Copyright

by

Joeny Quan Bui

2012

The Thesis Committee for Joeny Quan Bui
Certifies that this is the approved version of the following thesis:

Development of Software Architecture to Investigate Bridge Security

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor: ______________________________________
Eric Williamson

____
Oguzhan Bayrak

Development of Software Architecture to Investigate Bridge Security

by

Joeny Quan Bui, B.S.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2012

Dedication

To my loving parents and my dearest Shina, being the rock on which I rest.

Acknowledgements

I would first like to acknowledge the mentorship of Dr. Williamson. Through his guidance and this research, I realized my potential as an engineer. I would also like to acknowledge Eric Sammarco, a trusted colleague who provided me with insight and wisdom in both engineering and life. Finally, I would like to acknowledge the US Army Corp of Engineers for their support, and the MUSE office for all the help.

Abstract

Development of Software Architecture to Investigate Bridge Security

Joeny Quan Bui, M. S. E.

The University of Texas at Austin, 2012

Supervisor: Eric Williamson

After September 11, 2001, government officials and the engineering community have devoted significant time and resources to protect the country from such attacks again. Because highway infrastructure plays such a critical role in the public's daily life, research has been conducted to determine the resiliency of various bridge components subjected to blast loads. While more tests are needed, it is now time to transfer the research into tools to be used by the design community.

The development of Anti-Terrorism Planner for Bridges (ATP-Bridge), a program intended to be used by bridge engineers and planners to investigate blast loads against bridges, is explained in this thesis. The overall project goal was to build a program that can incorporate multiple bridge components while still maintaining a simple, user-friendly interface. This goal was achieved by balancing three core areas: constraining the graphical user interface (GUI) to similar themes across the program, allowing flexibility

in the creation of the numerical models, and designing the data structures using object-oriented programming concepts to connect the GUI with the numerical models.

An example of a solver (prestressed girder with advanced SDOF analysis model) is also presented to illustrate a fast-running algorithm. The SDOF model incorporates the development of a moment-curvature response curve created by a layer-by-layer analysis, a non-linear static analysis accounting for both geometric non-linearity as well as material non-linearity, and a Newmark-beta-based SDOF analysis. The results of the model return the dynamic response history and the amount of damage.

ATP-Bridge is the first software developed that incorporates multiple bridge components into one user-friendly engineering tool for protecting bridge structures against terrorist threats. The software is intended to serve as a synthesis of state-of-the-art knowledge, with future updates made to the program as more research becomes available. In contrast to physical testing and high-fidelity finite element simulations, ATP-Bridge uses less time-consuming, more cost effective numerical models to generate dynamic response data and damage estimates. With this tool, engineers and planners will be able to safeguard the nation's bridge inventory and, in turn, reinforce the public's trust.

Table of Contents

Acknowledgements v

Abstract vi

Table of Contents viii

List of Tables xii

List of Figures xiii

Chapter 1: Introduction 1

Thesis Outline 2

Chapter 2: Background, Motivation, and Challenges 3

Terrorist Threats Against Bridges 4

Experimental Research and Current Practice 5

Current Practice 7

Purpose of the Research Project 9

Goal of the Software 11

The Complexity of the Problem Domain 12

The Challenge of Managing the Developmental Process 14

The Flexibility Possible through Software 15

The Problems of Characterizing the Behavior of Discrete System 16

Summary 17

Chapter 3: Software Architecture and Design 19

Program Flow Chart 19

Program Paradigm 22

Modular Programming 23

Object-Oriented Programming 23

Abstraction 24

Encapsulation 25

Inheritance....26
Polymorphism....26
Event-Driven Programming....27
ATP-Bridge Paradigm....27
Summary....30

Chapter 4: Data Structures....31
Data Structure Relationships....31
Nexus Assembly....32
Load Assembly....33
Structural Component Assembly....34
Class Naming Convention and Standard....35
Applied Encapsulation....36
The Role of Inheritance....39
Polymorphism in Practice....41
Structural Component Class....43
Physical Attributes and Methods....44
Analysis....47
User Interface and Graphics Engine....49
Load Class....51
Nexus Class....55
Additional Data Types....57
Summary....59

Chapter 5: Graphical User Interface and Graphics Engine....60
Graphical User Interface Overall Design....61
Navigational Control....64
Tree-View Summary Control....64
Viewer Setting Control....67
Direct3D Graphical Environment....68
Vertices and Primitives Types....68
Vertex Buffers and Index Buffers....70

Graphics Engine Components 71
JQB Elements 72
Node Element 72
Plane Element Superclass 73
Triangle Element 75
Quadrilateral Element 77
Mesh Component 79
Graphics Object 85
Graphics Engine Cycle 87
JQB Graphics Engine Class 88
JQB Blackboard Class 94
JQB Component 97
Summary 102
Chapter 6: Prestressed Girder Model 103
Experimental Work 104
Advanced Single-Degree-of-Freedom Algorithm 108
Moment-Curvature Relationship 109
Concrete Material Model 112
Mild Steel Reinforcement Model 115
Prestressing Strand Material Model 117
Dynamic Increase Factor (DIF) 120
Layer-by-Layer Moment-Curvature Analysis 123
Bilinear Moment-Curvature Diagram 125
Resistance Function 127
Static Analysis 128
Load Spatial Distribution 129
Incremental-Iterative Method 133
Geometric Non-linearity 135
Material Non-Linearity 137
Generalized Single-Degree-of-Freedom System 137

Forcing Function 140
Solving the Equation of Motion with Newmark-beta Method 141
Example of Girder Model 142
Girder Single-Degree-of-Freedom Model 142
Comparison of SDOF Model with FEM Model 149
Summary 151

Chapter 7: Summary, Recommendations, and Conclusions 152
Summary of Research Program 152
Snapshots of the Anti-Terrorist Bridge Planner...154
Recommendations for Future Work...158
Recommendations on Prestressed Girder Solver for Future Work............160

Appendix A: Programming Glossary 164

Appendix B: 3D Mathematics 167
Vector Algebra...167
Matrix Algebra...168
Scaling..168
Rotation..169
Translation 169

Appendix C: Frame Element in ATP-Bridge 171
Elastic and Geometric Stiffness Matrix (McGuire et al, 2002) : 172
Member Load Vector:...172
Equations for Triangular Load Fixed-End Moments and Shear
(Kassimali, 1999):...173

Appendix D: Incremental-Iteration Variable Definitions.............................174

Appendix E: Newmark-beta Average Acceleration (Chopra, 2006) 175

References..177

Vita 183

List of Tables

Table 5.1: Triangle Prism Node Table....82

Table 5.2: Triangle Prism Triangle Table....83

Table 5.3: Triangle Prism Quadrilateral Table....84

Table 6-1: Power Formula Constants for the Prestressing Stress-Strain Diagram....119

Table 6-2: Dynamic Increase Factor for Far Range....121

List of Figures

Figure 2.1: Documented Worldwide Terrorist Attacks on Public Transportation Infrastructure (Jenkins & Butterworth, 2010) 3

Figure 2.2: Percentage of Bridge Targeted in Industrialized Nation between 1980 and 2006 (Jenkins, 2001) 5

Figure 2.3: Percentage of Different Bridge System in the U.S. Inventory as of 2011 (National Bridge Inventory, 2011) 10

Figure 3.1: General Structural Analysis Program Flow 19

Figure 3.2: Blast-Component Structural Analysis Flow Chart 20

Figure 3.3: Reinforced Concrete Column Blast-Component Structural Analysis Flow Chart 21

Figure 3.4: Steel Plate Blast-Component Structural Analysis Flow Chart 22

Figure 3.5: ATP-Bridge Paradigm 29

Figure 4.1: Nexus Assembly Class Model Diagram 32

Figure 4.2: Load Assembly Class Model Diagram 33

Figure 4.3: Structural Component Assembly Class Model Diagram 34

Figure 4.4: Typical Class Notation and Layout 36

Figure 4.5: Subroutine *SetLocation(…)* Example 38

Figure 4.6: Return Function *GetLocationY(…)* Example 39

Figure 4.7: Data Storage of Sub-Class 40

Figure 4.8: Sub-class Data Type Variable 41

Figure 4.9: *SetMesh()* Subroutine Example 42

Figure 4.10: ***Structural Component*** Class Phases 43

Figure 4.11: ***Structural Component*** Class Physical Attributes and

Methods Diagram.....44

Figure 4.12: Plate Class Overload *SetGlobalDimension(…)* Subroutine 46

Figure 4.13: Structural Component Class Analysis Diagram.....48

Figure 4.14: Structural Component Class User Interface and
Graphics Engine Diagram.....50

Figure 4.15: Load Class Analysis Diagram.....52

Figure 4.16: ***Nexus*** Class Analysis Diagram.....56

Figure 4.17: Optional Data Types.....58

Figure 4.18: Overriding Optional Data Type *StrengthInputs()* Example.....59

Figure 5.1: Graphical User Interface Schematic.....61

Figure 5.2: Graphical User Interface Collapse Navigation Control.....63

Figure 5.3: Tree-View Control General Information Section.....64

Figure 5.4: Tree-View Control Structural Component Section.....65

Figure 5.5: Tree-View Control Load Case Section.....66

Figure 5.6: Viewer Setting Control.....67

Figure 5.7: *Line List* Primitive Type (Miller, 2004).....69

Figure 5.8: *Triangle List* Primitive Type (Miller, 2004).....70

Figure 5.9: Vertex Buffer and Index Buffer (Thorn, 2005).....71

Figure 5.10: ***Node*** Class Diagram.....73

Figure 5.11: JQB Element.....75

Figure 5.12: Triangle Element.....76

Figure 5.13: ***Triangles*** Element Class Diagram.....76

Figure 5.14: Quadrilateral Element.....77

Figure 5.15: ***Quadrilateral*** Element Class Diagram.....78

Figure 5.16: ***Mesh Component*** Class Diagram.....80

Figure 5.18: Triangle Prism ***Node*** Element 82

Figure 5.19: Triangle Prism ***Triangle*** Element..83

Figure 5.20: Triangle Prism ***Quadrilateral*** Element 84

Figure 5.21: ***Graphics Object*** Structure Diagram...86

Figure 5.22: Graphics Engine Flow Chart 87

Figure 5.23: ***JQB Graphics Engine*** General Class Diagram 89

Figure 5.24: ***JQB Graphics Engine*** Camera Class Diagram 91

Figure 5.25: Zooming the Camera 92

Figure 5.26: Panning the Camera..93

Figure 5.27: Rotating the Camera..94

Figure 5.28: ***JQB Blackboard*** Class Diagram Method() 95

Figure 5.29: *OneFrameRender()* Method 96

Figure 5.30: ***JQB Component*** Class Diagram Properties 98

Figure 5.31: ***JQB Component*** Class Diagram Methods() 99

Figure 5.32: *ComponentFrameRender()* Subroutine Rendering Algorithm........101

Figure 6.1: Bridge Destroyed in Iraq from Truck Bomb

(The Washington Post, al- Mokhtar, 2009) 104

Figure 6.2: Colorado Bulb-Tee Test Specimen Dimensions

(Matthews, 2008) 105

Figure 6.3: Test Specimen Above-Detonation Scenario

(Matthews, 2008) 106

Figure 6.4: Test Specimen Below-Girder Detonation Load Case

(Matthews, 2008) 107

Figure 6.5: Fiber Diagrams of the Test Specimen 110

Figure 6.6: f'_c = 8,500 psi Concrete Model Stress-Strain Curve........................115

Figure 6.7: 60 ksi Reinforcement Material Model...117

Figure 6.8: 270 ksi Low-Relaxation Prestressing Strand Material Model...........120

Figure 6.9: Dynamically Adjusted Concrete Stress-Strain Curve
(Department of Defense, 2008)...122

Figure 6.10: Dynamically Adjusted Mild-Steel Reinforcement 123
Stress-Strain Curve (Department of Defense, 2008) 123

Figure 6.11: Colorado Bulb-Tee Moment-Curvature 124

Figure 6.12: Bilinear Moment Curvature...127

Figure 6.13: Load Distribution 130

Figure 6.14: Pressure-Time Curve for Free-Air Explosion
(Department of Defense, 2008)...131

Figure 6.15: Blast Distribution Variation with Respect to Standoff
(Department of Defense, 2008)...132

Figure 6.16: Incremental Iteration (McGuire, Gallagher, and Ziemian, 2000) ...134

Figure 6.17: Equivalent Single-Degree-of-Freedom System..............................138

Figure 6.18: Pressure-Time History at Girder Center and Left/Right Edge 143

Figure 6.19: Resistance Function...145

Figure 6.20: Forcing Function 146

Figure 6.21: Displacement/Velocity/Acceleration Time History
at the Mid-Span...148

Figure 6.22: Girder FEM Displacement-Time History at Mid-Span..................149

Figure 7.1: Geometry Form for Prestressed Girder 155

Figure 7.2: Load Form for Prestressed Girder 157

Figure 7.3: Graphics Engine Rendering of Prestressed Girder...........................158

Figure D.1: Frame Element Degrees-of-Freedom
(McGuire, Gallagher, and Ziemian, 2000)....................................171

"That's been one of my mantras – focus and simplicity. Simple can be harder than complex: You have to work hard to get your thinking clean to make it simple. But it's worth it in the end because once you get there, you can move mountains."

Steve Jobs (1989)

Chapter 1: Introduction

In the decade following September 11, 2001, government officials and the engineering community have devoted time and resources to protect the country from such attacks again. The highway bridge infrastructure, a system utilized daily by most Americans, is considered a potential terrorist target because it is a public symbol, and an attack can cause major disruptions to the local economy. Before the last decade, only a limited number of studies considered the response of bridge components subjected to blast loads. Although buildings and bridges have similar structural components, there are a number of behavior variations that warrant additional exploration. Furthermore, most bridge engineers have limited experience in blast-resistant design principles.

With these concerns, researchers at the University of Texas at Austin were tasked with the creation of *Anti-Terrorist Bridge Planner* (ATP-Bridge) PC software. The purpose of this software is to transition the knowledge gained through research over the last decade into a user-friendly software that allows bridge engineers and planners to investigate the response of different bridge components to a postulated terrorist threat scenario involving explosives. Analysis results provided by the software can be used to conduct vulnerability assessments of planned or existing bridges, and the information can also be used to determine whether structural hardening is needed to protect a critical bridge component. The project is funded by the US Department of Homeland Security (DHS) and is overseen by the US Army Corp Engineer Research Development Center. The software is designed to analyze a variety of bridge components, from reinforced concrete columns to steel plates on a bridge tower. As new research becomes available, additional components can be incorporated into the software.

This thesis outlines the development of the software architecture and lays out the different programming concepts used to address the identified challenges. The main challenge in developing ATP-Bridge is balancing the desire to create user-friendly software while still incorporating multiple bridge components with different loading conditions, material behavior, and modes of failure. The scope of this thesis covers the motivation for the research, the theory behind the software architecture, and how the software interacts with the user. This thesis also includes an example of a bridge component numerical model (prestressed girder) and describes how it is implemented inside the program.

THESIS OUTLINE

The thesis is divided into seven chapters, including this introductory chapter. Chapter 2 provides an overview of threats against bridges, prior experimental testing against bridge components, current practices, research motivation, and current design challenges. Chapter 3 describes the program flow path, the different programming styles, and general aspects of object oriented programming that are relevant to the current study. Chapter 4 explains the relationships between the core data structures, and it gives a thorough explanation on how they were implemented in practice. Chapter 5 lays out the schematics of the graphical user interface and provides an in-depth discussion on the graphics engine. Chapter 6 presents an advanced single-degree-of-freedom model using a layer-by-layer moment curvature response curve and non-linear static analysis for the resistance function. Finally, Chapter 7 ends the thesis with further recommendations for the software and the prestressed girder model.

Chapter 2: Background, Motivation, and Challenges

There have been terrorist threats against US interests, both domestically and abroad, dating back well before September 11, 2001. Events like the 1998 bombing at the US embassies in Tanzania and Kenya (FEMA 427, 2003), the 1993 World Trade Center truck bombing at the North Tower (NRC, 1995), and the Oklahoma City bombing of the Alfred P. Murrah Federal Building (NRC, 1995) highlight the need for structures to be designed with consideration given to blast protection. After the tragic events of September 11, 2001, there has been a renewed focus amongst policy makers about the security and readiness of the country to defend against such attacks across a wide range of infrastructure (FHWA, 2003).

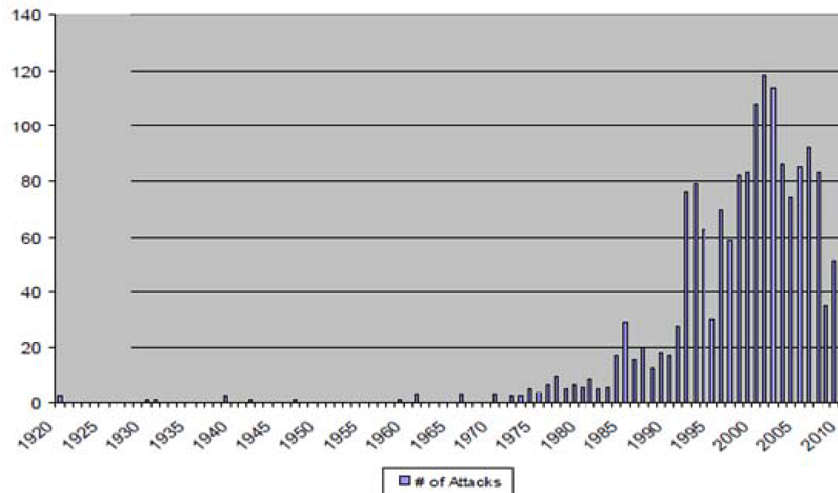

Figure 2.1: Documented Worldwide Terrorist Attacks on Public Transportation Infrastructure (Jenkins & Butterworth, 2010)

The protective design community has previously focused on military structures and federal buildings. Yet, as shown in Figure 2.1 there has been a worldwide increase in documented terrorist threats against public transportation infrastructure starting roughly around the 90s. Over the last decade, however, awareness of the vulnerability of public highway infrastructure has been raised, with concerns not only from a safety perspective but also an economic and socioeconomic perspective (Williamson, et al. 2010).

TERRORIST THREATS AGAINST BRIDGES

The Mineta Transportation Institute (MTI) documented 1633 worldwide terrorist attacks against public surface transportation infrastructure since the first quarter of 2010, including 161 attacks against highway infrastructure (Jenkins and Butterworth, 2010). Although there has been only one documented bridge attack on US soil—the 1977 Route 1 Bridge in Florida Homestead and Key West—there have been many reported incidents during the last decade that show bridges are potential targets for terrorists. On February 16, 1982, Oakland's Bay Bridge was targeted by an unknown terrorist leaving 40 lbs of liquid explosive beneath the bridge (Jenkins, 1997). On June 29, 1993, the George Washington Bridge was the target of nine arrested terrorists (Jenkins, 1997). The Brooklyn Bridge has been targeted in the past, with plans to bring down the bridge with various methods such as cutting through the suspension cables (Weiser, 2011).

Even though all the bridges described above are iconic bridges, they are only a small portion of the total bridge inventory within the US. Data have shown that it is more likely that typical bridges are targeted, as shown in Figure 2.2 (Jenkins, 2001). Bridges

can be the only entrance to a community, and they can be an important route for critical infrastructure like power plants, water treatment facilities, or shipping yards. If a series of attacks were coordinated, it could cause devastating economic consequences to the surrounding communities and beyond.

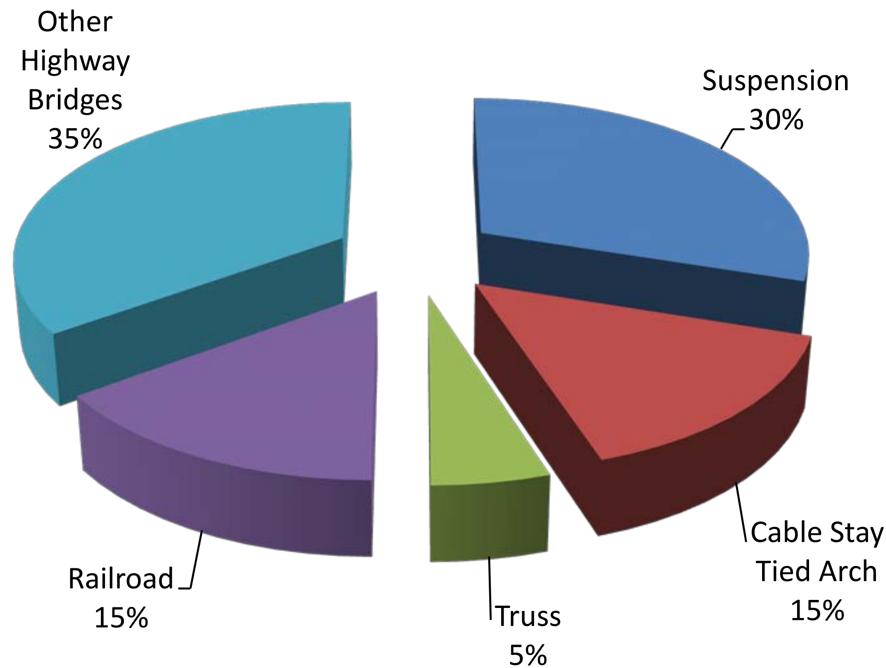

Figure 2.2: Percentage of Bridge Targeted in Industrialized Nation between 1980 and 2006 (Jenkins, 2001)

EXPERIMENTAL RESEARCH AND CURRENT PRACTICE

Within the last decade, several researchers have studied the effects of blast loads acting against different bridge components. The previous projects consider different bridge systems and materials.

For example, researchers at the University of Texas at Austin performed half-scale tests on bridge columns (Williams, 2009). This work was funded by the National Cooperative Highway Research Program. The researchers studied the behavior of rectangular and circular columns, varying the transverse reinforcement detailing and the overall dimensions to determine the governing failure mechanisms associated with different threats and design parameters. Knowledge from that research led to blast-resistant design guidelines, which are detailed in NCHRP Report 645 (Williamson, et al., 2010).

Bruneau and Fujikura from SUNY Buffalo studied the response of quarter-scale concrete pier-bent models (Bruneau, 2010). Their goal was to prevent breaching, which is characterized by the complete loss of concrete at a localized cross-section, from occurring in the piers. The study led them to include a composite steel jacket along the height of the column. Their primary conclusion was that the addition of the composite jacket increased the ductility on the column.

Aside from columns, research has been conducted on other critical bridge components. Prestressed girders were the focus of a recent study by Cofer (2012). This project included full-scale tests against typical AASHTO girder designs for both above- and below-deck threats. Data collected from the tests were used to validate detailed finite element models. After achieving a satisfactory single girder model, they extrapolated the model to include a full cross-section of a composite girder deck. The main conclusion reported is that the primary mode of failure is concrete rubbelization and shear failure.

The Federal Highway Administration (FHWA) funded research involving the blast testing of one-quarter scale representations of steel suspension bridge towers. Testing for this project was carried out by the Engineer Research and Development Center (ERDC) of the US Army Corp of Engineers (Ray, 2006). The objective of the test program was to determine the blast resistance of steel plates subjected to a variety of threats, considering a wide range of support conditions. The measured steel plate response was compared with results from numerical simulations to determine how accurate the models were at predicting the failure mechanism. ERDC also considered potential retrofit options and evaluated them during the test program.

The cited projects above provide insight to the blast-resistant design community on the actual behavior of different bridge components subjected to explosive threats. To protect the approximately 600,000 bridges in the US, however, there needs to be a design tool that is capable of analyzing the majority of the bridge components found in practice.

CURRENT PRACTICE

Currently, there are multiple approaches available for analyzing structures subjected to blast loads. The tools range from approximating a component as a single-degree-of-freedom (SDOF) system to a coupled fluid-structure finite element analysis. Although seismic and blast loadings on bridges share similar traits, designing for seismic loads will not necessarily satisfy the requirements for blast (Holland, 2008). The methods used for a given project will depend upon the experience and expertise of the

project personnel, the budget, and the desired accuracy from the client (Winget and Williamson, 2000).

Although it might be desirable for an engineer to model a bridge globally, it is usually not necessary to do so for assessing bridge vulnerability to potential blast threats. Because the pressure from a blast wave attenuates quickly with distance, damage is typically localized around the detonation site (Cofer, 2012). Thus, studying individual bridge components can provide valuable insight on how a bridge performs when subjected to an explosion while maintaining simplicity in the engineering models used to predict response.

Determining whether a given bridge should be designed for blast protection should be based on a vulnerability assessment. A vulnerability assessment, such as the one suggested by AASHTO (2002), should determine the importance and criticality of a bridge relative to the whole infrastructure network. If a bridge is deemed to be at risk, it should be assessed at an individual component level. Ray (2007) recently proposed a method for determining the risk associated with individual bridge components based on the occurrence, vulnerability, and importance of the individual members.

If by choice of the owner or through the outcome of a formal vulnerability assessment it has been determined that a bridge requires protection from blast loads, there are several options available. The most efficient means of protecting highway bridges is during the design phase through planning and site layout (Winget, et al., 2010). Planning could include increased surveillance, fencing off critical components, or increasing the

lighting around dark areas. Because these options are not always available, it is important to have available tools for retrofitting and analysis.

Purpose of the Research Project

The main objective of this project is to develop the Anti-Terrorist Planner for Bridges (ATP-Bridge) software. This work is funded by the Department of Homeland Security and overseen by the US Army Corp of Engineers. The software is created for a Windows-based PC, with a graphical user interface (GUI), navigation control scheme, and 2D and 3D rendering engine. These components will be explained in subsequent chapters.

The intent of the software is to provide a simple tool for bridge engineers to use when designing bridges to resist blast and other extreme loads. There is a great need to protect our domestic infrastructure; however, with the economic reality that there are limited resources available, it is not possible to have separate, detailed analyses for every bridge. Therefore, this software will not only allow engineers and planning personnel to quickly evaluate whether a bridge needs further protection, but it will help bridge engineers during the design process.

The software is to be a clearinghouse of experimental research done in the last decade and to be expandable to allow the incorporation of new data as it becomes available. To reflect the diversity of bridge systems around the US as shown in Figure 2.3, the ATP-Bridge software is component-based rather than focusing on a single system.

