THESIS REPORT

Master's Degree

Interactive Finite Element Analysis of Highway Bridges

by C.A. Hudson
Advisor: M. Austin

M.S. 93-16

The Institute for Systems Research is supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), the University of Maryland, Harvard University, and Industry

Abstract

Title of Thesis: Interactive Finite Element Analysis of Highway Bridges

Name of degree candidate: Christopher A. Hudson

Degree and year: Master of Science, 1993

Thesis directed by: Associate Professor Mark Austin
Department of Civil Engineering

Despite the well established benefits of using finite element methods, commercially available finite element packages have not received wide-spread application to the analysis of highway bridges. This is because they do not have the special features needed by bridge engineers.

In particular, the lack of pre- and post-processors suited for the finite element analysis of bridges is one of the major reasons why the running of finite element programs, as a standard part of bridge design, has always been a time consuming process. Currently available bridge software provides few facilities to easily set up, edit, and query information about the design problem description, and the response output to various truck loading conditions. Often the designer is given a textual echo of the input, but no graphical verification that a problem has been well formed. This problem is particularly acute when the design problem

definition, and ensuing finite element analysis, needs to be repeated several times.

Furthermore, general purpose finite element packages often do not provide for automatic generation of moving truck and lane loads for the maximum effect.

The purpose of this study is development of an interactive graphically based pre-processor for the finite element analysis of highway bridges with high-speed engineering workstations. The pre-processor gives engineers the choice of using both keyboard and mouse styles of interaction to describe and edit bridge geometries, set boundary conditions, and place trucks at the desired positions on the bridge. For our program, entitled XBUILD, the finite element software along with the pre-processor and post-processor, execute on separate workstations. They commmunicate via message passing and socket-based Interprocess Communication.

Dedication

First and foremost to my family for their guidance and support. Secondly, to all the faculty of the Department of Civil Engineering.

Acknowledgements

I would like to sincerely thank my advisor, Dr. Mark A. Austin for his vision, guidance and patience during this project. I also wish to thank all of the students, faculty, and staff of the Institute of Systems Research for their support and encouragement.

Most of all I would like to express my deepest appreciation to my thesis committee for their time, effort, and service. In alphabetical order, these three faculty members are:

1. Dr. Pedro Albrecht
2. Dr. Mark Austin
3. Dr. Charles Schwartz

Contents

Section **Page**

List of Tables vi

List of Figures vii

1 DISTRIBUTED FINITE ELEMENT COMPUTATIONS 1

1.1 Introduction 1

1.2 Trends in Computer Aided Design and Engineering 2

1.3 Distributed Finite Element Computations 11

1.4 Objectives and Scope 11

1.5 Reading Level 14

2 THE ARCHITECTURE OF XBUILD Version 2.0 15

2.1 Overview of Architectural Components 15

2.2 Graphical User Interface 16

2.2.1 Features of the XBUILD Pre-Processor 19

2.3 Input Data Files 29

2.3.1 Objects and Bridge Sub-Structures 29

2.3.2 Writing and Posting Files 30

2.3.3 Post-Processor 31

2.4 Finite Element Libraries 39

2.5 Interprocess Communication 40

3 OVERVIEW OF COMPUTER GRAPHICS AND IPC MODULES 42

3.1 Data Representation Approaches in Computer Graphics 42

3.1.1 Homogeneous Coordinates 43

3.1.2 View Transformation Matrix 46

3.2 Forward Mapping from World to Screen Coordinates 51

3.3 Reverse Mapping from Screen to World Coordinates 58

3.4 Pushing and Popping of Matrix Stacks 60

3.5 A Discussion on Three Data Structures for our System 64

3.5.1 Object Data Structure 64

3.5.2 Truck Data Structure 68

3.5.3 Data Structure for IPC Message Passing 71

3.6 IPC Modules and Their Relationship with the Command Language 75

3.7 Logic Behind the Construction of the IPC Modules 81

3.8 Working at the Object Level 83

4 ANALYSIS OF THREE ENGINEERING SYSTEMS 84

4.1 3-D Frame Structure 84

4.2 Cantilever Plate 92

4.3 Multi-Span Truss-Bridge Structure 95

5 CONCLUSIONS AND FUTURE WORK 99

5.1 Conclusions 99

5.2 Future Work 104

Bibliography 110

List of Tables

Number		Page
2.1	Example of a Finite Element DataFile	36
2.2	List Module for IPC System	40
2.3	Example of Block IF Code for IPC System	41
3.1	The Object Type Data Structure Modules	66
3.2	Language Constructs for the YACC Grammar	67
3.3	Sample Datafile for AASHTO HS20 Truck	69
3.4	Grammar Rules for Truck Objects	70
3.5	Dictionary of Tokens for Language Constructs	71
3.6	Representative Sample Data Structures For IPC System	79
3.7	The Enumeration Type of Data Structure Modules	81
4.1	Boundary Conditions for 3-D Frame Example	85
4.2	Sample Datafile for Plate Example	93
4.3	Parser Algorithm for YACC Comment Statement	96

List of Figures

Number		Page
1.1	CAD Model.	5
1.2	Intelligent Integrated Interactive CAD Architecture.	7
1.3	Decomposability.	9
1.4	Composability.	9
1.5	Syntactical Units developed in our grammar.	10
1.6	Typical Client-Server Modules.	12
1.7	Network Architecture for Optimization Algorithm.	13
2.1	Network Architecture for XBUILD Version 2.0.	17
2.2	Typical Example of the Graphical User Interface Window System.	17
2.3	Display of Cycle Panel Buttons.	21
2.4	Graphics Mode Window for Movement of Object.	21
2.5	Informal sequence of pushing and popping matrix stacks.	22
2.6	3D View Volume.	33
2.7	Orthogonal View Volume.	34
2.8	Sample of Pull-Down Menu Item.	34
2.9	Interactive Procedure for Plate.	35
2.10	Depiction of Nodal and Element Level Sequences.	35
2.11	Diagram of Hash Table with Typical Storage Scheme of Objects.	37
2.12	One Storey Frame.	37

2.13 Deflected Shape for One Storey Frame. 38

3.1 Homogeneous Coordinate Plane. 45

3.2 Transforming world-coordinates to window-coordinates. 49

3.3 Figure Depicting Definitions of Parameters for View Setup. . . . 50

3.4 Relationship of Eye-Screen Coordinate Systems. 51

3.5 The (xs,ys,zs) plane showing details of the perspective projection. 52

3.6 Orthographic View Volume Definitions. 54

3.7 Orthogonal View Volume Definitions. 55

3.8 Single Point Perspective Projection. 56

3.9 Single Point Parallel Projection. 57

3.10 Two formal sequences of pushing and popping matrix stacks. . . 63

3.11 Hierarchical Object Tree for Typical Bridge Structure. 65

3.12 AASHTO HS20, H20 Truck Definitions. 77

3.13 Cross-Section of Highway Bridge Structure. 78

3.14 IPC Data Structure Hierarchy. 78

3.15 Object made of Three Beam-Col Elements. 80

3.16 Example of Message Passing System in Data Structure Form. . 80

3.17 Steel Roofing Structure Object. 83

4.1 One Storey Frame - Example No. 1. 86

4.2 Graphical Display of One Storey Frame. 90

4.3 Rotation Display of One Storey Frame. 91

4.4 Plate-Cantilever Example. 92

4.5 Deflected Shape for Plate-Cantilever Example. 94

4.6 Multi-Span Truss Bridge Example. 97

4.7 Translation of Multi-Span Truss Bridge Beam-Col. 98

Chapter 1

DISTRIBUTED FINITE ELEMENT COMPUTATIONS

1.1 Introduction

Despite three decades of research in finite element analysis methods, and significant advances in computer hardware and software, most engineers are still analyzing bridges with simplified pre-1960's methods. During the 1950's, computers were designed as **number crunchers** for physicists and missile designers alone [19]. Slide rules were the only form of computational assistance available to engineers, and approximate methods for analysis were necessary simply to get the job done. By the mid-1960s, corporations such as IBM and General Motors were conducting large research projects, the results of which shaped the study of interactive computer graphics and Computer-Aided Design (CAD) into a viable, useful, and exciting research field.

Today there is strong evidence - see, for example, reference [10] - that bridge owner's are paying a high price for the engineer's convenience of reduced computational effort. Instead of using approximate analysis techniques and semi-

empirical formulas based on distribution factors, highway bridges should be analyzed using the finite element method. However, at this time, finite element methods of bridges have failed to gain wide-spread acceptance among bridge engineers for several reasons:

1. Despite the wide range of commercially available finite element programs in the marketplace, many lack suitable pre- and post-processors for the application to bridge structures. Not only is the setup of finite element descriptions of bridge structures tedious, but program output is often restricted to textual data file format. Interpretation of structural behavior is sometimes difficult because engineers may not have the tools to view the entire bridge geometry (or a substructure) from an arbitrary 3D (or planar) view, or to rotate the existing view to another arbitrary 3D view.

2. Special features for bridge analysis, such as lane loadings and automatic generation of moving truck loadings, are often missing. Providing engineers with the ability to work directly with standard definitions of truck types or loading patterns simplifies the process of setting up problem descriptions for the finite element program.

3. The checking of AASHTO design constraints for truck and lane loadings is left to the engineer rather than computed by the program itself.

1.2 Trends in Computer Aided Design and Engineering

Computer-Aided Design (CAD) may be defined as use of the computer to aid in the design of an individual part, system, or subsystem [23]. Computer-Aided

Engineering (CAE) is use of a computer to aid in the manufacturing or production of a part. Together, the goals of CAD and CAE are improved quality and safety of a product, reduced engineering time (achieved by fewer design iterations and prototypes), and reduced cost. As pointed out by Chin and Chanson [7], advances in computer hardware and software are being driven by the needs and market demands of CAD/CAE end-users.

In the opinion of Alan Kay, a fellow at Apple Computers, the user interface is the most important component of a CAD/CAE program for complex engineering systems [15]. If the on-screen view of an application is clumsy, slow, or unresponsive, then the application will be judged of low quality. Good user interfaces provide for interactive computer graphics; the latter may be partitioned into `static` and `dynamic` regions. Static regions are characterized by passive operations, such as the computer presenting stored information to an observer in the form of a picture. Mathematical functions, pie charts, and bar graphs are examples of this form of computer graphics [12]. In the dynamic portion of computer graphics, the observer has the capability to manipulate – scale, rotate, translate, adjust eye location – three dimensional pictures in real time. As we will see in Chapter Three, manipulation of large objects may require many arithmetic computations. Consequently, it is the dynamic region of computer graphics that research on algorithms, data storage techniques, and user interfaces is currently being conducted [12]. User interfaces may be enhanced with pull down menu systems which segregate a large number of application commands into logical groups. For large application programs, the parser generator Yet Another Compiler Compiler (YACC) may be used to design and implement command line interfaces dedicated to a specific problem domain [14].

Future CAD/CAE systems will employ a host of computing technologies,

including interactive graphically-based user interfaces, networking, parallel and distributed computing architectures, new programming languages, and artificial intelligence. With rapid advances in the computational and graphical power of engineering workstations currently taking place, the use of workstations to merely visualize a structure, or to provide basic analysis of a completed design may soon be deemed inadequate. Instead, engineers may expect significant computer assistance in (a) the synthesis of preliminary design alternatives (i.e. conceptual design), (b) tradeoff analysis of design performance versus costs for design alternatives (i.e. design optimization), (c) strategies of construction, (d) analysis of partially constructed structures, and (e) detailed analysis, checking, and life-time cost analysis of the final design. To ensure that better designs are produced in less time, all of the above tasks (a)-(e) should operate within a single integrated computing environment, and be interactive. The heart of the integrated design and analysis problem lies in the formulation of: (a) data representations for structures that contain enough information to be useful at multiple levels of abstraction within the design process, and (b) algorithms that can operate on these models [4], [5].

Unfortunately, the technology needed to implement these steps is still in its infancy. Common limitations of todays CAD systems include the following [3]:

1. Conceptual design stages are not supported by CAD tools.
2. CAD cannot manage inaccurate, incomplete, inconsistent information.
3. Models and systems are not integrated.
4. CAD does not check errors.
5. Data input is a problem.

6. Terminology of the task domain cannot be understood.

7. CAD has only poor problem solving abilities.

Moreover, conventional CAD systems do not support models of design processes. By design processes, we mean a sequence of activities to create an engineering system. A typical example might be that of a procedure to construct a reinforced composite concrete beam. In this respect, Finger and Dixon categorize design process models as follows [11]:

1. A **descriptive model** explains how design is done.

2. A **cognitive design process model** explains the designer's behavior.

3. A **prescriptive process model** shows how design is done.

4. A **computable design process model** expresses a method by which a computer may accomplish a specified (design) task.

One way of mitigating these deficiencies is to build a supervisor into the CAD architecture. Figure 1.1 shows a schematic of current CAD systems which are

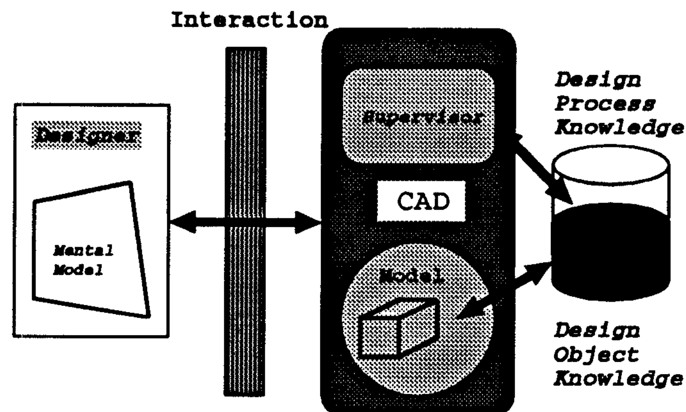


Figure 1.1: CAD Model.

controlled by an intelligent supervisor. The architecture recognizes that each designer will have his/her conceptual idea of how an object should behave in a particular environment; they depict in their mind the description of the model object. Tomiyama calls the latter the `mental model`, and emphasizes the importance of ensuring that the mental model mimics the designer's own image [26]. The role of the supervisor is to aid designers in entire designing processes. This aid comes from integrated models with rich functions for various kinds of design activities [26].

One area of research interest is Intelligent Computer Aided Design (ICAD). Tomiyama suggests that future ICAD systems should be integrated, intelligent, and interactive [26]. We say that a CAD system is intelligent if it provides services, (or incorporates advanced features such as a supervisor) beyond conventional CAD systems. ICAD systems should contain information from multiple problem domains, and knowledge on how a design entity should perform/behave in its environment [26]. This knowledge will be used to automate design processes (a combination of `top-down` and `bottom-up` strategies), to provide engineers with advice in decision making processes, and in the computation of optimal designs. A tentative architecture for an ICAD system is shown in Figure 1.2. Figure 1.2 is an elaboration of Figure 1.1. We see that the designer's `mental model` is now linked with an intelligent user interface, which in turn, is controlled by a supervisor. The supervisor is linked to knowledge of the design object, design processes, and the application interface. The application interface is used to control applications such as geometric modeling systems, technology analysis systems (e.g. finite element analysis packages for highway bridges), and expert systems. We say that a CAD system is intelligent if it provides services, (or incorporates advanced features such as a supervisor) beyond conventional

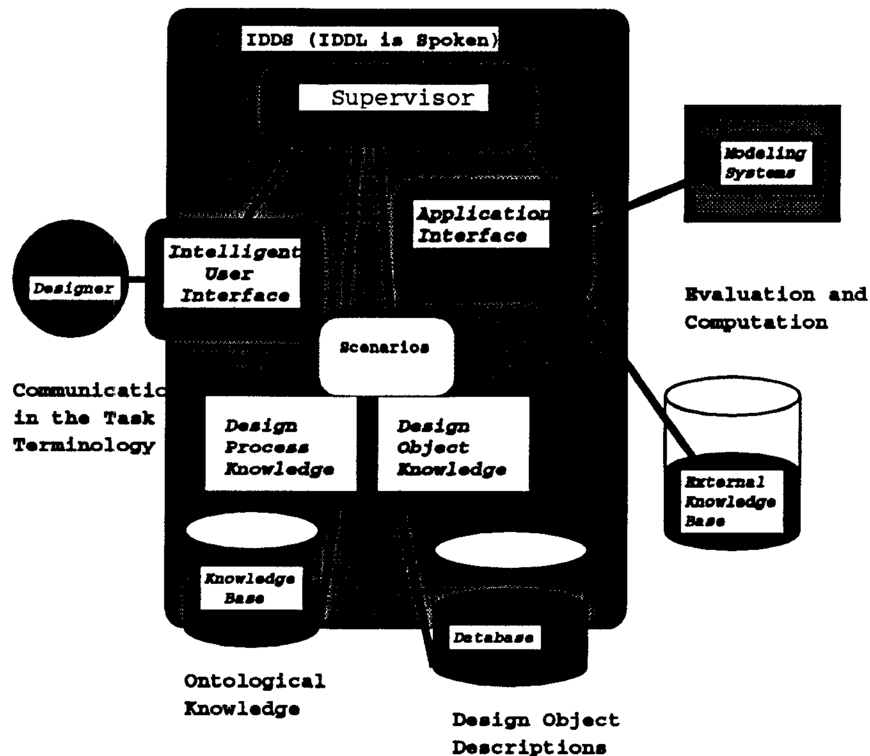


Figure 1.2: Intelligent Integrated Interactive CAD Architecture.

CAD systems.

In Figure 1.2, the keyword IDDL (Integrated Data Description Language) is associated with the kernel language of the ICAD system. We can now give a specific example of how the language for an ICAD system would differ from a conventional CAD system. Generally speaking, engineers know that the term `modulus of elasticity` means "slope of the stress-strain curve." Because current CAD systems cannot connect `terminology` with `intended use`, the onus for correct usage is left to the engineer. The hope is that ICAD systems will incorporate an intelligent supervisor, which will have the ability to understand engineering terminology to link concepts of objects, attributes of objects, and worlds to describe design objects [26]. So, for example, if the user accidentally misused the term `modulus of elasticity` in an expression, the supervisor

could alert the `design knowledge` bin of the ICAD system, and an appropriate error message would be returned. This concept of intelligence is closely related to the field of `knowledge engineering`.

Bertrand Meyer indicates that software systems, such as ICAD, should be assembled from `modules of code` – data structures, algorithms, and interfaces – that correspond to syntactic units in a `problem domain language` [17]. Coutaz writes that the language used by a computer and a user is defined by three components [9]. These three components are the semantics, syntax, and lexicon. We abbreviate his definitions:

> *semantics*: defines the meaning of sentences.
>
> *syntax*: defines the construction of sentences from a set of predefined syntactic units.
>
> *lexicon*: defines the construction of syntactic units from a vocabulary.

Conversational languages and computer languages are composed of grammars and syntactical units (or parts of speech). Grammar is the treatment of connected words as they are used for expression of thought, and without confusion or error. Grammars are divided into two components, parts of speech and sentences. In the English language, verbs and nouns are examples of parts of speech (or syntactical units). Sentences are groups of words that express a complete thought.

The design of ICAD-like software systems is expected to proceed in a combination of top-down/decomposable and bottom-up/composable procedures. Figure 1.3 depicts Meyer's definition of decomposability; it is the derivation of software elements from a family of problem specifications. Figure 1.4 illustrates

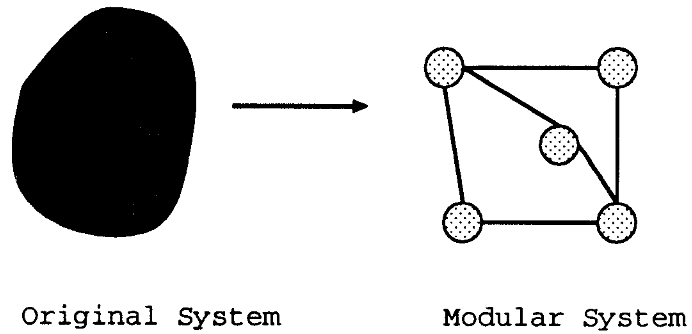


Figure 1.3: Decomposability.

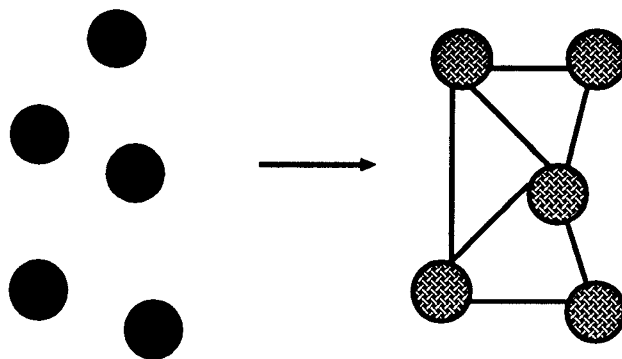


Figure 1.4: Composability.

the concept of composability; it is the production of software elements which may be freely combined with other modules to produce new software systems.

Overview : In the development of the XBUILD keyboard language, we wanted to use combinations of **verbs** and **nouns** to describe how actions should be applied to an object(s), and to use keywords that an engineer would be familiar with (e.g. stress, strain, moment, and shear). A schematic of verb and noun keywords is shown in Figure 1.5.

For the design of XBUILD, the concept of decomposability means separation of the finite element pre- and post-processor, and the main finite element analysis package. As will be explained in more detail in Chapter 2, the two components run on separate engineering workstations. Concepts of composability are needed

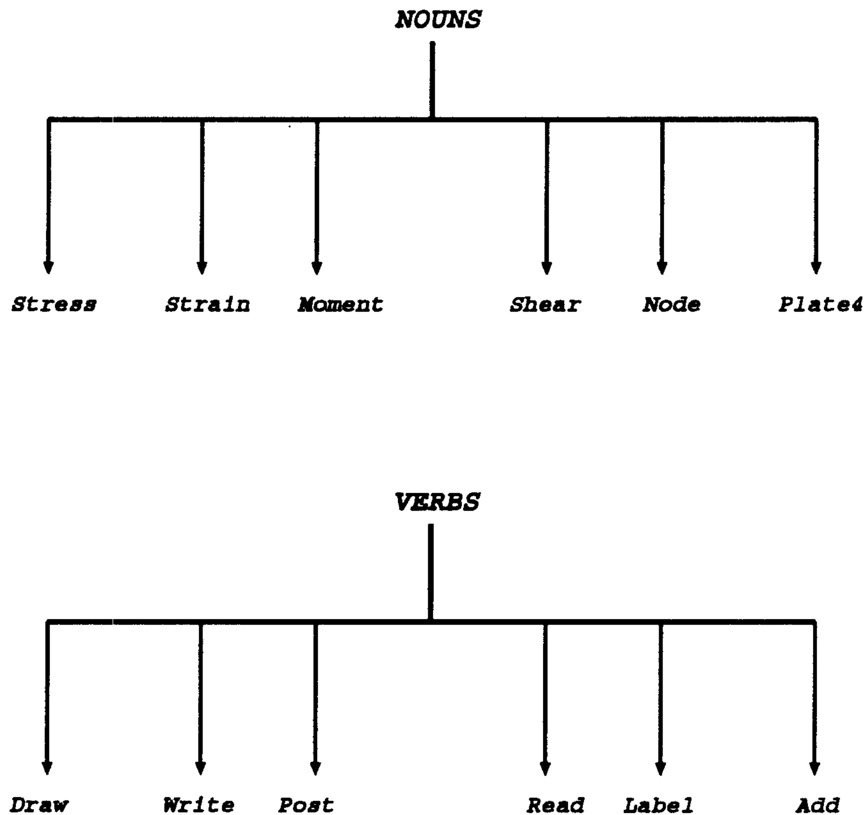


Figure 1.5: Syntactical Units developed in our grammar.

to build the distributed finite element system from components – in this case, the three largest software components: (a) the pre- and post-processor, (b) the finite element analysis package, and (c) software for message-based communication via Interprocess Communication. In XBUILD, composability also includes the use of graphics libraries, math libraries, and numerical libraries.

