Average Refresh Rate for View 4 (FPS)	14	48
Average Refresh Rate for View 5 (FPS)	7	20

It is obvious that the CORBA based simulator is better in both calculation time and refresh rates. The stand-alone simulator requires a better computer in order to provide smooth animation and real-time response, while one SP running on a powerful computer can serve many SRs simultaneously without compromising any performance. The only additional requirement is a network connection, which is quite inexpensive to set up and generally exists in today's computing environment.

5.2.4 Reliability Test

The communication between the SP and SR uses intensive *oneway* calls. A oneway request is a request for which no reply is received. Therefore a oneway request cannot return any results and there is no guarantee that the oneway request is properly executed by a server. A oneway operation may be lost and never delivered to the server. The specification guarantees that it will be delivered at most once. Therefore, the reliability of receiving and processing of one way calls poses a potential problem to the CORBA based simulator. ORBs from different vendors provide distinct implementations for oneway calls. Since no documentation was found about ORBacus's oneway call reliability, we ran the simulator under several conditions to test the lose-rate of oneway calls.

For those tests, one SP was connected with 4 SRs. The numbers of oneway calls, which were sent from one side and received and executed at the other side, were recorded. Results are presented in Table 5.2. From the table we can see that, in our experiments, none of the oneway calls are lost. It seems that the delivery of oneway calls in ORBacus is guaranteed.

Table 5.2: Oneway call sent and received for different SP concurrency model combinations.

	SR1				SR2				SR3				SR4			
	SP to SR		SR to SP		SP to SR		SR to SP		SP to SR		SR to SP		SP to SR		SR to SP	
	S.	R.	S.	R.	S.	R.	S.	R.	S.	R.	S.	R.	S.	R.	S.	R.
B-R SP	1724	1724	1673	1673	1012	1012	955	955	368	368	330	330	3309	3309	3217	3217
T-R SP	3169	3169	3091	3091	2428	2428	2357	2357	1745	1745	1631	1631	345	345	221	221
B-T SP	4898	4898	4801	4801	4559	4559	4367	4367	2661	2661	2382	2382	2176	2176	2002	2002
B-P3 SP	4532	4532	4152	4152	4068	4068	3846	3846	2477	2477	2124	2124	1822	1822	1702	1702

S.: Sent R.: Received

Chapter 6

Conclusions

6.1 Achievements

In this thesis, a CORBA based distributed simulation of a claming machine with 3D graphics visualization has been developed. The simulation is built upon the mathematical models of both the crane and the claming device, together with the model of hydraulic driven units. All models are programmed in an Object-Oriented Programming (OOP) manner so that these models can be reused with the dynamics complexity hidden from the user.

An interactive user interface with 3D animation has also been designed. 3D models of the crane, the claming device and the objects in the scene were constructed using AutoCAD and were later imported into the program. Several ways to produce the animation have been discussed and an infrastructure for fast animation under the Windows operating system has been built. OpenGL was employed to render rich and fast animation. Different views aiming to facilitate the operators in accomplishing the task was designed and implemented. Critical Mass Lab's simulation toolkits were incorporated to contribute to the realistic physics based behaviors among objects in the scene.

The smooth animation and real-time dynamics calculations are computationally heavy. Therefore a trial stand-alone program could not produce satisfactory results. CORBA is ideally suitable for abstracting away network-specific details in object oriented programming. Using CORBA and the predefined interfaces, the restrictions of tight computation power requirement were removed and this allows the separation of the dynamics calculation and animation rendering to a service provider and service receivers respectively. The main problem encountered when using CORBA for real-time applications is the extra overhead introduced by the abstraction layer. With deliberately designed independent hybrid client/server structure and object factory facility, request executions were distributed in calculation objects with their own threads. Non-blocking message sending/receiving allow the program to endure occasional large time lags.

The main contributions of this work are: (1) the development of mathematical models of the claming device and the crane together with the hydraulic driven units; (2) the utilization of the OpenGL graphics Application Programming Interface (API) and Microsoft Foundation Classes (MFC) to form an animation platform on the Windows operating system; (3) the incorporation of MathEngine's toolkits to produce physics based natural behavior of interacting objects; and (4) the implementation of CORBA technology to distribute the dynamics calculation over the network to achieve a real-time effect.

This simulation software can be used to train new operators and let them become familiar with real machine control and task performance. It also lays a foundation for future controller design and human-machine interface, which can assist operators in accomplishing their tasks.

6.2 Future Development

Possible future development could include the incorporation of a controller in controlling such a claming machine. Controller designs can be done on the simulator.

Furthermore, the dynamics model of the crane can be extended to allow tipping over of such machines. If the operators can be trained on such an animated simulator, chances of misoperation can be greatly reduced.

More interactive features can be added to the simulator. Sound effects that will provide more realistic presence experience can be incorporated. Graphics elements that mimic real world objects like a wood log or debris floating on the water can also be added to the scene. The claming device should be able to interact properly with these objects. This allows users of the simulator to perform more realistic pick-and-place tasks.

References

Attoui, A. "An Object Oriented Model for Parallel and Reactive Systems", *Proc. IEEE CS 12th Real-Time Systems Symp.*, San Antonio, TA, 1991, pp. 84-93.

Booch, G., "Object Oriented Analysis and Design with Applications", The Benjamin/Cummings Publishing Company Inc., Redwood City, CA, 1994.

Box, D., "Essential COM". Addison-Wesley, Reading, MA, 1997.

Cobb, M. and Shaw, K., "Trends in Distributed Object Computing", *Parallel and Distributed Computing Practices journal*, special issue on Distributed Object-Oriented Systems, Vol. 3, No. 1, 2000.