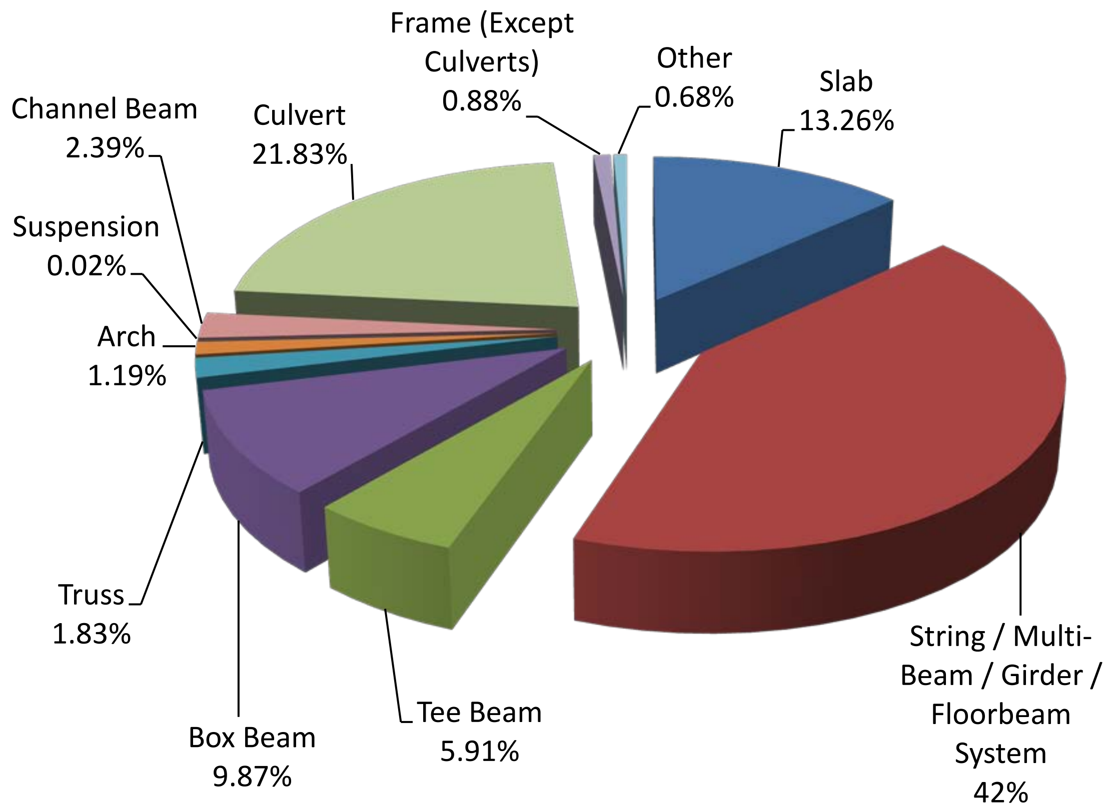

Figure 2.3: Percentage of Different Bridge System in the U.S. Inventory as of 2011 (National Bridge Inventory, 2011)

Structural analysis options available vary depending upon the complexity of the component being analyzed. Some bridge components can be accurately modeled as a single-degree-of-freedom system, whereas other components are complex and require correlation with empirical data. Although finite element models, if used correctly, are considered more accurate than SDOF or empirical models, they require a large investment in time and computational resources.

ATP-Bridge does not require users to decide the most appropriate method of analysis for a given component. Rather, the software has built-in algorithms that have been developed by researchers on this project to provide accurate predictions of response. The algorithms have been validated against test data and detailed finite element models. With the adopted software development approach, the user interface and the analysis modules are self-contained, making it possible for analysis modules to be coded by different developers without needing to know the details of the interface.

A possibility for component-based software is to compute local damage caused by a blast load and then map the damage back to a static global structural analysis program. With the new global model, an engineer can evaluate the residual capacity of a bridge system without needing to do a costly non-linear analysis of the entire structure.

GOAL OF THE SOFTWARE

"The complexity of software is an essential property, not an accidental one." (Brooks, 1986)

The ATP-Bridge software is organized to analyze individual bridge components rather than a complete system, which leads to several development challenges. Of primary importance is creating a simple and intuitive user interface that supports a multitude of different components, each having different material properties and response characteristics. To create a user-friendly interface, the steps required to define input parameters or to review the results must be similar and predictable across the different components. Doing so requires the core data structure of the program to be generic and

have the ability to store different types of data. This requirement for a generic data structure makes it possible for the interface, including the navigational control system and the 3D graphics environment, to behave seamlessly amongst the different components.

The goals for the software create a complex set of challenges for its development. Booch (1994) divides these complexities into four different elements: (1) the complexity of the problem domain, (2) the challenge of managing the developmental process, (3) the flexibility throughout the process, and (4) the problem of characterizing a discrete system. The following is a discussion of these different elements as it relates to ATP-Bridge.

The Complexity of the Problem Domain

“This external complexity usually springs from the ‘impedance mismatch’ that exists between the users of a system and its developer: users generally find it very hard to give precise expression to their needs in a form that developers can understand.” (Booch, 1994)

The problem domain can be generically described as required information that is needed to find the solution to a problem. It is important for the client and the developer to communicate their individual vision of the program throughout the software development period.

The general framework for the ATP-Bridge software is a multi-component blast design tool. Single-component analysis software requires a user interface for the inputs and outputs along with a solver to calculate the desired results. The challenges are magnified with the creation of a software package that can handle multiple types of individual components, each having their own different sets of inputs and outputs. Another challenge is maintaining and balancing requirements that might contradict each other for different components. Completely integrating all components to a standard interface is difficult without knowing in advance all the different components that the software may eventually include. This is due to the fact that the scope of all the components is not fully defined, and it is not possible to fully define them because ongoing research may require new interpretations of existing data. Thus, it is not clear how to enforce standardization for the user interface or even if it is desirable to do so because future components might be unnecessarily constrained due to current choices.

Another challenge to the problem domain concerns how the software evolves with time. Success is measured not only on how accurate and well developed the software is, but also on how useful it is to the protective design community at large. The software needs to address the problems that users face. The decision to make it component-based reflects the research work done to date. Nonetheless, even during the early stages of program development, consideration must be given to whether the software may eventually allow for a global bridge model to be analyzed or whether other output (e.g., vulnerability) may be needed.

The Challenge of Managing the Developmental Process

"The fundamental task of the software development team is to engineer the illusion of simplicity – to shield users from this vast and often arbitrary external complexity." (Booch, 1994)

There are a variety of different software development models that exist today, with classical ones such as the "waterfall" model and the stepwise refinement model, to the more modern ones such as rapid prototyping and the joint application development model (Scacchi, 2001). All the different models have the same overall goal of providing a conceptual scheme for managing the development process. Each model may vary in its implementation, but all development models include planning, requirement analysis, functional specification, detailed component design specification, debugging, and maintenance (Scacchi, 2001).

A concern when developing large software projects is carefully managing the development cycle. With poorly managed projects, it becomes increasingly difficult to make small changes without having a big impact on code that was developed earlier. The inertia of the software will become too hard for the development team to overcome as the size of the software increases. Therefore, it is important for the team to constantly iterate and reevaluate the efficiency of the software.

With a large team that is working on different areas, there will inherently be fragmentation amongst the groups, with models and interface development moving at different paces. To accommodate this challenge, there needs to be early planning and

coordination amongst the team on how the different areas mesh together. It is necessary for the software to be well documented and follow strict guidelines to become predictable for the developers to maintain the integrity of the software over its lifetime.

There are issues that arise when managing the software development process in a university research setting. It is always desirable for the development team to be intact throughout the full cycle of the software development. Because of the turnover of graduate students coming into the program, however, it is not possible to maintain the same development team for its full duration. Therefore, as stated above, it is necessary to keep things well documented.

Despite the best efforts of the developers, software will have bugs, and users may experience crashes or incorrect functioning of a program. Maintenance beyond the development stage is a consideration that the client needs to consider. Although most bugs should be corrected during the debugging cycle, software in commercial settings will have issues that the development team will be unable to predict. A program with a graphical user interface only increases the unpredictability, where concerns about the physical environment change for each user and the 3D graphical environment depends on physical hardware like the graphics card.

The Flexibility Possible through Software

"Software offers the ultimate flexibility, so it is possible for a developer to express almost any kind of abstraction. This flexibility turns out to be an incredibly seductive

property, however, because it also forces the developer to craft virtually all the primitive building blocks upon which these higher-level abstractions stand." (Booch, 1994)

Software that is poorly organized and too complex to use will frustrate the target audience and discourage them from using it. That is why it is critical that users find it intuitive and responsive to how they think. The problem is that users have different styles and behave differently from each other. Although it is impossible for software to satisfy everyone, it is important to develop a flexible program that gives multiple ways to progress through an analysis and to communicate and visualize material in more ways than one.

Different people learn in different ways, with some being more visual while others are more hands-on. These different learning styles are similar for a person using computer software. That is why successful software needs to have multiple ways to perform the same task. Different people are more attuned to clicking menu items or toolbar buttons, while others prefer using the keyboard to navigate through the software. All this additional flexibility added to the program adds to the challenges and requires additional planning and coding.

The Problems of Characterizing the Behavior of Discrete System

"Because we execute our software on digital computers, we have a system with discrete states. By contrast, analog systems such as the motion of the tossed ball are continuous system." (Booch, 1994)

In analyzing structures subjected to blast loads, there is a large variability not only in the characterization of the explosive but also in the way component response is computed. Aside from errors associated with the variability of the input parameters, implementation of the solution on computers only adds to the error due to the limits of compound round-off and data type manipulation. While these limits are unavoidable, it is important for program developers to understand and account for them in their software design.

Another concern is the validity of the component models beyond the developers' intent. All the component models should be verified for accuracy within a reasonable range, but it is impossible to check every input a user may provide. Users could generate a section property that is valid but inherently unstable, apply a load that is theoretically possible but outside the range of knowledge, or trigger an equation that will return a result close to infinity. To limit these difficulties, a vigorous debugging process is necessary.

SUMMARY

This chapter provides an overview of the vulnerability of transportation infrastructure, previous experimental research, and current design practice. To allow bridge engineers the ability to analyze structural components subjected to blast loads, the primary objective of this research is to develop a user-friendly software application that runs efficiently on a personal computer. Following a brief description of the desired

software capabilities, the goals and challenges of developing such a program are discussed. The next chapter details how the software design challenges are addressed in this study.

Chapter 3: Software Architecture and Design

To ensure that current and future versions of ATP-Bridge are robust, thorough planning was devoted to the development of its global framework. Although there has not been a universally adopted definition for software architecture in the computer science community, a good explanation provided by Microsoft (2009) describes it as the interaction of the major program elements with each other and the hardware. Software architecture also takes into consideration the needs of various stakeholders: the users, the clients, and the developers. This chapter details the overall software framework and paradigms.

PROGRAM FLOW CHART

Examining a high-level view of solving any structural analysis problem, there are three generic stages involved as shown in Figure 3.1. First is the [Pre–Processor] segment, where the problem domain is defined and the scenario is understood. Second is the [Analysis] segment, where the program will choose a solution method (empirical equations, direct stiffness method, finite element method, etc.) to solve the problem domain. Lastly, the [Post-Processor] segment is where the results are processed into desired output (producing graphs, contour plots, etc.) and reported to the user.

Figure 3.1: General Structural Analysis Program Flow

For the analysis of a structural component subjected to blast, the three segments from the general cases are further refined into blast-specific segments. The [Pre-Processor] segment is divided into the [Define Component Geometry] and [Define Blast Loads] segments, and the [Post-Processor] segment is divided into the [Failure Mode] and [Display Results] segments (Figure 3.2). The [Define Component Geometry] could consist of many different parameters depending on the type of component being analyzed. Examples include the global dimensions, shape, material properties, and boundary conditions. The [Define Blast Loads] segment can consist of many different parameters that are specifically associated with the component being analyzed, such as charge weight and diameter, charge location, and proximity to a reflecting surface such as the deck. The [Failure Mode] segment consists of local failure and global failure. Output from the [Display Results] segment varies depending on the fidelity of the component model used. Examples of the different results are displacements, spall length, and breach length.

Figure 3.2: Blast-Component Structural Analysis Flow Chart

Further refinement is possible for specific components, as shown in Figure 3.3 and Figure 3.4 for a reinforced concrete column and steel plate, respectively. With further refinement, there is greater divergence in common segments that are shared

among the different components. Comparing the flow chart of the reinforced concrete column with the steel plate, the two differ largely on the type of geometry declarations, including parameters such as support conditions and material properties. For loads, both components utilize similar parameters with the exception that reinforced concrete columns need information about the deck to calculate reflections. These variations in required input only compare two structural components. With each additional structural component, there will be even more variation, and planning the interface becomes a significant challenge.

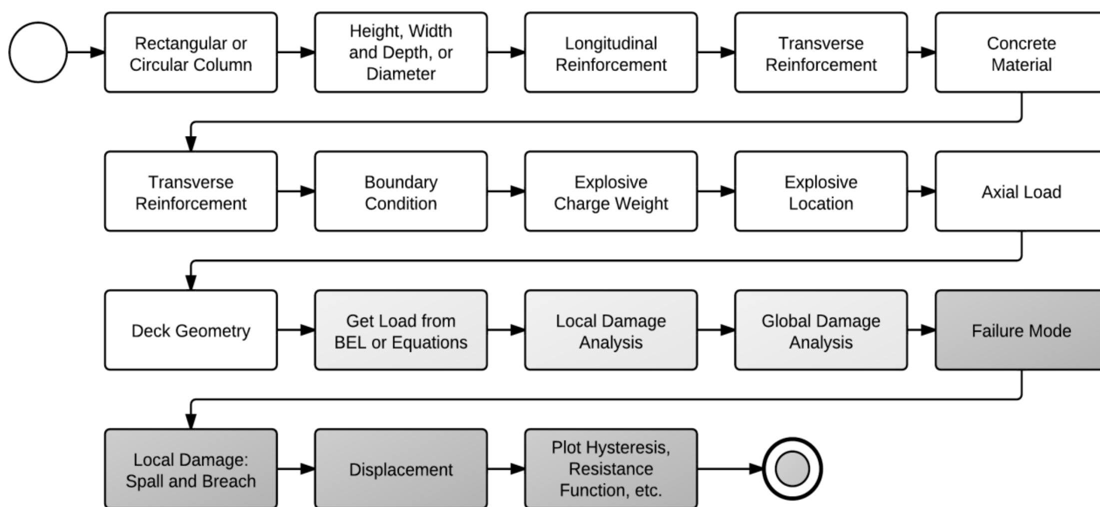

Figure 3.3: Reinforced Concrete Column Blast-Component Structural Analysis Flow Chart

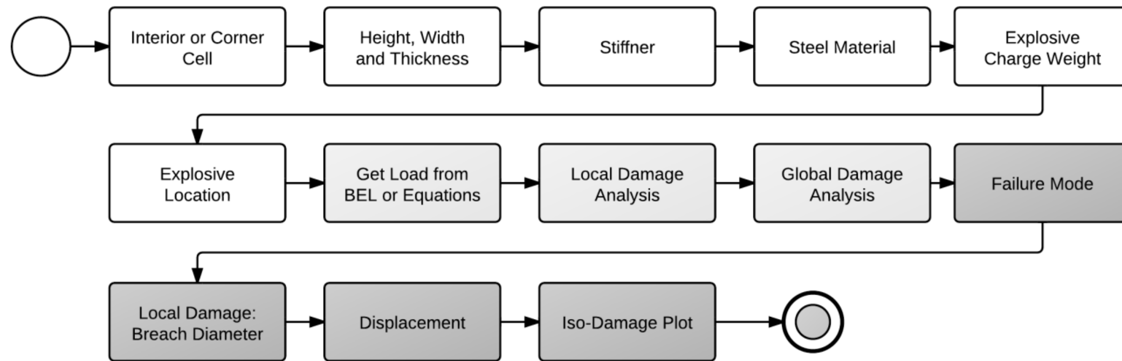

Figure 3.4: Steel Plate Blast-Component Structural Analysis Flow Chart

To avoid problems associated with the creation of component-specific input and output, ATP-Bridge is organized and planned according to a blast-component structural analysis flow chart. Each component is organized in broad segments and with the interface design for a generic component type. This generic flow path addresses the challenge of adding new components into the software as the project progresses.

PROGRAM PARADIGM

After determining the overall strategy in the software flow path, the next stage is to determine the appropriate paradigm. Paradigm is used in this thesis as the structure, style, and relationship regarding how the program is coded and developed. There are many different paradigms that can be used during software development, with some being more efficient than others at solving specific tasks. This thesis does not cover all of the different possibilities. Rather, it focuses on the ones used in ATP-Bridge. The

following sections provide brief explanations of the paradigms used along with references where additional information can be found.

Modular Programming

The general concept of modular programming focuses on creating independent, interchangeable components that complete a task within a module. Defined by Turner (1980) as the "locality of reference", a module needs to be self-contained and can only reference things inside its scope and passed through its boundaries. The data passed to a module must have clearly defined data types. This separation of tasks lends itself well to team projects, where different developers can focus on their specific concerns and easily maintain the work inside the modules they are developing.

Object-Oriented Programming

Object-oriented programming was developed to overcome the underlying security issues with reusing modules. There are two types of object-oriented programming: class-based and prototype-based (Craig, 2000). The focus in this thesis is on the former. Any references made to object-oriented programming in this thesis assumes the class-based approach. Because object-oriented programming is an important concept within the architecture of ATP-Bridge, the following paragraphs give a thorough description of the important features.

At its most fundamental level, object-oriented programming models all problems as *classes* and *objects*. As stated in Craig (2000), the concept of classes can be thought of

as a set of objects, a program structure, a template that produces objects, or a data type. Each definition is correct, but none fully defines the whole entity.

Classes hold both data and methods wrapped into one 'data type'. Objects, conversely, take all the attributes of the class and create an independent instance of that class, where it then is possible to be passed and returned through functions and subroutines, assign variables, and store arrays and data structures. A clear difference between classes and objects is that a class does not exist at runtime, whereas an object does.

To distinguish between user-defined classes and objects in this thesis, user-defined classes will be designated with "bold-italic" font while objects of that class are designated with just "italic" font. For example, a class of ***Animals*** will have objects of that class designated as *tiger*, *bears*, and *fish*.

What makes object-oriented programming a powerful concept and a popular programming style is its adherence to its principles, as discussed in Brooch (1994). This project uses the concepts of abstraction, encapsulation, polymorphism, and inheritance. The following sections give a general description and examples of those concepts.

Abstraction

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide(s) crisply defined conceptual boundaries, relative to the perspective of the viewer." (Booch, 1994)

Data abstraction is the concept of factoring out all the unnecessary details and capturing only those details that are relevant to the problem domain. An example of abstraction is trying to calculate payroll for an employee. When calculating payroll, it is unnecessary to know the employee's eye color, what car they drive, or their favorite food—those are unnecessary details. The information that is needed to be abstracted is the employee's social security number, salary, and mailing address. Abstraction is essential to modeling.

Encapsulation

"Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation." (Booch, 1994)

Encapsulation is information hiding. There is an interface for the public to access, but the internal workings are restricted and hidden. A typical example of encapsulation is a person driving a car. Most drivers do not know how cars are built, how engines work, or even how radios are connected. That knowledge is not required for individuals to drive a car. The information provided to the driver is how to start the ignition, where the gas pedal is, and how to shift gears. The benefit of encapsulation in programming comes when there is a need to improve a code's efficiency. When encapsulation is utilized correctly, there is no need for something previously written to know that a change had been made as long as it returns the same values.

Inheritance

"Hierarchy (inheritance) is a ranking or ordering of abstraction." (Booch, 1994)

Inheritance is an important concept that is used to achieve flexibility. Inheritance makes a family of classes by designating *super-classes* and *sub-classes*. Super-classes can be described as parent classes or base classes, where sub-classes are child classes, derived classes, or inherited classes. This concept leads to a hierarchal tree, where there can be many sub-classes for one super-class. Super-classes hold traits that are applicable to sub-classes, such as data types, subroutines, and functions. Sub-classes, meanwhile, inherit the attributes of the super-class, which reinforces reusability of codes. Therefore, sub-classes can reference codes from the super-classes, but super-classes cannot reference code from the sub-classes. An example of hierarchy is the animal kingdom, with reptiles and mammals both being a sub-class of animals, and tigers and bears being sub-classes of mammals. Traits like eating and sleeping are shared amongst all animals; having fur is specific to mammals.

Polymorphism

"The term means literally 'many formed' and refers to the property of object-oriented languages that they permit routines to have more than one type assignment." (Craig, 1994)

As defined in Craig (1994), there are three primary types of polymorphism: *genericity*, *inclusive polymorphism*, and *ad hoc polymorphism*. Genericity is an idea that

relates with inheritance, where a super-class and sub-class share methods with the same signature but return different results when called. Inclusive polymorphism is the idea that methods defined for a super-class will be made available to its sub-classes. Ad hoc polymorphism is commonly called *methods overloading*; methods with the same name but with different signatures (parameters inside the method calls) will access different algorithms.

Event-Driven Programming

ATP-Bridge, as well as most GUI software, uses *event-driven programming*. Conceptually, this type of programming is based on interaction with the user and does not have a pre-determined sequential order. ATP-Bridge reacts to triggered events, such as clicking the mouse or typing a letter on the keyboard. Once an event is triggered, a set of operations is performed that were written for the specific event. A key component to user-friendly software is developing multiple paths to a desired destination, and ATP-Bridge provides a variety of ways to carry out specific tasks including use of pull-down menus, shortcut icons, and text entered from the keyboard. See Faison (2006) for a detailed discussion on event-driven programming.

ATP-Bridge Paradigm

As stated in Van Roy (2009), most sizable software has two or more program paradigms that fully describe the relationships among different elements. With the demand of maintaining a flexible and robust program, ATP-Bridge uses a mixture of

event-driven programming, object-oriented programming, and modularized programming at different levels.

At the highest level, the program uses the modular paradigm, dividing the software into two segments: a front-end and a back-end. The front-end segment incorporates the interface and the data structures, with both adhering strictly to an object-oriented paradigm. In addition to the object-oriented paradigm, the interface also adheres to the event-driven paradigm to interact with the user. The back-end module incorporates a solver for the different bridge components that are analyzed. Those solvers are unconstrained to any specific paradigm, and other developers may determine the most appropriate paradigm for it. Figure 3.5 provides a visual description of the program organization.

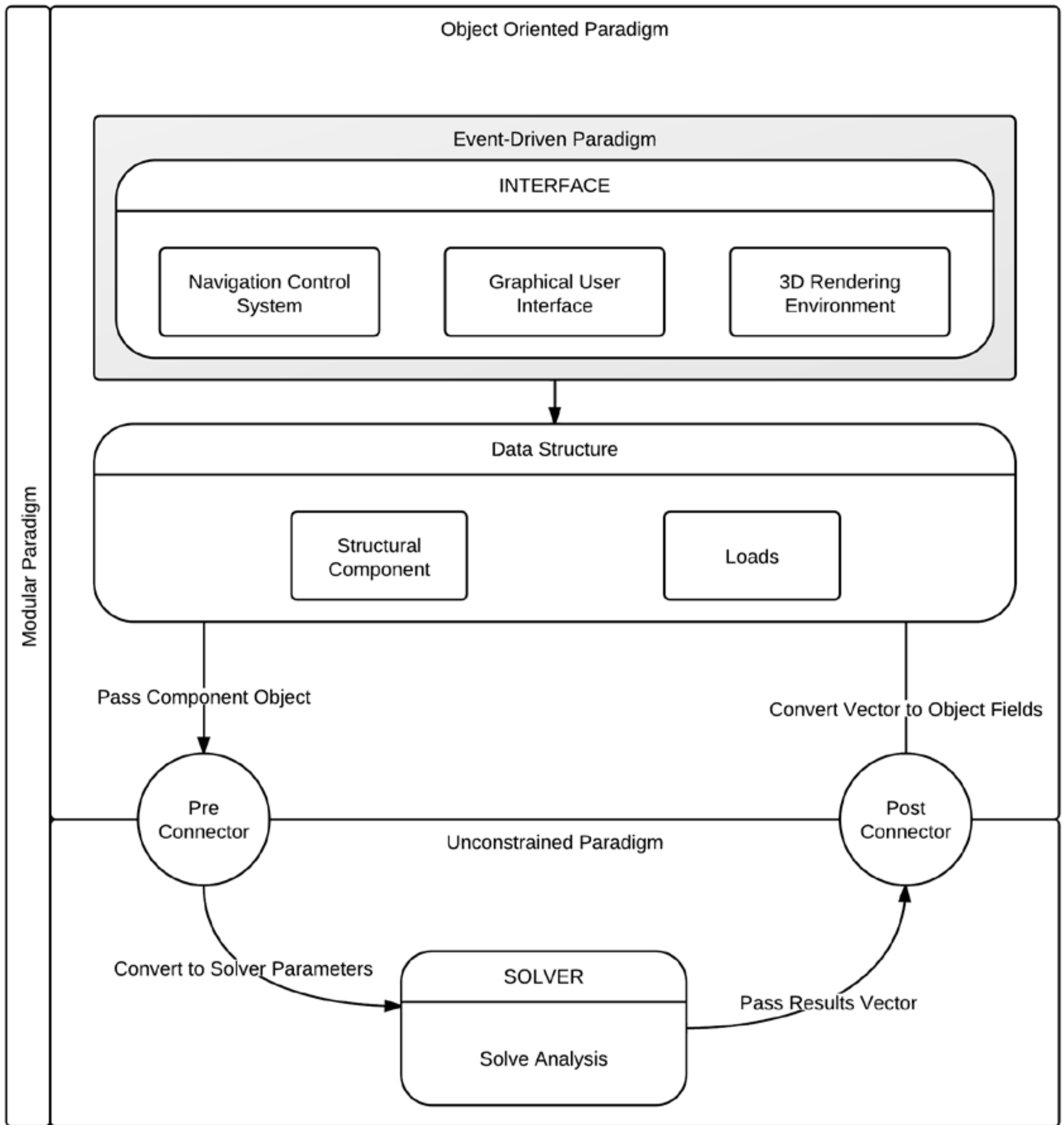

Figure 3.5: ATP-Bridge Paradigm

The user interacts with the interface to characterize the problem parameters. Once the user is satisfied with the parameters, the information is stored in data structures and saved until triggered for analysis. When the user initiates the command to carry out

an analysis, assuming both the corresponding structural component and load classes are complete (i.e., have parameters that fall within appropriate ranges and completely define the problem being solved), the program will convert the data structures into appropriate parameters for the specific solver. Once the analysis is complete, the results are converted back into parameters to store in the data structures. Users can then prompt the interface to see the results.