Summary : The field of CAD/CAE and interactive computer graphics has continued to evolve since its inception in the mid-1950's. Today, research and development efforts are focused towards the creation of intelligent user interfaces, feature based pre-processors, and the implementation of ICAD principles to the design object. If the details of ICAD systems can be worked out, then the hope is that when a designer makes a query, the computer system will

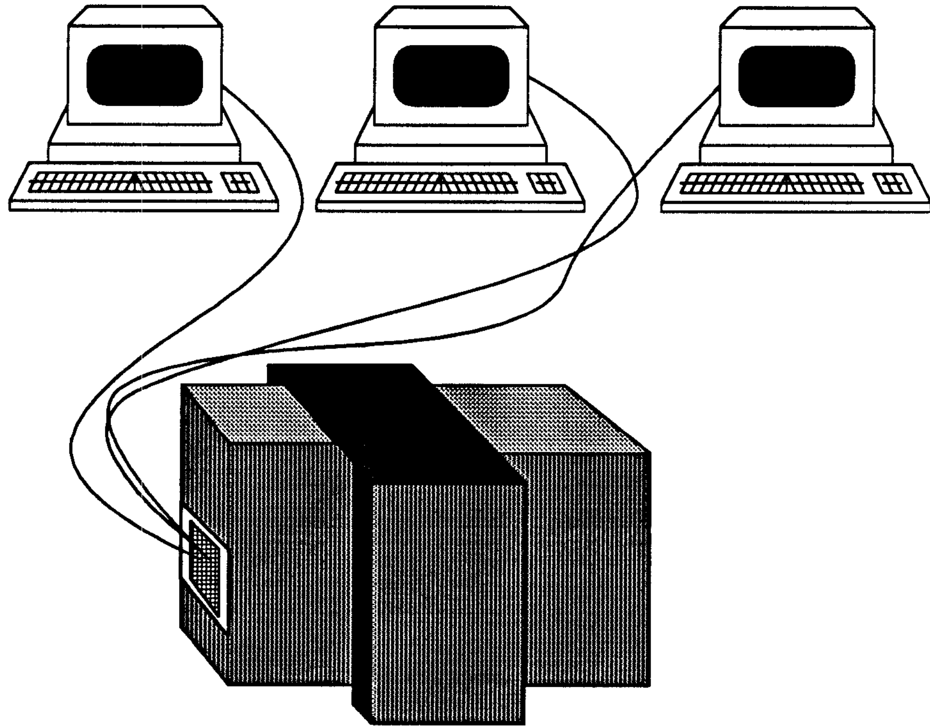
suggest improvements. Efforts are also underway to incorporate ideas from solid modeling and the finite element method into the analysis of highway bridge structures. For details, see Preston [21].

1.3 Distributed Finite Element Computations

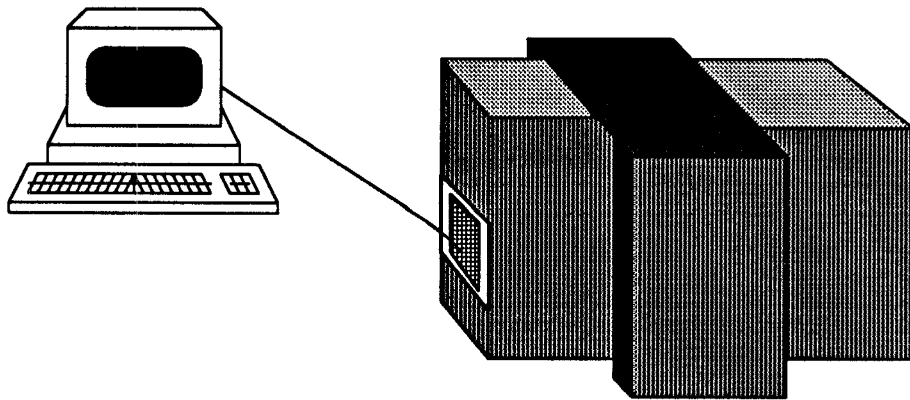
Personal computers and workstations are becoming increasingly popular in the engineering workplace. This trend is due in part to huge advances in networking technology. Networking allows multiple computers/workstations to share file systems, input/output devices (e.g. scanners, printers), and users to communicate via file transfer program (ftp), telnet, and E-mail. Networking also provides engineers with the opportunity to develop software systems that exploit the combined computational power of multiple workstations. Two arrangements of low-speed client workstations and a single high-speed finite element server are shown in Figure 1.6. Recently, Voon has studied the optimal design of two and three dimensional steel frames in a distributed computing environment [27]. His architecture was composed of a network of loosely coupled SUN-SPARC workstations; components of the architecture included a user interface, a process manager, and $1 \cdots n$ simulators, as shown in Figure 1.7. This type of architecture was used to test a new class of Feasible Sequential Quadratic Programming algorithms (FSQP) in a Distributed Network Computing (DNC) environment.

1.4 Objectives and Scope

For the past 29 months, the writer has been developing a distributed computational environment for the analysis and design of steel highway bridges. The



Model 1: Three Client Workstations
Connected to Single Server.



Model 2: Single Client Workstation
Connected to Single Server.

Figure 1.6: Typical Client-Server Modules.

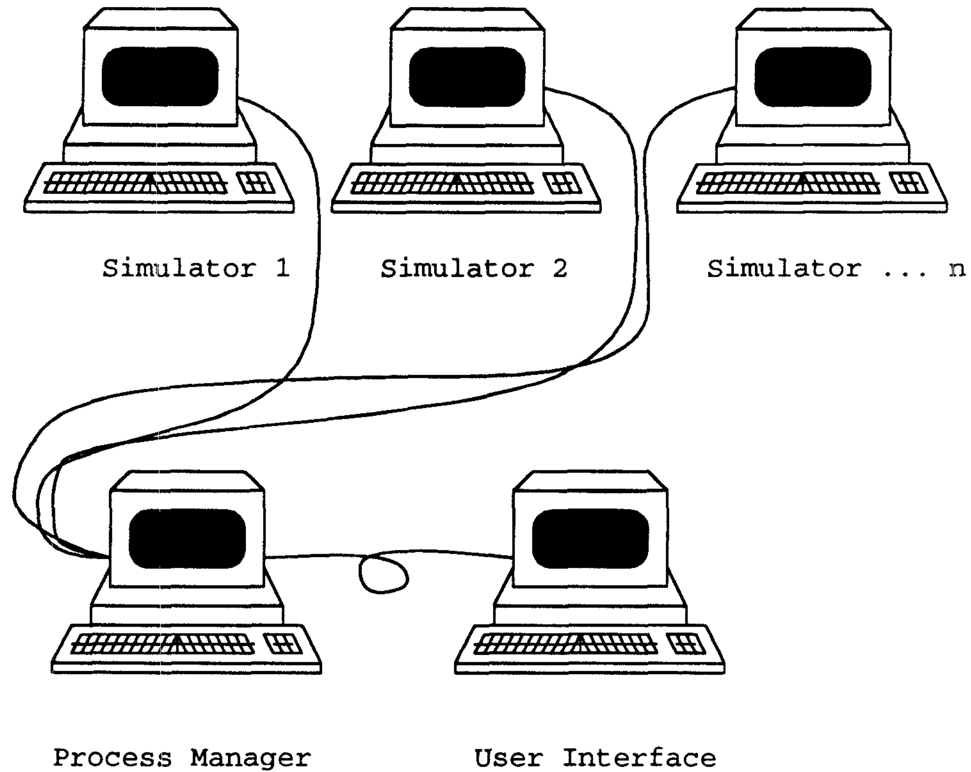


Figure 1.7: Network Architecture for Optimization Algorithm.

purpose of this project was to investigate the benefits of finite element analyses in a distributed computing environment. The scope of this project was restricted to the simple client-server model shown in the lower half of Figure 1.6. In our model, the client is a low-cost engineering workstation that runs a SunView-based pre- and post-processor for finite elements. The finite element analysis program executes on a separate high-speed workstation; it is the server. The principle objectives of this work are included below.

1. To construct a message passing data structure format for the Interprocess Communication system, which should enhance the engineer's interaction with the ensuing finite element analysis.

2. The generation of interactive procedures for creating highway bridge engineering systems in the pre-processor phase.

3. Develop prototype modules that will one day be used in an object-oriented design system for highway bridges.

The vehicle for this work is the workstation model consisting of a high resolution bit mapped screen, a multiwindow user interface model with mouse and keyboard input, multi-tasking, and network connectivity.

This thesis discusses the background and implementation of the prototype system. Chapter Two describes the architecture of XBUILD Version 2. Chapter Three explains our use of the data structures for the Interprocess Communication system, and how the program's modules and data structures work in conjunction with language constructs. Chapter Four describes the implementation of three structural systems in the prototype program. Conclusions and suggestions for future work are given in Chapter Five.

1.5 Reading Level

Readers of this thesis are assumed to be familiar with the finite element method as described in many textbooks. The reader is also assumed to be familiar with basic computer terminology (such as a workstation, mouse, window, menu, keyboard), the C programming language, and the compiler construction tool YACC.

Chapter 2

THE ARCHITECTURE OF XBUILD Version 2.0

The *XBUILD V. 2.0* software was developed for use on Sun Microsystems SUN/3, SUN/4, & SUN-SPARC workstations executing the UNIX 4.3 BSD operating system and SunView, the first windowing system available on Sun workstations. The 'C' programming language was used for this implementation because of the ease with which powerful and complex data structures may be defined, and also because it allowed for the use of the SunView libraries in the development of the point-and-click graphical user interface.

2.1 Overview of Architectural Components

The architecture of the computational environment is shown in the upper half of Figure 2.1. One workstation is used for the graphical user interface (GUI), and a second workstation is dedicated to running finite element analyses. The GUI and finite element method (FEM) packages communicate via message passing on socket-based Interprocess Communication (IPC) [13]. Coulouris and Dollimore define a socket as a software abstraction for a communication device

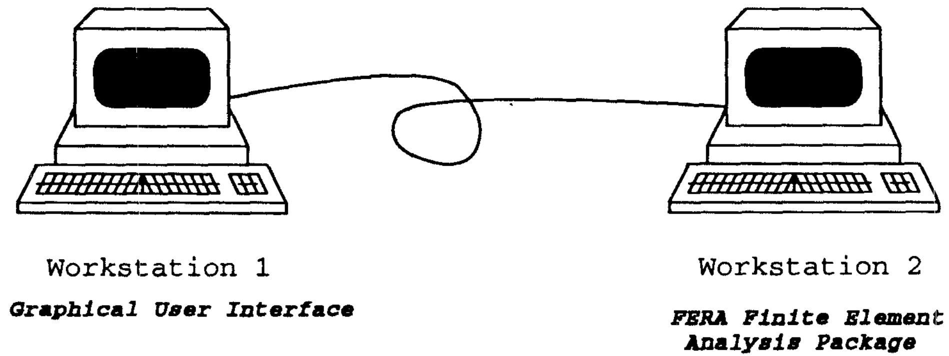
which creates an endpoint for communication [8]. This socket enables the two inter-connected computers to both transmit and receive messages. For further information on sockets and interprocess communications, the reader is referred to Voon [27] and Byrne [6].

Command sequences are always initiated at the graphical user interface - see diagram in lower-half of Figure 2.1. The engineer types or selects a menu/button item to activate the code in a callback function. This message will contain instructions for XBUILD to follow. Using a series of three data structures, the encoded message information is passed through the socket from the GUI to the workstation running the finite element program. The FEM interprets this information and carries out the instruction. Data generated by the finite element program is returned in information packages (data structures). The details of data structures for message passing will be explained further in Chapter 3.

2.2 Graphical User Interface

Figure 2.2 shows a typical layout of sub-windows in the XBUILD Graphical User Interface GUI. The GUI has five subwindows: (a) A bar of pulldown menus; (b) TTY terminal emulation window for keyboard input/textual output; (c) A canvas for displaying and manipulating three-dimensional views of the bridge, and (d)-(e) plan and elevation views of bridge cross-sections. From Figure 2.2, we see that the X-Z plane window is displayed in icon form. The TTY window is located on the lower third of the screen. The functionality of the GUI is divided into pre- and post-processing phases of finite element analyses. Kay lists six heuristics for designing a successful user interface [15]:

1. Identify the real end-user community



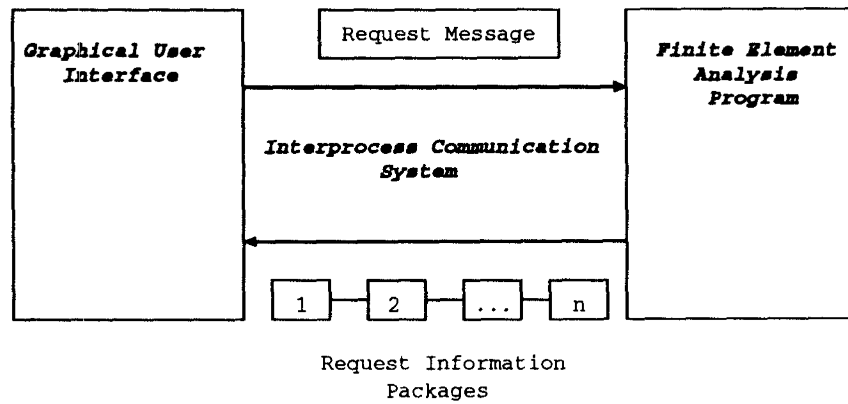


Figure 2.1: Network Architecture for XBUILD Version 2.0.

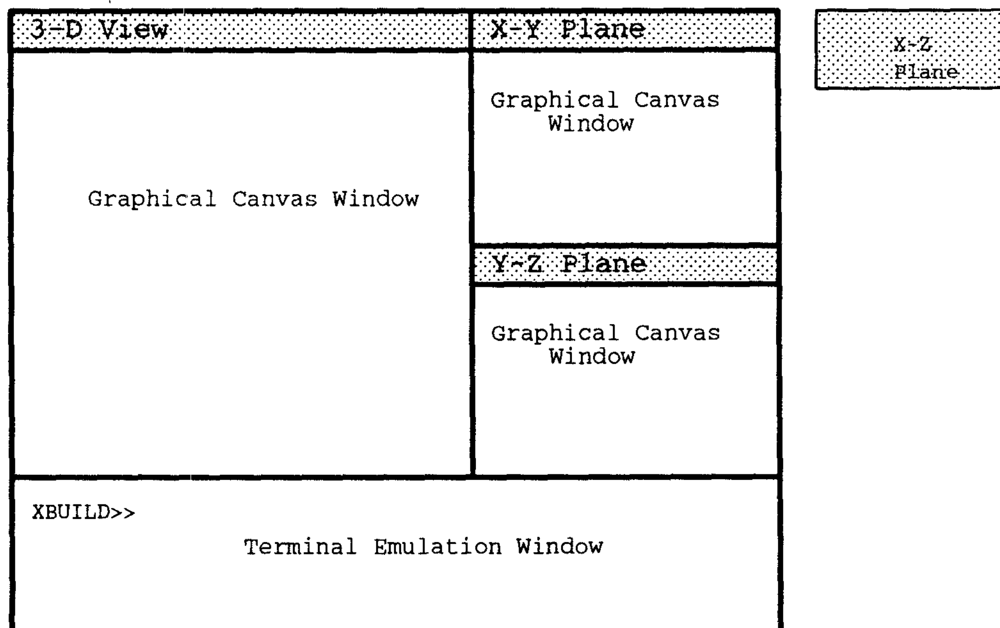


Figure 2.2: Typical Example of the Graphical User Interface Window System.

2. Carefully target the end-user community

3. Show users the interface on screen, not on paper

4. Mock up or prototype the interface to demonstrate the feel of operation

5. Use several groups of end-users as the specification progresses

6. Specify performance goals and metrics

Principles of good program design also include: [7].

abstract: The design concepts should be separated from the implem entation details. A program should hide the design decisions and data structures.

structured: A large program should be decomposed into components of a manageable size with well-defined relationships between the components.

modular: The internal design of each component should be localized so that is does not depend on the internal design of any other component.

concise: The code should be clear and understandable.

verifiable: The program should be easy to test and debug.

Lewi et al. mention that systems which cannot cope with the evolution of their environment (users, processes, machines, organization) are doomed to fail [16]. They also mention that software development is the process of continuously transforming, adapting, and redefining approximations. Moreover, it is

imperative that step by step composition and decomposition must be one of the most important characteristics of software development [16].

We incorporated some of these heuristics and characteristics in the design of our user interface, the IPC data structure hierarchy, and the command line messages used in the post-processing phase of the program. Our intention was to create an interactive user interface having keyboard and graphical interaction not available with most computer software application programs today. For instance, many user interfaces lack a graphical interaction for adding elements, adding plates, or applying loads and boundary conditions. They also lack a command language specifically developed for most engineering applications. Examples of graphical interaction will be elaborated upon in the remainder of this section. Command language techniques will be discussed in Chapter 3.

2.2.1 Features of the XBUILD Pre-Processor

The preparation of input files for finite element analyses can be a very time consuming process. Bridge engineers need pre-processors that will graphically display a highway bridge model and automatically generate input files for a finite element analysis. The software designer needs to develop interface strategies (graphical procedures using the mouse, typed keyboard expressions, or data file input commands) which expedite the pre-processing phase. In the XBUILD GUI for pre-processing, the engineer creates the desired window option (3D view, XY plane, YZ plane, XZ plane), inputs coordinate locations to set up the view transformation matrix, constructs new objects, applies loads to objects, and sets boundary conditions.

The Relationship of Objects to the GUI/Pre-Processor

Bridge engineers often find that the design of a highway bridge can be simplified if it is viewed as an assembly of objects (or sub-structures). Bridge diaphragms, parapets, girders, or bridge decks are examples of objects an engineer might be concerned with. A bridge deck will be modeled as a series of plate finite elements; each plate finite element may/may not have the same material and dimensional properties. An engineer may wish to store these objects, along with their respective material and dimensional properties, inside a hash table for future retrieval (A hash table functions very similar to a filing cabinet). The engineer may also want to edit existing bridge objects stored inside the hash table. Permissible edit operations include material or section properties, coordinate location(s), boundary conditions, and AASHTO [2] live loading patterns.

The next eight sub-sections describe the software development of the XBUILD GUI pre-processor and how an engineer may use it to construct objects. An engineer will work through this step-by-step procedure during this process of setting up a bridge geometry with its applicable loads.

Eye Location

Many of todays computer programs in engineering do not have an algorithm to control the position of an object on the screen via mouse input. Our algorithm is designed to work in the 3-D View graphics window. To initiate the procedure, the engineer points the cursor and clicks the mouse (`point and click`) on the appropriate cycle panel button shown in Figure 2.3. Now the following window menu, as seen in Figure 2.4, appears on the screen – it allows the engineer to choose either translation, or rotation, along with an appropriate magnification factor. In this context, the translation term limits an object to a rigid body

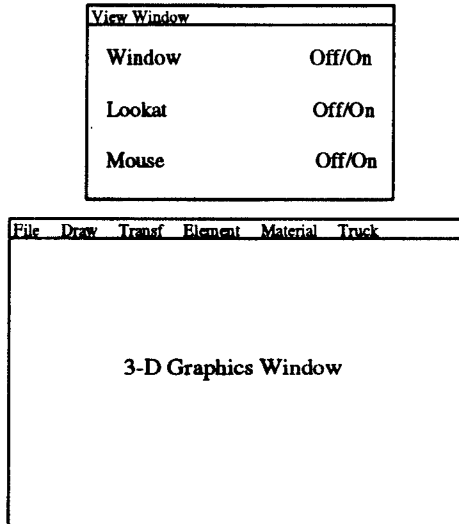


Figure 2.3: Display of Cycle Panel Buttons.

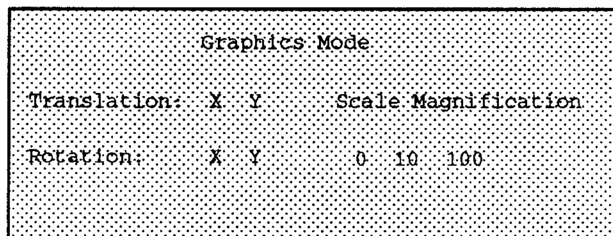


Figure 2.4: Graphics Mode Window for Movement of Object.

translation, while the rotation term limits it to a rigid body rotation. Both cases occur in either/both the x and y axis directions according to the `screen coordinate axis`. With this stage of the procedure complete, the engineer can adjust the position of an object by simply moving the mouse in a horizontal or vertical position. To begin the interactive eye algorithm, the program reads the pixel location of the mouse as the starting point, and then with the corresponding movement, records each new pixel location as an addition to, or subtraction from, the original point. As we will see in Chapter 3, all operations for computer graphics (translation, scaling, rotation, window and lookat) are computed with matrices of homogeneous coordinates. The new scalar value is inserted

into the respective translation or rotation matrix, and the new translation or rotation matrix is pre-multiplied with the current view transformation matrix. Each new scalar value requires a new matrix operational procedure. Hence, the

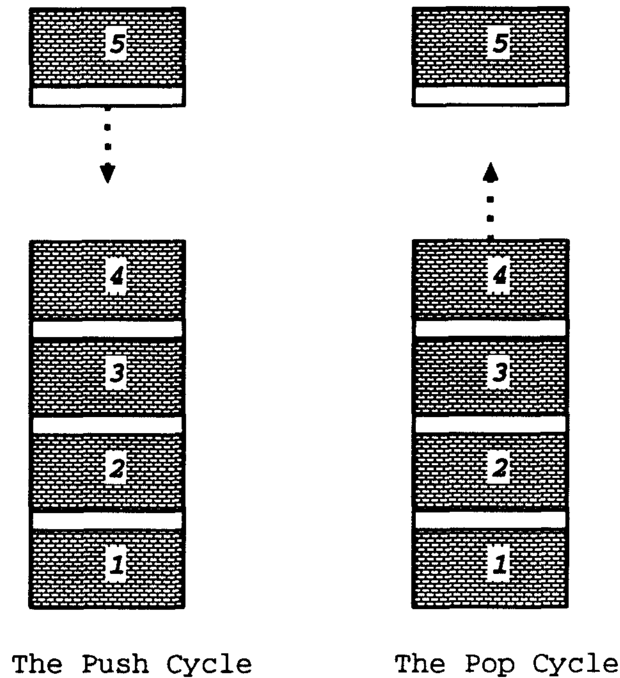


Figure 2.5: Informal sequence of pushing and popping matrix stacks.

old view transformation matrix must be "popped off" the matrix stack and new one "pushed on" the matrix stack. Figure 2.5 shows an informal definition of pushing/popping matrices. From Figure 2.5, the numbers (1,2,3,4,5) represent any of the aforementioned types of matrices. We will give a working example of **popping** and **pushing** matrix stacks in Chapter 3.

Windows

The XBUILD GUI has four graphics windows - three orthogonal views (X-Y plane, Y-Z plane, and X-Z plane) and one 3-D view, which all may be decreased or increased in size. For the windows containing orthogonal views, engineers

may look down any axis in either direction. In the X-Y planar orthogonal view, for example, the eye will be oriented along the Z-axis, with a Z-coordinate value at either $+\infty$, or $-\infty$.

Terminal Emulation Window

The Terminal Emulation Window, TTY, shown in Figure 2.2, is for command line input, and for the display of textual output from XBUILD. As mentioned in Chapter 1, keyboard expressions match the syntactical units of a YACC grammar. The character string `XBUILD >>` is the default prompt. XBUILD uses the prompt as a way to control the program modes. For instance, if the engineer wants to add a node to a structure then she/he would first need to set the prompt mode. This is done by typing the following command.

```
XBUILD >> add node    ⇒    XBUILD_add_node >>
```

While XBUILD is in the `add_node` mode, the post-command action is as if the command already started with add node. There are similar commands for adding boundary conditions, nodal loads, and beam-column and plate finite elements. Each prompt directs the program to a different subroutine in the XBUILD software constructs. The reader will see many of these different prompt modes throughout this report.

Elements

Procedure: To start the element procedure, the engineer uses the XBUILD command language to set the material type and section type using the following examples as guides.

```
XBUILD >> set material type to STEEL4
```

```
XBUILD >> set section type to W36x245
```

The term STEEL4 refers to the type of steel material found in the preface of the LRFD Manual [1], and W36x245 refers to a rolled thirty-six inch beam with a nominal weight of 245 lbs/ft. To add a new section, the user simply changes the material type, the section type, or both.