Dias, J.M.S., Galli, R., Almeida A.C.M., Belo C.A.C. and Rebirdão J.M. "mWorld: A Multiuser 3D Virtual Environment", *IEEE Computer Graphics and Applicatio*ns, Vol. 17, No. 2, pp. 55-65, 1997.

Farin, G., "Curves and surfaces for Computer Aided Geometric Design: A Practical Guide", Academic Press Inc., New York, NY, 1990.

Garcia-Alonso, A., Serrano, N. and Flaquer, J., "Solving the Collision Detection Problem", *IEEE Computer Graphics and Application*s, Vol. 14, No. 3, 1994, pp. 36-43.

Gobbetti, E., Balaguer, J. and Thalmann, D., "VB2: An Architecture for Interaction in Synthetic Worlds". *Proceedings of the ACM UIST '93*, Atlanta, 1993, pp. 167-178.

Gobbetti, E. and Balaguer, J. F., "Virtuality Builder II: On the Topics of 3D Interaction". In Virtual Worlds and Multimedia, John Wiley, New York, NY, 1994.

Gokhale, A. and Schmidt, D. C., "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks". *Transactions on Computin*g, Vol. 47, No. 4, 1998, pp. 391-413

Henning, M. and Vinoski, S., "Advanced CORBA Programming with C++". Published in the Addison-Wesley Professional Computing Series, New York, 1999.

Isdale, J., "What Is Virtual Reality?". http://vr.isdale.com/WhatIsVR/, 1999.

Ishikawa, Y., Tokuda, H., and Mercer, C. W., "An Object-Oriented Real-Time Programming Language", *IEEE Compute*r, Vol. 25, No. 10, 1992, pp. 66-73.

Kane, K., "An Implementation Model for Time-Triggered Message-Triggered Object Support Mechanisms in CORBA-Compliant COTS Platforms", *IEEE 1st Int'l Symp. on Object-oriented Real-time dependable Computing (ISORC),* Kyoto, Japan, 1998, pp. 12-21.

Kilgard, M. J., Blythe, D. and Hohn, D., "Rendering System Support for OpenGL Direct", Silicon Graphics, Inc. Graphics Interface '95, 1995.

Kim, K. H., "Object Structures for Real-Time Systems and Simulators", *IEEE Compute*r, Vol. 30, No. 8, 1997, pp. 62-70.

Kleiman, S., *et al.*, "Programming with Threads", Sunsoft Press, 1996.

Luh, J.Y.S., Walker, M.W. and Paul. R.P.C., "On-line Computational Scheme for Mechanical Manipulators", *ASME Journal of Dynamics Systems, Measurement and Control*, Vol. 102, 1980, pp.69-76.

Critical Mass Labs Inc., Critical Mass Labs Simulation toolkit developer documentation, http://www.mathengine.com/_mathengine_corp/_downloads/docs.html, 2001.

Merritt, H.E., "Hydraulic Control Systems", Wiley, New York, NY, 1967.

Microsoft, MSDN Library Visual Studio 6.0 Release, 1998.

Neider, J., Davis, T. and Woo, M., "OpenGL Programming Guide", Addison-Wesley, New York, NY, 1993.

Object Management Group. "The Common Object Request Broker: Architecture and Specifi-cation" Revision 2.0, Framingham, MA, 1995.

Object Oriented Concepts (OOC), Inc., "ORBacus For C++ and Java", 2000.

Ozer, J., "3D Computing". *PC Magazine*, Vol. 17, No. 11, 1998, p. 118.

Paul, R., "Robot Manipulators: Mathematics, Programming, and Control", The MIT Press, Cambridge, MA, 1981.

Richter J., "Advanced Windows", 3 rd Edition, Microsoft Press, 1997.

Robertson, G., Card, S. and Mackinlay, J., "The Cognitive Coprocessor Architecture for Interactive User Interfaces". *Proc. of ACM SIGGRAPH/SIGCHI 1989 Symp. on User Interface Software and Technology*, 1989, pp 10 – 18.

Rogerson, D., "Calling All Members: Member Functions as Callbacks", Microsoft Developer Network Technology Group, Available at: http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dnarvc/html/ msdn_callb.asp, 1992.

Schilling R. J., "Fundamentals of Robotics: Analysis and Control", Prentice Hall, Enalewood Cliffs, NJ, 1990.

Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F., "Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects", Vol. 2. Wiley & Sons, New York, NY. 2000.

Segal, M. and Akeley, K., "The OpenGL Graphics System: A Specification", Version 1.0., Silicon Graphics, Mountain View, CA. 1993.

Shaw, C., Liang, J., Green, M. and Sun, Y., "The Decoupled Simulation Model for Virtual Reality Systems", *Proc. CHI '92.* Monteray, CA., 1992, pp. 321–328.

Shaw, C., Green, M., Liang, J. and Sun, Y., "Decoupled Simulation in Virtual Reality with The MR Toolkit". *ACM Transactions on Information Systems*, Vol. 11 No. 3, 1993. pp 287-317.

Surridge, B., "Modeling and Simulation of a Cable-Driven Grapple System Machine", B.Sc Thesis, Mech. & Indus. Engineering Department, The University of Manitoba, 1996.

Takashio, K., and Tokoro, M., "DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems", *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOCSLA)*, Vancouver, BC, 1992, pp. 276-294.

Wollrath, A., Riggs, R., and Waldo, J., "A Distributed Object Model for the Java System", *USENIX Computing Systems*, 1996.