This careful arrangement among the three paradigms ensures reliability, robustness, reusability, and security. It also provides the flexibility necessary for this software to mature. The pure separation from the modular paradigm allows the different components of the software to be developed by different developers concurrently without needing constant interaction. An important benefit from the modular paradigm is its clear separation of the least sensitive information—the interface and data structure—with the more sensitive information—the solvers and external loading software. This separation compartmentalizes the information not only from the users but from the developers of the program. Similar to the military concept of "need to know", information that is not necessary for the different developers to accomplish their tasks is left hidden.

SUMMARY

This chapter provides a description of the overall flow path of ATP-Bridge, explains modular, object-oriented, and event-driven programming, and presents the ATP-Bridge Paradigm. The next chapter describes in detail the main data structures: structural component, loads, and nexus.

Chapter 4: Data Structures

Data structure is defined by Lafore (2003) as "the arrangement of data in a computer's memory." This chapter focuses on the core ATP-Bridge software data structures: the ***Structural Component***, the ***Load***, and the ***Nexus*** class. The collection of these three data structures is used by all the different components inside the software, similarly to how the heart and circulatory system run all the major organs inside a body. Without them there is no path to connect the user with the interface, the interface with the solver, and the solver to the results.

Data Structure Relationships

From Chapter 3, it is clear that there should be at least two generic data structures in ATP-Bridge: the ***Structural Component*** and the ***Load*** classes. In addition to these two classes, there is one more important data structure: the ***Nexus*** class. *Nexus* is defined in the Oxford dictionary as "a connection or (a) series of connections linking two or more things." The ***Nexus*** class used in this software links the front-end segment with the back-end segment of the software. The front-end segment consists of the graphical user interface, whereas the back-end segment consists of the bridge component model and Bridge Explosive Loading (BEL) software, developed by the US Army Corps of Engineers to predict blast loads against bridge components (USACE-ERDC, 2004).

The relationship among these three important data structures is important to the overall organization of the software. Below are three different relationships that were

considered for the software: the Nexus assembly, the Load assembly, and the Structural Component assembly. The names of these three options relate to the class that is the top tier of the associated assembly.

Nexus Assembly

Shown in Figure 4.1 is an assembly of the ***Nexus*** class, with one ***Structural Component*** and ***Load*** class contained inside the ***Nexus*** class. The benefit of this model is that the interior classes are independent of each other; therefore, the order in which they are defined is unimportant.

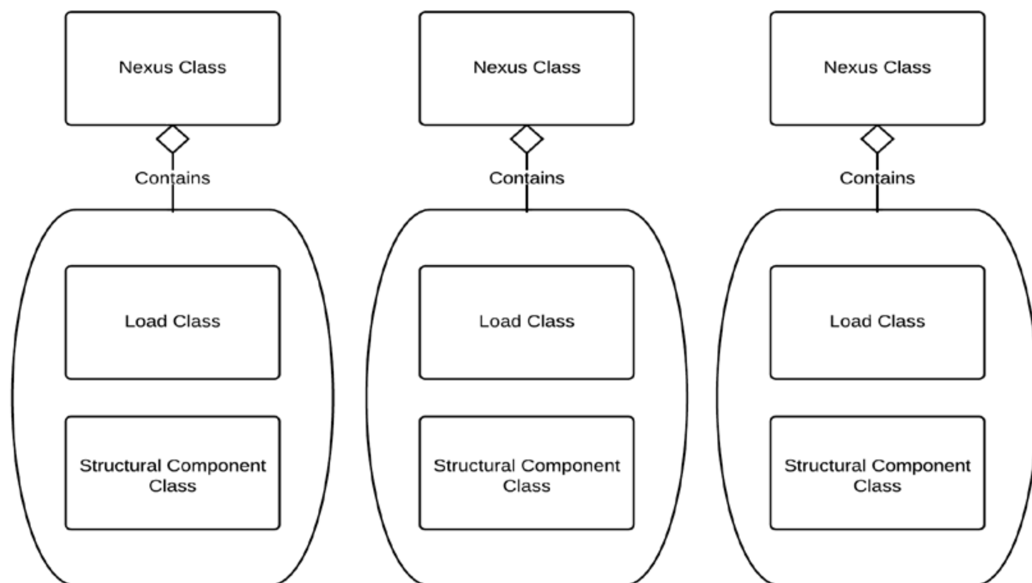

Figure 4.1: Nexus Assembly Class Model Diagram

One drawback of this assembly occurs when a user wants to observe the behavior of different loading conditions on the same bridge component. Under these conditions, the program requires the same ***Structural Component*** class to be declared multiple times,

adding unnecessary and redundant data types. In addition, when changes are required for the ***Structural Component***, multiple actions are necessary to change the same parameter through the different assemblies.

Load Assembly

In the Load assembly, the ***Load*** class is the host of the assembly, as shown in Figure 4.2. Contained inside the ***Load*** class is an array of ***Structural Component*** classes; within a ***Structural Component*** class there exists one ***Nexus*** class.

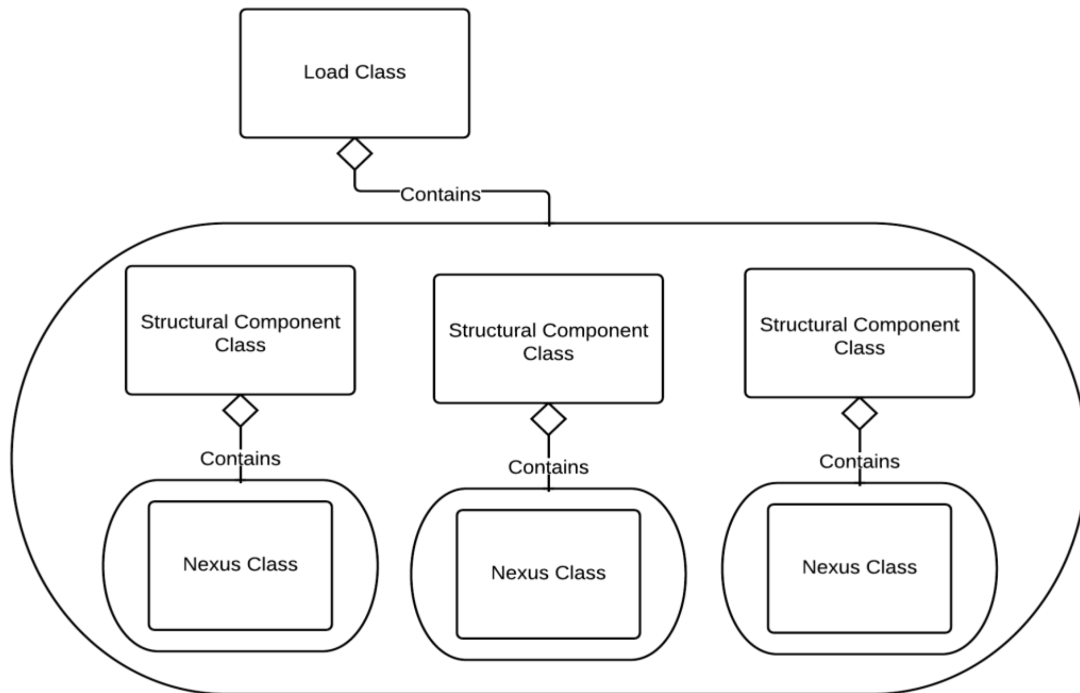

Figure 4.2: Load Assembly Class Model Diagram

A good illustration of this assembly is a single explosive relative to an entire bridge. A designer can define one explosive in coordinate space and then define multiple

bridge components (columns, beams, bent, slabs, etc.) with respect to that explosive. With this information defined, it is possible to calculate the response of each component independently of the others by simply considering the relative distances from the explosive.

Structural Component Assembly

Finally, the third assembly is similar to the second one but with the major difference of the ***Structural Component*** class being the host instead of the ***Load*** class. This assembly is shown in Figure 4.3. The benefit of this assembly is that it follows the procedure used in practice to design against blast loads.

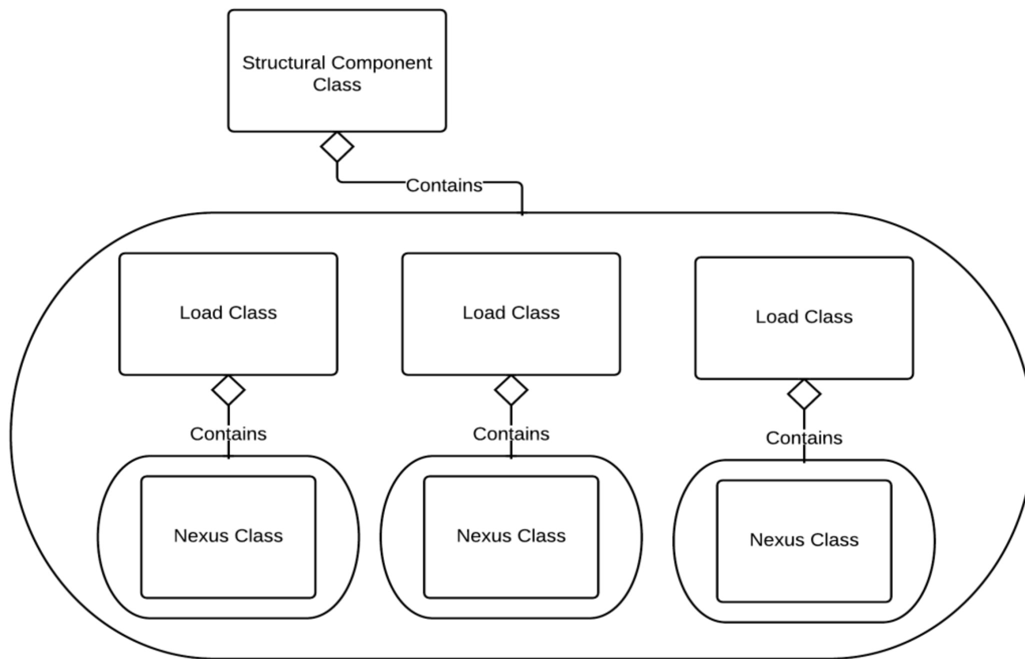

Figure 4.3: Structural Component Assembly Class Model Diagram

In practice, engineers would determine the critical component in a structure, and then they would run a series of possible loading scenarios to determine the component's response. Designers would vary the standoff distances or charge weight to determine its overall vulnerability. Lastly, a determination of desired protection or retrofit needed for the component is considered.

CLASS NAMING CONVENTION AND STANDARD

A thorough explanation of the definitions, standard organization, naming convention, and keywords will enhance the readability of the remaining parts of this chapter. The front-end segment of the software is coded primarily in Visual Basic.NET (VB.NET); therefore, the keywords defined below rely on standard Visual Basic syntax. Although the keywords vary across different programming languages, the concepts are nearly the same. The following is a series of definitions and keywords. For a complete list, see the glossary included at the end of this thesis.

- 'MustInherit' – specifies that a class cannot be used to declare a new object and that it only exists as a super-class that must be inherited.
- 'MustOverride' – specifies that a method has no body in the current class, and it must be overridden in the sub-class before use.
- 'Overridable' – specifies that a method has a body in the current super-class, but it can be overridden by a sub-class.
- 'Private' – specifies that one or more declared programming element are accessible only from within the class.

- ‘Property’ is a special keyword inside Visual Basic used to give a controlled interface for the internal data type inside the class.
- ‘Protected’ – specifies that one or more declared programming elements are accessible only from within the super-class or from a sub-class.
- ‘Public’ – specifies that one or more declared programming elements are accessible inside and outside the class.

Applied Encapsulation

In the ATP-Bridge software, all variables defined inside a class are not accessible outside the class, except for certain exceptions. Figure 4.4 shows a generic class diagram indicating the different properties and methods used.

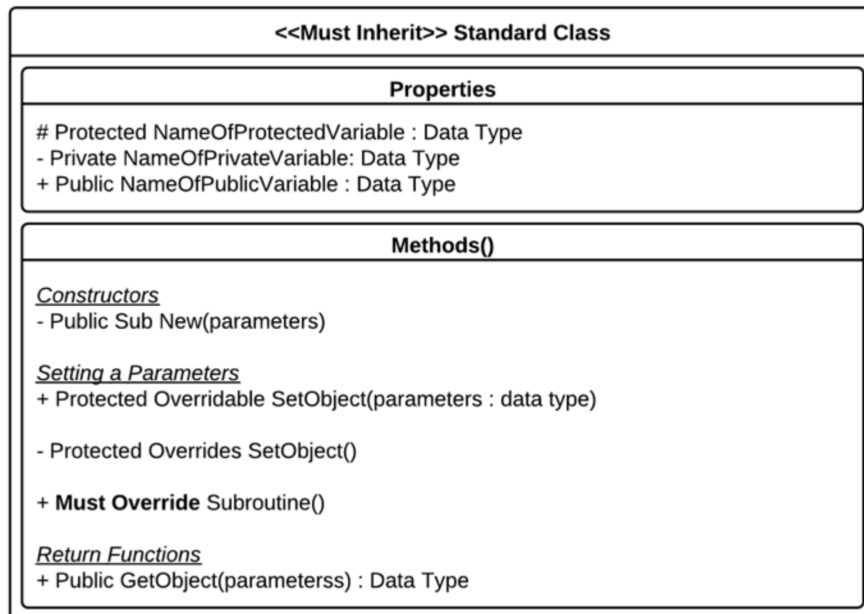

Figure 4.4: Typical Class Notation and Layout

The naming convention for all ‘private’ or ‘protected’ variables that are hidden from outside a class but accessible inside a class include the prefix ‘*My*’ to designate its scope, such as *MyLocations*. This naming convention provides a clear designation of the scope of the class global variables, the variables used in the parameter list, and the local variables used inside methods. As a result, no public access to class variables is possible aside from defined interfaces that allow control to the developer regarding how they are used. This requirement adheres to the principle of encapsulation. Exceptions to this rule include system variables from the Visual Studio environment and defined constant variables that do not change at runtime.

All subroutines that strictly populate the variables begin with the prefix ‘*Set’*, an example being the *SetLocation(…)* subroutine. The subroutine parameter list ensures that data passed into the class are the correct data type, allowing no implicit conversion in the software. Once the correct parameters are passed, the subroutine can check if they are within the correct range before storage.

Real, physical variables that define actual objects have not only magnitude but units of reference as well. It is therefore necessary that both the magnitude and unit are stored as a pair to define one physical variable. Because magnitude is a ‘double’ data type and unit is a ‘string’ data type, one way to create an attached pair using one data type is through a two-dimensional array of the ‘object’ data type. An example of this is shown in Figure 4.5, with the second column being the ‘double’ data type and the third column being the ‘string’ data type.

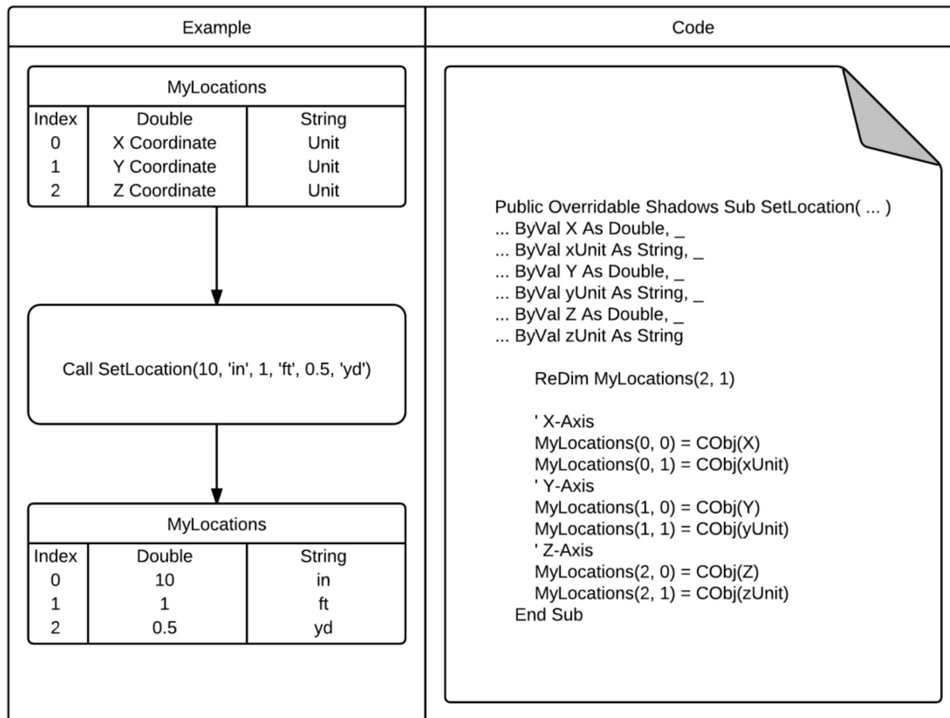

Figure 4.5: Subroutine *SetLocation(…)* Example

In contrast to the 'Set' subroutine, functions that retrieve variables begin only with the '*Get*' prefix. An example is shown in Figure 4.6. The function *GetLocationY(Unit : String)* returns the *Y* coordinates of the component back to the user. To return a value the user must designate a desired unit it wishes to return in, leaving the conversion process to take place "behind the scenes" and hidden from the caller. This process is another example of encapsulation.

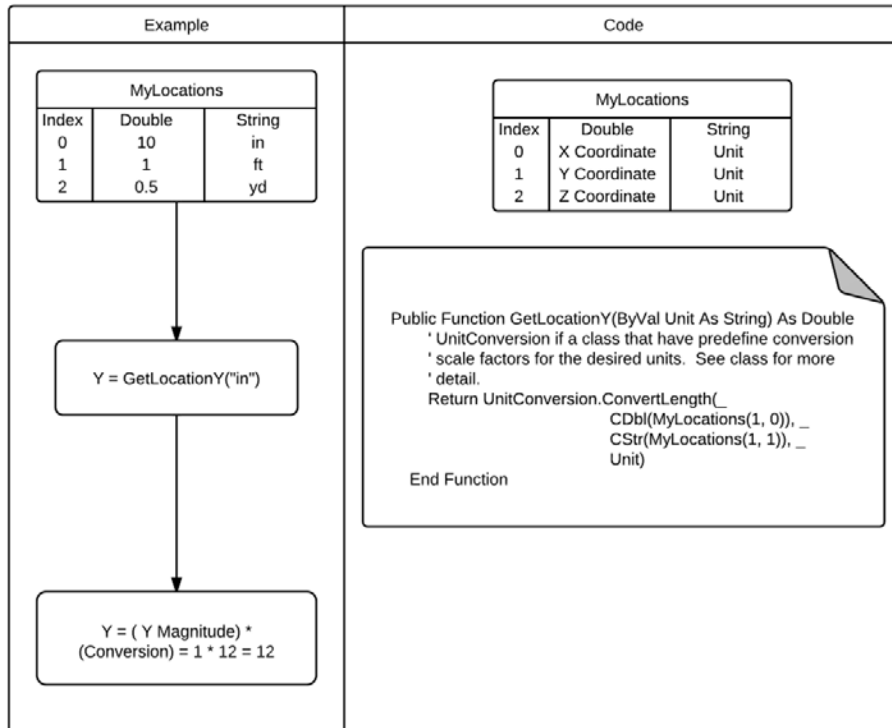

Figure 4.6: Return Function *GetLocationY(…)* Example

The Role of Inheritance

Inheritance is the central concept that allows the ATP-Bridge software to connect the different areas of the program, allowing the graphical user interface and the 3D rendering environment to be created with reliability and robustness. This is all possible because development of the user interface is designed for the ***Structural Component***, ***Load***, and ***Nexus*** super-class, not for any specific bridge component like concrete column or steel plate. Hence, when developing new components in the future, instead of being concerned about how to change the user interface to accommodate the new components,

the developer would inherit the super-class ensuring that it has all the required properties and methods necessary to integrate into the software.

For example, information the graphic engine reads from an array of ***Structural Component*** is shown in Figure 4.7. The graphics engine does not need to be changed to render the ***Column***, the ***Plate***, or any other bridge component. As long as those sub-classes inherit the ***Structural Component***, the graphics will populate the graphics engine based off of the configuration inside the class.

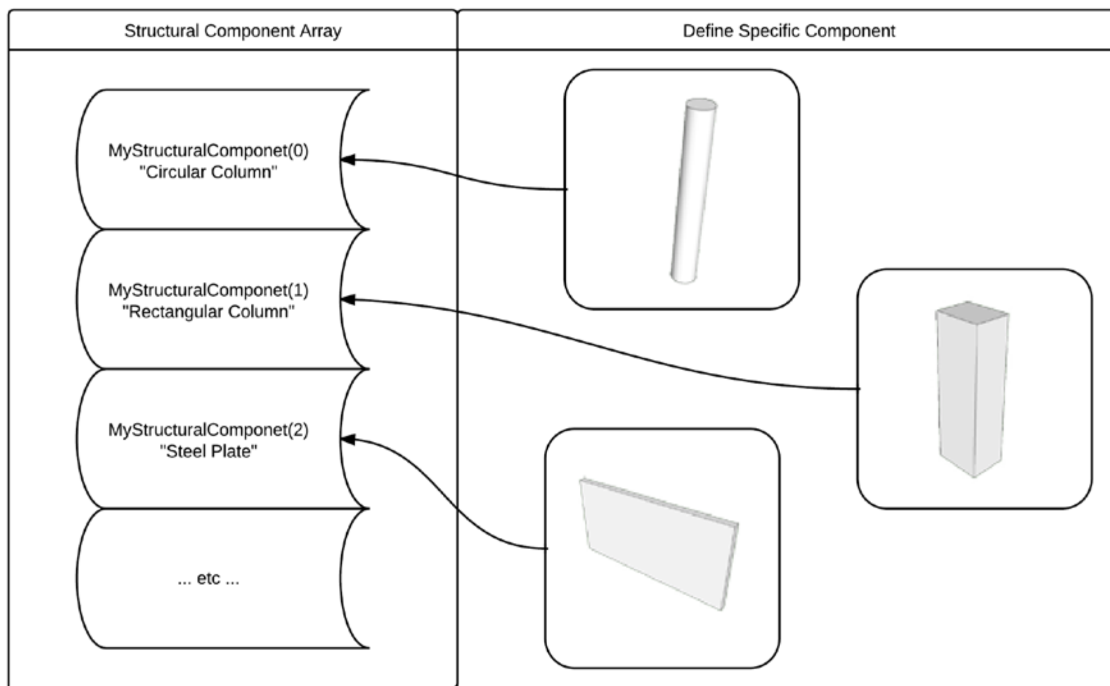

Figure 4.7: Data Storage of Sub-Class

An additional requirement specifically for this software is that all sub-classes should only have data types that are inherited from the super-class. The super-class should provide the generic necessary data types for the sub-class, or, for special

circumstances, the sub-class should use one of the optional data types provided in the super-class. To create a usable variable for the sub-class, the 'Property' keyword should be used as shown in Figure 4.8. Sub-classes can be visualized as a template of the super-class, customizing different data types for the specific components.

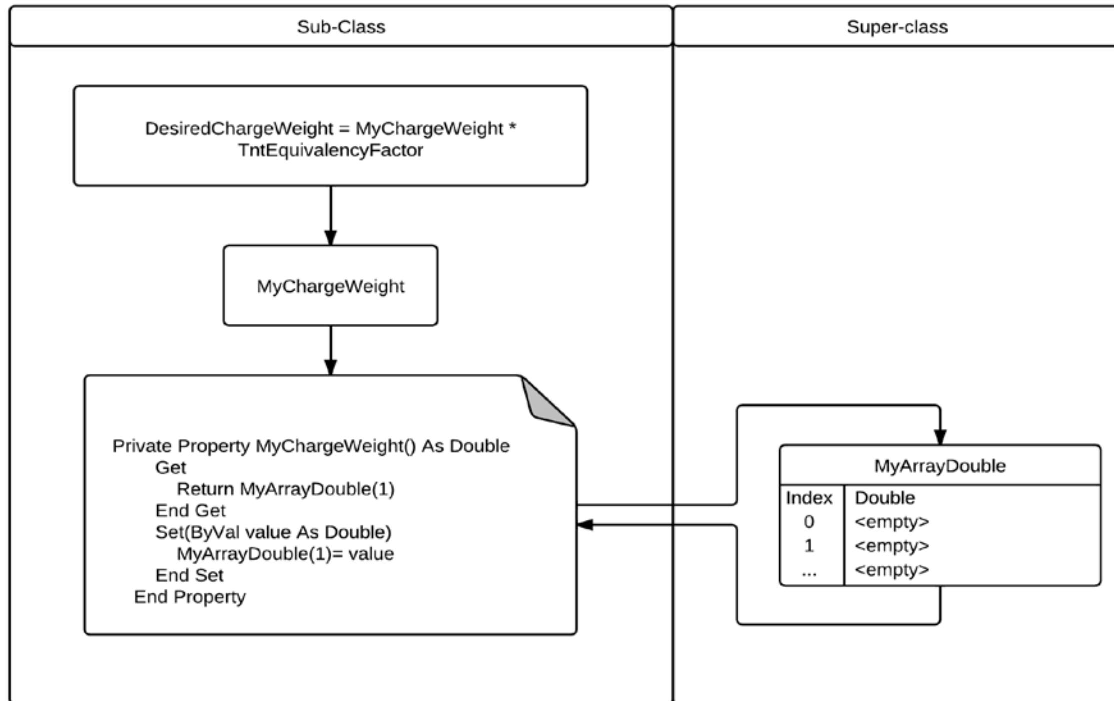

Figure 4.8: Sub-class Data Type Variable

Polymorphism in Practice

With the requirement that a sub-class inherit all the methods, it is necessary to have a mechanism to alter necessary methods to customize it to the sub-class. Polymorphism is used to overcome this obstacle, allowing sub-classes to override the signature of the super-class to create a new body of code. Some methods designated with

the keyword 'MustOverride' are required by the sub-class to override it. Thus, when a method that is overridden is called, it will be redirected to the specific sub-class. Figure 4.9 shows the polymorphism of the *SetMeshes()* subroutine, with the same method in the structural component producing different results in the graphics engine.