Our procedure for adding beam-column elements from Node i to Node j involves the familiar rubber band technique [12] (i.e. If we take a rubber band and hold one end fixed, we can stretch it to almost any reasonable distance). When starting this algorithm, the engineer will treat Node i as a fixed reference point and Node j will become the target destination, or endpoint. To begin the algorithm, the engineer places the cursor on a node inside one of the orthogonal view windows. By depressing the left mouse button, one procedes through the algorithm by dragging the mouse from Node i to Node j. As this process is taking place, the engineer will see a green line mimic the behavior of a rubber band. The program will trace the movement of the cursor and set up the appropriate mathematical relationship. By reverse mapping from pixel coordinates to world coordinates, the program can ascertain which node the user has started with and which node the user has ended with. This information can now be provided to all of the other three windows.

In cases where graphical input of elements is impractical, the user may add them via keyboard input. For example, the command

```
XBUILD_beam_col >> [STEEL4, W36x245] from Node 1
                     to Node 3 by 1
```

will attach one beam-column element to Nodes 1 and Node 2, and a second beam-column element from Node 2 to Node 3. Each beam-column will have a

material type of `STEEL4` and a section type of `W36x245`.

Unfortunately, the graphical procedure for adding beam-column elements can only be initiated in one of the orthogonal windows. This is due to the fact that in a 3D projection the window matrix `Persp` (see Chapter 3) involves perspective division based on a truncated, or trapezoid shaped, view volume. Referring to Figure 2.6, if a series of lines is projected from the `back clipping plane` to the `front clipping plane`, the angle of intersection between `each line` and the `front clipping plane` will not be ninety degrees. However, for an orthogonal projection, (see Figure 2.7), the view volume is rectangular in shape. In this case, if we draw a series of lines from the `back clipping plane` to the `front clipping plane`, the angle of intersection between `each line` and the `front clipping plane` will be ninety degrees. Thus, all the projectors (or series of lines) are parallel to the viewplane. A mathematical argument for this fact will be presented later in Chapter 3.

Plates

Procedures for adding plate finite elements are similar to those for beam-column elements. The first step is to set the default types for the material type and thickness. We now give a typical expression for setting the material type and thickness.

```
XBUILD >> set material type to STEEL4
XBUILD >> set plate thickness to 2 inches
```

We initiate the graphical plate procedure, by pressing the appropriate pull-down menu button as shown in Figure 2.8. This puts XBUILD in the add plate4 prompt mode; i.e.

```
XBUILD_add_plate4 >>
```

Four node plate finite elements may be added via either keyboard or graphical input. If the graphical option is selected, the engineer will use the `point and click` philosophy. The procedure is started by depressing the left mouse button and moving the cursor. Shortly afterwards, the engineer will see a rectangular box form on the canvas. The box stretches in a style similar to the rubber-band technique [12]. The goal is to stretch the rectangular box over the desired four nodes. Once the four nodes are enclosed, the user can end the procedure by releasing the left mouse button. Now the program will check mathematically to see if the four nodes lie on a plane.

Whenever the user cannot stretch the rectangular box over the desired four nodes due to their projection, the engineer has the option of adding plates via keyboard input. For example, in the command expression,

```
XBUILD_add_plate4 >> Nodes {4,5,6,7} with 2 inches thickness
                      and STEEL4 material
```

the numeric sequence is the nodal connectivity list, and the term STEEL 4 is the type of material according to the LRFD Manual [1]. Figure 2.9 represents two successful plate additions – notice that an interaction in one window will trigger a response in the other three projection windows. Our algorithm draws hatched plate polygons. This method is particularly useful when the engineer is studying structures such as buildings or highway bridges because it allows the engineer to see through the super-structure to the sub-structure. By simply changing the material type, or thickness, the engineer can create a new plate element section and store it by name inside the hash table. Engineers may find this feature useful when they are working with an object such as a highway

bridge which involves several plate elements.

Forces

Due to their complexity, we could not develop an interactive graphical procedure for adding external forces and moments. The following four scripts of code demonstrate keyboard input for nodal forces, linear loads, concentrated loads, and area loads.

nodal force: `XBUILD_add_nodalload >> 20 kips in neg_z direction at Node 4`

lin load: `XBUILD_add_linload >> 20 N/m in pos_x direction along x{0:5}, y{0:10}`

conc. load: `XBUILD_add_concload >> 20 kips in pos_y direction at (10,10,10)`

area load: `XBUILD_add_areaload >> 20 N/(m*m) in neg_x direction y{0:5}, z{0:10}`

The syntax of the expression x{0:5}, y{0:10} means x (0,1,2 $\cdots$ 5) and y (0,1,2,3 $\cdots$ 10) respectively. All types of external loads may be shown in any window. The loads are distinguished from each other by their respective color. A nodal load is yellow, a concentrated load is green, a distributed load is purple, and an area load is gold. All of the loads are represented as solid polygons.

Boundary Conditions

The FERA finite element package allows for six degrees of freedom at each node; three translations (dx,dy,dz) and three rotations (rx,ry,rz) [24]. When developing the boundary condition phase of the pre-processor, we decided to

adopt some the of same terminology already implemented by FERA [24]. From the expression,

```
XBUILD_bound_cond >> Node { 2,3 } DOF[dx,dy,rz]
```

we see that the term DOF[ ] contains the terms [dx,dy,rz]. This means that the engineer is applying full-fixity to the displacements in the x and y directions, and full fixity to the rotation about the z axis. Each kinematic degree of freedom may be suppressed in any combination with the others. When all of the kinematic degrees of freedom are to be fixed, the term DOF[dx,dy,dz,rx,ry,rz] may be abbreviated to DOF[All]. Nodes that represent a free condition are colored blue, a translation is white and a rotation is cyan.

Difficulties with Graphical Input of Complex Structures

If the engineer is analyzing an object that is very large in one or more directions, then there may be some problems with adding finite elements graphically. The reason behind this principle is that the coordinate difference, (max - min), forces the viewport and window to scale the screen coordinate data. The end result is a graphical display which appears smashed together rendering the display unpleasant to the viewer. Here is a civil/structural engineering example. Let's say the engineer was analyzing a very tall bridge. The (x, y, z) coordinate locations of a girder's web and flange will be minute compared to the pier or abutment dimensions. Making the coordinate window size large enough to accomodate the larger dimension will push small objects very close together on the computer screen. As a result, it may be impossible for the user to distinguish the different nodes on these particular segments of the girder. Similar problems will occur for large multi-span highway bridge structures, modeled with hun-

dreds of DKQ plate elements. Adding individual plate elements via keyboard or graphical input will be cumbersome, particulary if the structure is very long.

2.3 Input Data Files

As part of the development of the pre-processor, it was decided to give the user an option of placing all of the command line expressions inside a datafile. Table 2.1 gives an example of such a file. Notice that the command line expressions in Table 2.1 are very similar to the expressions previously given in the chapter. The command syntax for reading a file is the following.

```
XBUILD >> read file "<file_name>"
```

2.3.1 Objects and Bridge Sub-Structures

In XBUILD Version 2.0, objects may be constructed from one or more finite elements that has associated with it a numeric nodal sequence. This nodal sequence will be used as the basis to retrieve the analysis results on the FEM side the architecture. We give an example of an object created with a numeric nodal sequence below.

```
XBUILD >> deck = Nodes { 1,2,3,4 }
```

This command creates an object called "deck" and assigns to it four nodes (1,2,3,4). The object is then loaded into the hash table for present/future use. Figure 2.10 gives a diagram showing specifically how the nodal and element level sequences of this object are interrelated. From Figure 2.10, we see that each `plate (P14)` element has four nodes associated with it while a `beam-column (BC)` and `geometric exact rod (GER)` has two nodes. Figure 2.11 shows a

typical diagram of a hash table. This particular hash table has six **hash nodes** with nodes 1 and 5 each having two **buckets**. The hash nodes are essentially an array of data structures while the buckets comprise a linked list. The objects are placed inside the hash table according to an arithmetic expression using the character string which forms their name. We constructed our YACC grammar such that the engineer builds objects from a nodal, or element level base. The command is typed inside the TTY window and then parsed into it's lexical tokens. Let's say the engineer was studying a particular girder of a bridge structure. Using some of the commands previously given in the pre-processing phase, the engineer can construct the object. If the object already exists inside the hash table, the engineer can load the object with the following command:

```
XBUILD >> girder = Beam_Col { 1 }
```

This expression is defining an object called "girder" and associating with it a beam-column element which has previously been labeled as the first beam-column element inside our database system.

2.3.2 Writing and Posting Files

Before the engineer can begin the post-processing phase of XBUILD, one must generate an input file for the FEM program.

```
XBUILD >> write file "<file_name>"
```

All of the data stored inside the modules for the pre-processor are now emptied. In turn, command line expressions are formulated using this data. These expressions are now compatible for the FEM to read. To post a file across the network, the user must type the following expression.

```
XBUILD >> post file "<file_name>"
```

Once this file has been completely posted across the network, the FEM program will begin to conduct the analysis. We give a formal example of the "posting procedure" in Chapter 3. At this time, the engineer only needs to understand the syntax of the expression. Assuming the file has been posted across the network successfully, the engineer can now begin the post-processing phase and begin to ask queries about the girder.

2.3.3 Post-Processor

The XBUILD post-processor allows engineers to query information about the finite element analysis results or about the problem setup. Response information that may be queried includes nodal displacements, shear reactions, moment reactions, and strains. The user has the option of plotting or drawing (in an appropriate canvas window) or displaying the analysis results (in the TTY window). For instance, one may be interested in the deflection response of the girder defined earlier. A command for this query is as follows.

```
XBUILD_send_message >> print deflection in neg_y direction
                                for Beam_Col {1}
```

Remark: We note that some of the command line expressions establish the feature of encapsulation. Taenzer et al. give a formal definition for encapsulation [25].

> *encapsulation*: means that objects can be treated as black boxes. All that must be known about an object is its protocol or interface, namely, what messages it responds to and what they mean in terms of the object's behavior.

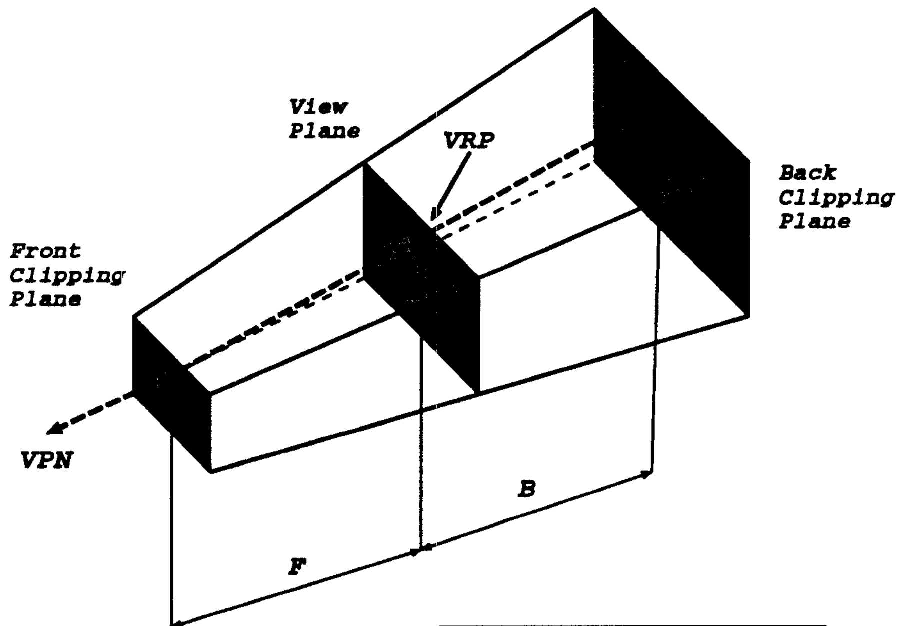
`Encapsulation` is a frequently used technique in `object-oriented programming` and allows the engineer to focus their `mental model` solely on the sub-object(s) they want to study. For the post-processing phase, we developed command messages that would allow the user to request specific information on kinematic displacements (and/or response quantities) of the particular structure(s) the user wished to study. By restricting the graphical view to only the information requested, and interfacing the command language with the `designer's image` of the analysis, we are essentially encapsulating the object. We feel that encapsulation sharpens the engineer's ability to create a mental model of the design object. `Encapsulation` will be discussed further in Section 3.2.

We now give another working example of the post-processing phase procedure. Figure 2.12 shows a picture of a one storey frame. The dimensions and material properties of the frame are not relevant for this case. If the user wanted to view the nodal rotations for the beam-column elements then one would type the following command at the prompt.

```
XBUILD_send_message >> draw All rotations for All Beam_Col
```

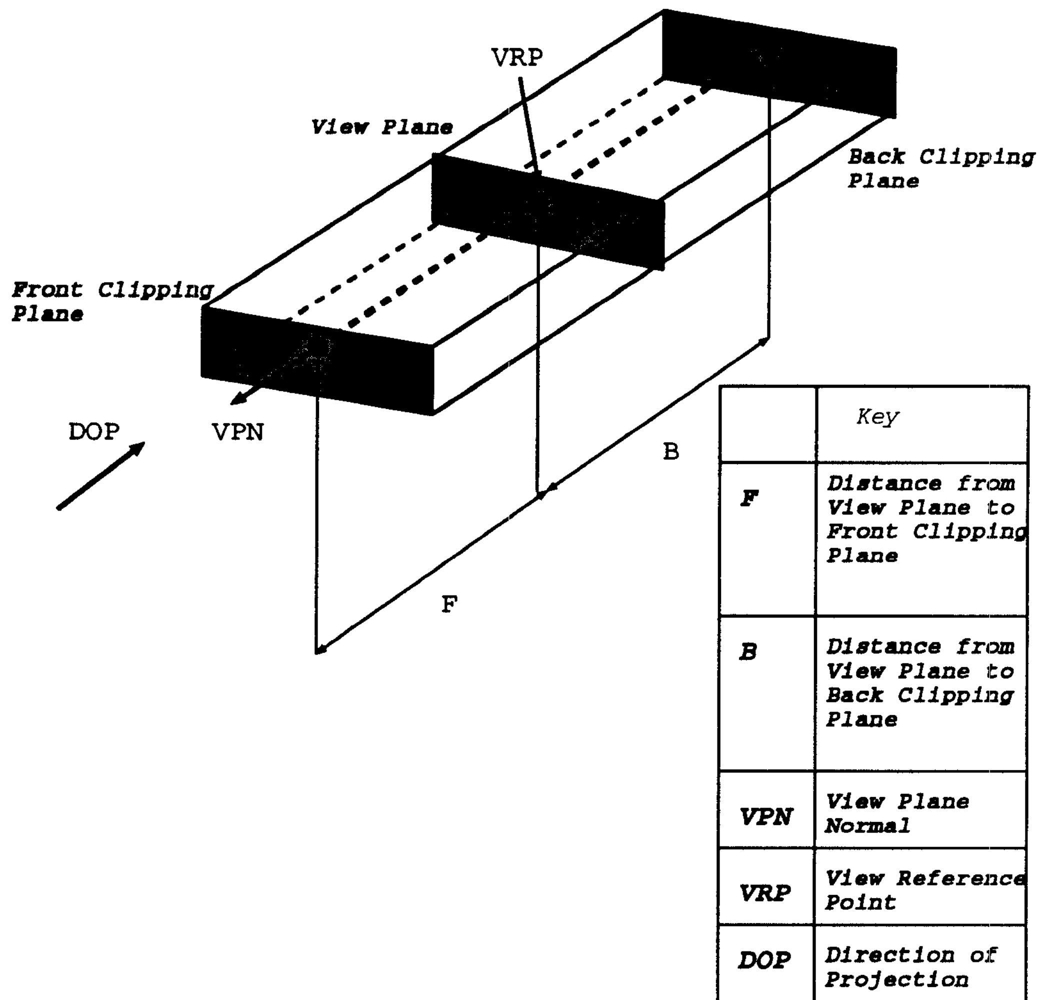
Our program would subsequently generate the rotation analysis plot shown in Figure 2.13. Since the engineer will know the orientation of the coordinate axis on the screen and the element type, the engineer has a preconceived notion of how the displacement pattern will look for this structure. This principle is related to Tomiyama's idea of knowledge engineering [26]. Specifically, the user has depicted a `mental model` inside their mind.

This is one small example of an expression that user can generate in the post-processing phase. After the results have been studied, the user can clear the window and ask for another request pertaining to this structure.



	Key
F	Distance from View Plane to Front Clipping Plane
B	Distance from View Plane to Back Clipping Plane
VPN	View Plane Normal
VRP	View Reference Point

Figure 2.6: 3D View Volume.



	Key
F	Distance from View Plane to Front Clipping Plane
B	Distance from View Plane to Back Clipping Plane
VPN	View Plane Normal
VRP	View Reference Point
DOP	Direction of Projection

Figure 2.7: Orthogonal View Volume.

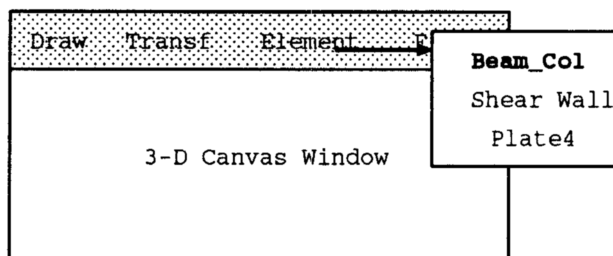


Figure 2.8: Sample of Pull-Down Menu Item.

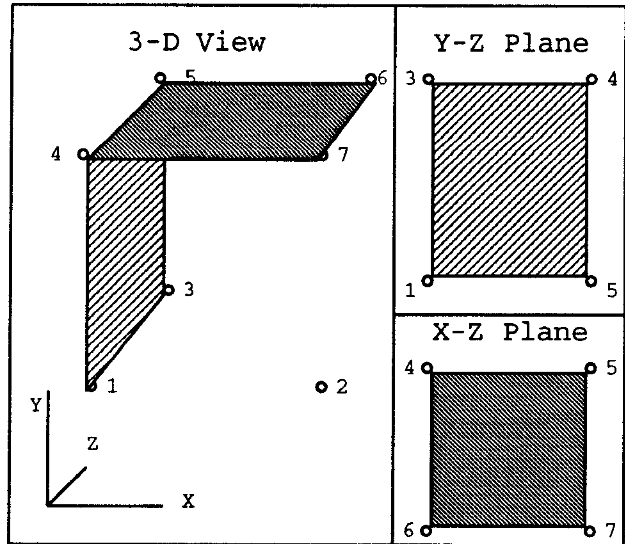


Figure 2.9: Interactive Procedure for Plate.

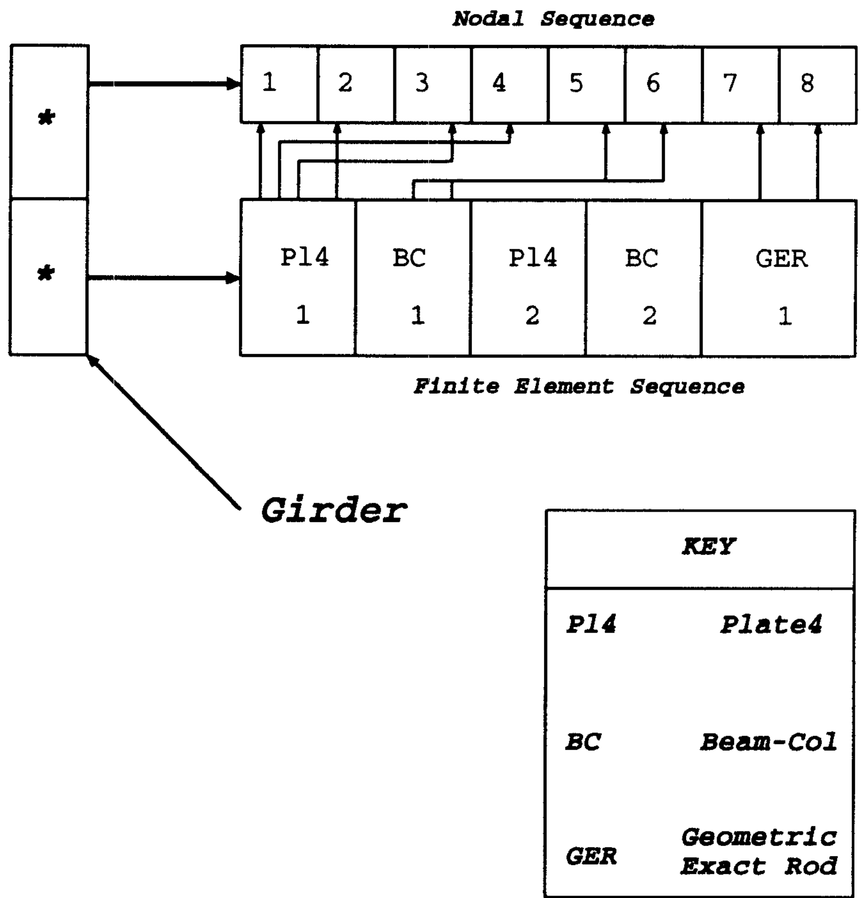


Figure 2.10: Depiction of Nodal and Element Level Sequences.

```
  set plate thickness to 2 inches;

/* Setting the prompt mode to add node */

   add node;

/* Creating the node coordinates */

   z {   0.0} inches  y {    0.0} inches  x {   0.0} inches;
   z {   0.0} inches  y {    0.0} inches  x { 500.0} inches;
   z {   0.0} inches  y {    0.0} inches  x {1000.0} inches;

       ....... code deleted .......

/* Setting the prompt mode to add bc_elmt */

   add bc_elmt;

/* Adding the element sections to the datafile */

   [STEEL4, W16x89] (1,7),   (7,13),  (13,19);
   [STEEL4, W16x89] (15,21), (1,2),   (2,3);

       ....... code deleted .......

   /* Setting the prompt mode to add plate4 */

   add plate4;

/* Adding the plate sections to the datafile */

   Nodes {2,1,4,5} with 8 inches thickness and STEEL1 material;
   Nodes {3,2,5,6} with 8 inches thickness and STEEL1 material;


       ....... code deleted .......
```

Table 2.1: Example of a Finite Element DataFile

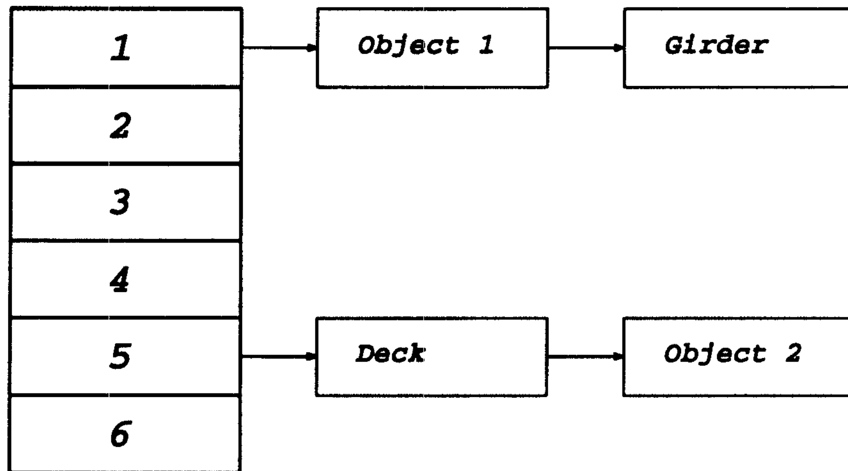


Figure 2.11: Diagram of Hash Table with Typical Storage Scheme of Objects.

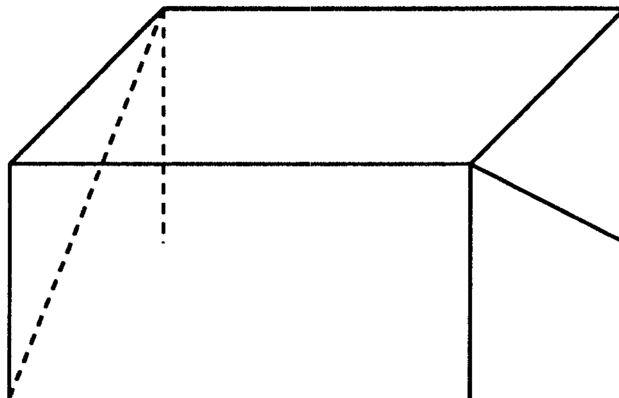


Figure 2.12: One Storey Frame.