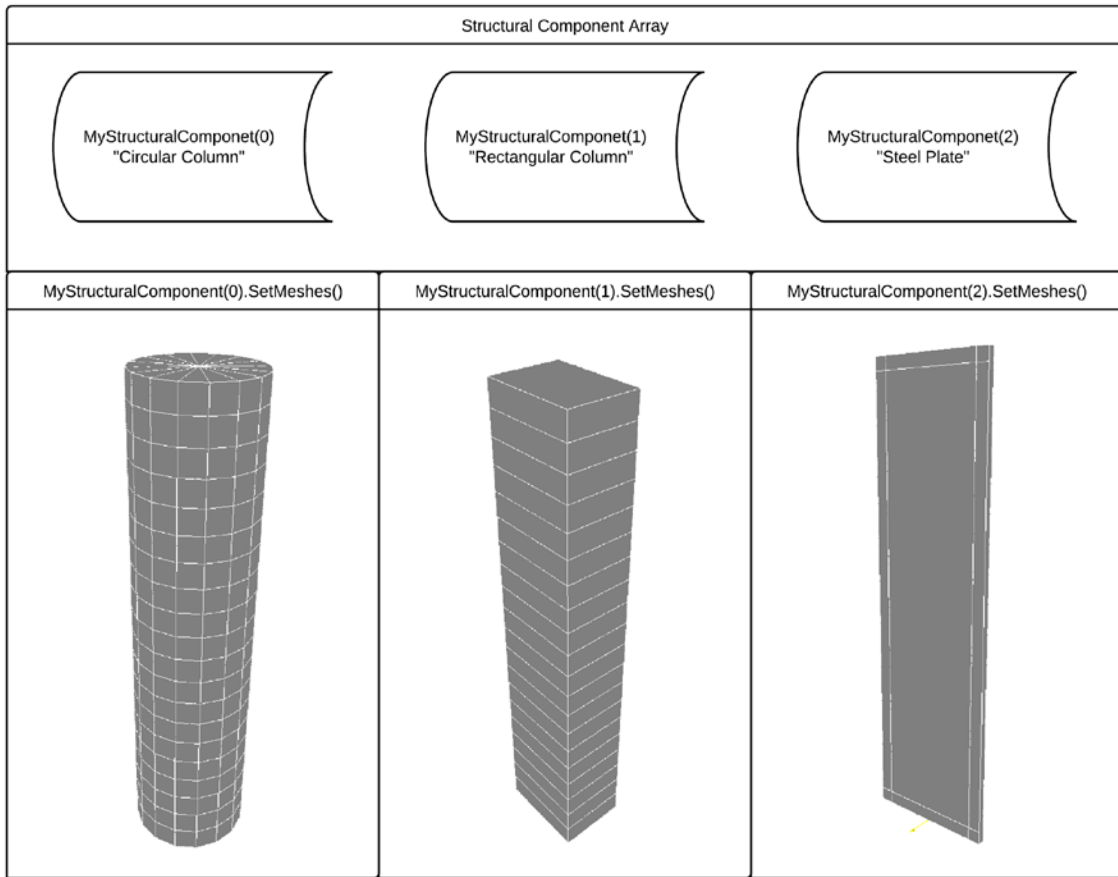

Figure 4.9: *SetMesh()* Subroutine Example

STRUCTURAL COMPONENT CLASS

The ***Structural Component*** class is the super-class for all bridge components and stores all the properties used inside its sub-class. This class can be divided into three distinct phases: physical attributes and methods, analysis, and user interface and graphics engine interactions. The phases are shown in Figure 4.10.

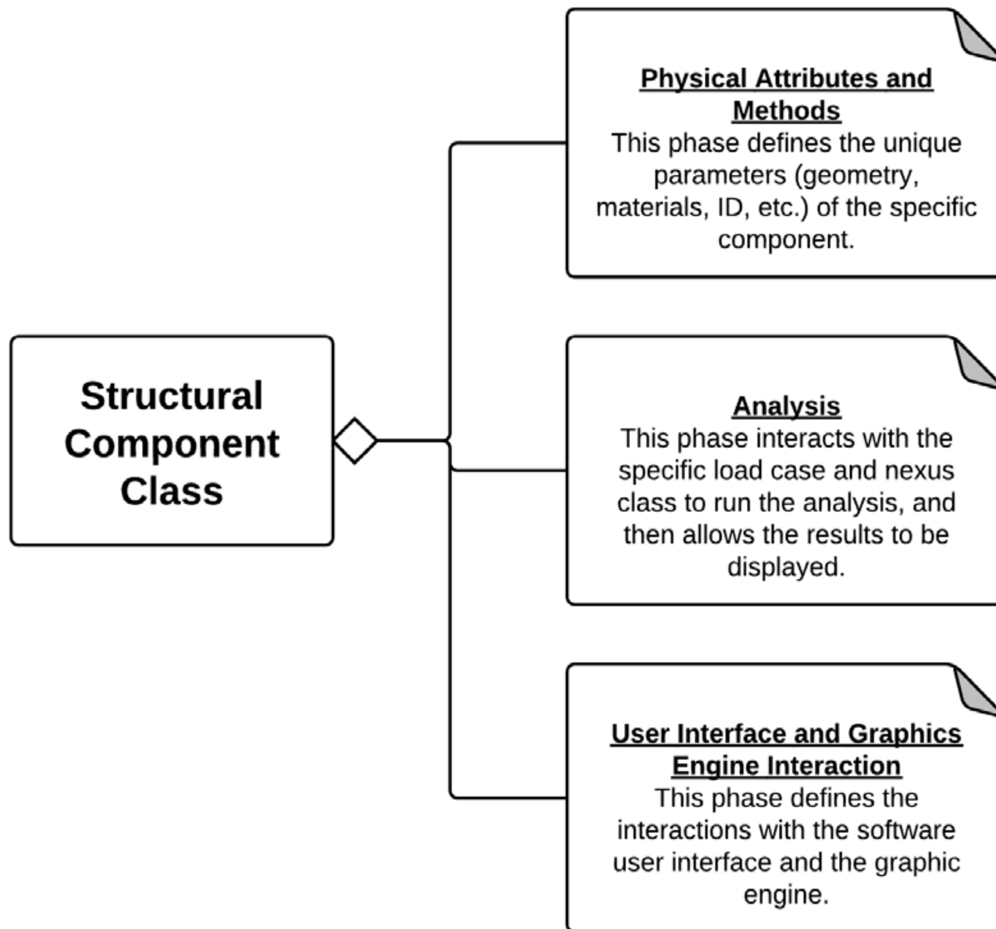

Figure 4.10: ***Structural Component*** Class Phases

Physical Attributes and Methods

The first phase defines the physical properties of a component such as units, geometry, materials, etc. The following class diagram, Figure 4.11, and subsequent text provide a description of the important properties and methods in this phase of the class.

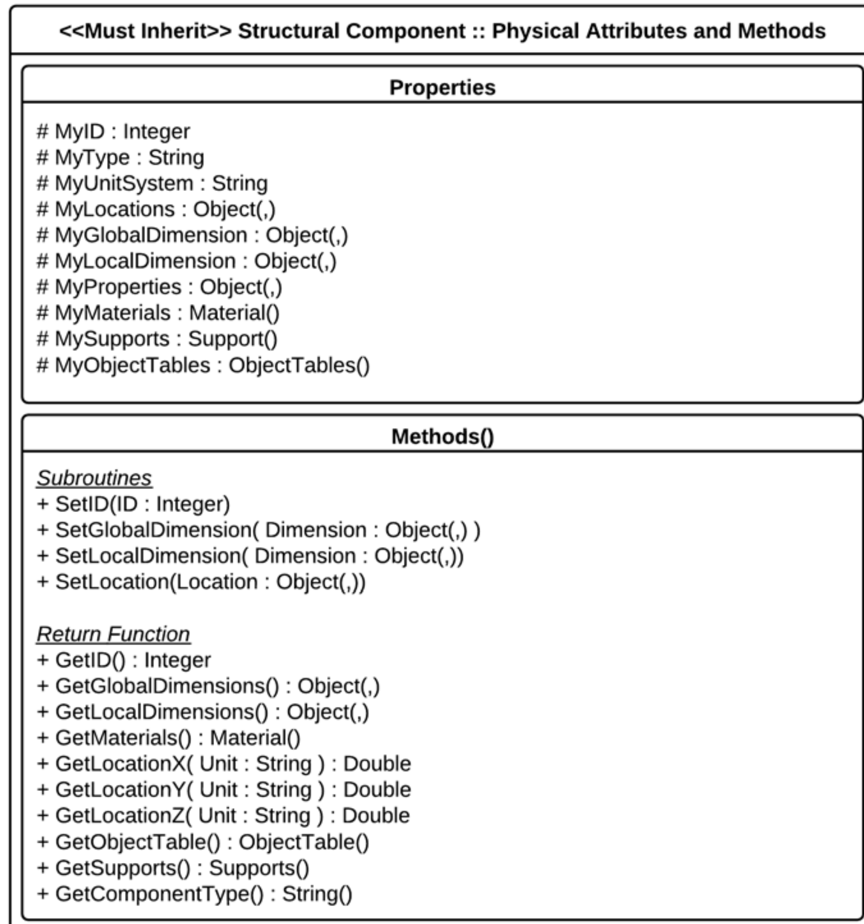

Figure 4.11: ***Structural Component*** Class Physical Attributes and Methods Diagram

The *MyID* variable stores the ID of a bridge component as an integer data type. This variable is unique for each bridge component of the project and is initialized in the

constructor of the class, though the *SetID(…)* subroutine allows the ID to be changed when one or more structural components are removed. The *GetID()* function, conversely, retrieves the ID for display in the user interface.

The *MyUnitSystem* variable stores the unit system of the project as a string data type, with the values being either 'US' or 'Metric'. After the variable is set in the constructor, it cannot be changed for the duration of the object's existence. This is because all the other properties inside the class and the hosted classes are determined relative to the unit system.

The *MyType* variable stores the bridge component type as a string variable, such as 'Circle Column', 'Rectangle Column', 'Steel Plate', etc. This attribute is used by the user interface to determine which bridge component sub-class is to be created. The user interface retrieves this variable with the *GetComponentType()* function.

MyGlobalDimension is a two-dimensional array that stores the global or overall dimensions of each bridge component such as height, width, depth, etc. The generic *SetGlobalDimension(…)* subroutine copies the two-dimensional array passed to it, but it is more common to overload this subroutine and have multiple parameter lists to set the array. Figure 4.12 provides an example of an overloaded subroutine. The *GetGlobalDimension()* function is the generic return method to pass back an entire array, but it is more common for the sub-class to have a specific function to return just one dimension. For example, the *GetHeight(Unit : String)* function returns the total height in the unit given inside the parameter list.

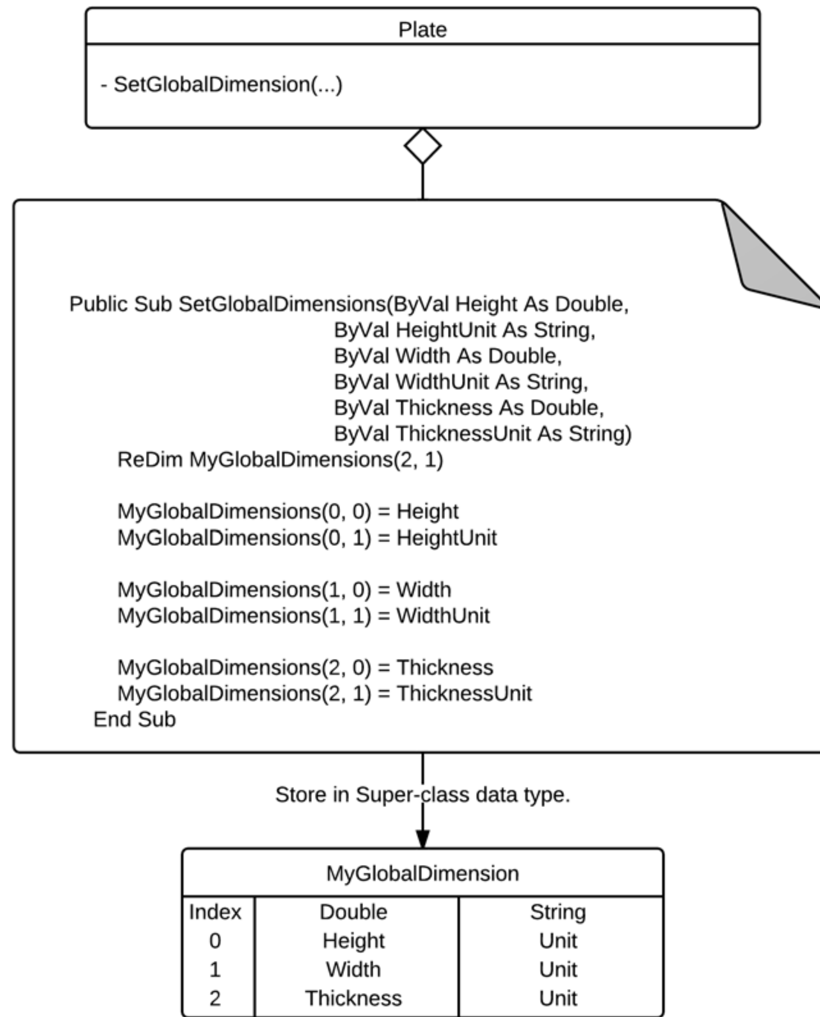

Figure 4.12: Plate Class Overload *SetGlobalDimension(…)* Subroutine

MyLocalDimension performs a similar function to *MyGlobalDimension* but instead defines local parameters that are usually specific to the cross-sectional area information of a bridge component. An example of this is in the ***Column*** sub-class where this array is used to store the concrete cover and transverse rebar size information. The

SetLocalDimension(…) subroutine and the *GetLocalDimension(…)* function have equivalent roles as the global dimension case.

The *MyLocation* object array stores the *x*, *y*, and *z* coordinates of a bridge component with respect to its origin, which is specified during its development for the specific bridge component. This variable is declared during the constructor of the class and can be modified later with the *SetLocation(…)* subroutine. The *GetLocationX(…)*, *GetLocationY(…)*, and *GetLocationZ(…)* are used to retrieve the value of the origin with respect to a desired unit.

MyMaterials is an object array of the ***Material*** class. The ***Material*** class collects information on common material properties such as yield strength, ultimate strength, modulus of elasticity, etc. The ***Material*** class is a super-class and must be inherited by sub-classes such as the ***Concrete*** or ***Steel*** sub-class. The caller retrieves this variable by the *GetMaterials()* function.

MyObjectTables is an object array that holds a two-dimensional matrix of generic 'object' data types. The ***ObjectTables*** class was created to address the need to store data information with attributes similar to a spreadsheet. For the reinforced concrete column, the tables hold the reinforcement layout information such as bar size, location, units, etc. The caller retrieves the array with the *GetObjectTable()* function.

Analysis

The analysis phase of this class handles the interaction with the ***Load*** class, the ***Nexus*** class, and running the analysis. The class diagram shown in Figure 4.13 and

subsequent paragraphs provide a brief explanation of the properties and methods related to this phase.

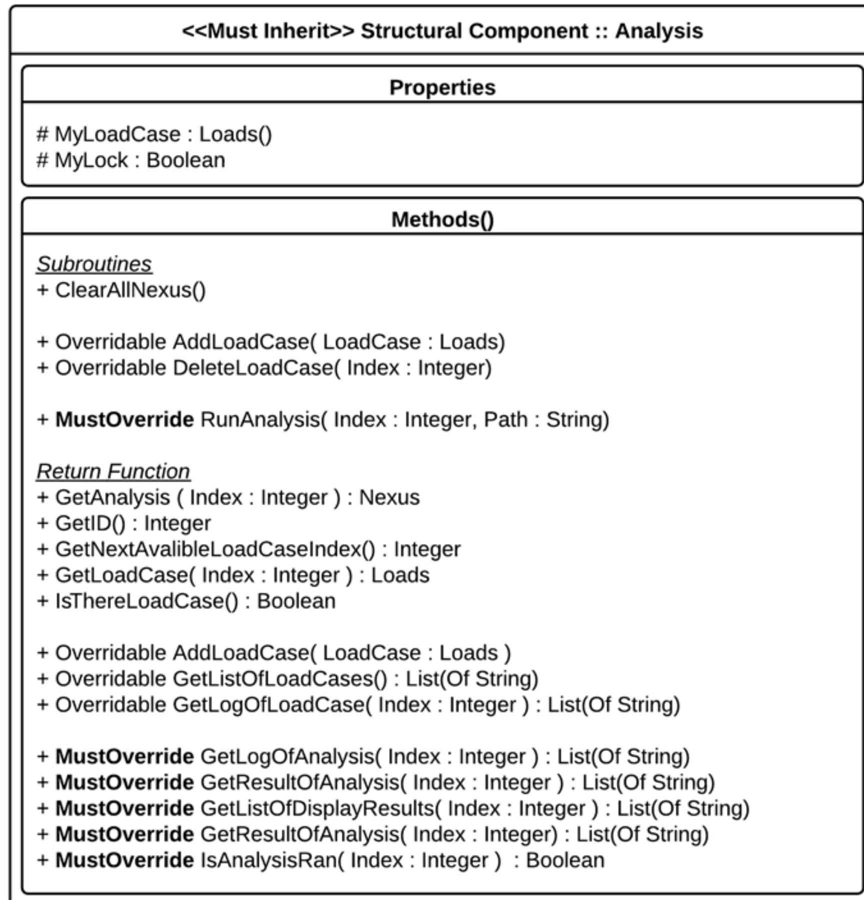

Figure 4.13: Structural Component Class Analysis Diagram

The *MyLoadCase* array stores all the ***Loads*** objects that are linked to a given bridge component. The *AddLoadCase(…)* subroutine is used to add the load case to the end of the array. Similarly, the *DeleteLoadCase(…)* subroutine removes the specified load case designated by the *Index* variable. When the user needs to know if the array is empty, the *IsThereLoadCase()* performs that function. The *GetListOfLoadCase()*and

GetLogOfLoadCase(…) are two functions that return a list of string variables for the user interface to communicate with the user the available load cases.

The *MyLock* variable is used to determine if the properties of the bridge component are locked from changes. Initially, when the bridge component is created, this logical variable is 'false'; once a single analysis is completed, the variable is set to 'true', locking the bridge component parameters from being changed. Locking the bridge component prevents the user from changing problem parameters unintentionally. When the user wants to change the properties of the 'locked' bridge component, the user interface will prompt the user. If the user chooses to continue, the variable is 'unlocked' and all the results are cleared with the *ClearAllNexus()* subroutine, disposing all the ***Nexus*** sub-class inside all the load arrays.

The *RunAnalysis(…)* subroutine is an important method that must be overridden by the sub-class. This method triggers the call to create the *Nexus* class inside *MyLoadCase[Index]*. To return this analysis for post-processing, the user interface calls the *GetAnalysis(…)* function. The *GetLogOfAnalysis(…), GetResultOfAnalysis(…), GetListOfAnalysis(…),* and *GetResultsOfAnalysis(…)* are all the remaining analysis functions used to convey information of the analysis back to the user interface.

User Interface and Graphics Engine

This phase of the class interacts with the User Interface and Graphics Engine. Figure 4.14 and the following paragraphs provide a brief explanation of the properties and methods related to this phase.

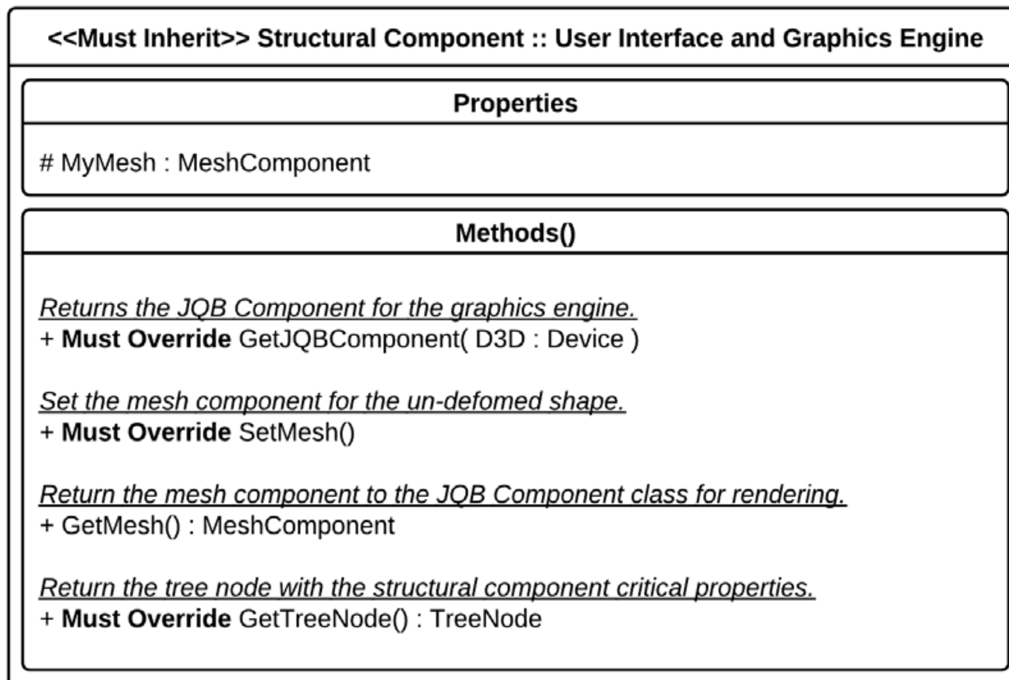

Figure 4.14: Structural Component Class User Interface and Graphics Engine Diagram

MyMesh is an object of the ***Mesh Component*** class, which defines the geometry of the graphics object that is rendered by the graphics engine. The subroutine *SetMeshes()*, which must be overridden, creates a ***Mesh Component*** sub-class with the specific mesh fidelity associated with a specific bridge component. Additional information on the construction of the ***Mesh Component*** class is presented in the next chapter.

The *GetJQBComponent(…)* function creates a ***JQB Component*** class and returns it to the GUI to be loaded into the graphics engine. *T*he ***JQB Component*** class is used by

the graphics engine to render the component. It does this by collecting information from the mesh component and converts it into a graphics format that the graphics engine uses for rendering. A detailed explanation of the ***JQB Component*** is presented in the next chapter.

The *GetTreeNode()* function returns a windows form ***Tree Node*** object to the caller. The ***Tree Node*** class is used by the GUI to present information about the ***Structural Component*** in a tree-view format. The function returns all of the specific properties of the class, including geometry, material, load case, etc. The ***Tree Node*** is a class provided in the Microsoft.NET environment and is further described in the next chapter.

LOAD CLASS

The ***Load*** class is a super-class that holds all the information concerning the loading scenario associated with a given bridge component. This class defines explosive parameters such as physical dimension, location, and type to retrieve the pressure-time history from BEL, an external software using fundamental equations and empirical data to calculate blast loads acting against bridge components (USACE-ERDC, 2004). Each bridge component should have a corresponding ***Load*** sub-class with all the inherited properties, but it should also be able to accommodate any special loading conditions. For example, the ***Column Load*** class inherits the ***Load*** class, but it also takes into account column axial load and deck geometry. These two parameters are stored in *MyExternalLoads* and *MyExternalGeometry*, respectively. The class diagram shown in

Figure 4.15 and subsequent paragraphs provide a brief explanation of the properties and methods in this class.

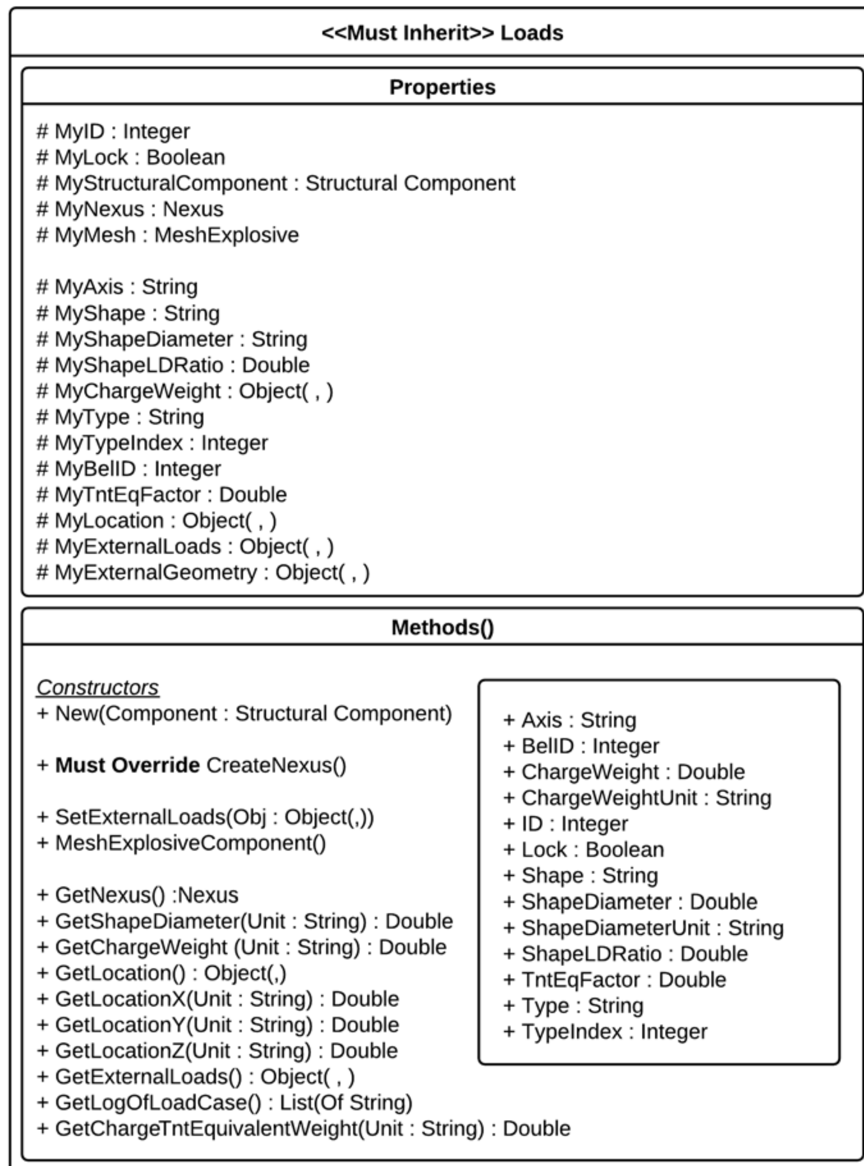

Figure 4.15: Load Class Analysis Diagram

The *MyID* variable provides a unique identifier for the ***Load*** class relative to the bridge component, similarly to the ID explained earlier when describing ***Structural Component***. *MyStructuralComponent* is a reference variable allowing the ***Load*** class to call the ***Structural Component*** public methods. The *MyLock* variable is also similar to the variable with the same name in the ***Structural Component*** class, but it is only applicable for the specific load case. Therefore, if one analysis is done on *MyLoadCase*[1] but not on *MyLoadCase*[2], then *MyLoadCase*[1] and the bridge component are locked for this case but not *MyLoadCase*[2].

MyNexus is an object of the **Nexus** class connecting the ***Structural Component*** with the ***Loads*** class. Unlike the ***Structural Component*** with multiple ***Loads***, there is only one ***Nexus*** per ***Loads*** object. Therefore, when the ***Structural Component*** → *RunAnalysis()* subroutine is triggered, it calls the ***Loads*** → *CreateNexus()* subroutine to create a new ***Nexus*** class for the analysis. If the user decides to change a parameter of the ***Loads*** class, the *ClearNexus()* subroutine is called to delete the ***Nexus*** class.

The *MyAxis* string variable determines the axis along which the explosive is allowed to vary; the possible choices are locking it on the '*x*-axis' or '*y*-axis' or allowing complete movement in space with an '*xy*-axis' value. With the axis known, the *x*, *y*, and *z* coordinate can be stored in *MyLocations* array. With the location of the explosive and the location of the bridge component specified, it is possible to determine the height of the explosive above the ground surface, the standoff distance of the explosive from the bridge component, and the blast face side of the bridge component.

MyBelID, *MyType*, and *MyTypeID* are all variables used to determine the type of explosive. The user chooses the explosive type from a list of available options, and the results are stored. *MyType* and *MyTypeID*, respectively, are string descriptions of the explosive type and index location inside that list. After the explosive type is selected, a corresponding BEL ID is found from a list and stored in the *MyBelID* variable. The BEL ID is an input for BEL that is used to correctly characterize the blast load.

MyShape, *MyShapeDiameter*, *MyShapeLDRatio*, *MyChargeWeight* and *MyTntEqFactor* are variables that define the overall geometry and magnitude of the explosive. *MyShape* defines the explosive as either a cylindrical or spherical charge. *MyShapeDiameter* defines the gross diameter of the explosive. *MyShapeLDRatio* is the length-to-depth ratio used to define the geometry of a cylindrical charge. *MyChargeWeight* defines the charge weight specific to the explosive type. With those parameters, BEL is used to generate pressure-time histories for analysis. *MyTntEqFactor* is an additional variable used in some analyses in place of BEL to convert the explosive specified by the user to an equivalent weight of TNT.

With all the explosive parameters set, the subroutine *MeshExplosiveComponent()* is called to create *MyMesh*, an object of the ***Mesh Explosive*** class. ***MeshExplosive*** is similar to the ***MeshComponent*** class in the ***Structural Component*** class, but it can only render BEL spherical and cylindrical charges. The *GetMesh()* function likewise returns the *MyMesh* variable to the JQB Component for rendering.

NEXUS CLASS

The generic ***Nexus*** class has only a few objects and data types that must be inherited because the purpose of the ***Nexus*** class is to tailor the strict format of the ***Structural Component*** and ***Load*** classes to an external solver. The solver can vary from case to case depending upon the component being analyzed.

The constructor requires two reference variables, ***Structural Component*** and ***Load***, for initialization, and is stored in *MyStructuralComponent* and *MyLoad*. The ***Nexus*** class takes the two objects and tailors their properties to parameters that are specific to the solver used for analysis. Figure 4.16 shows a class diagram of the ***Nexus*** class.

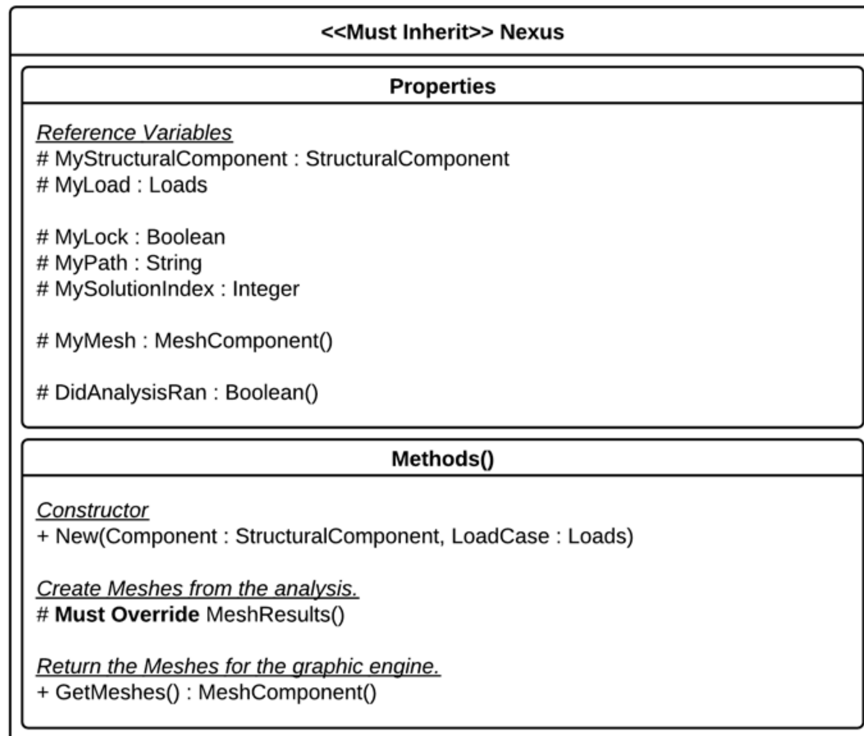

Figure 4.16: ***Nexus*** Class Analysis Diagram

For example, the analysis for the ***Column Nexus*** runs through two separate solvers, one for local damage and the other for global damage. First, the ***Column Nexus*** class needs to determine if the column failed locally. Thus, a set of parameters is passed into the reinforced concrete column local damage solver. Once the local damage analysis is completed, three possible results are returned to the nexus: breaching, spalling, or no damage. If there is no damage or if the spall length is less than the depth of the column, the analysis continues by considering global response and calling the global damage solver. If the column is breached or the damage is too severe, the column is considered compromised without the need for carrying out a global response analysis.

Once the analysis is completed, the results are meshed and displayed in the graphics engine. The sub-class must override the *MeshResults()* subroutine, which meshes the post-processing results to the bridge component and stores it in the *MyMesh* array. The ***Mesh Component*** in the array is a sub-class of the class, inheriting all its properties while tailoring the rendering specifically to the results. The *GetMeshes()* function returns the array to the graphics engine for rendering.

Some additional properties and methods are shared with all of the ***Nexus*** sub-class. *MyPath* is a string variable that stores the external location to output any results from the solvers. *MyLock* is used similarly to the previous two classes to determine if a previous analysis exists. Finally, the *DidAnalysisRan()* function is used to determine if an analysis ran to completion.

ADDITIONAL DATA TYPES

For the previous three classes, there is a collection of data types that are declared in the super-class (Figure 4.17). These data types are available to be used by the sub-class for specific parameters that are not currently addressed by the super-class.

Single Data Type

- MyCharacter : Char
- MyString : String
- MyInteger : Integer
- MySingle : Single
- MyDouble : Double
- MyBoolean : Boolean
- MyObject : Object

Array Data Type

- MyArrayCharacter : Char()
- MyArrayString : String()
- MyArrayInteger : Integer()
- MyArraySingle : Single()
- MyArrayDouble : Double()
- MyArrayBoolean : Boolean()
- MyArrayObject : Object()

Two-Dimensional Data Type

- MyAdditionalCharacter : Char(,)
- MyAdditionalString : String(,)
- MyAdditionalInteger : Integer(,)
- MyAdditionalSingle : Single(,)
- MyAdditionalDouble : Double(,)
- MyAdditionalBoolean : Boolean(,)
- MyAdditionalObject : Object(,)

List of Data Type

- MyListOfCharacter : List(of Char) ()
- MyListOfString : List(of String) ()
- MyListOfInteger : List(of Integer) ()
- MyListOfSingle : List(of Single) ()
- MyListOfDouble : List(of Double) ()
- MyListOfBoolean : List(of Boolean) ()
- MyListOfObject : List(of Object) ()

Figure 4.17: Optional Data Types

The desired way to utilize these data types is through the 'Property' keyword, as shown in Figure 4.18. Effectively, the keyword allows the sub-class to consider the data type as defined inside the sub-class even though it is actually stored in a super-class variable.

```
ReadOnly Property StrengthInputs() As Single()
  Get
    Dim Strength(MyListOfSingle(1).Count - 1) As Single

    For i As Integer = 0 To UBound(Strength)
      Strength(i) = MyListOfSingle(1).Item(i)
    Next

    Return Strength
  End Get
End Property
```

Figure 4.18: Overriding Optional Data Type *StrengthInputs()* Example

SUMMARY

This chapter includes a description of the fundamental data structures used in the ATP-Bridge software, a brief overview of the programming style used to accomplish the object-oriented programming structure, and a thorough explanation of all the important properties and methods inside the super-class. The next chapter details the graphical user interface, gives a brief summary of DirectX, and explains the graphics engine.

Chapter 5: Graphical User Interface and Graphics Engine

If data structures are the heart of ATP-Bridge, the graphical user interface (GUI) is the body. The GUI is designed to interact with the user through simple and intuitive forms that convert user input into required variables for the solvers. The first part of this chapter provides a description of the overall GUI design and structure, the ***Navigation*** control, and the ***3D Rendering*** viewer. The second part of this chapter includes a detailed description of the graphics engine, which is broken into three sections: *Direct3D Graphical Environment*, *Graphics Engine Components*, and the *Graphics Engine Cycles*.

The *Direct3D Graphical Environment* section provides a high-level explanation on Direct3D fundamental concepts regarding how rendering is performed. For more information on Direct3D, see books by Miller (2004) and Luna (2008). The next section, *Graphics Engine Component*, builds on this information and presents a new set of classes that are common to the structural engineering community—nodes, triangles, and quadrilateral elements—and hides all the primitive features of Direct3D from future developers. The last section, *Graphics Engine Cycle*, explains how the graphics engine component operates to render the 3D environment.

Throughout this chapter, the 'JQB' prefix is used as a special designation for many of the classes. The prefix is used to distinguish between graphics engine classes and non-graphics engine classes.

GRAPHICAL USER INTERFACE OVERALL DESIGN

The GUI environment starts with the main form (Figure 5.1), which is divided into three different segments. The first segment is the ribbon area that holds the ***Menu Item*** control and the ***Quick Icon*** control. The ***Menu Item*** control has a series of standard menu items that is typical throughout all Windows software, establishing a familiar option to navigate through the software. The ***Quick Icon*** control contains a subset of the buttons included in the ***Menu Item*** control, having small icons that provide quick shortcuts for users. The typical items are the *Define Geometry*, *Define Loads*, and *Run Analysis* buttons.

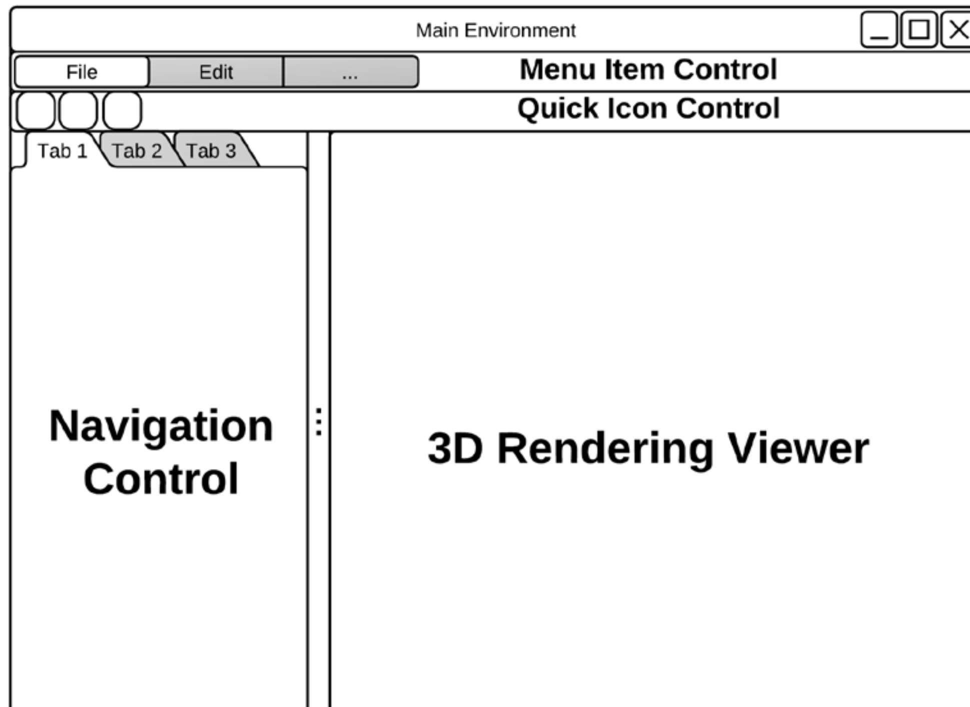

Figure 5.1: Graphical User Interface Schematic

The second segment is the ***Navigation*** control. This control interacts with all the bridge components inside the project. There are two sub-controls inside the ***Navigation*** control. The first is the ***Tree-View Summary*** control, which displays all the different bridge components inside the project and their respective load cases. The second is the ***Viewer Setting*** control, which gives a user the ability to switch the display in the ***3D Rendering*** viewer to show the explosive, component geometry, local damage, and displacements.

The last section is the ***3D Rendering*** viewer, the central visual component used to communicate the scenario with the user. The viewer is used to present a 3D representation of the bridge component being analyzed and the explosive. It is designed to give the user information not only on the scale of the component but also the position of the explosive in space, the extent of damage, and the level of deflection.

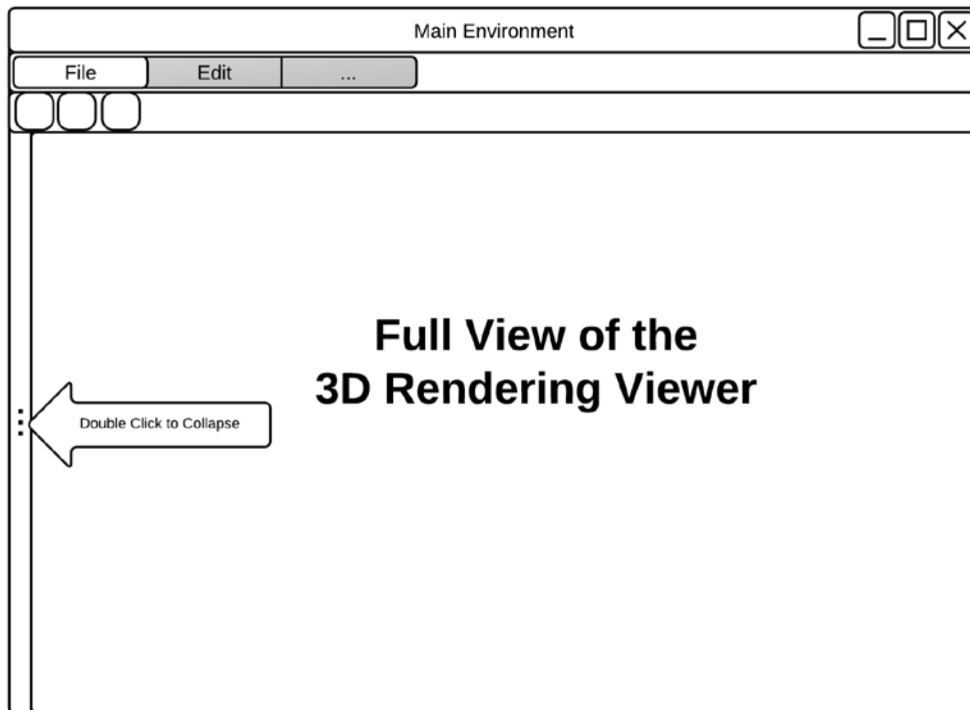

Figure 5.2: Graphical User Interface Collapse Navigation Control

A concern in designing the GUI is presenting information concisely without cluttering the screen with too many options. That is why the GUI is designed with the ***3D Rendering*** viewer having the largest screen area, and it is used to render as much information about the scenario as possible. Because the screen size varies across different computers, the GUI is designed to allow the ***Navigation*** control to collapse to the sides, thereby increasing the screen space. Figure 5.2 provides a schematic of the collapse ***Navigation*** control.

NAVIGATIONAL CONTROL

The ***Navigation*** control main component is the tab page container, allowing the two sub-controls (***Tree-View Summary*** and ***Viewer Setting*** control) to be added into the container. If future controls are required, the ***Navigation*** control is able to accommodate future expansion.

Tree-View Summary Control

The ***Tree-View Summary*** control is designed to present all the critical information of the bridge components in a project. One of the GUI design goals was to keep all the information in front of the user, allowing them to verify that information is correctly entered.

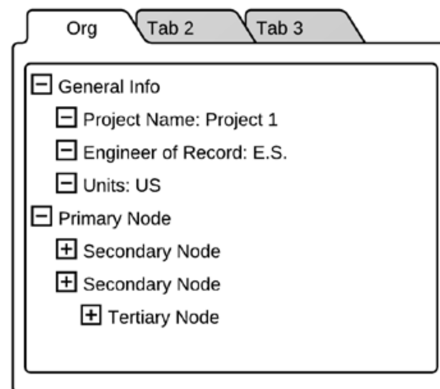

Figure 5.3: Tree-View Control General Information Section

The first primary node in the control presents the generic information provided when a project is created as shown in Figure 5.3. The information includes the project name, engineer of record, and unit system. Right-clicking on the nodes brings up a form

that prompts the user to change the project name and engineer of record, but the unit system is not allowed to be changed after a project is created.

All primary nodes thereafter hold bridge component information. Although it is not desirable to present all the bridge component information, certain information like global dimensions and material properties are desirable. Figure 5.4 presents a schematic of some bridge component tree-nodes.

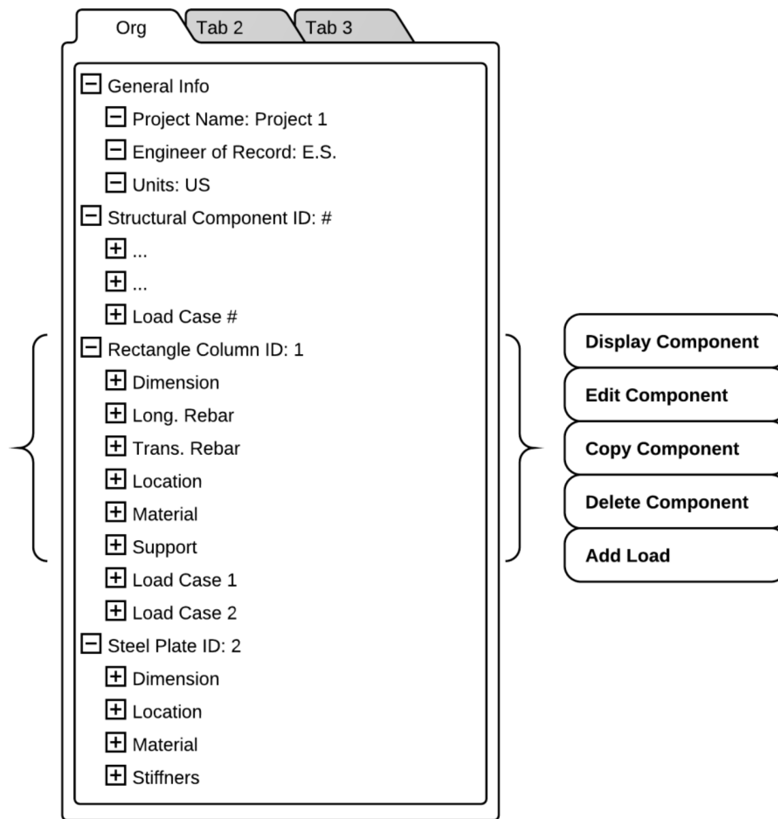

Figure 5.4: Tree-View Control Structural Component Section

Each bridge component tree-node has a respective right click menu list that gives the ability to edit, copy, or delete the component. The menu also gives the user the ability to add load cases to the specific component.

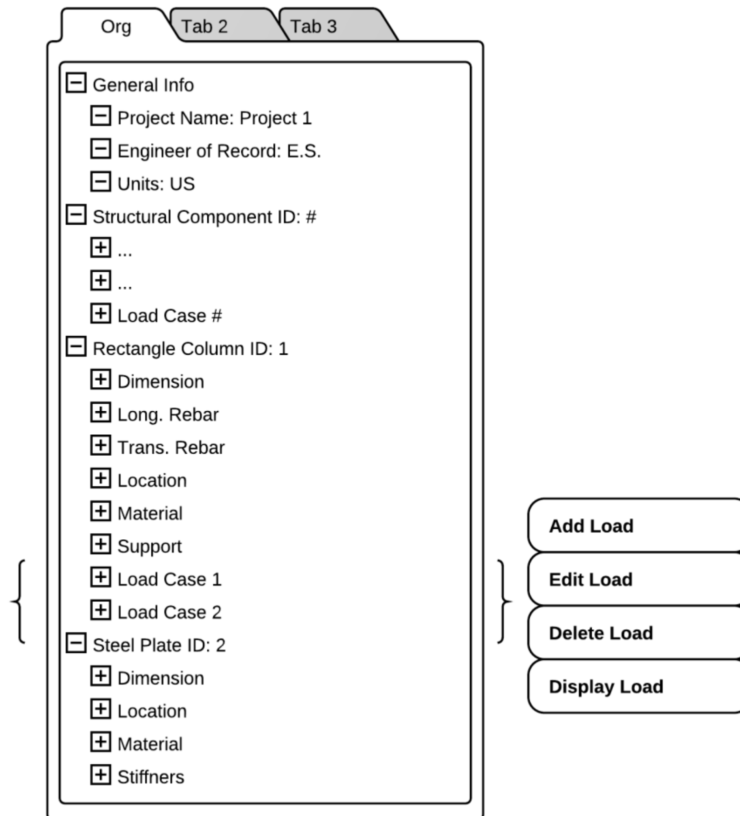

Figure 5.5: Tree-View Control Load Case Section

If a bridge component has a load case, the load case will be displayed in a secondary tree-node for that bridge component. The load cases, like the bridge components, need to contain essential information, such as charge weight and location. The menu options for the loads are ‘add’, ‘edit’, and ‘delete’.

Viewer Setting Control

The ***Viewer Setting*** control is used to communicate with the graphics viewer regarding the information that is to be displayed. The control has options to switch among bridge components, load cases, and results as shown in Figure 5.6 below.

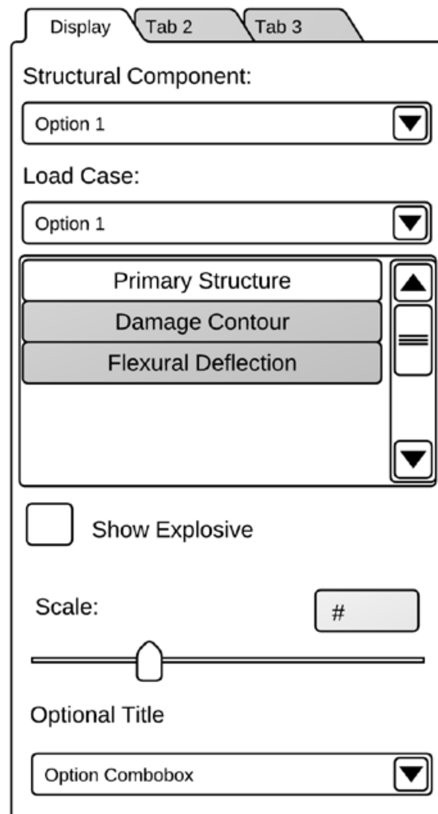

Figure 5.6: Viewer Setting Control

Additional settings specific to the bridge components are displayed in the optional panel sections below the 'Show Explosive' checkbox. Examples of possible features in the optional panel are the scale factor, the plane color, and the damage color.

DIRECT3D GRAPHICAL ENVIRONMENT

The graphical environment used inside ATP-Bridge for 3D rendering is Direct3D. Direct3D is an application programming interface (API) giving developer's access to the user's graphics hardware (Luna, 2008). Direct3D provides a standard syntax for 3D rendering used across different hardware, ensuring continuity exists as long as the hardware is Direct3D compatible. Below is a brief introduction on vertices, primitive types, vertex buffer, and index buffer. These key concepts are needed to understand how the graphics engine is created.

Throughout the remaining part of the chapter, there are references to *vectors* and *matrices*. Vectors and matrices are components used in 3D mathematics to render 3D environments into a 2D monitor. Vectors are used to represent either points in space or direction. Matrices are used to manipulate vectors, allowing operations such as translation, rotation, and scaling. A concise explanation on 3D mathematics can be found in Appendix A, while the books by Luna (2008) or Thorn (2005) can be consulted for additional information.