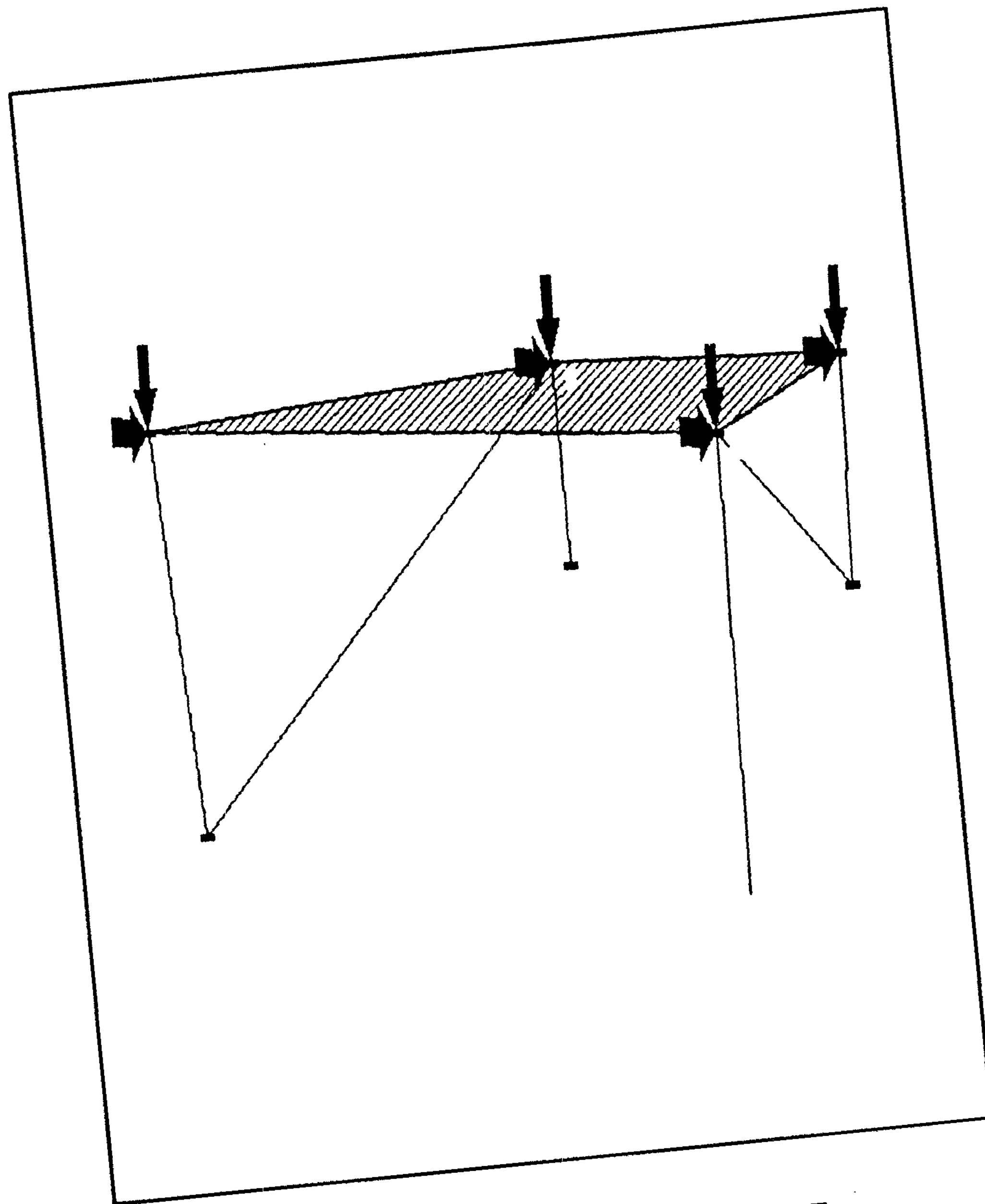


Figure 2.13: Deflected Shape for One Storey Frame.

2.4 Finite Element Libraries

During the time our work was being developed on the pre-processor and post-processor, the work on the finite element libraries proceeded at a much slower pace. Therefore, the analysis of the structures, shown in Chapter 4, is limited to a static analysis. The finite element package supports analysis of linear elastic structures, possibly containing rigid body components. The library of finite elements includes `plane stress-plane strain`, two- and three-dimensional frame elements, and the DKQ plate element. For more information on the finite element program, the reader is referred to Sondhi [24].

2.5 Interprocess Communication

Effective software development allows for upgrades in existing data structures and algorithms. The Interprocess Communication system enables the user to represent the dynamic procedural aspect of the object-oriented paradigm via sending and receiving messages [26]. The communication language that we developed contains keywords which are passed from one workstation to another by the socket-based capabilities of the Interprocess Communication system. For instance, the keywords in

```
XBUILD_send_message >> draw translation in neg_y direction
                          for All Beam_Col
```

are `translation` and `Beam_Col`. From Table 2.2, we see that the list data structure for the message passing system contains four data fields. The data fields

```
typedef struct list {                  typedef enum {
        char         direction[6];             IPC_NODE     = 1,
        int          list_no;                  IPC_ELEMENT  = 2,
        DATA_TYPE    datatype;                 DKQ_PLATE    = 3,
        ELEMENT_TYPE etype;                    IPC_GIRDER   = 4,
} LIST, *LIST_PTR;                     } ELEMENT_TYPE;

typedef enum {
        MESSAGE        = 1,
        DISPLACEMENT   = 2,
        ANGULAR_DISPL  = 3,
        SHEAR_FORCE    = 4,
        MOMENT_COUPLE  = 5,
        STRESS_RESULT  = 6,
        STRAIN_RESULT  = 7
} DATA_TYPE;
```

Table 2.2: List Module for IPC System

that record the keyword information are `etype` and `datatype`. A workstation on the FEM end will interpret this keyword information as follows: On the

FEM end, the bytes of information are read from the socket which were passed through it from the GUI end. While each field of data is being read from the message packet, it is transferred into data structures on the FEM end for future use. This data is then passed through a loop structure which is interconnected with logical if statements. These statements segregate the type of information that can be retrieved. Table 2.3 gives a sample of this code. The method upon

```
case DISPLACEMENT: case ANGULAR_DISPL:
    retrieve_displ_data(tp->displ_type, fe_data,
                        l_ptr[i-1].list_no-1,i-1);
    strcpy(fe_data[i-1].direction,
           l_ptr[i-1].direction);
    fe_data[i-1].node_no  = l_ptr[i-1].list_no;
    fe_data[i-1].datatype = l_ptr[i-1].datatype;
    fe_data[i-1].etype    = l_ptr[i-1].etype;
break;
case SHEAR_FORCE:
    if (l_ptr[i-1].etype == IPC_NODE) {
      retrieve_shear_data(tp->sheartype, fe_data, i-1);
      strcpy(fe_data[i-1].direction,
             l_ptr[i-1].direction);
      fe_data[i-1].node_no  = l_ptr[i-1].list_no;
      fe_data[i-1].datatype = l_ptr[i-1].datatype;
      fe_data[i-1].etype    = l_ptr[i-1].etype;
    }
```

Table 2.3: Example of Block IF Code for IPC System

which to retrieve the displacements, shear reactions, moments reactions, and stresses was dependent upon the finite element libraries that were previously setup.

Chapter 3

OVERVIEW OF COMPUTER GRAPHICS AND IPC MODULES

This chapter describes the data structures and algorithms XBUILD uses in its user interface, and for message passing between components of the client/server model. In this Chapter, we discuss how modules in the GUI-IPC-FEM system should have an open and closed format. A change in one lower level module should not significantly affect the function of modules higher in the hierarchy.

3.1 Data Representation Approaches in Computer Graphics

Pictures ultimately consist of points and a drawing algorithm to display them. The coordinate triplets representing these points are usually stored in some form of a database. Unfortunately, complex pictures require sophisticated databases and data structures. For example, one way to represent the stress contour pattern of a simply-supported beam is to use a very complex form of a tree hierarchy data structure called a matrix quadtree. Similarly, complex objects may encom-

pass several lower level geometric data structures which, when coupled with data structures that store kinematic displacements, can form a very complex system. Some examples of complex data structures include B-Trees, quadtrees, or matrix-quadtrees.

3.1.1 Homogeneous Coordinates

Two types of coordinate systems which the engineer can use to represent coordinate data are the cartesian and homogeneous coordinate systems. Whereas cartesian coordinate systems represent world coordinate data in $\mathbb{R}^n$ space, homogeneous coordinate systems represent world coordinate data in $\mathbb{R}^{n+1}$ space. For the purposes of illustration, suppose that we want to graphically display or manipulate a point $M(x,y)^T$ in $\mathbb{R}^2$ space. The group of equations

$$M' = T + M \quad (Translation) \tag{3.1.1}$$

$$M' = S * M \quad (Scaling) \tag{3.1.2}$$

$$M' = R * M \quad (Rotation) \tag{3.1.3}$$

represent translation, rotation, or scaling transformations in cartesian coordinates. Here M' is the result of mathematical computation, and T, R, and S are defined as follows

$$\mathrm{T(dx,\ dy)} = \begin{pmatrix} dx \\ dy \end{pmatrix} \qquad \mathrm{S(sx,\ sy)} = \begin{pmatrix} sx & 0.0 \\ 0.0 & sy \end{pmatrix}$$

$$\mathrm{R}(\Theta) = \begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix}$$

Notice that the equation for translation involves addition, whereas the equations for scaling and rotation involve multiplication. Hence, translation is not a

matrix linear transformation. The general form of the above translation equation is now given.

$$Ax + b \tag{3.1.4}$$

We see that this equation involves a linear mapping if b = 0.0. We would like to be able to work with the three different matrix operations (translation, scaling and rotation) in a consistent way (i.e. multiplication). This would make it easier to form compound expressions. Compound expressions are frequent occurences in complex graphical operations and may require the concatenation of several different matrices (i.e. the need for a matrix stack). One way to achieve this goal is to use homogeneous coordinates.

For the case of a two-dimensional system, we would represent a coordinate triple using the following notation – (x, y, W). Although, the homogeneous triplet is in $\mathbb{R}^3$ space, it is actually representing a coordinate in $\mathbb{R}^2$ space. For instance, if we have a coordinate triple defined as $(1,2,1)$, then $(2,4,2)$ is a multiple of it. The net result is a line in $\mathbb{R}^3$ space. If the W coordinate is set to one, then (x, y) pairs now lie in the plane in the $(x, y, 1)$ system as shown in Figure 3.1. Using homogeneous coordinates, we get the following matrix equations for translation, scaling and rotation.

$$M' = T * M \quad (Translation) \tag{3.1.5}$$

$$M' = S * M \quad (Scaling) \tag{3.1.6}$$

$$M' = R * M \quad (Rotation) \tag{3.1.7}$$

where M' = new $(\mathbf{n} \times \mathbf{1})$ matrix and T, R, and S $\in \mathbb{R}^{(3\times3)}$ space

$$\text{T(dx, dy)} = \begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ dx & dy & 1.0 \end{pmatrix} \qquad \text{S(sx, sy)} = \begin{pmatrix} sx & 0.0 & 0.0 \\ 0.0 & sy & 0.0 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$$

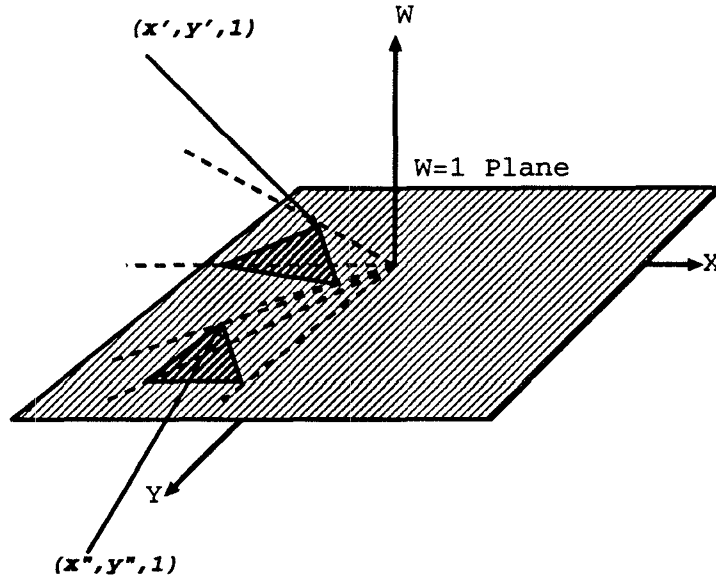


Figure 3.1: Homogeneous Coordinate Plane.

$$R(\theta_y) = \begin{pmatrix} cos\theta & -sin\theta & 0.0 \\ sin\theta & cos\theta & 0.0 \\ 0.0 & 0.0 & 1.0 \end{pmatrix} \qquad M(x, y) = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

These matrices apply for a left-handed coordinate system. Now we see that each matrix operation involves multiplication and that two successive translations, rotations or scalings may be represented as a matrix product. We give an example for two successive translations and leave scaling and rotation for the reader.

$$M'(dx_2, dy_2, dx_1, dy_1) =$$

$$T(dx_2, dy_2) * T(dx_1, dy_1) = \begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ dx_2 & dy_2 & 1.0 \end{pmatrix} * \begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ dx_1 & dy_1 & 1.0 \end{pmatrix}$$

$$= \begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ dx_1 + dx_2 & dy_1 + dy_2 & 1.0 \end{pmatrix}$$

In every case, the second translation function T(dx_2, dy_2) is pre-multiplied with the first translation function T(dx_1, dy_1).

Remark on Homogeneous Coordinates: If integers are used to store the position of a point (x, y, W), then the range of integer numbers is 2^{n-1} - 1, where n = the number of bits in the word. When an integer number outside of this range is encountered, the user cannot control the outcome of the result [23]. Even if the engineer attempted to alleviate this problem using absolute and relative coordinates, the result would be the same [23]. In both cases, the computer could encounter a range of numbers outside of it's storage capability. Hence, the printed result would be either underflow or overflow. The way out of this dilemma is to use real number homogeneous coordinates. However, the use of real number homogeneous coordinates introduces some additional constraints. For instance, Rogers states that their use creates some loss in speed, and some loss in resolution [23].

3.1.2 View Transformation Matrix

Before an object can be displayed on the computer screen, the object's world coordinate system must be mapped to the screen coordinate system as represented by pixel locations. Many computer graphics programs construct an algorithm for this procedure using two components, namely the window and viewport matrices. Because matrix multiplication is associative, it is possible to create a matrix expression that is the product of several smaller expressions. One such

matrix expression is that of the view transformation matrix (VTM). The (VTM) is constructed to allow the user flexibility in setting up the eye location of the viewing system. There are many types of views a user may create. For further reading on the types of views one may setup, the user is referred to Foley & Van Dam [12].

Window Systems

To be able to display world coordinate data from the object to pixel coordinates on the screen, world coordinates must be transferred into screen coordinates (represented as pixel locations). The first step is to construct the window matrix. `PerspW` is the matrix for an orthographic projection while `ParllW` is the matrix for an orthogonal projection. The window coordinates are input using the `world coordinate system`.

$$\text{PerspW} = \begin{pmatrix} 2.0 * n/(r-l) & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 * n/(t-b) & 0.0 & 0.0 \\ (r+l)/(r-l) & (t+b)/(t-b) & -(f+n)/(f-n) & -1.0 \\ 0.0 & 0.0 & -2.0 * f * n/(f-n) & 0.0 \end{pmatrix}$$

$$\text{ParllW} = \begin{pmatrix} 2.0/(r-l) & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0/(t-b) & 0.0 & 0.0 \\ (r+l)/(r-l) & (t+b)/(t-b) & -2/(f-n) & 0.0 \\ -(r+l)/(r-l) & -(t+b)/(t-b) & -(f+n)/(f-n) & 0.0 \end{pmatrix}$$

The definitions of the terms used in these matrices are given as follows:

$on = near$

$of = far$

$ob = bottom$

$$ot = top$$

$$ol = left$$

$$or = right$$

Viewport Systems

The window matrix and viewport coordinates work in tandom to represent the world-coordinate data. The viewport is responsible for capturing the window data that can be represented on the display device. A viewport can be described as a rectangle with coordinate values ranging from 0.0 to 1.0. A window system may contain more than one viewport. The coordinate values are defined in the **`screen coordinate system`**.

If the window system and the viewport system have the same size or proportions, then the object will not suffer from any scaling effects. If they differ, then the object will need to be scaled such that it will fit inside the window. Essentially, we can say that the window-viewport system determines which pixels can be drawn on the screen and which cannot. This process is known as `clipping` the world-coordinate data. Foley & Van Dam's definition of a window-viewport relationship is shown in Figure 3.2 [12]. The first diagram shows the dimension of the window. This window is then translated to the origin as seen from the second diagram. Next, the window is scaled to the size of the viewport. Along with Figure 3.2, the matrix representation for this transformation is given for M_{vw}.

$$M_{vw} = [T(u_{min}, v_{min}) * S\left[\frac{u_{max}-u_{min}}{x_{max}-x_{min}}, \frac{v_{max}-v_{min}}{y_{max}-y_{min}},\right] * T(x_{min}, x_{min})]$$

where T = translation matrix and S = scaling matrix

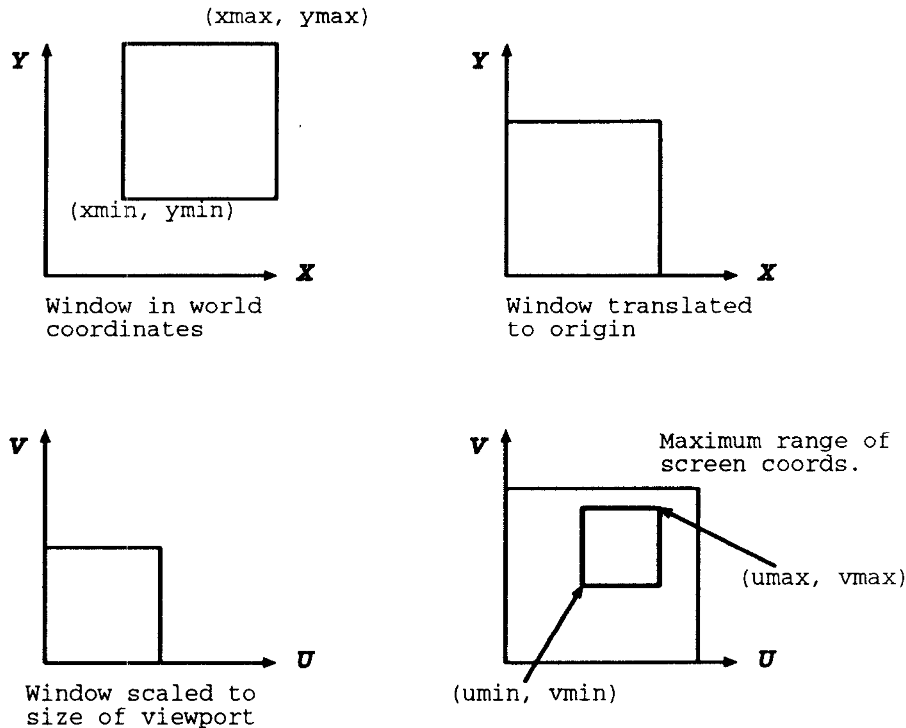


Figure 3.2: Transforming world-coordinates to window-coordinates.

$$M_{vw} = \begin{pmatrix} \frac{u_{max}-u_{min}}{x_{max}-x_{min}} & 0.0 & -x * \frac{u_{max}-u_{min}}{x_{max}-x_{min}} + u_{min} \\ 0.0 & \frac{v_{max}-v_{min}}{y_{max}-y_{min}} & -y * \frac{v_{max}-v_{min}}{y_{max}-y_{min}} + v_{min} \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$$

Lookat Functions

Orthographic view matrices are defined in terms of two coordinate triplets and an angle of twist. As seen in Figure 3.3, one coordinate triplet is located as the viewing endpoint and a second is located where the person would be standing in $\mathbb{R}^3$ space. We designate the coordinate triplet known as the viewing endpoint as **(vx,vy,vz)** and the coordinate triplet of the viewer as **(px,py,pz)**. Together their vector forms the line of sight. The angle of twist allows the user to rotate the eye position about the z axis in the eye coordinate system.

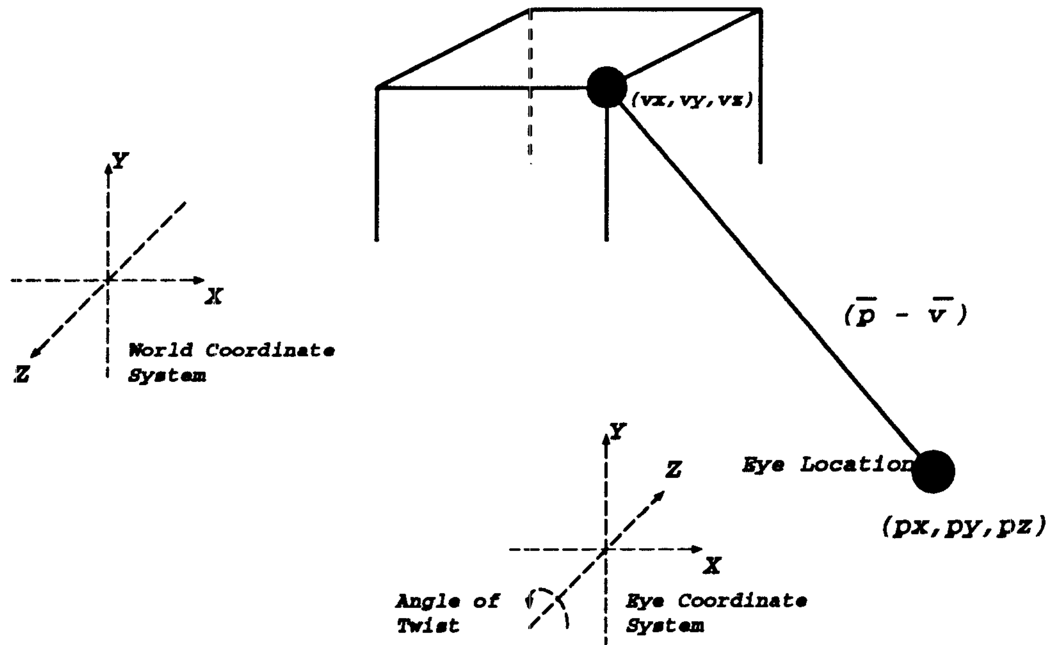


Figure 3.3: Figure Depicting Definitions of Parameters for View Setup.

To view an object on the screen, the engineer will need to input the coordinate locations for (px,py,pz) and (vx,vy,vz). These two coordinate locations form the argument list for the `lookat` function. The `lookat` matrix will then be premultiplied with a rotation matrix followed with the `PerspW` matrix, and finally a translation matrix to form the view transformation matrix VTM. It should be noted, however, that the coordinates in the `world coordinate system` will first need to translated and rotated to the origin of the `screen coordinate system`.

3.2 Forward Mapping from World to Screen Coordinates

Now that the primary matrix components of a computer graphics system have been discussed, we can focus our attention on the procedure to transform world coordinate data to pixel coordinate data. This transformation involves three different coordinate systems. We previously displayed the relationship between the world coordinate system and the eye coordinate system in Figure 3.3. Figure 3.4 shows the relationship between the eye coordinate system and the screen coordinate system [19]. After the VTM has been setup, the world coordinate

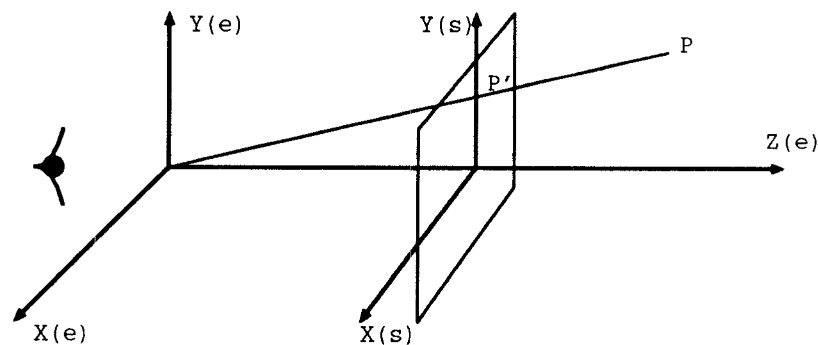


Figure 3.4: Relationship of Eye-Screen Coordinate Systems.

data is ready to post-multiplied with the VTM. A matrix expression for this mathematical computation is shown below.

$$[x^*, y^*, z^*, w^*]^T = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} * [x, y, z, 1]^T$$

where $[x^*, y^*, z^*, w^*]$ = view coordinates, $[x, y, z, 1]$ = homogeneous world coordinates, and $[a \cdots p]$ = real number expressions

Upon multiplication, the new coordinate quadruple (x^*, y^*, z^*, w^*) is in the eye coordinate system. Now the goal is to move from the eye coordinate system to the screen coordinate system. We will use the mathematical relationship for a perspective projection to establish this transformation.

From Figure 3.5, we see that by using similar triangles $x_s/D = x_e/z_e$ and $y_s/D = y_e/z_e$ [19]. Dividing by the screen size,S, the equations for x_s and y_s become dimensionless.

$$x_s = (D * x_e)/(S * z_e) \tag{3.2.8}$$

$$y_s = (D * y_e)/(S * z_e) \tag{3.2.9}$$

where S = screen size, and D = distance from eye to screen window

An alternative method involves using the size and center coordinates of the canvas window or screen. In the screen coordinate system, the terms V_{sx}, V_{sy} are the size of the canvas window, while the terms V_{cx}, V_{cy} are the location of the center. These four terms are related to the variable S in Figure 3.5 [19].

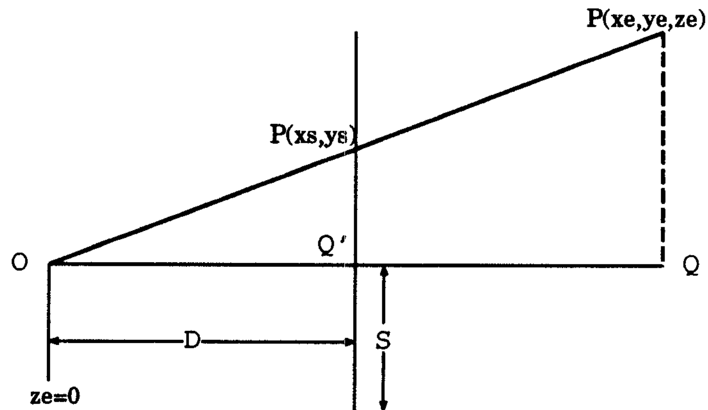


Figure 3.5: The (xs,ys,zs) plane showing details of the perspective projection.

We begin with terms that represent the location of the screen center. To find the center of the screen in the x-direction, we need the coordinate locations

of the left and right side. If we add these two terms and divide by two, our expression would be as follows.

$$V_{cx} = (left + right)/2.0 \qquad (3.2.10)$$

For the y-direction, we would get a similar expression.

$$V_{cy} = (top + bottom)/2.0 \qquad (3.2.11)$$

The size of the screen, in both the x and y directions, is formulated by using the same terms but this time we subtract their coordinate values.

$$V_{sx} = (left - right)/2.0 \qquad (3.2.12)$$

$$V_{sy} = (top - bottom)/2.0 \qquad (3.2.13)$$

Now we are ready to develop the equations for the pixel coordinates or screen coordinates.

$$device_x = x_e * V_{sx}/w_{ex} + V_{cx} \qquad (3.2.14)$$

$$device_y = y_e * V_{sy}/w_{ey} + V_{cy} \qquad (3.2.15)$$

where $w_{ex} = D * z_e/S$, $w_{ey} = D * z_e/S$.

Hence, the equations for $(device_x, device_y)$ are just another way to write those for (x_s, y_s).

The equations for $device_x$ and $device_y$ can now be placed in matrix form. In this section, we will derive the terms for `ParllW` and leave `PerspW` for the reader. If we set the variables w_{ex} and w_{ey} equal to one, then the equations for $(device_x, device_y)$ become the following.

$$device_x = x_e * V_{sx} + V_{cx} \qquad (3.2.16)$$

$$device_y = y_e * V_{sy} + V_{cy} \qquad (3.2.17)$$

This only provides us with the expressions for the x and y direction. However, for the z-direction we get a similar expression.

$$device_z = z_e * V_{sz} + V_{cz} \tag{3.2.18}$$

In this case the term V_{sz} refers to the spacing size of the two clipping planes

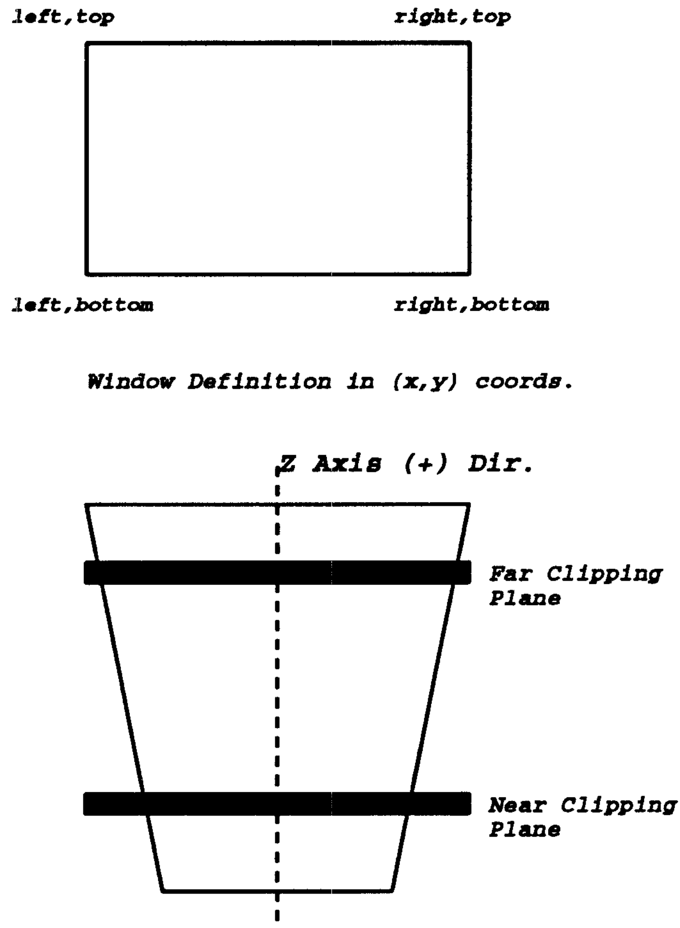


Figure 3.6: Orthographic View Volume Definitions.

shown in Figure 2.7 while the term V_{cz} refers to the z-coordinate location of the view plane. Figures 3.6 and 3.7 show two dimensional diagrams of the 3-D view volumes and orthogonal view volumes defined in Figures 2.6 and 2.7 respectively. These figures show the clipping planes and their relationship to

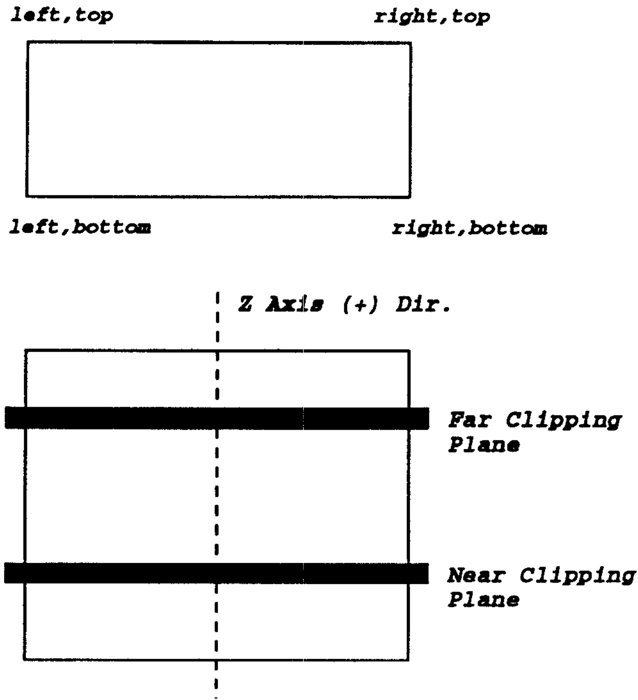


Figure 3.7: Orthogonal View Volume Definitions.

the view volume. The fundamental role of the clipping planes is to determine which coordinate data will be `clipped out`.

$$device_z = z_e * V_{sz} + V_{cz} \tag{3.2.19}$$

Placing $device_x$, $device_y$ and $device_z$ in matrix form, we get the following expression.

$$\begin{pmatrix} x_s \\ y_s \\ z_s \\ w_s \end{pmatrix} = \begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix} * \begin{pmatrix} V_{sx} & 0.0 & 0.0 & 0.0 \\ 0.0 & V_{sy} & 0.0 & 0.0 \\ 0.0 & 0.0 & -V_{sz} & 0.0 \\ -V_{cx}/V_{sx} & -V_{cy}/V_{sy} & -V_{cz}/V_{sz} & 0.0 \end{pmatrix}$$

Figures 3.6 and 3.7 show two dimensional diagrams of the 3-D view volumes and orthogonal view volumes defined in Figures 2.6 and 2.7 respectively. These figures show the clipping planes and their relationship to the view volume. The

fundamental role of the clipping planes is to determine which coordinate data will be `clipped out`.

Figure 3.8 shows a single point perspective which may help in the understanding of the expressions for $(device_x, device_y)$ [23]. Figure 3.8 is the perspective projection of a point (x, y, z) to the point (x^*, y^*, z^*). Using similar triangles, we can determine the mathematical relationship between the two sets of coordinate points.

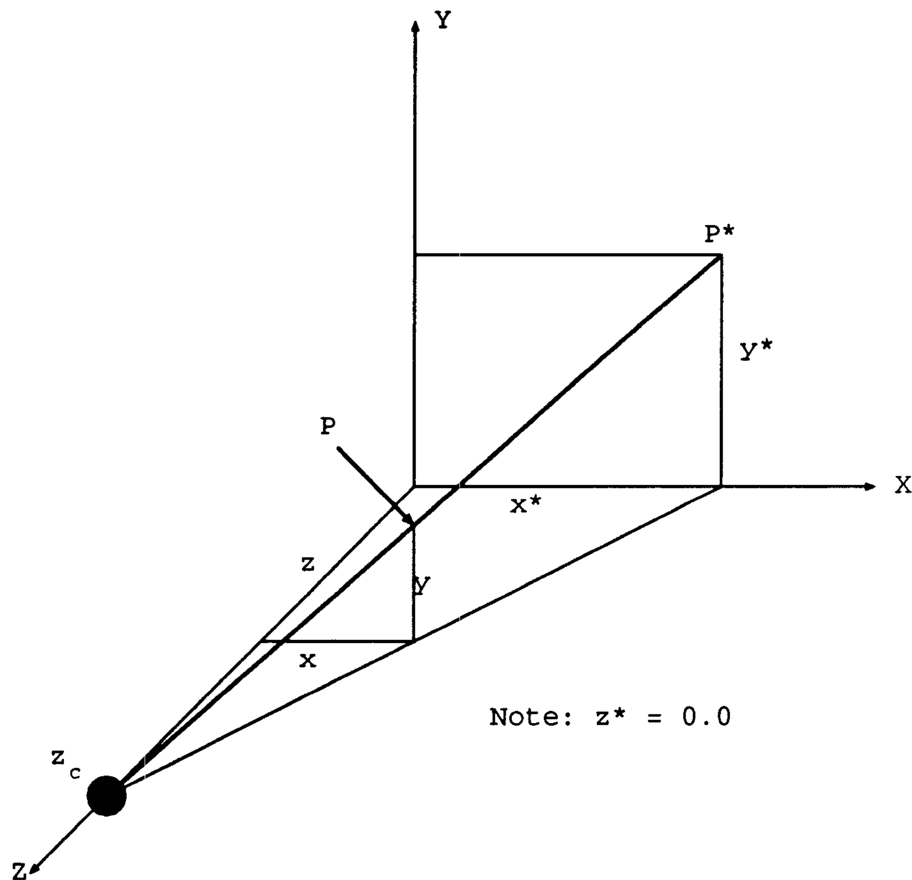


Figure 3.8: Single Point Perspective Projection.

$$x^*/z_c = x/(z_c - z) \tag{3.2.20}$$

or

$$x^* = x/(1 - z/z_c) \tag{3.2.21}$$

$$y^*/\sqrt{x^{*2} + z_c^2} = y/\sqrt{x^2 + (z_c - z)^2} \qquad (3.2.22)$$

or

$$y^* = y/1 - (z/z_c) \qquad (3.2.23)$$

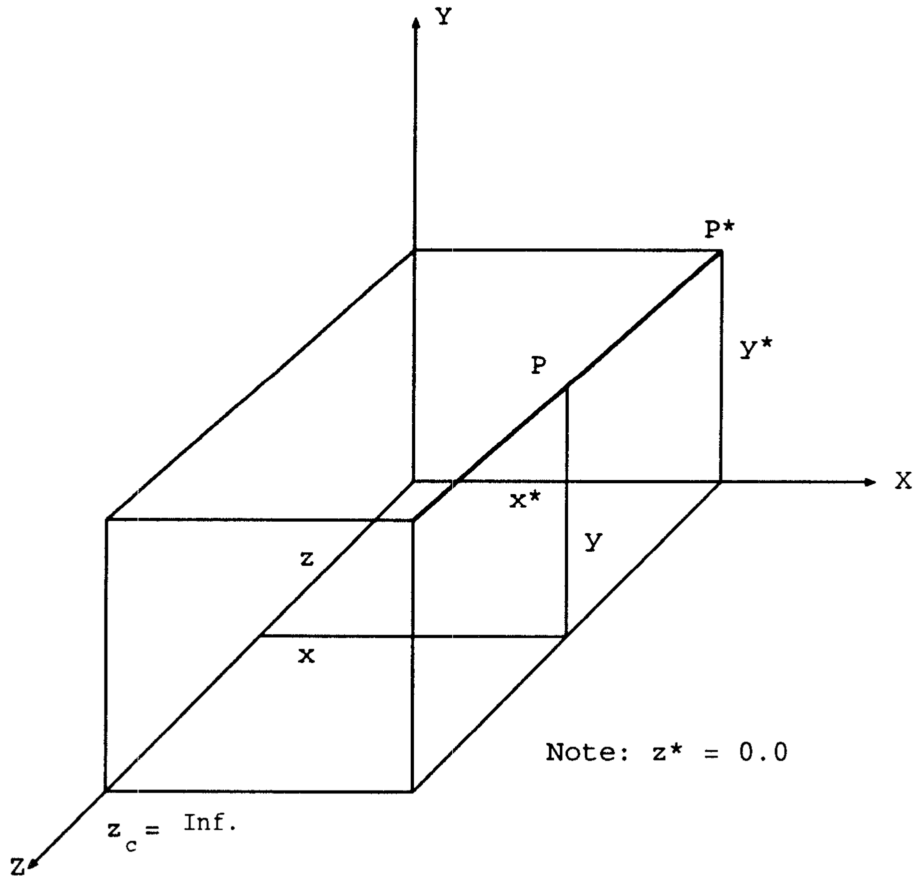


Figure 3.9: Single Point Parallel Projection.

For a parallel projection, we would get a very different relationship. The parallel projection of a point (x, y, z) to the point (x^*, y^*, z^*) is depicted in Figure 3.9. From Figure 3.9, and Figure 2.7, we can see that all of the projectors are parallel to the viewplane. Now, since all of the projectors are parallel to the viewplane, the mathematical relationship is direct. To be specific, the only loss in precision occurs when the program converts the world coordinates from a

real value to an integer value. We now provide the mathematical relationship between the two sets of coordinate points.

$$x^* = x \tag{3.2.24}$$

$$y^* = y \tag{3.2.25}$$

Hence, if one wanted to add a beam-column element from `node i` to `node j`, they could take advantage of this relationship and develop a graphical interactive procedure. We decided to use this procedure for the graphical addition of beam-column elements.

3.3 Reverse Mapping from Screen to World Coordinates

Now that the progression from world coordinates to screen coordinates has been discussed, we will mention how this procedure can be reversed for orthogonal projections. Naturally, the logical sequence would be from screen coordinates to world coordinates. The first step is to transform the (x, y) pixel coordinates back to the eye coordinate system. We can accomplish this by using the equations for x_s and y_s which were derived in the previous section.

$$x_e = (x_s - V_{cx})/V_{sx} \tag{3.3.26}$$

$$y_e = (y_s - V_{cy}) + V_{sy} \tag{3.3.27}$$

Since we are now working in the eye-world coordinate system, our ultimate goal is to transform these eye coordinates back into world coordinates. However, it should be mentioned that in the eye-world coordinate system, the terms V_{sx}, V_{cx}, V_{sy} and V_{cy} refer to the window dimensions not the canvas dimensions.

This next paragraph defines the terms which are used to form the final equation for the transformation process. The first four variables are related to the perspective division for the x-coordinate.

$$x1 = ParllW[1][1] - ParllW[1][4] * x_e \tag{3.3.28}$$

$$y1 = ParllW[1][2] - ParllW[1][4] * y_e \tag{3.3.29}$$

These two equations can actually be written as follows.

$$x1 = V_{sx} - (S * z_e/D) * x_e \tag{3.3.30}$$

$$y1 = V_{sy} - (S * z_e/D) * y_e \tag{3.3.31}$$

However, for an orthogonal projection D = ∞. Thus, our expressions become $x1 = V_{sx}, y1 = 0$. Our next two equations are related to the perspective division in the y-coordinate.

$$x2 = ParllW[2][1] - ParllW[2][4] * x_e \tag{3.3.32}$$

$$y2 = ParllW[2][2] - ParllW[2][4] * y_e \tag{3.3.33}$$

Reducing these equations, we get a much nicer form.

$$x2 = 0 - (S * z_e/D) * x_e \tag{3.3.34}$$

$$y2 = V_{sy} - (S * z_e/D) * y_e \tag{3.3.35}$$

Again, since D=∞, our equations simplify to $x2 = 0, y2 = V_{sy}$. Now we are ready to move on to the terms that are involved with the scaling.

$$s1 = ParllW[4][4] * x_e - ParllW[4][1] \tag{3.3.36}$$

$$s2 = ParllW[4][4] * y_e - ParllW[4][2] \tag{3.3.37}$$

$$s1 = (S * z_e/D) * x_e + V_{cx}/V_{sx} \tag{3.3.38}$$

$$s1 = (S * z_e/D) * y_e + V_{cy}/V_{sy} \tag{3.3.39}$$

If we insert all of the above terms in the following manner, we can achieve the world coordinates in both the x and y directions.

$$world_x = (x2 * s2 - s1 * y2)/(y1 * x2 - y2 * x1) \tag{3.3.40}$$

$$world_y = (x1 * s2 - s1 * y1)/(x1 * y2 - x2 * y1) \tag{3.3.41}$$

In conclusion, the equations to transform pixel coordinates to world coordinates for orthogonal projections using our window matrix `ParllW` is as follows.

$$world_x = ((x_e + V_{cx_{window}})/V_{sx_{window}}) + V_{sx_{window}} \tag{3.3.42}$$

$$world_y = ((y_e + V_{cy_{window}})/V_{sy_{window}}) + V_{sy_{window}} \tag{3.3.43}$$

3.4 Pushing and Popping of Matrix Stacks

In general, matrix multiplication is not commutative (i.e. AB $\neq$ BA). This property of matrix algebra leads to computational difficulties when we are rotating or translating an object on the screen. Consider the problem of translating an object in the x-direction in real time. The step-by-step procedure is as follows:

1. Setup the View Transformation Matrix (VTM).

2. Input the (x,y,z) coordinate values into the argument list of the translation function, T_1.

3. Pre-Multiply the Translation Matrix times the View Transformation Matrix.

4. "Pop off" the Current Translation Matrix from the stack. This leaves the (VTM) matrix.

5. Repeat as Steps (2,3,4) as necessary.

This procedure gives us a very simple mathematical relationship.

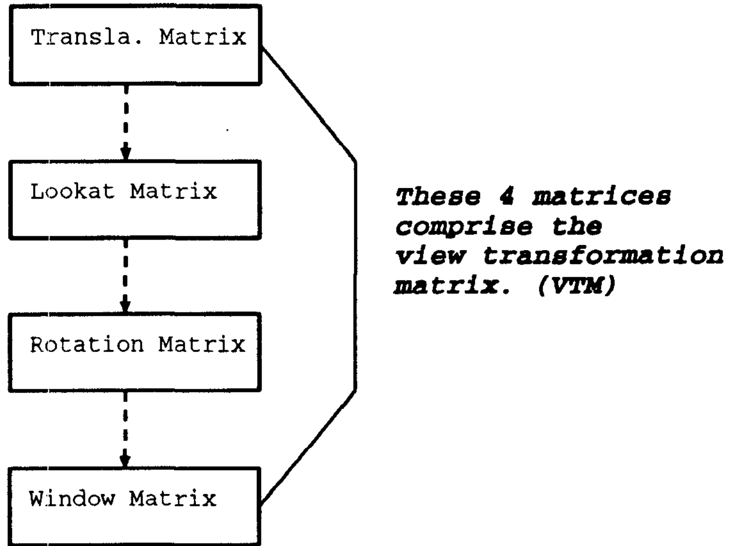
$$M' = T_1 * VTM \tag{3.4.44}$$

We notice the VTM remains unchanged once it is setup and each time a new coord location is given, the old coordinate location is not needed. We cannot, however, simply multiply a new translation matrix, T_2, with the previous mathematical expression. This would give us an expression as follows:

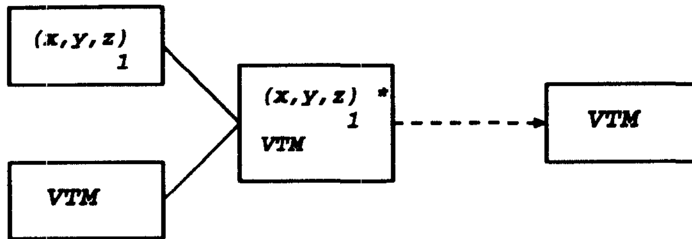
$$M'' = T_2 * T_1 * VTM \tag{3.4.45}$$

Now we have two translation matrices stacked on top of a view transformation matrix. This relationship does not function the same as the one given for M'. The expression for M'' means to take the newly translated engineering system, and then to translate the eye location along with the engineering system. The behavior of this formula does not match that for M'. To make them similar, we need to remove the old translation matrix, T_1, from the matrix equation for M'. The only way to do this mathematically is to find it's inverse, but the inverse of a matrix may not exist. Computer scientists have come up with an alternative solution that is based on the following observation. If we knew ahead of time that we wanted to perform operations with the VTM, it would make sense to store this original matrix so that we could always return back to it. Once we have finished our translation or rotation sequence, we could easily discard this new matrix, $(T_1 * VTM)$, and begin again with the initial (VTM). This allows us to always have the same fixed reference point in $\mathbb{R}^{(n \times n)}$ space.

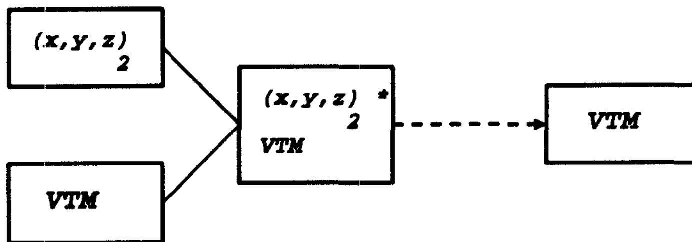
From Figure 3.10, we see that for `pushing`, the coordinate translation matrix is stacked on top of the VTM giving us the product $(T * VTM)$. For `popping`, the product, $(T * VTM)$, is removed from the matrix stack leaving us with VTM.



(Pushing Sequence for (VTM)



STEP 1 (Pushing) STEP 2 (Popping)



STEP 3 (Pushing) STEP 4 (Popping)

Figure 3.10: Two formal sequences of pushing and popping matrix stacks.