Vertices and Primitives Types

Vertices in Direct3D are vectors describing a point in 3D space. Vertices hold additional attributes beyond position, such as lighting, normal component, textures, etc. For the ATP-Bridge graphics engine, the only additional attribute for vertices is color. Vertices are the main building block used to render lines and planes.

All graphics drawn in Direct3D are derived from either a point, line, or a triangle *primitive*. *Primitives* in Direct3D are different types of rendering options; the basic list includes *Point List*, *Line List*, *Line Strip*, *Triangle List*, *Triangle Strip*, and *Triangle Fan*. Primitives are rendered with colors set at the vertices. Therefore, if the colors of the vertices are different along the primitive, it will be interpolated. The two primitive types used in the ***JQB Graphics Engine*** are the *Line List* and the *Triangle List*, which are presented below. More information on the other types can be found in the book by Luna (2008).

Line List is the primitive type that defines an individual line, which simply consists of two vertices for each line. Figure 5.7 provides a visual diagram.

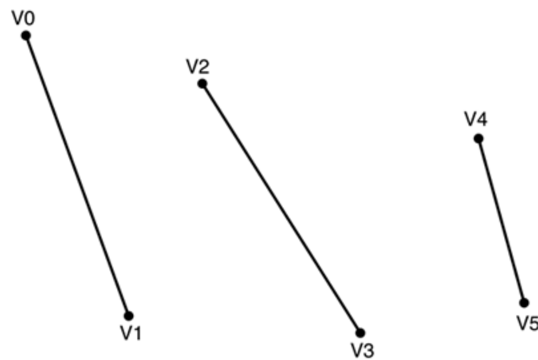

Figure 5.7: *Line List* Primitive Type (Miller, 2004)

Triangle List is the primitive type that requires three vertices to render one triangle element. Figure 5.8 provides a visual diagram.

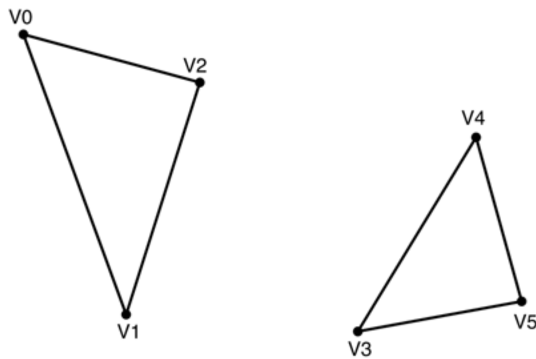

Figure 5.8: *Triangle List* Primitive Type (Miller, 2004)

Vertex Buffers and Index Buffers

A *vertex buffer* is used to load an array of vertices into the graphics device. Using a vertex buffer minimizes the time to render because it reads the data straight from the graphics device rather than from the system memory (Miller, 2004).

A drawback of using only the vertex buffer is that it requires declaring multiple vertices at the same point to draw primitives that share common vertices (Figure 5.9). Aside from requiring multiple vertices at the same location, it also requires multiple processes of the vertices to load into the graphics card. One way to improve efficiency is by using indices and an index buffer.

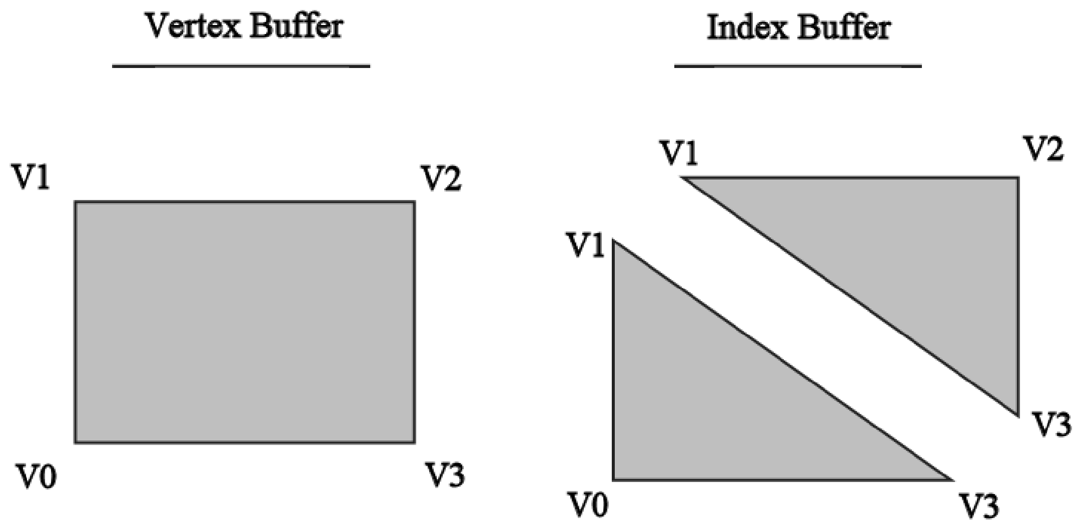

Figure 5.9: Vertex Buffer and Index Buffer (Thorn, 2005)

Indices, unlike vertices, are simply integer values that are either *short* or *long* data types. Indices are loaded into the index buffer, similarly to how vertices are loaded into the vertex buffer. Each index has a value that corresponds with a vertex location inside its respective vertex buffer. With the index buffer, the graphics engine can access the vertices more than once to declare a primitive. The index buffer always exists with a vertex buffer; without it, there is nothing to reference.

GRAPHICS ENGINE COMPONENTS

The ***3D Rendering*** viewer links with ATP-Bridge graphics engine, used to render bridge components and explosives. It is also used in post-processing results. The graphics engine utilizes a collection of Direct3D and ATP-Bridge custom classes to facilitate the rendering. The list of custom classes includes ***JQB Element***, ***Mesh***

Component, ***JQB Blackboard***, and ***JQB Component***. These custom classes are described in detail in the following sections.

JQB Elements

The graphics engine rendering of bridge components is designed utilizing common structural analysis elements: ***Node***, ***Triangle***, and ***Quadrilateral***. These elements provide program developers the simplicity of meshing a component by automating the processes of creating vertices, indices, vertex buffers, and index buffers.

Node Element

Node is a class that defines a point in space where other planar elements connect. The class stores the x, y, and z global coordinate space with respect to the origin, shown in Figure 5.10. *MyX*, *MyY*, and *MyZ* are the respective variables that hold the coordinates and are only accessible inside the class. The variables are set through the constructor *New(…)* or through the property keywords *X()*, *Y()*, and *Z()*.

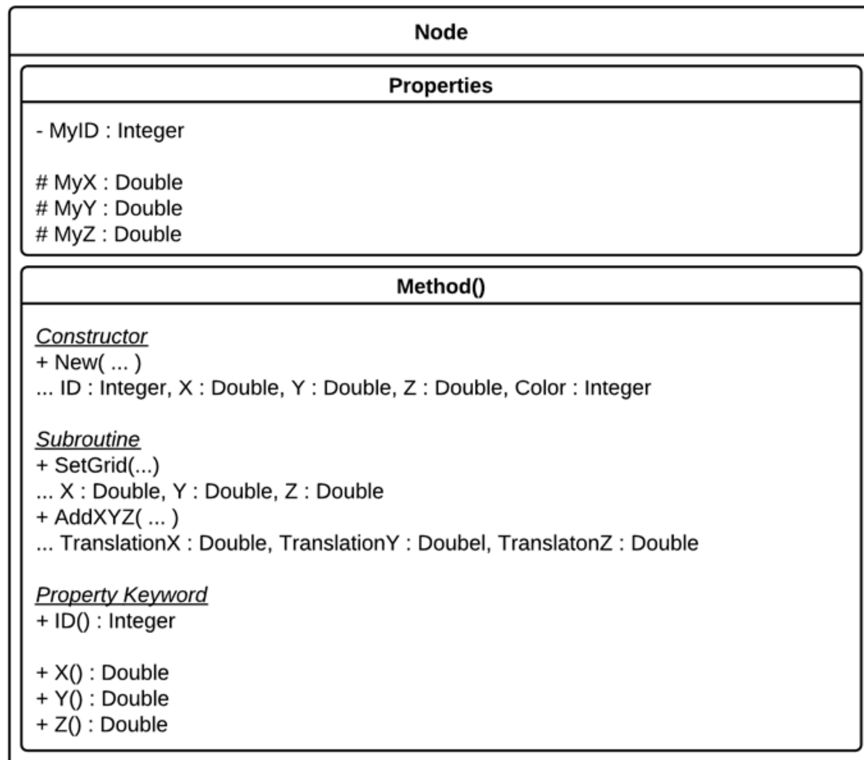

Figure 5.10: ***Node*** Class Diagram

In addition to the coordinates, the ***Node*** class stores the properties *MyID* and *MyColor*. *MyID* is a unique index used by the planar ***JQB Elements*** to identify a node. *MyColor* stores the color index used when rendering elements. Both of the values are set in the constructor and are accessible through the property keyword *ID()* and *Color()*.

Plane Element Superclass

The ***JQB Element*** class is a super-class of a generic plane element (Figure 5.11) that must be inherited by the sub-class. In ATP-Bridge, there are currently two elements

used to draw planes: the ***Triangle*** and ***Quadrilateral***. It is possible, however, to inherit the class so that solid elements can be rendered.

The ***JQB Element*** class has two variables: *MyID* and *MyArrayOfNodeID*. *MyID*, the unique element identifier, is an integer variable that can be changed with the property keyword *ID()*. *MyArrayOfNodeID* is an array of integers used to store the ID of the ***Node***.

As shown in Figure 5.11, there are four functions that are used to assemble the index buffer: *GetTriangleIndices()*, *GetTriangleIndicesCount()*, *GetLineIndices()*, and *GetLineIndicesCount()*. The sub-class must inherit all four functions. The function *GetTriangleIndices()* returns an *short* type array used to render one *Triangle List* primitive. The function *GetLineIndices()* returns a *short* type array used to render *Line List* primitives. The functions *GetTriangleIndicesCount()* and *GetLineIndicesCount()* returns the number of triangle indices and the number of line indices, respectively.

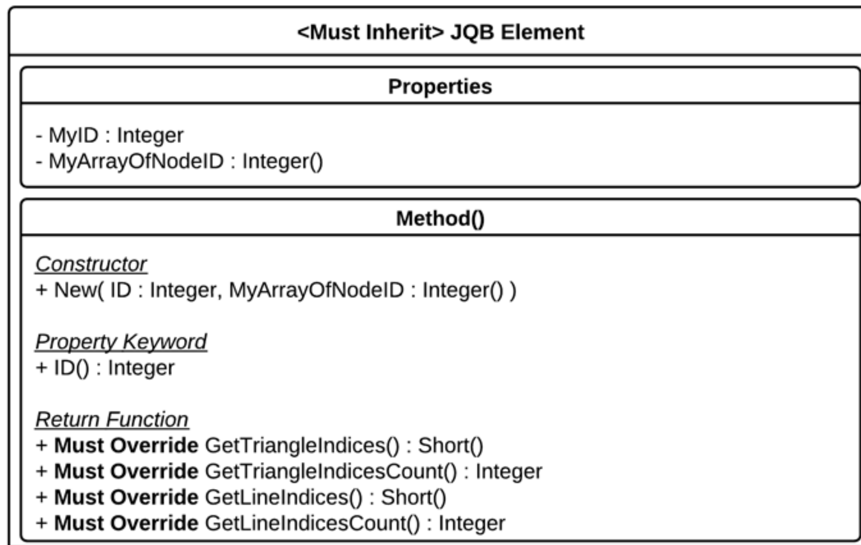

Figure 5.11: JQB Element

Triangle Element

The ***Triangle*** element is a planar element with three nodes. It has the class diagram shown in Figure 5.12. Nodal IDs are set through the constructor and can be changed using the property keywords *NodeAID()*, *NodeBID(…)*, and *NodeCID()*.

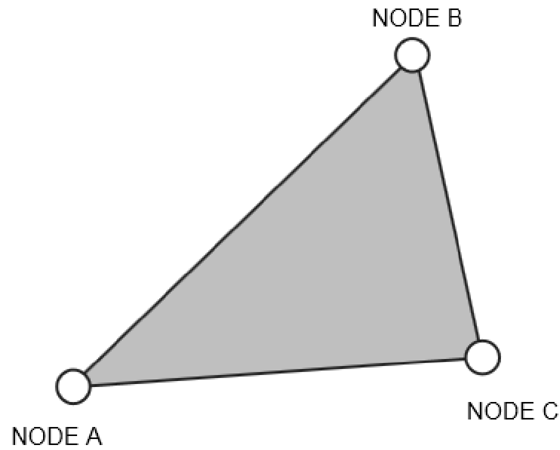

Figure 5.12: Triangle Element

As shown in Figure 5.13, the four functions from ***JQB Element*** class are overridden and return the values shown in Equation (5-1), Equation (5-2), Equation (5-3), and Equation (5-4).

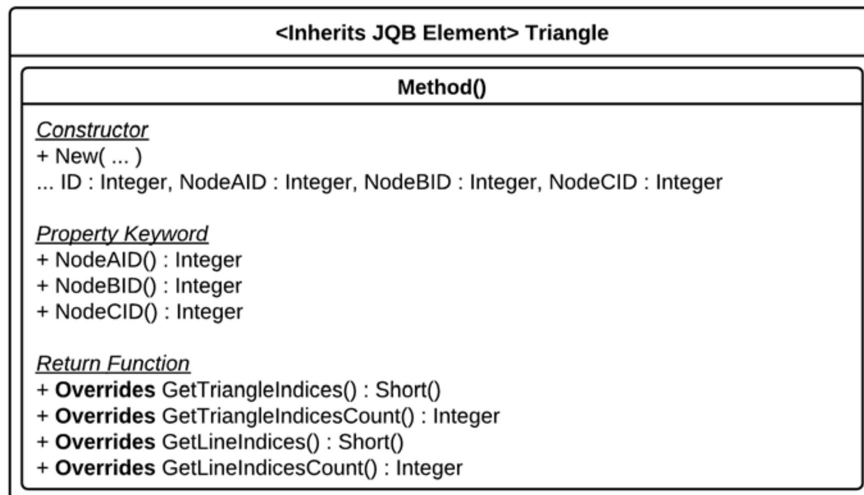

Figure 5.13: ***Triangles*** Element Class Diagram

$$GetTriangleIndices() = \begin{bmatrix} Node\ A\ ID \\ Node\ B\ ID \\ Node\ C\ ID \end{bmatrix} \quad (5\text{-}1)$$

$$GetLineIndices() = \begin{bmatrix} Node\ A\ ID \\ Node\ B\ ID \\ Node\ B\ ID \\ Node\ C\ ID \\ Node\ C\ ID \\ Node\ A\ ID \end{bmatrix} \quad (5\text{-}2)$$

$$GetTriangleIndicesCount() = 3 \quad (5\text{-}3)$$

$$GetLineIndicesCount() = 6 \quad (5\text{-}4)$$

Quadrilateral Element

The ***Quadrilateral*** element is a planar element with four nodes (Figure 5.14). Nodal IDs are set through the constructor and can be accessed using the property keywords *ID()*, *NodeAID()*, *NodeBID()*, *NodeCID()* and *NodeDID()*.

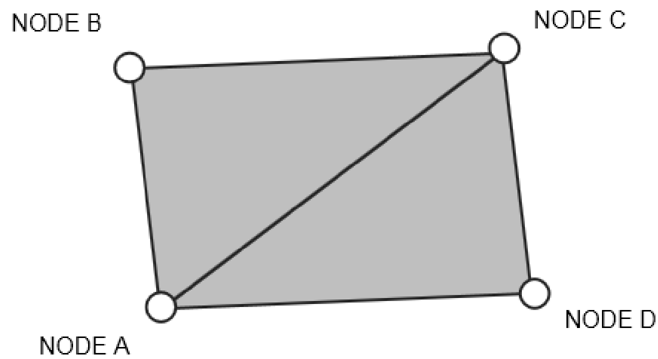

Figure 5.14: Quadrilateral Element

As shown in Figure 5.15, the four functions from ***JQB Element*** class are overridden and return the values shown in Equation (5-5), Equation (5-6), Equation (5-7), and Equation (5-8).

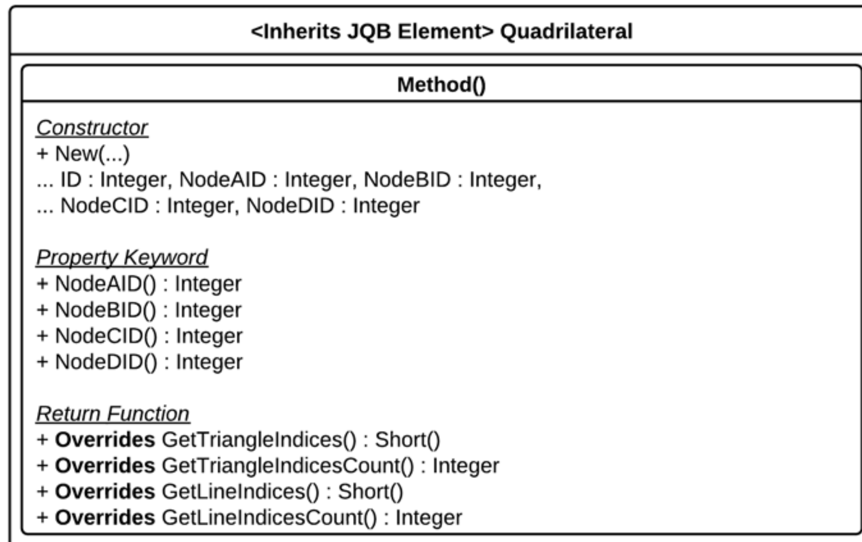

Figure 5.15: ***Quadrilateral*** Element Class Diagram

$$GetTriangleIndices() = \begin{bmatrix} Node\ A\ ID \\ Node\ B\ ID \\ Node\ C\ ID \\ Node\ A\ ID \\ Node\ D\ ID \\ Node\ C\ ID \end{bmatrix} \tag{5-5}$$

$$GetLineIndices() = \begin{bmatrix} Node\ A\ ID \\ Node\ B\ ID \\ Node\ B\ ID \\ Node\ C\ ID \\ Node\ C\ ID \\ Node\ D\ ID \\ Node\ D\ ID \\ Node\ A\ ID \end{bmatrix} \tag{5-6}$$

$$GetTriangleIndicesCount() = 6 \tag{5-7}$$

$$GetLineIndicesCount() = 8 \tag{5-8}$$

Mesh Component

Mesh Component uses a collection of ***Node***, ***Triangle***, and ***Quadrilateral*** elements to assemble a 3D object. ***Mesh Component*** is a super-class and must be inherited by the sub-class for specific 3D objects, such as a cylindrical prism, a rectangular prism, or a rectangular plate. Figure 5.16 presents the class diagram for the ***Mesh Component***.

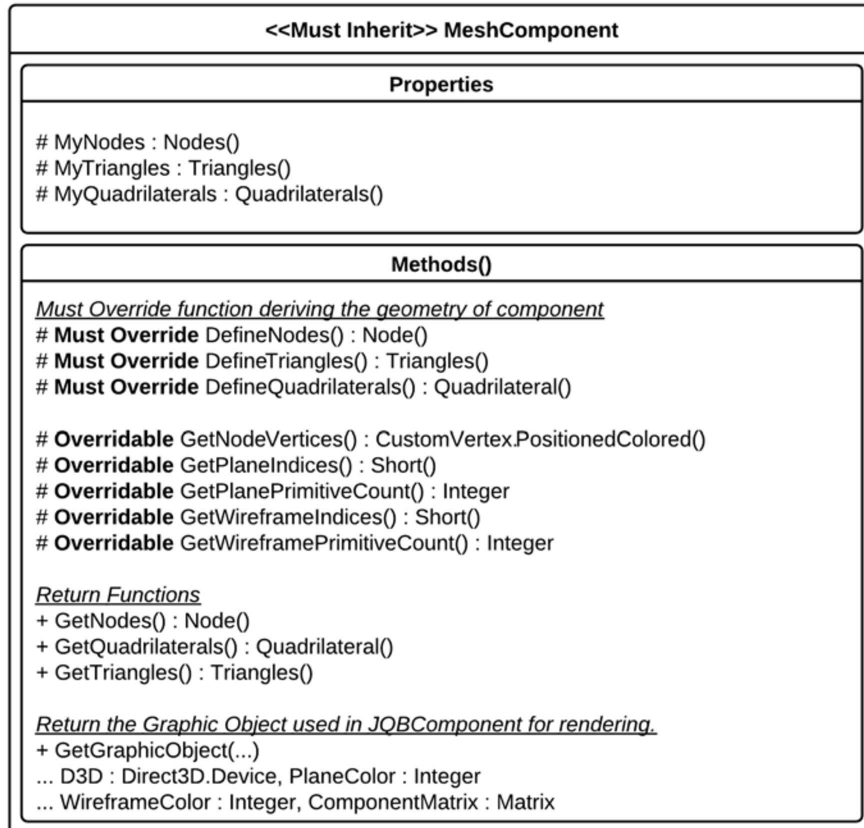

Figure 5.16: ***Mesh Component*** Class Diagram

In the sub-class constructor, the method defines the global dimensions, the mesh density, and any other additional parameters required to create the mesh. The sub-class then calls the functions *DefineNodes()*, *DefineTriangles()*, and *DefineQuadrilaterals()* to create *MyNodes*, *MyTriangles*, and *MyQuadrilaterals*. These three functions must be overridden in the sub-class with a meshing algorithm developed for the specific bridge component being rendered. To further explain these functions, a simple example of a triangle prism is presented, with a schematic shown in Figure 5.17.

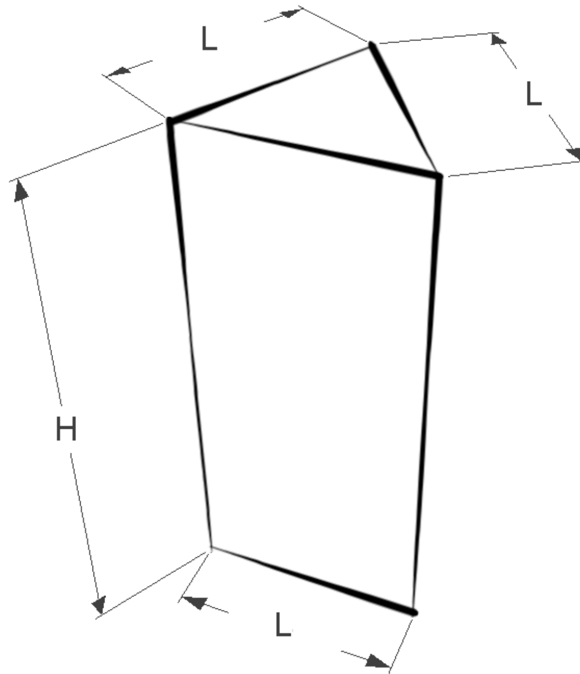

Figure 5.17: Triangle Prism Schematic

With the global dimension '*H*' and *'L'* defined and the mesh density set as '1' for all sides, the constructor will call the function *DefineNodes()* to construct the six perimeter nodes of the prism. As presented in Figure 5.18 and Table 5.1, the function defines the bottom three nodes at $z = 0.0$ and top three nodes at distance $z = H$.

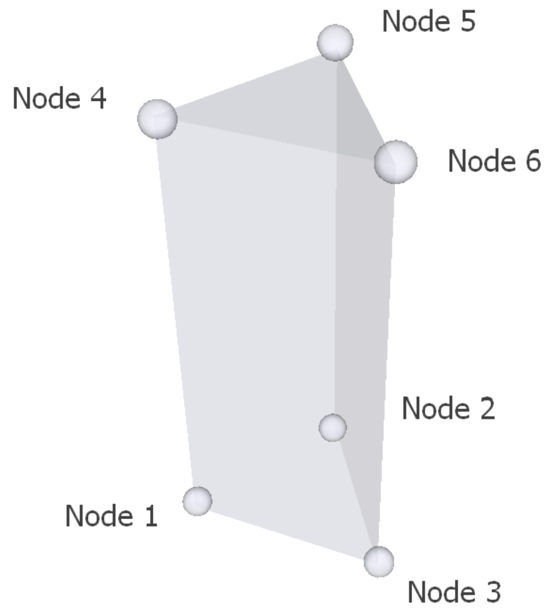

Figure 5.18: Triangle Prism ***Node*** Element

Table 5.1: Triangle Prism Node Table

Node ID	**X**	**Y**	**Z**
1	$-1/2\,L$	-$\sqrt{3/4}\,L$	0.0
2	$+1/2\,L$	-$\sqrt{3/4}\,L$	0.0
3	0.0	+$\sqrt{3/4}\,L$	0.0
4	$-1/2\,L$	-$\sqrt{3/4}\,L$	H
5	$+1/2\,L$	-$\sqrt{3/4}\,L$	H
6	0.0	+$\sqrt{3/4}\,L$	H

Following the nodal declaration, the ***Triangle*** element is declared using the *DefineTriangle()* function. Two triangles are constructed; one connects the top nodes and the other connects the bottom nodes.

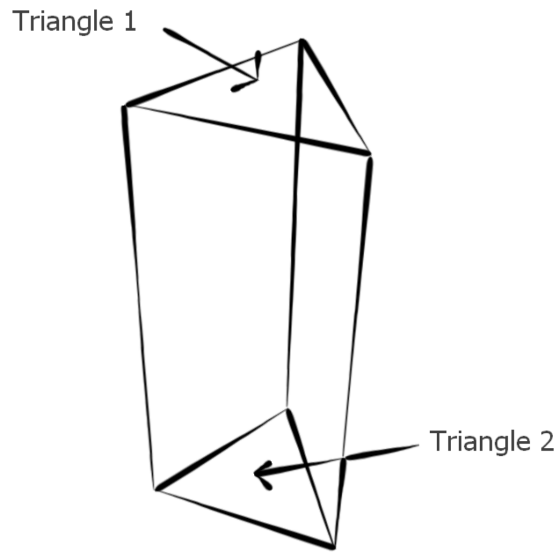

Figure 5.19: Triangle Prism ***Triangle*** Element

Table 5.2: Triangle Prism Triangle Table

Triangle ID	**Node A**	**Node B**	**Node C**
1	Node 4	Node 5	Node 6
2	Node 1	Node 2	Node 3

Following the triangle declaration, the ***Quadrailateral*** element is declared using the *DefineQuadrilateral()* function. There are three quadrilaterals constructed connecting the three sides of the prism.