3.5 A Discussion on Three Data Structures for our System

It should be noted that it was our intention to be able to create objects both graphically and with keyboard commands. Once these objects are created, they may be stored inside the hash table for future use or retrieval. These objects, in conjunction with the Interprocess Communication system, allow the user to develop a pseudo object-oriented programming system. By storing these objects in the hash table, the user may develop language messages specifically curtailed to questions about the behavior of this object, both before and after the analysis phase of the program. Throughout the next three subsections, we describe the functionality of three of the data structures associated with highway bridge objects and message passing.

3.5.1 Object Data Structure

Models of highway bridge systems can be broken down into several smaller component modules called sub-objects. For instance, associated with the structural object system can be the smaller components known as element types, nodal coordinates, loading conditions, boundary conditions, and material properties. An example for a highway bridge system is shown in Figure 3.11. The mechanism of breaking down the original structure into smaller ones is related to the principles of `object-oriented programming` (OOP). Two features of object-oriented programming that take advantage of this ability are `encapsulation` and `inheritance`. We previously discussed `encapsulation` in Section 2.2.2. A primary feature of inheritance is that it allows objects to be categorized into specific classes. These classes could be in terms of the objects behavior, material

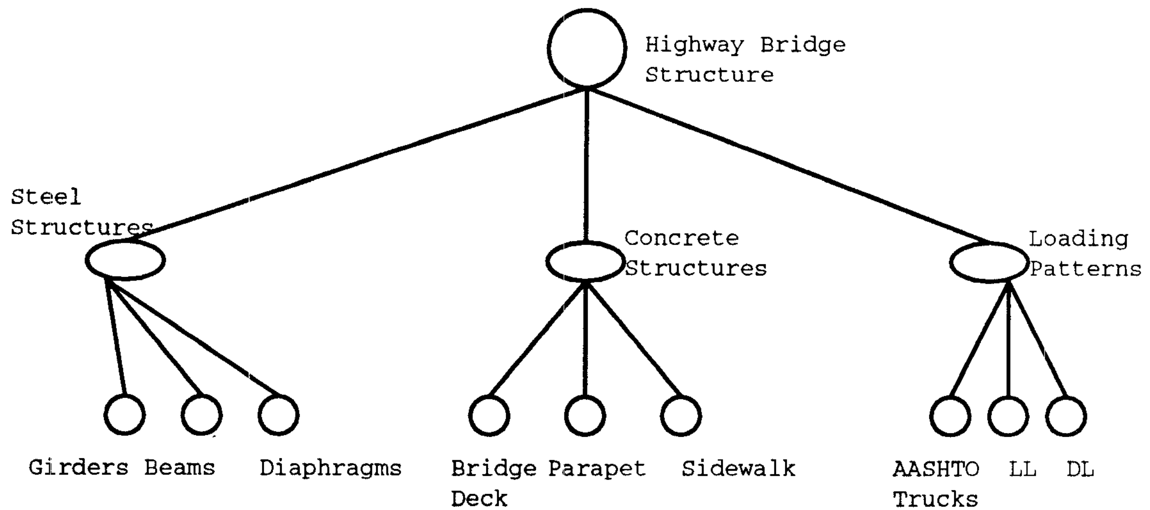


Figure 3.11: Hierarchical Object Tree for Typical Bridge Structure.

property, or function. Taenzer et al. give a formal definition for inheritance, namely [25]:

> *inheritance*: refers to the ability to describe new objects or classes of objects by specifying how they differ from other objects. In theory, this can be done without understanding the implementation details of the parent object. A user should be able to define new classes via subclassing by simply understanding the protocol of the parent class.

The feature encapsulation allows objects to be represented in terms of classes or instances. We can treat objects that behave the same according to a set of methods as a class, and call these objects instances of this particular class [26]. Shown in Table 3.1 are the modules which we used to build the object, plate, and element sub-structures respectively. One can see that the data fields inside the respective data structures are task specific. Task specific modules usually follow a top-down/decomposable hierarchical format. Essentially these types of

```
type struct object {
     char *name;
     int no_nodes;
     int no_elements;
     struct node *node;
     struct element *elmt;
     struct pl4_section *plate;
     struct conc_load *point_load;
     struct linr_load *linear_load;
     struct truck_list *list_truck;
     struct bcond_set *bound_cond;
     struct object *next;
} OBJECT, *OBJECT_PTR;

typedef struct pl4_section {
        char    *name_1;
        char    *name_2;
        float   thickness;
        int     node_count;
        struct  material *matr_ptr;
        struct  arglist *plate_list;
} PL4_SECTION, *PL4_SECTION_PTR;

type struct element {
     int nodes[2];
     struct material    *mat_p;
     struct bc_section  *sec_p;
} ELEMENT, *ELEMENT_PTR;
```

Table 3.1: The Object Type Data Structure Modules

systems can be modeled as an inverted tree.

Chin and Chanson define a programming language as `object based` if it supports objects as a language feature and `object-oriented` if it also supports the concept of inheritance [7]. Table 3.2 shows an example of our language constructs for building objects with either nodal or element level numeric lists.

The method of constructing objects from these three rules occurs using the following procedure.

1. Create a name for the object. Examples include (girder, diaphragm, I83 bridge).

```
assign_list :  VARIABLE '=' MATH2 '(' numer_list ','
                             numer_list ')' { ;}          <Rule 1>
            |  VARIABLE '=' NODES numer_list    { ;}     <Rule 2>
            |  VARIABLE '=' ELMTS numer_list    { ;}     <Rule 3>
            ;

numer_list : '{' numer_seq '}'      { ;}                 <Rule 4>
           | NUMERICAL_LIST         { ;}                 <Rule 5>
           ;

numer_seq  : expr                               { ;}     <Rule 6>
           | expr TO expr                       { ;}     <Rule 7>
           | expr TO expr BY expr               { ;}     <Rule 8>
           | numer_seq ',' expr                 { ;}     <Rule 9>
           | numer_seq ',' expr TO expr         { ;}     <Rule 10>
           | numer_seq ',' expr TO expr BY expr { ;}     <Rule 11>
           | numer_seq ':' EXCEPT expr          { ;}     <Rule 12>
           ;
```

Table 3.2: Language Constructs for the YACC Grammar

2. Give a nodal or element numeric sequence for the nodes/elements desired to formulate the geometry. Examples are as follows: girder1 = Nodes {1,2,3,4}

   Diaphragm = Beam_Col {20,21,22,23}.

Let's say for example that the engineer had the intention of studying the deflection behavior of a highway bridge deck with differing concrete compressive strengths. In order to satisfy Chin and Chanson's criteria and definition of object based, we need to be able to formulate an expression which can store the name of the object and it's respective physical data. We refer the reader to Table 3.2 and investigate the meaning of Rules (1-3). Rule 1 allows the engineer to combine two separate lists into one by finding their intersection, union, or difference. These three properties are directly related to the lexical token MATH2. Rules (2,3) allow the engineer to create objects from a numeric nodal or element level list. Finally, Rules (6-12) set options for the engineer to create the numeric

expressions. A formal example of creating an object with nodes is given below.

```
XBUILD >> "bridge_deck" = Nodes {1 to 80}
```

In continuation of our discussion of `object based` and `object-oriented`, we move on to Chin & Chanson's definition of `object-oriented`. We previously defined inheritance as the ability to describe new objects or classes of objects by specifying how they differ from other objects. Using the expression for "bridge_deck", let's say we are studying the deflection response of highway bridge decks with differing concrete compressive strengths. According to the definition of inheritance by Taenzer et. al, each bridge deck can be classified as a new object [25]. But, since the deflection behavior follows the same mathematical relationship, they actually belong to the same class. At the post-processing phase, the user will know intuitively that the analytical behavior will differ. These behavioral differences could be in the shear capacity, moment capacity, or deflection response. But, the important point to note is that the semantic constructs of the grammar do support a preconceived notion of an object's kinematic behavior. Hence, we have shown how our language constructs support both object-based and object-oriented features.

3.5.2 Truck Data Structure

We decided to treat AASHTO [2] trucks as objects because we found their parts such as (wheels, axles, equivalent loads, and spacing) were repetitious. If we look at the definitions for an AASHTO H20 and HS20 truck respectively, we see that the number of axles and their spacing is variant. (See Figure 3.12). Table 3.3 gives a finite element datafile for a typical AASHTO HS20 truck. This datafile was developed to take advantage of the similar sub-object properties between

the two trucks. Consequently, it was found that we could treat the axle spacing,

```
truck ("HS20") {
        truck_body {

    .... code deleted ....

        }
        axle {
           name          "front_axle";
           wheel_load     8 kips;
           wheel {        name "front_set";
                          center (0,1,24.5) ft;
                          radius 1.0;
           }
           wheel {        name "front_set";
                          center (10,1,24.5) ft;
                          radius 1.0;
           }
        }

    .... code deleted ....

        axle {
           name          "cab_axle2";
           wheel_load     32 kips;
           wheel {        name "front_set";
                          center (0,1,17.5) ft;
                          radius 1.0;
           }
           wheel {        name "front_set";
                          center (10,1,17.5) ft;
                          radius 1.0;
           }
        }
```

Table 3.3: Sample Datafile for AASHTO HS20 Truck

axle loads, number of axles, and number of wheels as component modules of the overall truck system. An AASHTO truck itself could also be a structural component of an even larger system. If we look at Figure 3.13, we see that we have a typical cross-section of a highway bridge loaded with four AASHTO HS20 trucks. Table 3.4 gives the language constructs for the truck objects. These rules are also broken down into a hierarchical level. The reader should note some of the keywords in these rules and how they correspond with some of

those listed in Table 3.4. It was mentioned earlier, that the basic components of an AASHTO truck are the same. The primary difference between an AASHTO H20 truck, and an AASHTO HS20 truck are the wheel spacing, the number of axles, and the loads for each axle. However, these are only scalar values. Hence, the lower hierarchical rules should be exactly the same. Table 3.3 gives a representative example of a data file for an AASHTO HS20 truck. We can see from Table 3.3 how the number of axles, and the number of wheels, can be put into a loop structure. This makes the lower hierarchical rules more flexible in their definition. Moreover, this makes it possible to develop very flexible and elementary keyword tokens associated with engineer's dictionary.

```
truck_command : TRUCK '(' STRING ')' '{' truck_list '}'   { ;}
              ;

truck_list :                                               { ;}
           | truck_list assign_numerical ';'               { ;}
           | truck_list NAME STRING ';'                    { ;}
           | truck_list TRUCK_BODY '{' body_list  '}'      { ;}
           | truck_list WHEEL      '{' wheel_list '}'      { ;}
           | truck_list AXLE       '{' axle_list  '}'      { ;}
           ;

wheel_list :                                               { ;}
           | wheel_list assign_numerical ';'               { ;}
           | wheel_list NAME STRING ';'                    { ;}
           | wheel_list CENTER coord_item LENGTH ';'       { ;}
           | wheel_list RADIUS expr ';'                    { ;}
           ;

axle_list :                                                { ;}
          | axle_list assign_numerical ';'                 { ;}
          | axle_list NAME STRING ';'                      { ;}
          ;
```

Table 3.4: Grammar Rules for Truck Objects

```
static struct {
      char *name;
      short type;
      } noun[] = {
         "axle",              AXLE,
         "name",              NAME,
         "type",              TYPE,
         "lanes",             LANES,
         "truck",             TRUCK,
         "wheel",             WHEEL,
         "plate",             PLATE,
         "center",            CENTER,
         "radius",            RADIUS,
         "plan_cabface",      PLAN_CABFACE,
         "plan_bodyface",     PLAN_BODYFACE,
         "cross_face",        CROSS_FACE,
         "bridge_deck",       BRIDGE_DECK,
         "elev_face",         ELEV_FACE,
         "distance",          DISTANCE,
         "direction",         DIRECTION,
         "magnitude",         MAGNITUDE,
         "section",           SECTION_TOKEN,
         "material",          MATERIAL,
         "thickness",         THICKNESS,
         "truck_body",        TRUCK_BODY,
         "wheel_load",        WHEEL_LOAD,
         "Node",              NODE,
         "Nodes",             NODES,
         "Elmts",             ELMTS,
         "Plates",            REC_PLATES,
         "bf",                BF,
         "tf",                TF,
         "web",               WEB,
      };
```

Table 3.5: Dictionary of Tokens for Language Constructs

3.5.3 Data Structure for IPC Message Passing

The functional purpose of the message data structure is to set up the parameters for the message and view enumeration types, indicate the number of structures generated in the request message list, and process any message strings. In turn, the task data structure contains seven data fields to represent the seven enumeration modules that store the necessary information about the generated request message list. It should be noted, our intention was to create a top-

down/decomposable hierarchy. The advantage of a top-down/decomposable hierarchy is that it allows the user to assign well defined tasks to specific modules. In contrast, the disadvantage is that it does not allow bottom-up/composability, nor does it allow certain modules to be reusable. Figure 3.14 shows a model of the IPC data structures hierarchy. In Section 3.5, we will discuss our philosophy of incorporating some programming principles of `object-oriented programming`. We attempted this practice by developing algorithms to generate objects both graphically and with keyboard commands using YACC. Although the absolute language constructs for the object-oriented programming phase have not been fully developed, we hope to convince the reader that by developing a command language, in conjunction with the Interprocess Communication, we will set the framework for its use.

In order to retrieve the analysis data from the FEM side efficiently, we had to devise a message passing system where the engineer could formulate expressions for reactive forces (shear, moment), kinematic degrees of freedom (translation, rotation), queries, and command instructions. If we look in Table 3.6, we see some of our data structures for the Interprocess Communication system. Inside each module, we see that there is a data field for each kinematic component, a data structure for each element type, as well as a data structure for each command type. These data structures contain the information that the FEM program will read and use to retrieve the requested information. To date, the engineer may request element types such as beam-columns and plates, or even objects that the engineer stored inside the hash table. Each element or object can be broken down into it's nodal form. A node is considered to have the lowest hierarchy in any structural object because all elements or sub-objects can be formed from a numeric nodal sequence. Hence when an expression is formed,

the program breaks down element types, or object types, into their respective nodal format. This information is passed across the network along with the appropriate element type as seen in the data structure titled `ELEMENT_TYPE` shown in Table 3.6. The following is an expression for the retrieval of displacements for beam-column elements.

```
XBUILD_send_message >> draw translation in neg_y direction
                         for Beam_Col { 1,2,3 }
```

The program will first determine the nodal format of the term Beam_Col { 1,2,3 } which may form the connectivity pattern for an object like that shown in Figure 3.15. This element type will be stored in the data field `IPC_ELEMENT` shown in Table 3.6. This information as well as the keyword `translation` will be passed across the network. A more detailed description is shown in Figure 3.16.

Our design of the message data structure(s) is strongly coupled to the design of the YACC grammar, and vice-versa. Details of our message passing protocol are illustrated in the schematic of Figure 3.16, and in the data structures of Table 3.6. Each message is a train of message components that always begins with a `Task_Module`. The `Task_Module` is defined by the data structure `TASK`, as shown in the middle box of Table 3.6. `TASK` contains information on the type of message packet to follow, and on the details of the list of finite element (nodes/elements) for which information is requested (GUI-to-FEM), or is being delivered (FEM-to-GUI). The data field `commandtype` reflects the intended purpose of a message – common purposes are: (a) to post a finite element data file from the GUI-to-FEM, (b) to report on intermediate finite element calculations, (c) to send/retrieve data, and (d) and close down the XBUILD system. The enumeration type named `DATA_TYPE` (see the upper box of Table 3.6) contains information on the overall type of information contained in the trailing packets

– typically, these will be one of the following: moments, shear forces, displacements, stresses, and strains. Enumeration types MOMENT_COMP, STRAIN_COMP and DISPL_COMP contain information on specific components of a response quantity type, for example bending component M_{xx}. The second component of a message is contained in MESSAGE_PACKET. The message packet contains information on how the message should be processed when it reaches it's destination. Common directives include printing a message to the screen, placing the message in a file, or drawing the response data on the screen of the GUI. Referring back to our example command, the keyword **draw** is stored as the enumeration data field **vtype** and **translation** is stored in **displcomp**. Part three of a message is an array of data structures of type LIST and of length **nostructs**, as stored in MESSAGE_PACKET. Again, details of LIST are shown in Table 3.6.

All of the data fields inside these modules are task specific. This concept follows the philosophy of decomposability discussed by Meyer [17]. Similarly, by inspection the data fields are well defined and easily understood. Moreover, the ease to add to, or delete data fields from these data structures can easily be accomplished without changing the overall function of the data structure.

An example of one of these messages is given below.

```
XBUILD_send_message >> draw displacements in neg_y direction
                                for all Plate4
```

From this expression, we see that the syntactical units are the terms **displacements**, **Plate4**, and **neg_y direction**. Referring back to Table 3.6, the reader can see that the data fields storing this information would be **trans_y** inside DISPL_TYPE and IPC_PLATE inside ELEMENT_TYPE.

3.6 IPC Modules and Their Relationship with the Command Language

This section describes the development of the command line messages for the pre-processor, the post-processor,and the Interprocess Communication stages of the program. It has already been cited in Chapter 1 that Meyer refers to the belief that the modules must correspond to the syntactic units in the language used and should be created and used in accordance with the principles of decomposability and composability [17]. We also mentioned that our language was composed of syntactic units. These units consist of words used to construct our engineering expressions. Hence, these words form a dictionary of engineering meanings. Throughout the construction of our dictionary, it was decided to use words that were familiar to the bridge engineer for specific tasks or procedures. A highway bridge engineer would want to place words in their dictionary that they were accustomed to using on a daily basis. Some of these words might include (girders, parapets, lane loading, dead load ...). Shown in Table 3.5 was an example of some the vocabulary words listed inside our dictionary. These words are understood by the engineer for their particular study of highway bridges. They correspond to either a truck object or a bridge object. All of these words are loaded into the hash table during the initialization phase of the program.

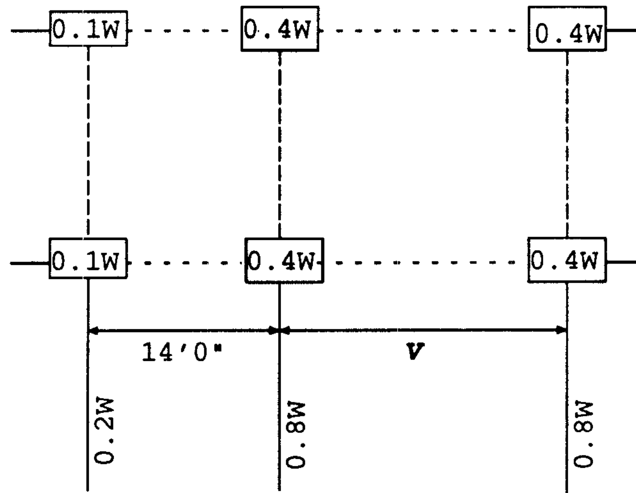
To begin to discuss the relationship of the IPC Modules and their relationship with the command language, we give an example of a rule constructed using lexical tokens with YACC. The rule is followed by a representative sample of it's expression counterpart. This rule and expression both have the same intention (i.e.) retrieve the kinematic response after the analysis phase has been completed.

```
DRAW kin_request direction for ALL Plate4
XBUILD_send_message >> draw translation in neg_direction
                       for all Plate4
```

To begin the process of building the message list on the GUI end, the element tokens are resolved into their nodal format. For this particular case, the nodal format would be comprised of the nodes which form the plate geometry. The syntactic units must be understood by the user. Hence, we have created a list of keywords suitable for this task. Our particular list is given below:

(translation, rotation, shear, moment, strainx, strainy, strainz)

We denote the correspondence of the kinematic request list with some of their respective enumeration data structures shown in Table 3.3. In particular, it can be seen how the expression above satisfies the end-user's needs for a request of a sub-structure or complete structure. Earlier, we referenced that objects can possess both behaviors and properties. Tomiyama writes that message passing allows objects to capture the semantics of a real world written in the program as long as the message passing is a good approximation of what is happening in the real world [26].



W = Combined weight on first two axles which is the same as for the corresponding H truck.

V = Variable Spacing - 14 ft to 30 feet inclusive

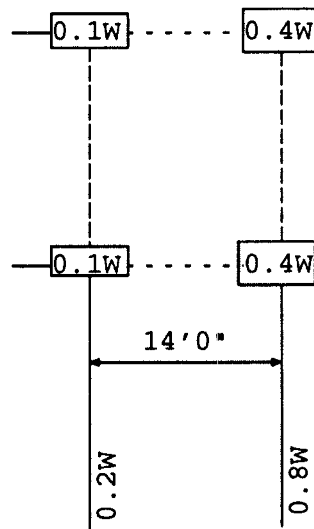


Figure 3.12: AASHTO HS20, H20 Truck Definitions.

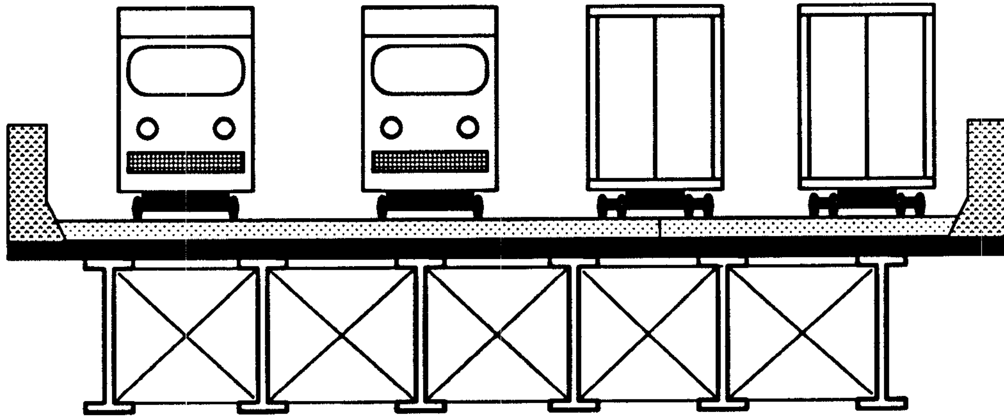


Figure 3.13: Cross-Section of Highway Bridge Structure.