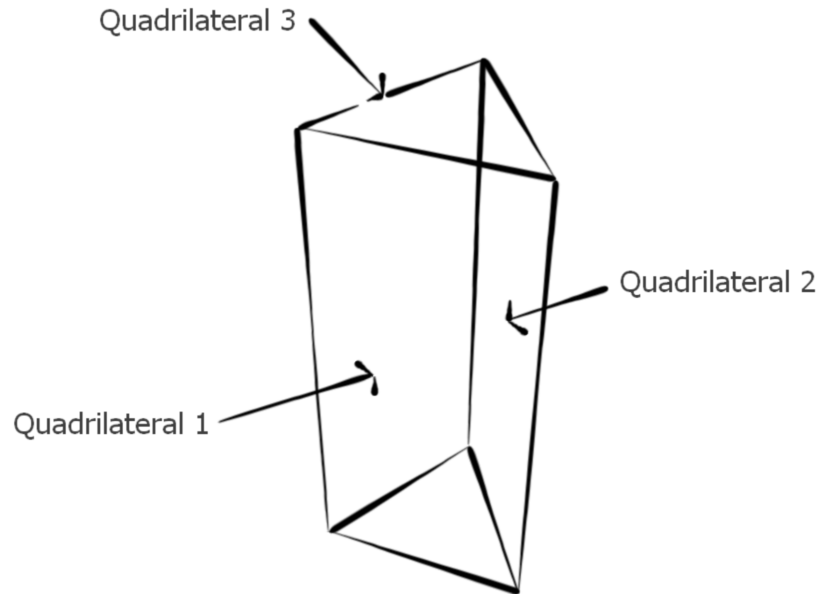

Figure 5.20: Triangle Prism ***Quadrilateral*** Element

Table 5.3: Triangle Prism Quadrilateral Table

Quadrilateral ID	Node A	Node B	Node C	Node D
1	Node 1	Node 4	Node 6	Node 3
2	Node 3	Node 6	Node 5	Node 2
3	Node 2	Node 5	Node 4	Node 1

Once the nodes, triangles, and quadrilateral elements are defined, these elements can be used to generate the necessary arrays and parameters to render the object in the

graphics engine. The function *GetGraphicObject(…)* returns the ***Graphics Object*** structure that is used by the graphics engine to render the component. The graphics object is described in the next section; in short, it stores the vertices and indices of a bridge component. The method calls the *GetNodeVertices()*, *GetPlaneIndices()*, *GetPlanePrimitiveCount()*, *GetWireframeIndices()*, and *GetWireframePrimitiveCount()* functions to create the ***Graphics Object***.

The function *GetNodeVertices()* processes the nodal information positions and colors into vertices to load into a vertex buffer. With the vertices loaded, the triangle and line indices list can be formed with the *GetPlaneIndices()*and *GetWireframeIndices()* functions, respectively. The number of triangle and line primitives are determined using the functions *GetPlanePrimitiveCount()* and *GetWireframePrimitiveCount()*, respectively.

Graphics Object

Graphics Object is a collection of data placed into a structure type that is used to render an object in the graphics engine. All data types inside the structure haves *public* access and therefore do not need any respective methods to set or return the data. Figure 5.21 shows a diagram of the structure.

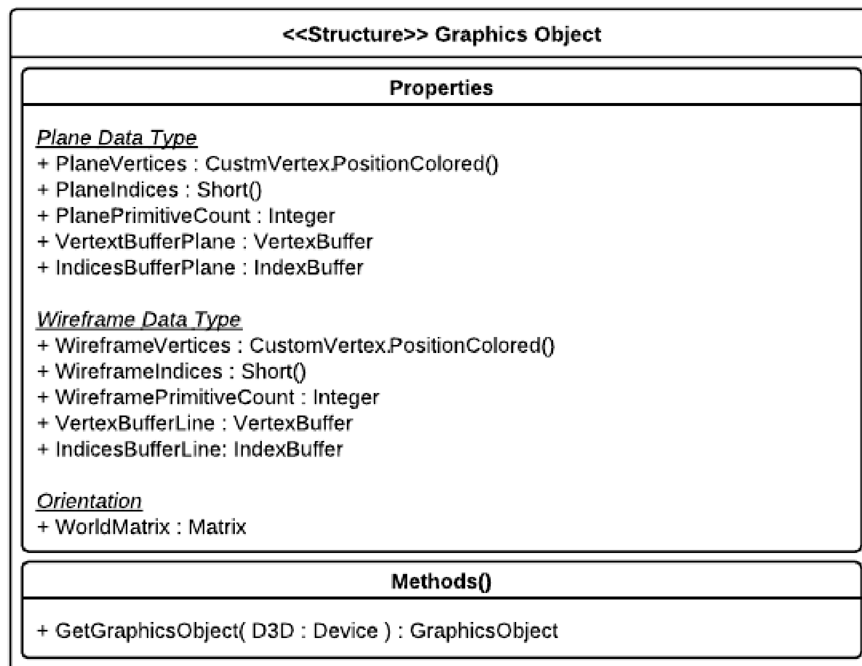

Figure 5.21: ***Graphics Object*** Structure Diagram

The graphics engine stores information for both the solid planes and the wireframe lines for a given component. Both require a combination of vertices, indices, and a number of primitives to create the vertex buffer and index buffer. The structure also stores the world matrix of the component. The world matrix is used to determine the orientation of the component from its local coordinate system to the global (or world) coordinate system. For a typical ***Graphics Object***, the world matrix is the identity matrix, meaning the local coordinate axes are the same as the global coordinate axes.

The one method that is available inside the structure is the *GetGraphicObejct(…)* function. This function passes to the graphics device the location where the vertex and

index buffer information will be loaded. It then returns the graphics object back to the graphics engine to display.

GRAPHICS ENGINE CYCLE

After meshing a given bridge component, the mesh is stored in the ***Structural Component*** object for rendering in the graphics engine. There are four different segments of the graphics engine cycle as shown in Figure 5.22: the GUI, ATP-Bridge Graphics Engine, the ***JQB Component,*** and the ***Structural Component***.

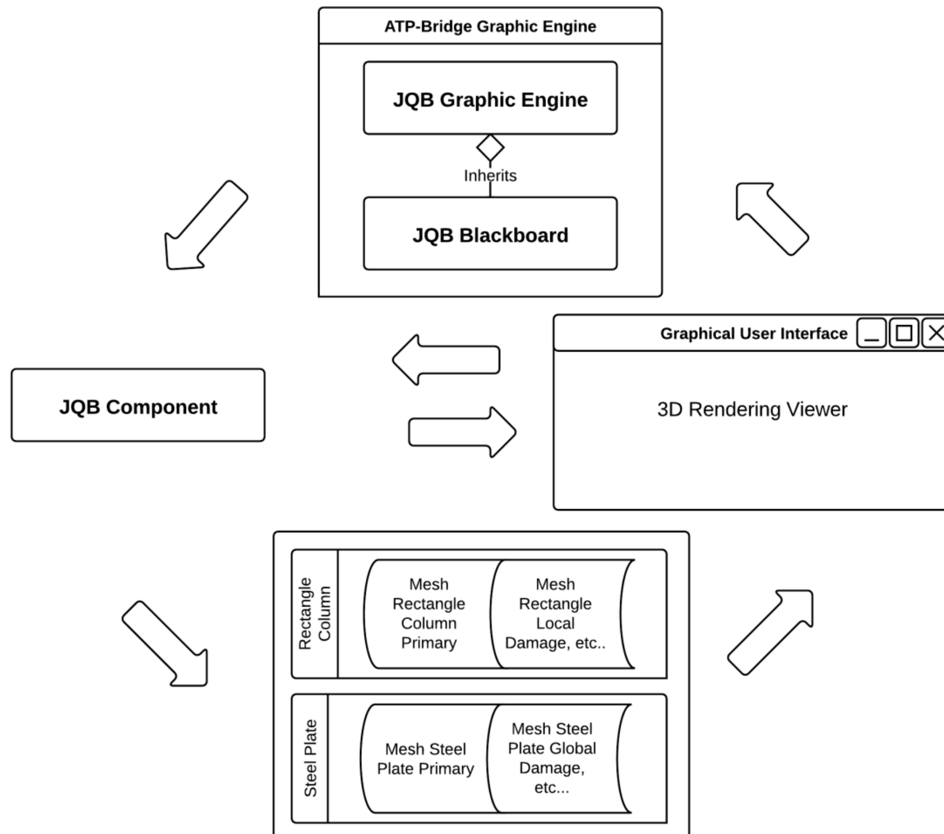

Figure 5.22: Graphics Engine Flow Chart

The cycle starts with a GUI-triggered event, such as a mouse click or a button being pressed. Such an event then communicates with the ATP-Bridge graphics engine to modify the environment. Afterwards, the graphics engine will update the frame by calling the ***JQB Component*** class to re-render the bridge component. The ***JQB Component*** takes the mesh component stored in the ***Structural Component*** class and renders it in the GUI. This four-step process completes a single frame of the graphics engine. The following sections detail the three additional classes in the cycle: ***JQB Graphics Engine***, ***JQB Blackboard***, and ***JQB Component***.

JQB Graphics Engine Class

The ***JQB Graphics Engine*** class helps create the virtual environment where bridge components are rendered. Because the class is designed only to initialize the correct graphics card parameters, setup the presentation space, and control the camera (explained later in this chapter), the class must be inherited by the sub-class to render bridge components. Figure 5.23 presents a partial list of the critical properties and methods inside the class.

MyDevice is a variable of the ***Device*** class from Direct3D and is the means to communicate with a PC's graphics card. The *SetDevice()* subroutine is called to establish the device, determining whether the computer will be using its graphics processor unit (hardware processing) or its central processing unit (software processing). The subroutine determines which window form the device is displayed to, and it establishes the different display properties used in the rendering. The details of these parameters are

omitted because it is beyond the scope of the thesis; additional information can be found in the book by Miller (2004).

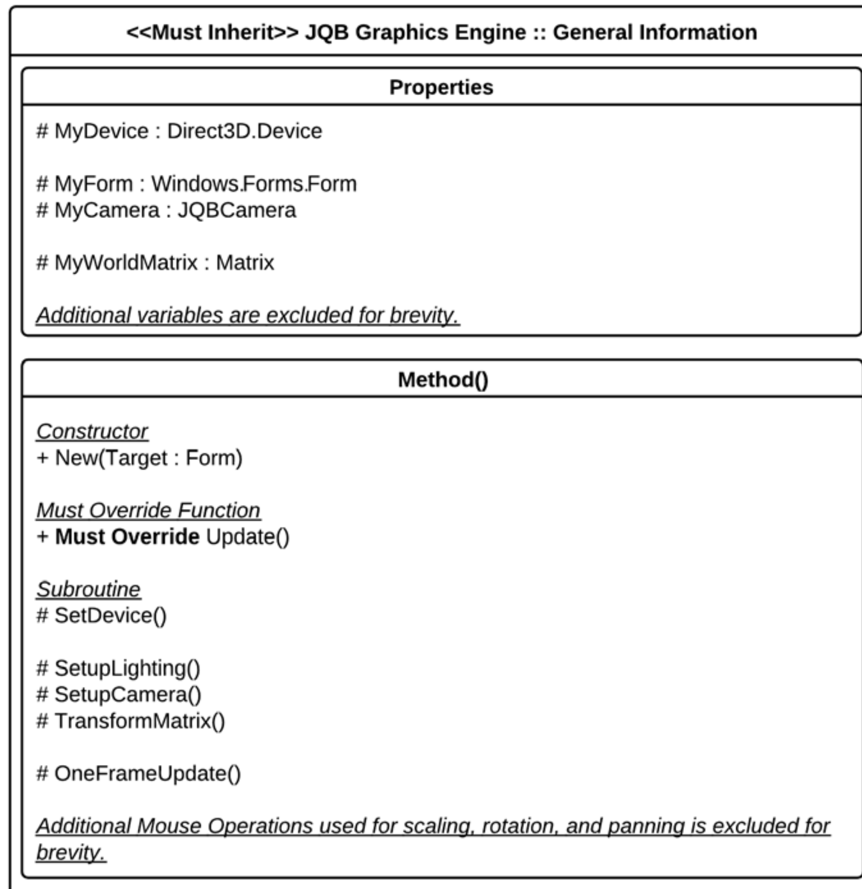

Figure 5.23: ***JQB Graphics Engine*** General Class Diagram

To animate the environment, the GUI calls the subroutine *Update()*, which re-renders the scene for the next frame. The *Update()* subroutine must be inherited by the sub-class and is designed to call the *OneFrameUpdate()* subroutine and any other subroutine in the sub-class to update the frame. The *OneFrameUpdate()* subroutine calls

the *SetupLight()* subroutine to update the lighting, the *SetupCamera()* subroutine to update the camera, and the *TransformMatrix()* subroutine to modify the world matrix that establishes the overall local coordinate axes relative to the global coordinate system. In the software, both the lighting and the world matrix currently does not change at run-time, with the lighting turned off and the world matrix set to the identity matrix. Of these three subroutine, *SetupCamera()* is further explained because it plays an important role in ATP-Bridge.

The *SetupCamera()* subroutine edits the variable *MyCamera* of the ***JQBCamera*** class. This class is developed according to Thorn's (2005) book, using a camera matrix that defines the location, target focus, and its angle of rotation. Figure 5.24 shows a list of properties and methods used to control the camera.

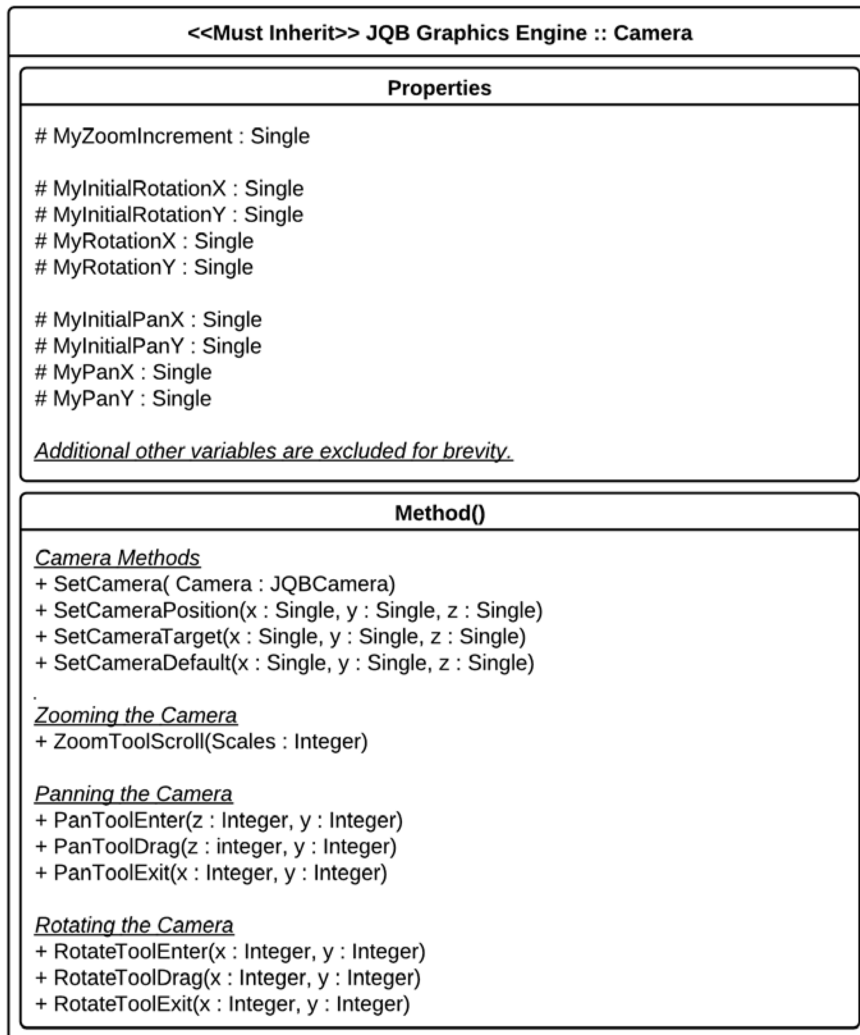

Figure 5.24: ***JQB Graphics Engine*** Camera Class Diagram

Three camera methods are used to initialize the camera: *SetCameraPosition(…)* defines the camera location in 3D space, *SetCameraTarget(…)* defines the point in space where the camera is focused, and *SetCameraDefault(…)* sets the default orientation of the camera. Three interactive motions are modeled in the ATP-Bridge graphics engine: zooming, panning, and rotating the camera.

Zooming the camera is activated when the user scrolls the middle mouse button, triggering an event that calls the *ZoomToolScroll(…)* subroutine in the graphics engine. This subroutine stores the amount of scrolling in a variable called *MyZoomIncrement*. The subroutine then updates the camera position forward and backwards with respect to the camera target; Figure 5.25 provides a visual description.

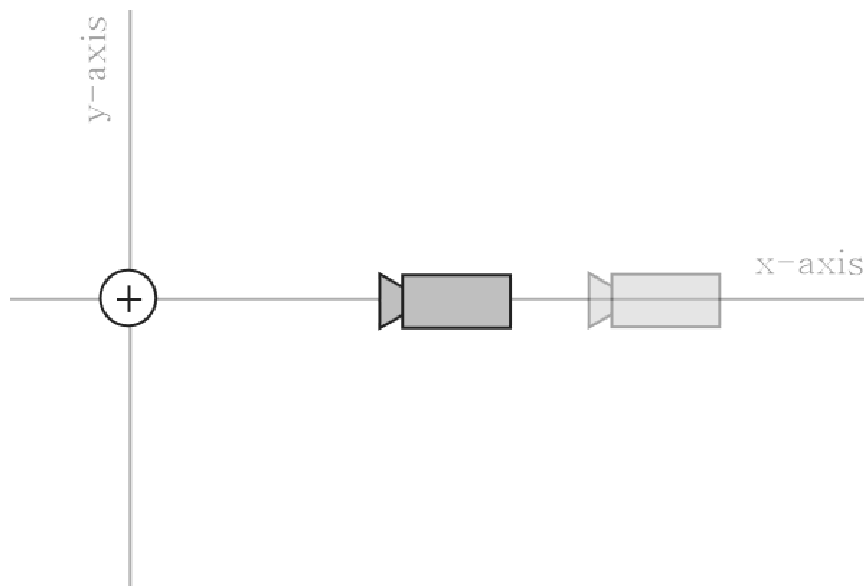

Figure 5.25: Zooming the Camera

Panning the camera is activated by the user clicking the left mouse button, triggering an event in the graphics engine that calls the subroutine *PanToolClicked(…)*. This subroutine tracks the initial and current (x, y) coordinates of the mouse and stores it in the *MyInitialPanX* and *MyInitialPanY* variables. If the button is still pressed and the user drags the mouse, the *PanToolDrag(…)* subroutine is triggered to update the current (x, y) coordinate in the variables *MyPanX* and *MyPanY*. To simulate panning, the camera

position and target position are translated by the same amount, as shown Figure 5.26. When the left button is released, the *PanToolUnclick(…)* subroutine is called to stop updating.

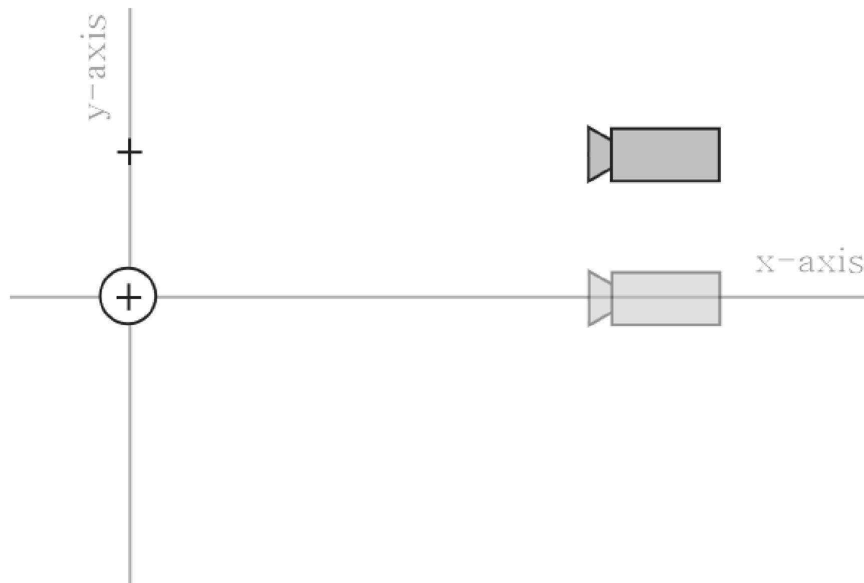

Figure 5.26: Panning the Camera

Rotating the camera is activated by the user clicking the middle mouse button, triggering an event in the graphics engine that calls the subroutine *RotateToolEnter(…)*. This method tracks the initial and current (x, y) coordinates of the mouse and stores it in the *MyInitialRotationX* and *MyInitialRotationY* variables. If the button remains pressed while the user drags the mouse, the *RotateToolDrag(…)* subroutine is triggered to update the current (x, y) coordinate in the variables *MyRotateX* and *MyRotateY*, and it then updates the next frame. To rotate the camera, the location of the camera pivots about the target position and the camera angle rotates pointing to the target location; see Figure

5.27 for a visual diagram. When the button is released, the *RotateToolUnclick(…)* subroutine is called to inform the graphics engine to cease updating.

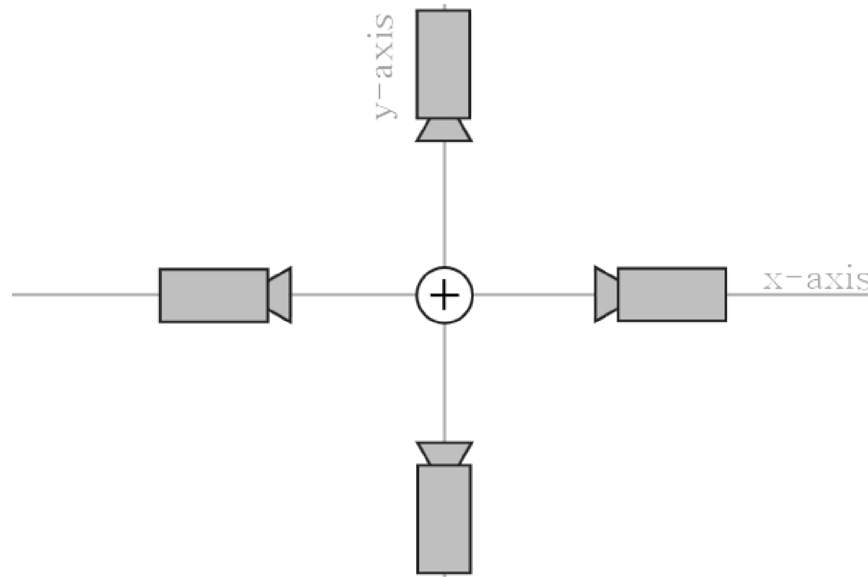

Figure 5.27: Rotating the Camera

JQB Blackboard Class

JQB Blackboard is the sub-class of the ***JQB Graphics Engine*** and is created to render objects in the graphics engine (Figure 5.28). *MyComponent* is the reference variable of the ***JQB Component*** class stored in the ***Structural Component*** data structure and is used to switch between different bridge components by changing the variable in the *SetJQBComponent(…)* subroutine. More information about this class is provided in the next section.

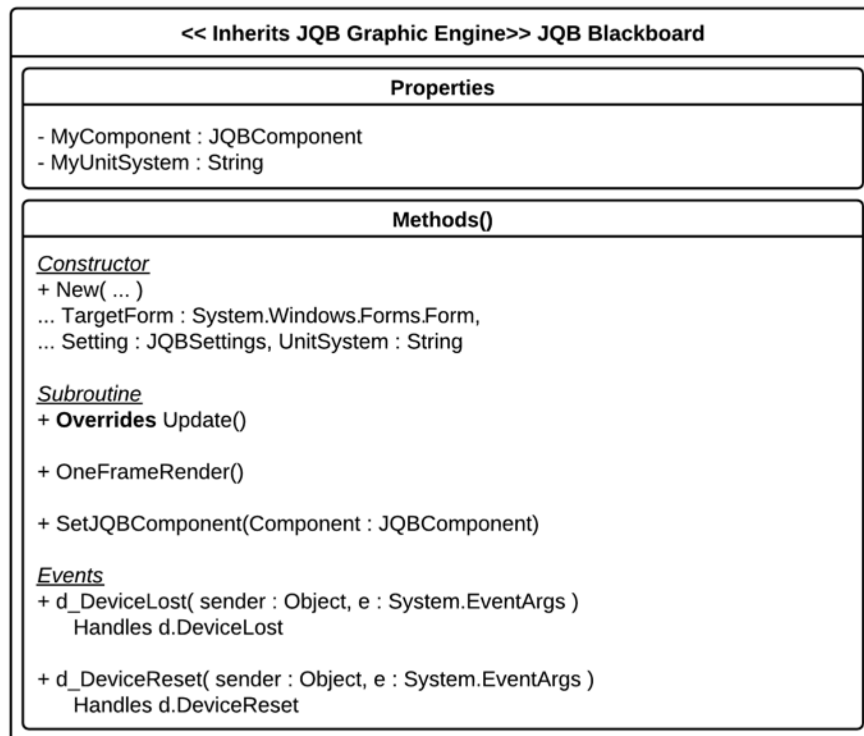

Figure 5.28: ***JQB Blackboard*** Class Diagram Method()

Once the ***JQB Component*** reference variable is stored, the component can then be rendered by calling the *Update()* subroutine. As previously stated, the *Update()* subroutine calls the *OneFrameUpdate()* subroutine from the super-class, but it also calls the *OneFrameRender()* subroutine declared in the sub-class. The *OneFrameRender()* subroutine is used to render objects onto the screen.