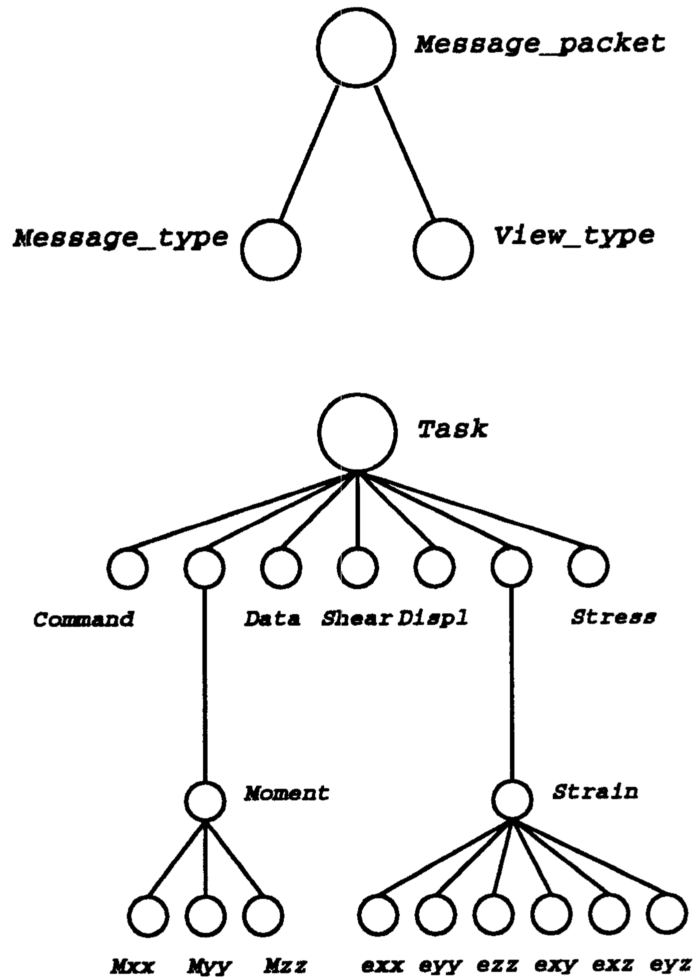


Figure 3.14: IPC Data Structure Hierarchy.

```
typedef enum {
    OPEN_FILE_MAN    =  1,
    CLOSE_FILE_MAN   =  2,
    CLOSE_CONNECTION =  3,
    WRITE_FILE_MAN   =  4,
    RETRIEVE_DATA    =  5
} COMMAND_TYPE;

typedef enum {
    MESSAGE        = 1,
    DISPLACEMENT   = 2,
    ANGULAR_DISPL  = 3,
    SHEAR_FORCE    = 4,
    MOMENT_COUPLE  = 5,
    STRESS_RESULT  = 6,
    STRAIN_RESULT  = 7
} DATA_TYPE;

typedef enum {
    MXX      = 1,
    MYY      = 2,
    MZZ      = 3,
    ALL_MOM  = 4
} MOMENT_COMP;

typedef enum {
    STRAINXX   = 1,
    STRAINYY   = 2,
    STRAINZZ   = 3,
    STRAINXZ   = 4,
    STRAINXY   = 5,
    STRAINYZ   = 6,
    ALL_STRAIN = 7
} STRAIN_COMP;

typedef struct task {
    COMMAND_TYPE  commandtype;
    DATA_TYPE     datatype;
    DISPL_COMP    displcomp;
    STRAIN_COMP   straincomp;
    STRESS_COMP   stresscomp;
    SHEAR_COMP    shearcomp;
    MOMENT_COMP   momentcomp;
} TASK, *TASK_PTR;

typedef struct message_packet {
    char          source[16];
    MESSAGE_TYPE  type;
    VIEW_TYPE     vtype;
    int           nostructs;
    char          message[80];
} MESSAGE_PACKET, *MESSAGE_PACKET_PTR;

typedef enum {
    IPC_NODE    =  1,
    IPC_ELEMENT =  2,
    DKQ_PLATE   =  3,
    IPC_GIRDER  =  4
} ELEMENT_TYPE;

typedef struct {
    char          direction[6];
    int           list_no;
    DATA_TYPE     datatype;
    ELEMENT_TYPE  etype;
} LIST, *LIST_PTR;
```

Table 3.6: Representative Sample Data Structures For IPC System

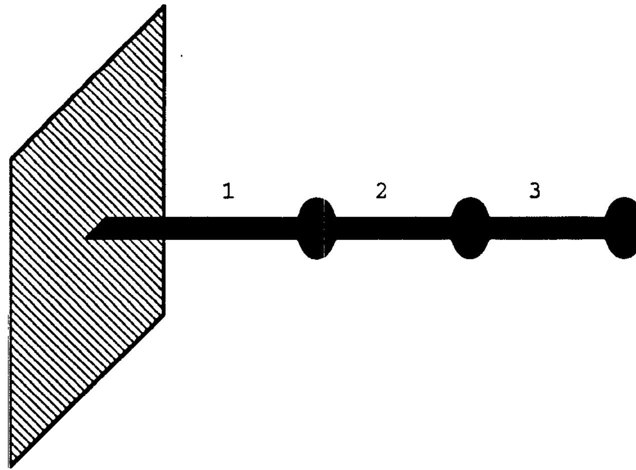


Figure 3.15: Object made of Three Beam-Col Elements.

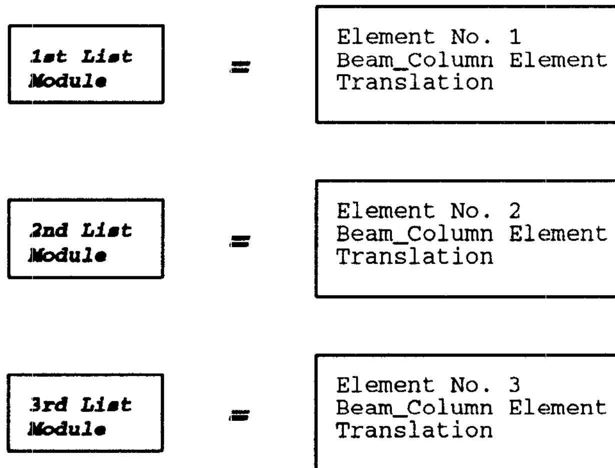


Figure 3.16: Example of Message Passing System in Data Structure Form.

3.7 Logic Behind the Construction of the IPC Modules

Throughout this section, we will mention the logic behind the development of the data structures shown in Table 3.6, as well as the enumeration types for the possible kinematic requests seen inside the task data structure. Their modular, flexible and decomposable format will also be discussed.

As one can see from Table 3.6, the type of analysis data that can be transferred across the network is quite unlimited in scope. From Table 3.7, a list is given of three of the enumeration type data structures constructed for the purpose of message passing. These data structures represent the moment types, the stress types, and displacement types. Inserting or deleting new data is very

```
typedef enum {
        MXX         = 1,
        MYY         = 2,
        MZZ         = 3,
        ALL_MOM     = 4
} MOMENT_TYPE;

typedef enum {
        STRESSXX        = 1,
        STRESSYY        = 2,
        STRESSZZ        = 3,
        STRESSXZ        = 4,
        STRESSYZ        = 5,
        STRESSXY        = 6,
        ALL_STRESS      = 7
} STRESS_TYPE;

typedef enum {
        TRANSX          = 1,
        TRANSY          = 2,
        TRANSZ          = 3,
        ROTATX          = 4,
        ROTATY          = 5,
        ROTATZ          = 6,
        ALL_TRANS       = 7,
        ALL_ROTATE      = 8
} DISPL_TYPE;
```

Table 3.7: The Enumeration Type of Data Structure Modules

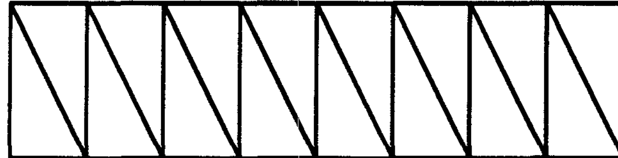
simple because the engineer can create or remove the specific data field from the

enumeration type data structure. For example, if the analysis system was a 2-D `plane stress - plane strain` problem, then only four data fields inside the stress enumeration type and strain enumeration type data structures, shown in Table 3.3, would be needed. The deletion of data fields, however, is not necessary because of the following reason. Meyer states that in order to maintain program modularity, a program must satisfy the modular decomposition technique where the modules are both open and closed [17]. He defines a module as open if it is still available for extension. In other words, the user should have the availability of adding or deleting fields inside the data structure without changing the scope of it's intended purpose. This process should not affect the function of the data structure itself, nor should it affect the overall function of the program. A module is closed if it is available for use by other modules. This means the specific field may be compiled and stored in a library for others to use. Thus, by looking back at Table 3.6 and Table 3.7, we can see that both the message and task data structures, and their respective enumeration data structures, are satisfying an open and closed format.

We see that by allowing certain response information to be added or deleted creates a modular programming environment. Thus, one can see how the format we used for our data structure design has been modeled around the objectivity of creating a very flexible, structured modular user interface. By setting up the software modules in an open and closed format, and allowing certain modules to fall into a top-down/decomposable hierarchical format, we have established the principle of implementing an object-oriented programming style for future work.

3.8 Working at the Object Level

It can be shown how an ordinary object can be broken down into its respective components for the entire object. Shown below, in Figure 3.17, is a specific example for such an object. This object could be a steel roofing truss for a warehouse structure. One can see that by creating this object graphically, or



Steel Truss Roofing Structure

Figure 3.17: Steel Roofing Structure Object.

with keyboard commands, the user can give the object a name and store it inside the hash table. In turn, using our language, the user can ask questions about the object concerning the nodal displacements, element shear, element moments, or any other pertinent information. These queries can be accomplished easily because associated with the object's name are its lower hierarchical components. These components might include nodal numbering, element numbering, nodal forces, element forces, boundary conditions, material properties etc., etc..

The fact that the engineer is now interfacing on an object level means that the engineer has a conceptual idea of the engineering system being studied or designed. At the object level, the user can ask questions about analysis process.

Chapter 4

ANALYSIS OF THREE ENGINEERING SYSTEMS

This chapter describes the step-by-step procedure for creating finite element models of simple structural systems. Tasks include nodal connectivity, element connectivity, loading conditions, boundary conditions, initial conditions, and the view transformation setup. Three complete problem descriptions are given. They are as follows: (a) `One Storey Frame Structure`, (b) `Cantilever Plate`, (c) `Multi-Span Truss Bridge Structure`. Where appropriate, screen-dumps will be given to provide a visual aid in the system formulation.

4.1 3-D Frame Structure

The `One Storey Frame` consists of four columns, four beams, and a plate element for the slab. There are eight nodal loads applied to the structure. Four loads act in the negative-y direction and four loads act in the positive-x direction. The boundary conditions are given in Table 4.1. Figure 4.1 gives a description of the geometry, the nodal loads and the boundary conditions for the structure.

To begin the graphical geometric setup of this structural system, the user

```
Node {2}        DOF [dx,dz,ry];
Node {3}        DOF [dx,dz,ry];
Node {7}        DOF [dx,dz,ry];
Node {6}        DOF [dx,dz,ry];
Node {1}        DOF [all];
Node {4}        DOF [all];
Node {5}        DOF [all];
Node {8}        DOF [all];
```

Table 4.1: Boundary Conditions for 3-D Frame Example

first needs to set default types for engineering units, material types, and section types. These default types are entered using the keyboard with the prompt inside the TTY window. A typical example for this system is shown below.

XBUILD ≫ `set length units to inches`

XBUILD ≫ `set material type to STEEL4`

XBUILD ≫ `set section type to W36x245`

XBUILD ≫ `set plate thickness to 2 inches`

Now the user can begin to enter the nodal coordinates. The coordinates may be entered either by a single expression or one command at a time. In Chapter 2, we mentioned how the engineer can change the prompt mode by typing the appropriate expression. The purpose of creating this prompt mode expression is that it frees the engineer from extraneous typing. For adding nodes, the prompt command is as follows:

XBUILD ≫ `add node`

XBUILD _ add_node ≫ `x {-10 to 10 by 5} inches y {-10 to 10 by 5} inches z {-10 to 10 by 5} inches`

XBUILD _ add_node ≫ `x {-500.0} inches y {0.0 } inches`

Figure 4.1: One Storey Frame - Example No. 1.

`z {-500.0}inches`

Once this stage of the problem setup is complete, the user can then move on to the mode for adding the beam-column elements. Recall from Chapter 2, that the user has the option of adding the beam-column elements either graphically, or by keyboard command. Since we already gave examples on how to add beam-column elements graphically, we will merely discuss their addition to the present system via keyboard command.

To begin, the user needs to place the prompt into the proper mode. This time the prompt should read for adding beam-columns. We give a typical example for adding beam-columns below this command. Note how the command depends on two initial conditions.

XBUILD `add beam_col`

XBUILD_add_beam_col ≫ `[STEEL4, W16x89] (1,2), (2,3), (3,4)`

Another element stage is the procedure for adding plates. This stage requires the specification of the plate thickness, material type and the nodal connectivity pattern. The next two commands place the prompt into the plate4 mode and add a plate to the engineering system respectively.

XBUILD ≫ **add plate4**

XBUILD _add_plate4 ≫ **Nodes {2,3,6,7} with 8 inches thickness and STEEL1 material**

Now that the elements have been added, the engineer may add nodal loads. At this time, the finite element program is only capable of interpreting command line expressions for nodal loads. We distinguish nodal loads from ordinary concentrated loads by the yellow color. Command sequences are now given for the prompt mode command and application command for adding a nodal load. It should be noted that nodal loads can be either forces or moments.

XBUILD ≫ **add nodalload**

XBUILD_add_nodalload ≫ **-4 kips in neg_y direction at Nodes {3}**

XBUILD_add_nodalload ≫ **+4 kips in pos_x direction at Nodes {2}**

After finishing the pre-processing commands for this structure, the user will get the following structural engineeering system displayed in Figure 4.2.

To begin the post-processing stage of XBUILD, the engineer needs to create an appropriate data file that the finite element program can read. Since the GUI and FEM nodes are uncoupled, a file must be created on the GUI end which can be passed through the socket to the FEM side. The commands to write and to post this file across the network are now shown.

XBUILD ≫ **write file "< file_name >"**

XBUILD ≫ **post file "< file_name >"**

If the posting of the file across the network is successful, the engineer will be given the default prompt displayed inside the TTY window, otherwise the engineer will be given an error message. This message will indicate that the number of bytes sent across the network, from the GUI end, does not match the number of bytes retrieved from the FEM end. This message is a critical message because it indicates that the program has lost information passed through the socket. Consequently, the program will cease operation at this point of execution. The following two commands represent a message for a successful transfer and an unsuccessful transfer respectively.

XBUILD ≫ `In Remote Simulator...CLOSE_FILE`

XBUILD ≫ `bytes lost in TASK transfer` !!

However, let's say that the user did pass the information successfully through the socket. Now the user is ready to begin the post-processing phase of the program. We set up the syntax of the grammar language such that the user could construct messages around the kinematic responses they desired. For instance, in the example of the `One Storey Frame` the user can ask relevant questions about the translations, rotations, shear reactions and moment reactions. Shown below are typical relevant queries a user may have about such a structure.

XBUILD _send_message ≫ `draw All translation in neg_y direction for Beam_Col{1,2,3}`

XBUILD _send_message ≫ `print All moment for Beam_Col{1,2,3}`

XBUILD _send_message ≫ `view All translation in neg_y direction for Beam_Col{1,2,3}`

Figure 4.3 shows the rotation plot of the One Storey Frame. The program gathers all of the rotation analysis data from the data structures using the

element connectivity pattern generated from the GUI end of the program. This information is then passed across the network to the respective data structures on the GUI end and then plotted on the screen. Where necessary, an appropriate magnification factor has been multiplied to the data. In conclusion, the reader can see that the questions one may ask are directly related to specific response information about the structure. Some of the information about a particular structure may not be necessary. However, by now the reader can get an idea as to why we broke up the IPC modules into a task-oriented hierarchy.

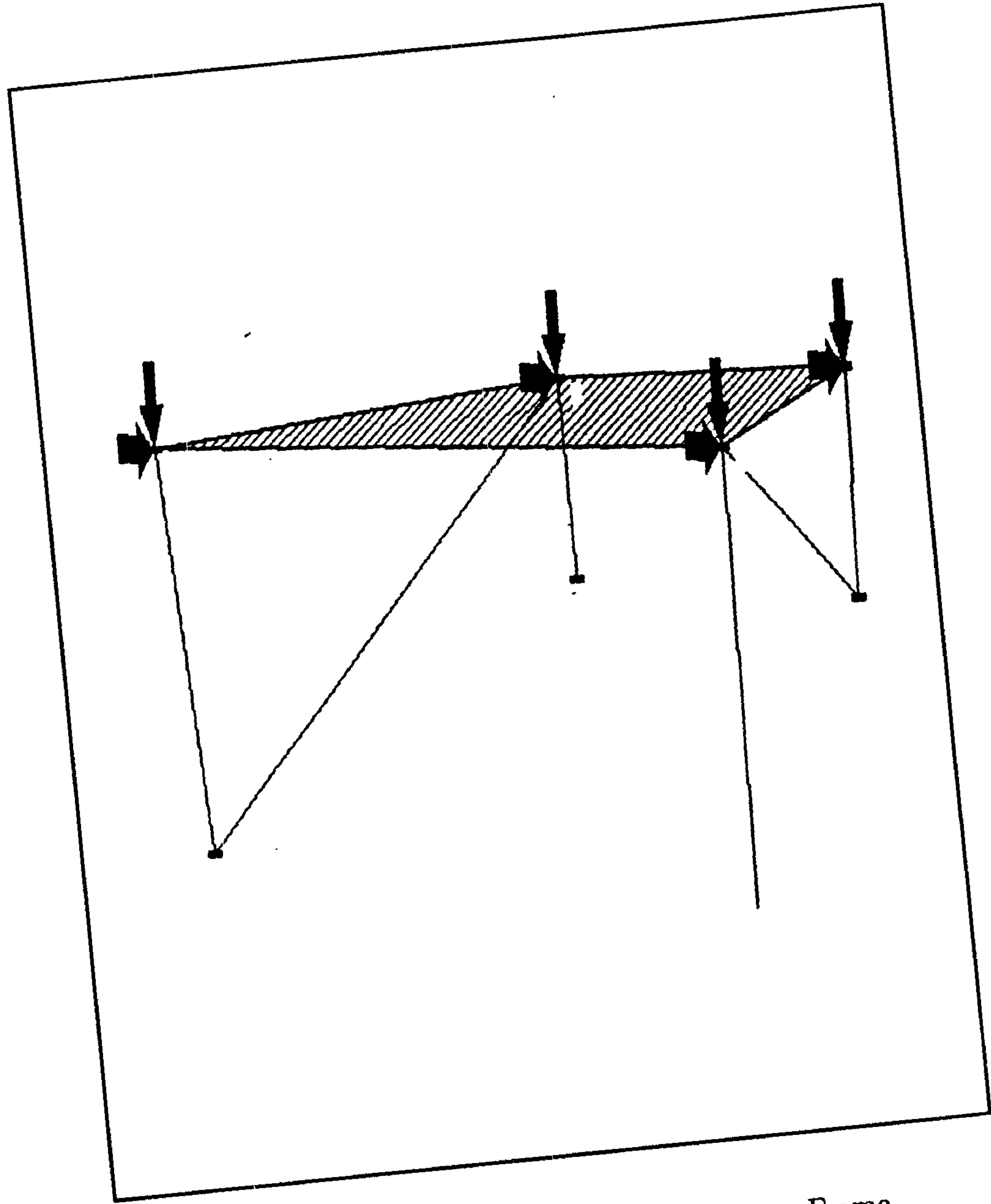


Figure 4.2: Graphical Display of One Storey Frame.

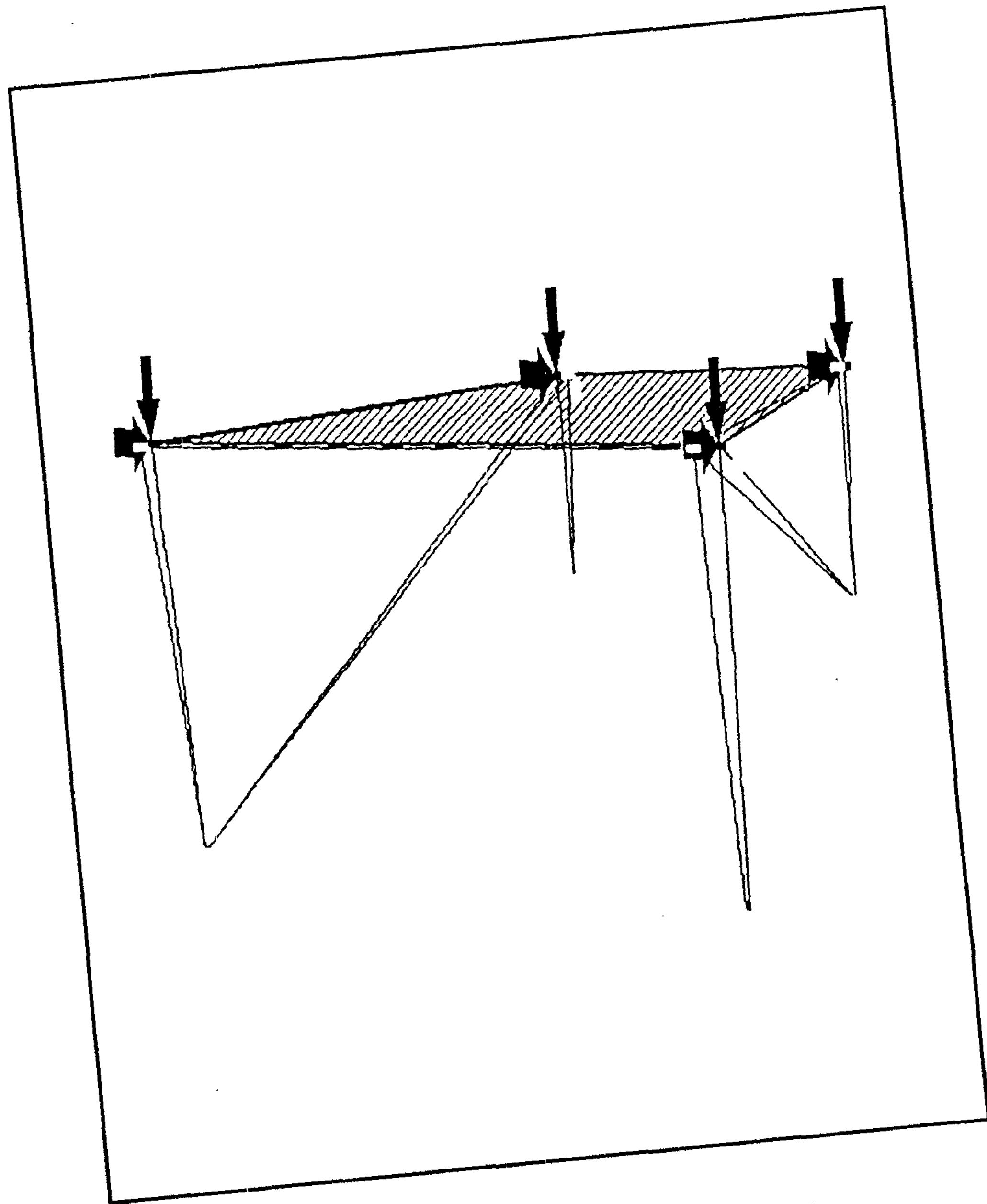


Figure 4.3: Rotation Display of One Storey Frame.

4.2 Cantilever Plate

The second structural system we will explain is a cantilever plate. The plate is eight inches thick with four nodal loads applied to the structure. The x and z dimensions of the plate are 160.0 inches x 480.0 inches respectively. The analysis algorithm for the plate assumes that the structure is modeled using the Discrete Kirchoff Quadralateral model. Unfortunately, at this time membrane action has not been implemented into the analysis algorithm.

The commands to set up the plate are relatively the same as in Example 1. Thus, we have decided to include segments of the code for the data file. We will then proceed with the post-processing phase and show a different set of commands that one may ask for this type of structure.

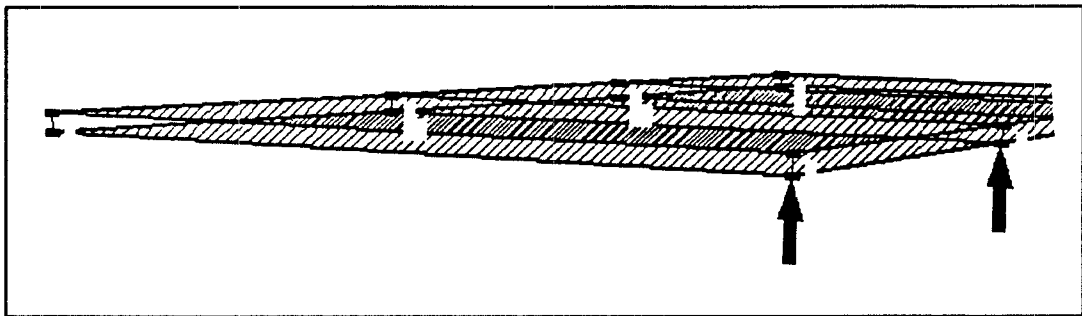


Figure 4.4: Plate-Cantilever Example.

After the data file, shown in Table 4.2, has been read, the engineer should get the following display in the 3-D graphics window. For a plate element, the finite element analysis package computes the strains, stresses, displacements, and reaction forces. In this example, we are going to show the translation of the plate in the `y-direction` caused by the four nodal forces. We can see from Figure 4.5, that the plate structure essentially behaves the same as a cantilever beam. Although it would not be practical to plot the strain results on the screen,

```
/* Setting the prompt mode to add node */

   add node;

/* Creating the node coordinates */

   x {-480.0}  inches  y { 104.0}   inches  z { 480.0}  inches;
   x {-320.0}  inches  y { 104.0}   inches  z { 480.0}  inches;

   x {-480.0}  inches  y { 104.0}   inches  z { 320.0}  inches;
   x {-320.0}  inches  y { 104.0}   inches  z { 320.0}  inches;

..... code deleted .....

/* Setting the prompt mode to add bc_elmt */

   add bc_elmt;

/* Adding the element sections to the datafile */

   [STEEL4, W36x160] (1,2),     (2,4),     (4,6),     (8,6),     (7,8);
   [STEEL4, W36x160] (1,3),     (3,5),     (5,7),     (3,4),     (5,6);

..... code deleted .....

/* Setting the prompt mode to add plate4 */

   add plate4;

/* Adding the plate sections to the datafile */

   Nodes {1,2,3,4}     with 8 inches thickness and STEEL4 material;
   Nodes {3,4,5,6}     with 8 inches thickness and STEEL4 material;

..... code deleted .....

/* Setting the prompt mode to fix bcond */

   fix bcond;

/* Fixing the proper nodal boundary conditions */

   Node {2,4,6,8}          DOF [dz,ry,rx,rz];
   Node {10,12,14,16}      DOF [dz,ry,rx,rz];

..... code deleted .....

/* Setting the prompt mode to add nodalload */

   add nodalload;

/* Adding the appropriate nodal loads to the structure */

  12 kips in pos_y direction at Nodes {12};
```

Table 4.2: Sample Datafile for Plate Example

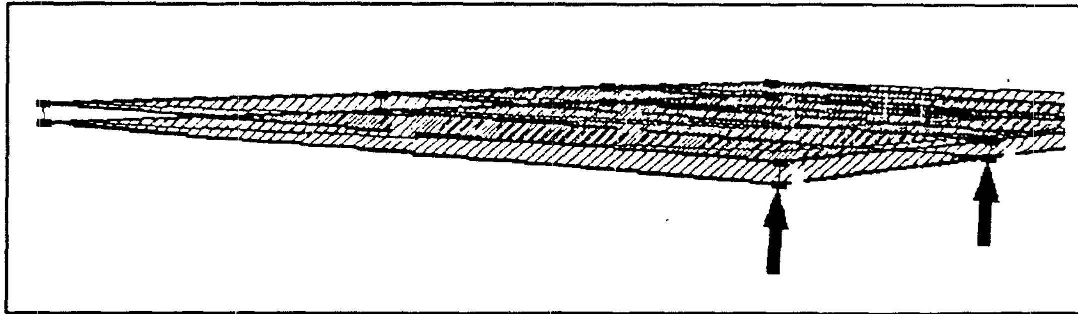


Figure 4.5: Deflected Shape for Plate-Cantilever Example.

the user can print them to the TTY window. One such command to do this would be the one given below.

XBUILD _send_message ≫ `print All strain for All Plate4`

The strain data would then be gathered from the data structures on the FEM side of the architecture, passed through the socket, and appropriately displayed inside the TTY window.

4.3 Multi-Span Truss-Bridge Structure

The final structure we will look at is a multi-span truss bridge. This bridge is comprised of beam-column elements and several plate elements. Nodal loads have been added randomly to the structure and do not represent any particular loading pattern. In this example, the reader can see how we have thickened the beam-column elements to highlight the analysis plot. Again, as in Example 1, the commands to set up the bridge geometry are essentially the same. Shown in Figure 4.6, is a sample display of the bridge geometry and its loading pattern. As with the first two examples, this object is comprised of both beam_col elements and Plate4 elements. The engineer may request specific information concerning these element types. For such a large structure though, it might be more beneficial to write all of the analysis data to a file. The engineer can easily add comment statements to separate the data. This creates a more tidy output format for several requests of response information.

Table 4.3 shows the 'C' code for the YACC parser algorithm for breaking up an expression into lexical tokens which form a comment statement. This code shows an added benefit of the XBUILD program. When the user is analyzing large objects, it might be beneficial to retrieve the information on paper. By having the ability to break up the data into segments, the engineer is given a neat spreadsheet format. Figure 4.7 shows a plot of the bridge where we requested all of the translations for the beam_col elements have been requested.

```
if ((c == '/') && (follow('*', TRUE, FALSE) == TRUE)) {
  i=0;
  start++;
  c = get_next_char(start);
  comment_buffer[i] = c;
  finished = FALSE;
  while (finished == FALSE) {
       c = get_next_char(start);
       if (c != '*') {
         i++;
         comment_buffer[i] = c;
       }

       if ((c == '*') && (follow('/', TRUE, FALSE) == TRUE)) {
         c = get_next_char(start);
         finished = TRUE;
         comment_buffer[i] = ' ';
         i++;
       }
       start++;
  }
  comment_buffer[i] = '\0';
  lineno++;
  return (COMMENT);
```

Table 4.3: Parser Algorithm for YACC Comment Statement

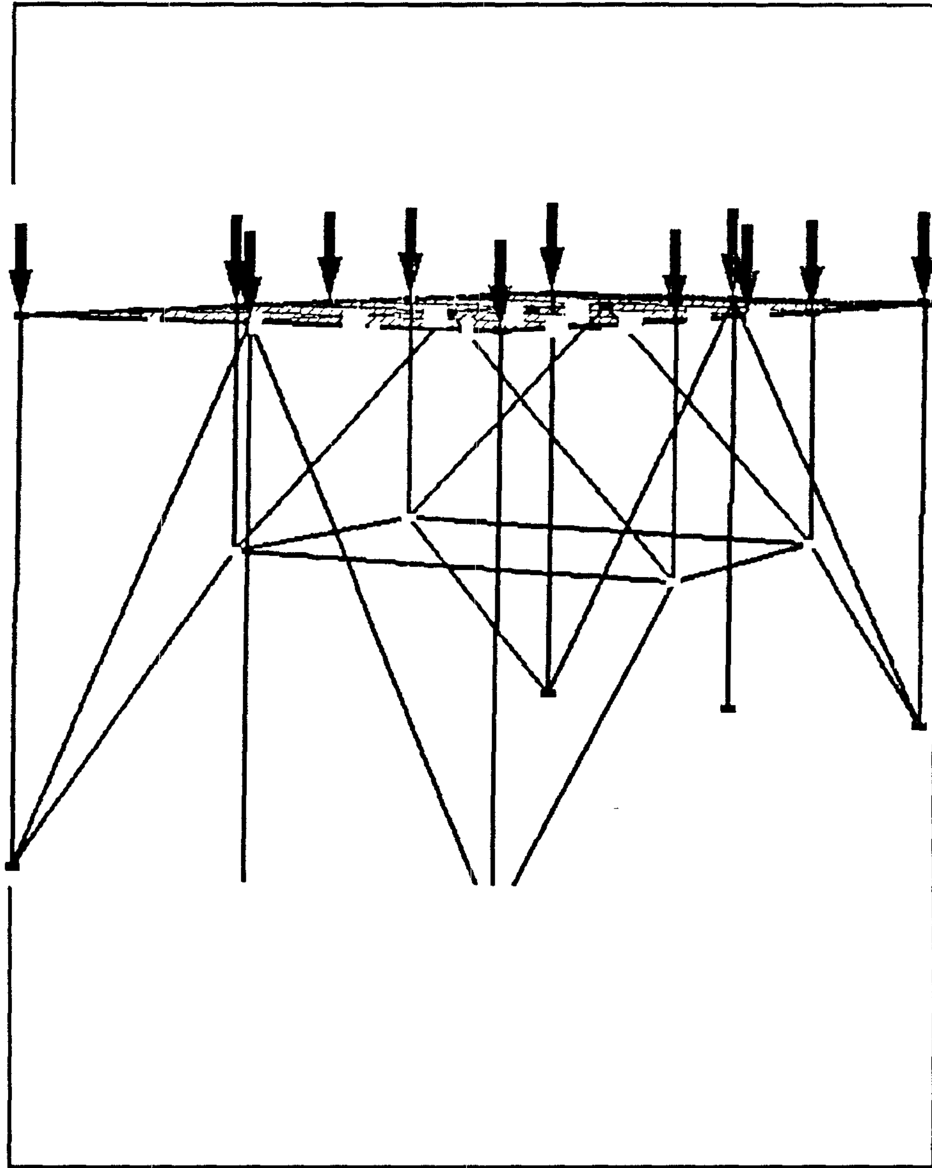


Figure 4.6: Multi-Span Truss Bridge Example.

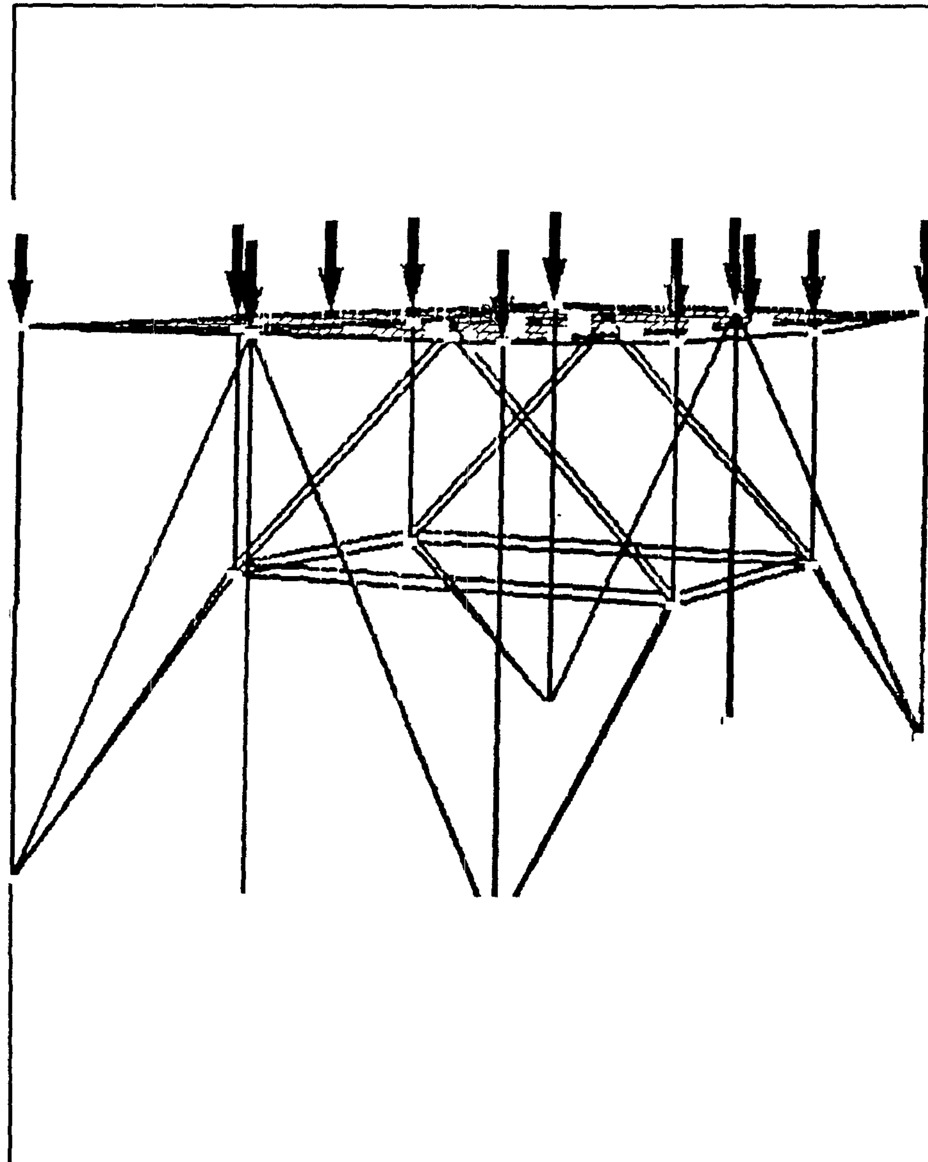


Figure 4.7: Translation of Multi-Span Truss Bridge Beam-Col.

Chapter 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

The result of this research is XBUILD Version 2.0. It is an interactive interface coupled with a message passing mechanism to aid in the design and analysis of structural engineering bridge systems.

As Kay mentioned, the user interface is the most important part of an application program because it is the only part the engineer can see [15]. Mondkar states most comprehensive (FEM) packages today are broken down into three distinct modules [18].

1. A pre-processor to provide a graphical approach to the geometric creation of the object along with the appropriate boundary conditions, load conditions and finite element mesh size.

2. An analysis stage that takes the data from the pre-processor and performs the structural analysis.

3. Finally, a post-processing stage which enables the user to observe and evaluate output from the analysis.

It was our intention to provide user friendly features to aid in the problem setup description. The problem setup (pre-processor stage) is not where the engineer wants to spend most of their time. Therefore, features which expedite this process were generated. This concept is related to the idea of developing an interactive user interface which meets the end-user's needs.

XBUILD provides the engineer with an opportunity to create highway bridge engineering systems with three primary interface strategies(keyboard, graphical, data file). The first choice allows the engineer to type each command from the keyboard. The commands were constructed using the YACC grammar and are appropriate for each stage of the pre-processing sequence to aid the engineer in setting up the problem formulation. The engineer can also set default parameters, enter the nodal coordinates, element connectivity, and load/boundary conditions. The second choice allows the engineer to add all of the finite elements with interactive graphical procedures. These graphical procedures are utilized once the coordinate nodes have been placed inside the canvas window. The mouse represents the primary hardware component which the engineer employs for this option. Using reverse mapping (screen coordinates to world coordinates), we have built an algorithm which allows the engineer to interact in all four (3D, XY plane, YZ plane, XZ plane) windows. The final choice places all the geometric commands, element connectivity sequences, and boundary conditions inside a data file. By subsequently typing an appropriate expression, the engineer can initiate the construction sequence to read each line inside the data file until the end is reached. The expressions inside the file are read using the compiler language YACC. The YACC grammar allows the user to customize command

line expressions for their specific task. A dictionary of engineering keywords comprise the syntax which form the expressions familiar to the highway bridge engineer.

The current version of XBUILD can only handle a very limited list of element types (two and three dimension elastic beams, DKQ plate elements). In the near future, the XBUILD library of finite elements will be expanded upon to geometrically exact rods, and beam-column elements containing material non-linearities. One goal of the work was to develop message-passing data structures that would lend themselves to concepts of encapsulation and inheritance. As such, future developments of XBUILD may include the use of languages that support these constructs. A natural choice would be C++.

The message passing modules follow a top-down hierarchy. This allows us to assign the data fields, inside these modules, task specific duties. The modules also follow an open and closed format. The data structures which store the response information favor a minimal maintenance problem if any future changes in the software elements are necessary. Programs which adopt this philosophy are called modular understandable. Modular understandibility is one of the five criteria Meyer lists a program should have in order to establish object-oriented capabilities. The others are enumerated below.

1. modular composability
2. modular decomposability
3. modular understandibility
4. modular continuity
5. modular protection

It is not possible to attain all of these criteria. However, all of the criteria are independent of each other. Therefore, the goal for the user is to understand each by definition and then construct a modular system with these ideas in mind. We have already discussed some of these topics in the text. The latter two features are particularly important to setup an object-oriented paradigm.

As we alluded to earlier in Section 3.4, the information is broken down into a specific request level. For instance, if the user was doing an analysis of a simply-supported beam in the x-y coordinate plane, then they might be interested in the moment in the z-direction. A component moment data field named `MZZ` is given for this operation. In turn, the user may be interested in the translational displacements in the y-direction. For this case, a data field named `TRANSY` was created.

Coupling the grammar with the data fields is necessary to establish a one-to-one correspondence. By doing so, the engineer is able to interact at a much higher syntactical level. This brings the ideals of the program closer to the principles of knowledge engineering. Knowledge engineering is considered a future direction all CAD programs must follow [26].

From the information in the preceding chapters, it is clear that the research field of interactive computer graphics is still growing and expanding. The next generation of CAD systems is likely to adopt a philosophy of artificial intelligence, geometric modeling, and feedback for design processes. Although the computer will never replace the thought and logic processes of man, they will certainly take on more added responsibility to alleviate the engineer of remedial or repetitive tasks.

Thus, we have demonstrated how the Interprocess Communication system allows the engineer to devise interactive procedures that can aid in the engineer's

knowledge of the structural system being studied. Any smaller system also has an opportunity to be encapsulated and inherited as well. Similarly any property, or material, information can be retrieved either before the analysis phase or after. By asking for information after the analysis phase, the user can interact with both the pre-processing and post-processing phases of the computer program.

5.2 Future Work

Our hardware and software components have the capability of running distributed or parallel finite element analyses. It would be easy to set up a pull-down menu item or panel button to begin an algorithm for an engineering system to be run in parallel. To be specific, one such expression that is concerned with the study of bridge decks is a typical example. An expression for simultaneous studies of bridge decks is given below.

```
XBUILD >> post files (deck1, deck2, deck3)
```

This typed expression instructs the computer program to retrieve the files titled *deck1*, *deck2*, and *deck3* from the current working directory. The data files would then be posted across the network through the socket based message protocol system. Subsequently, the analysis results would be transferred back through the socket to the GUI side with the appropriate display of the information.

Future work for XBUILD will also involve the implementation of a true object-oriented approach to highway bridge systems. By incorporating the object-oriented features encapsulation and inheritance, the engineer can categorize the bridge components into special classes and focus the analysis strictly on their behavior. B. Meyer lists five criteria a program should follow if it is to adopt the philosophy of object-oriented programming [17]. Since object-oriented programming (OOP) is a future direction we would like to pursue, we decided to incorporate some of his ideas. Pun and Winder indicate that object-oriented languages are becoming more and more popular [22]. The authors also mention how objects can possess both behaviors and properties of an entity. The entity can then be simulated by the sending and receiving of messages, and in turn, encapsulated or inherited [26]. Pun and Winder mention that object-oriented

programming is a benefit to the successful development of a user interface [22]. The IPC system demonstrates a principal part of this development.

Our work will also concentrate on the development of functions to check design constraints required by the AASHTO Bridge Design Manual [2]. The design constraints include limitations on lane widths, truck locations, and placement of parapets and wheel loads. The finite element library will be expanded so that curved girder bridges may be analyzed.

Enhancements to the pre-processor and post-processor also need to be completed. Many of the graphical routines could use some added features such as shading, and hidden surface removal. In turn, an algorithm needs to be developed that would allow the user to study the strain-stress patterns of the elements being analyzed. Certain finite element libraries also need to be generated which would aid in the types of analyses that can be done. However, as mentioned in the thesis, all updates need to be completed independently. This means any upgrade of any single node of the (GUI-IPC-FEM) should not affect the other two.

Much emphasis is now being put on feature based modeling. Mondkar states that most software developers are designing pre-processors with a (WYSIWYG - what you see is what you get) attitude [18]. This means the pre-processor is supposed to ideally represent the object that is going to be analyzed. This principle is very similar to Tomiyama's idea of the `mental model` [26]. Preston defines solid modeling as the process of building and analyzing geometrically complete representations of three dimensional solid objects [21]. He goes on to say that in a functional sense, a model is said to be geometrically complete if it is possible to answer questions about the geometry of the object. Potter says that if a 3-D model is to be useful then it must provide the information it

contains [20]. The idea of building a solid model, and then viewing the analysis, enables the user to create simulations concerning it's behavior as indicated by Potter [20].

In future versions of XBUILD, the engineer may enhance some of these ideas mentioned by Mondkar [18] and Preston [21]. In terms of modeling, the engineer may request a specific planar location anywhere along the corresponding view axis. For instance, let's say we have an object that extends from -10.0 to 10.0 meters in the x-direction and that we are in the y-z plane view window. Currently, the user may only look at the -10.0, or 10.0 meter x-axis planes. However, in the future, the user will be able to view planar locations inside these boundaries. Potter [20] mentions that in future CAD software, the user will be able to look at HVAC, electrical, or plumbing as-built drawings on a computer. Thus, one can see how these arbitrary planar views would come in handy. Future versions of our program will also allow the user to make planar cuts anywhere on the object and rotate the two pieces apart from each other. One can see how this feature would be very beneficial for a person studying the behavior of reinforced or prestressed concrete beams. For instance, an emerging area of research interest in computer graphics is to be able to actually move through the object. This will enhance 3-D building models as a presentation tool [20].

Bibliography

[1] Load & Resistance Factor Design, 1st Edition. American Institute of Steel Construction, 1986.

[2] Standard Specifications for Highway Bridges, 14th Edition. American Association of State Highway Transportation Officials, 1989.

[3] Akman, V., ten Hagen, P.J.W., Tomiyama, T. "A fundamental and theoritical framework for an intelligent CAD system". *Computer-Aided Design*, pages 352–367, 1990.

[4] Austin, M.A., Preston, J.P. "Solid Modeling of RC Beams, Part 1 : Data Structures and Algorithms". *Computing in Civil Engineering*, pages 389–403, 1992. American Society of Civil Engineers, New York, NY.

[5] Austin, M.A., Preston, J.P. "Solid Modeling of RC Beams, Part 2 : Computational Environment". *Computing in Civil Engineering*, pages 404–416, 1992. American Society of Civil Engineers, New York, NY.

[6] Byrne, R.H. "Interactive Graphics and Dynamical Simulation in a Distributed Processing Environment". University of Maryland, Institute of Systems Research. Technical Report # 90-7, 1990.

[7] Chin, R., Chanson, S. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[8] Coulouris, G.F., Dollimore, J. *Distributed Systems ; Concepts & Design.* International Computer Science Series, Addison-Wesley, 1989.

[9] Coutaz, J. "Architecture Models for Interactive Software". *Proceedings of the Third European Conference on Object-Oriented Programming*, pages 383–398, July 1989. East Midlands Conference Centre, University of Nottingham.

[10] Elnahal, M., Albrecht, P., Cayes, L.R. "Load Distribution in a Two-Span Continuous Bridge". Report FHWA-RD-89-101, Federal Highway Administration, McLean, VA "Turner-Fairbanks Highway Research Center", 1989.

[11] Finger, R., Dixon, S. A review of research in mechanical engineering design Part I: Descriptive, Prescriptive and computer based models of design processes. *Research Engineering Design*, pages 51–67, May 1989.

[12] Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F. *Computer Graphics Principle and Practice, Second Edition.* Addison-Wesley, Reading Massachusetts, 1990.

[13] Hudson, C.A., Austin, M.A. "Finite Element Post - Processor for Analysis of Highway Bridges". *Proceedings of the 8th Computing in Civil Engineering Conference, ASCE*, 1991. American Society of Civil Engineers, New York, New York.

[14] Johnson, S. C. YACC - Yet Another Compiler Compiler. Technical report, AT& T Bell Laboratories, Murray Hill, New Jersey, 1975.

[15] Kay, A. User Interface Design Methodologies. *Case : User Interface Design Methodologies.* McGraw-Hill, 1988.

[16] Lewi, J., Steegmans, E., De Man, J. "Object-Oriented Sofware Construction". *Proceedings of the Fifth Annual European Computer Conference,* pages 626–633, May 1991. Bologna, Italy.

[17] Meyer, B. *Object-Oriented Sofware Construction.* Prentice Hall Series in Computer Science, 1988.

[18] Mondkar, D., Uyei, J.H. "Pcs Ease the use FEA". *CAE,* pages 76–84, July 1987.

[19] Newman, W., Sproull, R. *Principles of Interactive Computer Graphics.* McGraw-Hill, New York, NY, 1979.

[20] Potter, C.D. "CAD in Construction". *CAE,* pages 34–39, November 1987.

[21] Preston, J.P. *"An Application of Solid Modeling Concepts to Design Software for Reinforced Concrete Beams".* PhD thesis, University of Maryland College Park, College Park, MD, 1991.

[22] Pun, W.Y., Winder, R.L. "A Design Method For Object-Oriented Programming". *Proceedings of the Third European Conference on Object-Oriented Programming,* pages 225–239, July 1989.

[23] Rogers, D.F., Adams, J.A. *Mathematical Elements for Computer Graphics Second Edition.* McGraw-Hill Publishing Company, New York, NY, 1990.

[24] Sondhi, J. "FERA : C-Based Finite Element Program". University of Maryland, Institute of Systems Research. Scholarly Paper, 1991.

[25] Taenzer, D., Ganti, M., Podar,S. "Problems in Object-Oriented Software Reuse". *Proceedings of the Third European Conference on Object-Oriented Programming*, pages 26–28, July 1991.

[26] Tomiyana, T. Intelligent CAD Systems. *New Trends in Computer Graphics IV*, pages 343–388, May 1991.

[27] Voon, B. K., Austin, M.A. "Structural Optimization in DNC". University of Maryland, Institute of Systems Research. Technical Report # 91-104.