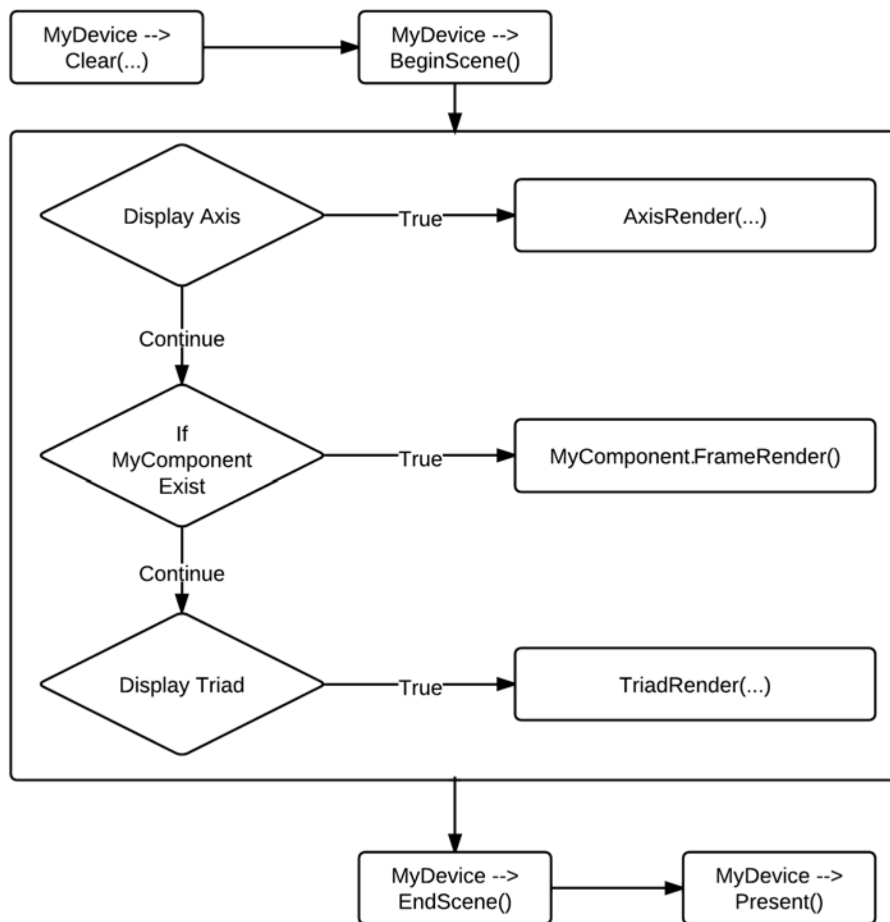

Figure 5.29: *OneFrameRender()* Method

A flowchart of the *OneFrameRender()* method is shown in Figure 5.29. The method begins by calling ***MyDevice*** → *Clear(),* a method used to wipe the existing data from the *buffer* and set the background color. The buffer is similar to a blackboard—a space used to draw the content. In ATP-Bridge there are two buffers: a front buffer that is being displayed currently and a back buffer that is used to draw the next frame. After a

frame is done rendering, the back buffer is swapped with the front buffer and presented. This process continues in reverse for the next frame.

To begin rendering in the buffer, the method *MyDevice* → *BeginScene()* is called to prepare the device. Once the device is ready for rendering, the first decision is to determine whether to render an axis on the screen. If it is to be rendered, the *AxisRender(…)* subroutine is called. Following that, the next step is to determine whether a component exists for rendering. If so, the class will call the *MyComponent* → *FrameRender()* subroutine inside the ***JQB Component*** class. The last decision concerns whether to display the triad defining the global origin in space. If it is to be displayed, the *TriadRender()* subroutine is called. After all the rendering is done, *MyDevice* → *EndScene()* is called to instruct the device that the buffer is ready to be presented. Finally, the program calls *MyDevice* → *Present()* to swap the back buffer with the front buffer and display the content of the back buffer.

JQB Component

The ***JQB Component*** class is the intermediary component connecting the ***JQB Blackboard*** with the ***Structural Component*** class. This class is a super-class and must be inherited by the sub-class. By allowing sub-classes, program developers are able to customize the rendering algorithm for each bridge component. See Figure 5.30 and Figure 5.31 for a list of important properties and methods for the class.

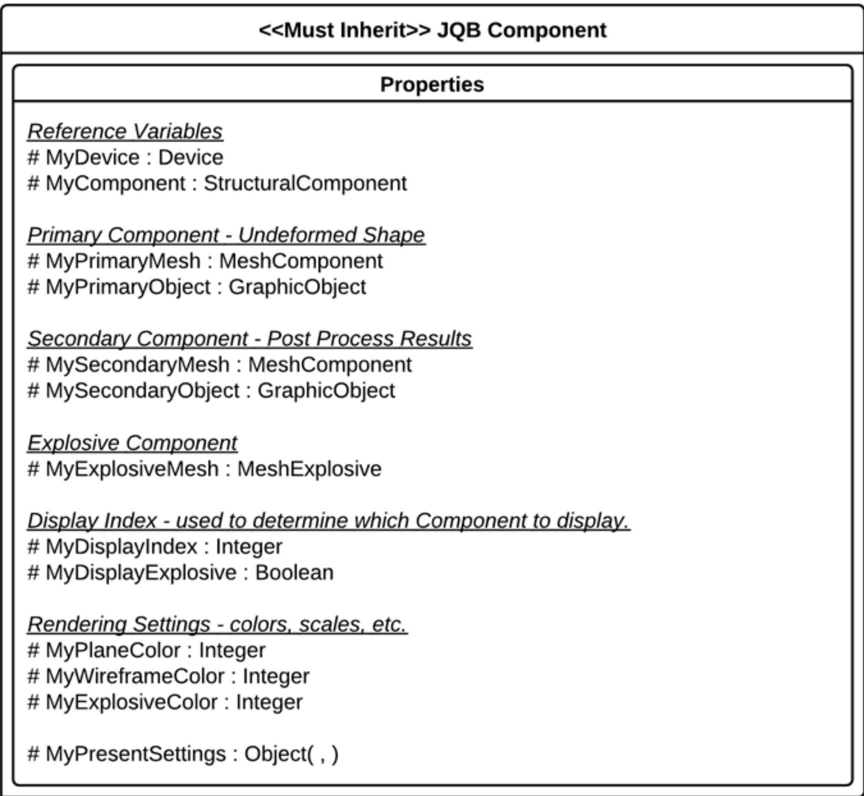

Figure 5.30: ***JQB Component*** Class Diagram Properties

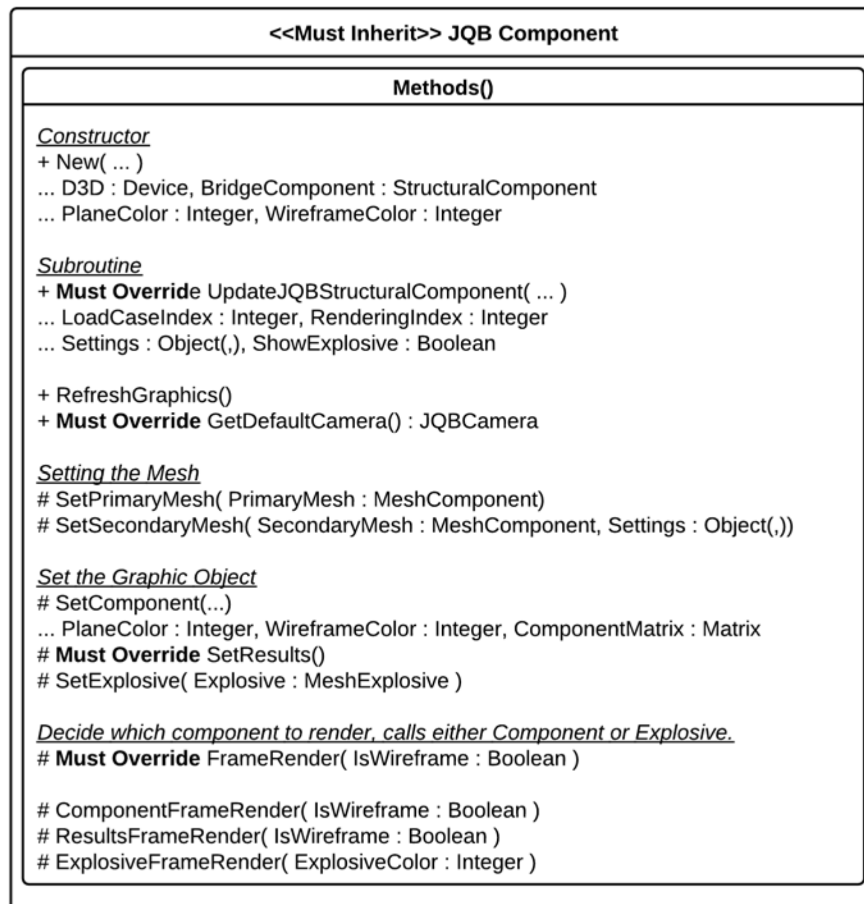

Figure 5.31: ***JQB Component*** Class Diagram Methods()

The class is initialized with the constructor *New(…)*, requiring both the ***Structural Component*** and the Direct3D ***Device*** declared in the graphics engine as reference variables. Inside the constructor, the *SetPrimaryMesh(…)* subroutine is called to store the undeformed mesh from the ***Structural Component*** class into the *MyPrimaryMesh* variable. Afterwards, the *MyPrimaryObject* graphics object is created by calling the *SetComponent(…)* subroutine. Similarly to the primary mesh and object, the

SetSeconaryMesh() subroutine and *SetResults()* are used to store the post-processing results mesh and graphics object, respectively. As for the *SetResults()* subroutine, it must be inherited for the sub-class so it is able to allow different options to render each specific bridge component.

The subroutine *UpdateJQBStructuralComponent(…)* is called by the GUI to specify which load case mesh explosive should be displayed, which results are to be displayed, and how the bridge components should be displayed. This subroutine must be overridden by the sub-class so that in can include the different variations of post-processed results for each specific bridge component. Given the index, the sub-class will determine which mesh and graphics object is loaded into the *MySecondaryMesh* and *MySecondaryObject* variables, respectively. The *GetDefaultCamera()* method must also be inherited by the sub-class. This function returns the default camera position set for a bridge component when it is first loaded in the graphics engine.

The *FrameRender(…)* subroutine is called by ***JQB Blackboard*** to initiate rendering of a bridge component. The subroutine must be overridden for the individual sub-class, and the subroutine chooses between the *ComponentFrameRender()* or *ResultsFrameRender()* subroutine to display bases from the *MyDisplayIndex* variable value. Similarly, the *MyDisplayExplosive* variable is used to determine if the explosive should be displayed, and if so the *ExplosiveFrameRender()* is called to render the explosive.

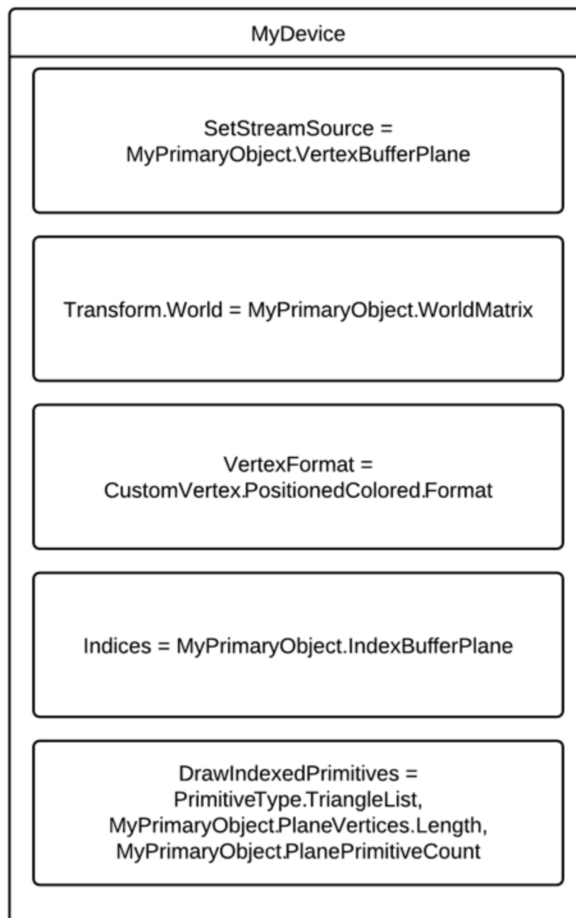

Figure 5.32: *ComponentFrameRender()* Subroutine Rendering Algorithm

There are sequences of steps required to render a graphics object into the device buffer. Shown in Figure 5.32 is the *ComponentFrameRender()* example to render the planes of the primary object. The first step involves calling the *SetStreamSource(…)* method and loading the vertex buffer from *MyPrimaryObject* to the device buffer. The second step sets the world matrix to the desired matrix to render the object. The third step sets the vertex format to the 'positioned color' to determine the type of vertices to render. The fourth step is to load the index buffer into the device. Finally, the last step is

to draw the primitive in the device by specifying the primitive type, the number of vertices, and the number of primitive types to render. This same sequence is used to render the *MySecondaryObject* variable, both for the planes and the wireframe. Once both sequences complete rendering the planes and wireframe of a bridge component, the results are displayed in the GUI ***3D Rendering*** viewer. This cycle then continues for the next frame to create a dynamic environment.

SUMMARY

This chapter includes a description of the overall GUI concepts and design, a brief introduction of Direct3D, and an explanation of the graphics engine structure. The next chapter steps through an example of a *solver*. The solver presented evaluates a simplified flexural model for a prestressed concrete bridge girder.

Chapter 6: Prestressed Girder Model

The last part of ATP-Bridge needed to complete the software cycle is comprised of the *solvers*. A solver takes information from a data structure, analyzes it, and displays it in the GUI. A solver is similar to a brain, which processes complex decisions that are then communicated back to the body. Solvers are numerical models that are independent among the bridge components included in ATP-Bridge. This arrangement allows each solver to choose the most appropriate model for the specific bridge component being analyzed.

For example, in ATP-Bridge, the reinforced concrete column's solver is an advanced single-degree-of-freedom model, whereas the steel plate's solver is a set of empirical equations. These two models are different, but both models provide the desired level of computational efficiency and accuracy when compared with available test data.

To illustrate the development of a solver, the flexural response of prestressed girders subjected to blast loading is investigated using a simplified single-degree-of-freedom (SDOF) model. Prestressed girders are used throughout the US highway infrastructure because of their low cost, superb strength, and ease of construction. Prestressed girders make up approximately 11% of the current bridge inventory in the US (Cofer, 2012). They perform a critical role in elevating the deck and spanning a bridge between the piers. When they have been targeted in the past, the bridges that were attacked sustained substantial damage. For example, an attack in western Iraq targeting a bridge where Iraqi soldiers were crossing caused the complete failure of some of the

girders as well as large portion of the deck, as reported in the Washington Post article by al-Mokhtar (2009). Figure 6.1 shows an above-deck view of the damage.

Figure 6.1: Bridge Destroyed in Iraq from Truck Bomb (The Washington Post, al-Mokhtar, 2009)

EXPERIMENTAL WORK

Although extensive research on prestressed girders has been conducted for static loading and traffic loading conditions, only limited data exist for the response of these components subjected to blast loads. To date, the only known experimental study was conducted by Cofer (2012) at Washington State University.

Blast testing was performed by the Engineer Research and Development Center (ERDC) of the US Army Corps of Engineers (Matthews, 2008). There were two loading scenarios—one case involving a detonation on top of the prestressed girder and the other case involving a detonation below the girder. The geometry of the test specimen was a

Colorado Department of Transportation bulb-tee section having a 3 ft.−6 in. in depth, 3 ft.−7 in. top flange width, and a 2 ft.–3 in. bottom flange width. The dimensions are shown on Figure 6.2. The girder spanned 68 ft.–4 in. and rested on two bearing pads.

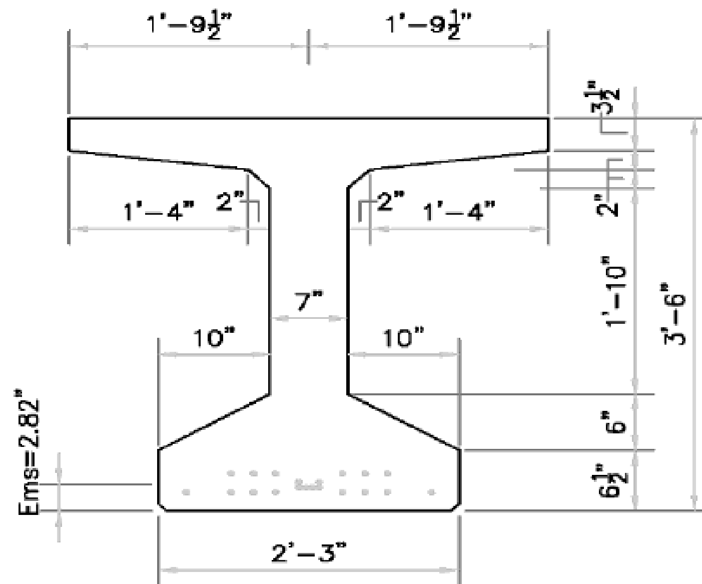

Figure 6.2: Colorado Bulb-Tee Test Specimen Dimensions (Matthews, 2008)

For the above-girder detonation scenario, the explosive was placed at a standoff of approximately X.XX ft. from the mid-span of the test specimen. The blast load resulted in rubblizing approximately 3 ft. – 6 in. of the concrete on the top flange and along the web. In addition, heavy cracking was observed along the web, and longitudinal cracking occurred along the underside of the flange (Matthew, 2008). Figure 6.3 shows the damage that occurred for the above-girder detonation scenario.

Figure 6.3: Test Specimen Above-Detonation Scenario (Matthews, 2008)

For the below-girder detonation scenario, the explosive was place at a standoff of XX.X ft. below the mid-span of the test specimen. The explosion destroyed 9 ft. of the top flange. Of the remaining concrete, less than on 1 ft. was rubblized, and heavy cracking occurred 10 ft. along the web. Figure 6.4 shows the damage experienced for the below-girder detonation scenario.

Figure 6.4: Test Specimen Below-Girder Detonation Load Case (Matthews, 2008)

For both scenarios, the prestressed girder failed due to local damage. Although the results cannot be directly compared with a flexural model, the experimental data collected can be used to validate a prestressed girder (FE) model. In turn, the FE model can be used to compare with a single-degree-of-freedom response model (SDOF) for investigating scenarios in which girder damage is not as severe as that observed in the test program described in this section. The FE validation was performed by Hendryx (2012); interested readers may consult his thesis for additional information. This thesis uses the validated model and compares it with the SDOF response model at the end of this chapter.

ADVANCED SINGLE-DEGREE-OF-FREEDOM ALGORITHM

The prestressed girder solver created for ATP-Bridge uses an SDOF analysis procedure that is consistent with the guidelines given in UFC 3-340-02 (Department of Defense, 2008), but it also includes several enhanced features based on work done by Sammarco (2012) for the analysis of reinforced concrete columns subjected to blast loads.

Sammarco's model analyzes both the flexural and dynamic shear response of reinforced concrete columns. In his model, the two response modes are uncoupled due to the notably different dynamic response characteristics. It is hypothesized for this research that the same computational algorithm can be applied to prestressed girders due to the largely different natural periods associated with the primary modes of response in flexure and in shear that occur for members with typical properties and geometries.

Sammarco's flexural model involves three major phases. The first phase is the development of a bilinear moment-curvature response curve, taking into account dynamic increase factors, confinement effects, and axial loads (Department of Defense, 2008). The second phase is the development of a non-linear resistance function that uses the flexural mode shape derived from the user-specified threat scenario. The resistance function is produced by performing a static analysis that accounts for geometric non-linearity and plastic hinge formation. The final phase is a non-linear SDOF analysis using the formulated resistance function and the pressure-time history generated from the BEL software (USACE-ERDC, 2004). The remaining sections of this chapter cover the

development of the three analysis phases for the prestressed girder model. A description regarding how the current model differs from Sammarco's model for reinforced concrete columns is also provided.

MOMENT-CURVATURE RELATIONSHIP

Like Sammarco's model, the first phase for the analysis of blast-loaded prestressed girders is the construction of the moment-curvature relationship. The moment-curvature relationship for the prestressed girder model differs from Sammarco's model in many respects. One concern with the model is the addition of prestressing strands that have no defined yield plateau and initial strain, requiring an additional term when calculating the moment-curvature relationship. Another concern is with the initial strain—the moment-curvature response of the prestressing model is put into initial reverse curvature at zero applied moment. Because of these differences, the moment-curvature relationship is first developed using a layer-by-layer analysis as described in the text by Collins and Mitchell (1997), then simplified into a bilinear model.

The layer-by-layer analysis consists of breaking a prestressed girder cross-section into small incremental areas with a respective material model. The concrete section is broken into thin rectangular fibers (Figure 6.5-a), and the mild-steel reinforcement and prestressing strand are modeled using circular fibers. Based on the assumption that plane section remains plane, strains at the centroid at each fiber can be found assuming a linear distribution of the strain through the depth of the cross-section. Stresses can then be found knowing the strain in a given fiber and the associated material model properties.

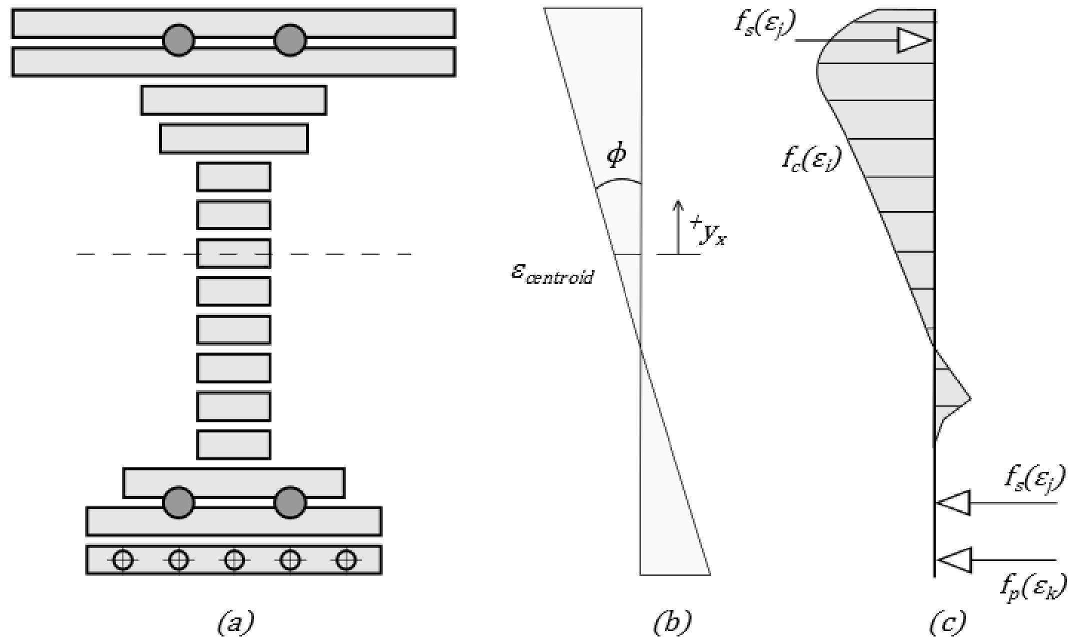

Figure 6.5: Fiber Diagrams of the Test Specimen

Two variables are needed to generate the moment-curvature relationship of a flexural member: curvature and axial strain at the centroid of the cross-section. With these two variables, strains at any point in the cross-section can be found using Equation (6-1). Figure 6.5-b shows the strain diagram along the cross section.

$$\varepsilon_x = \varepsilon_{centroid} + \phi\, y_x \tag{6-1}$$

where,

$\varepsilon_{centroid}$ = axial strain at the centroid of the cross-section

ϕ = curvature of the cross-section

y_x = distance from centroid to fiber being analyzed

In the algorithm used for this study, the curvature is first established, and then strain at the centroid is iterated upon until convergence is achieved. Iterations are needed because of the non-linear stress-strain relationship that exists in each fiber through the depth of the cross-section. To find the correct pair, all the internal forces within the section must be in equilibrium. The governing equation is presented in Equation (6-2), where the forces from the concrete fibers, the reinforcement fibers, and the prestressing fibers must be equal to all externally acting axial loads. Convergence is achieved when Equation (6-2) is satisfied within the desired tolerance. After ensuring convergence, the moment capacity of the section can then be calculated using Equation (6-3).

$$\sum_{h=1}^{l} N_h = \sum_{i=1}^{m} f_c(\varepsilon_i)\, b_i h_i + \sum_{j=1}^{n} f_s(\varepsilon_j)\, A_{s_j} + \sum_{k=1}^{k} f_p(\varepsilon_k)\, A_{p_k} \tag{6-2}$$

$$\sum_{h=1}^{l} M_h = \sum_{i=1}^{m} f_c(\varepsilon_i)\, b_i h_i y_{c_i} + \sum_{j=1}^{n} f_s(\varepsilon_j)\, A_{s_j}\, y_{c_j} + \sum_{k=1}^{k} f_p(\varepsilon_k)\, A_{p_k} y_{p_j} \tag{6-3}$$

where,

N_h	= axial loads acting along the section
M_h	= moments about the *z*-axis
y_c	= distance from the fiber to the centroid of the entire cross-section
$f_c(\varepsilon_i)$	= concrete stress at a given strain
b_i	= width of a fiber
h_i	= height of a fiber
$f_s(\varepsilon_j)$	= steel stress at a given strain

A_{s_j} = area of reinforcement

$f_p(\varepsilon_k)$ = prestressing strand stress at a given strain

A_{p_k} = area of the prestressing strand

Concrete Material Model

Concrete material properties must be modeled accurately to determine the response of a prestressed girder. Concrete complexity stems from its non-linear behavior and its different response characteristics in tension and compression. Because of these properties, concrete is modeled using two sets of equations for compression and tension.

In the concrete material model, the governing equation used for concrete in compression is shown in Equation (6-4). This equation is chosen because it takes into account high-strength concrete, which is commonly used in the construction of prestressed bridge girders (Collins and Mitchell, 1997).

$$\frac{f_c}{f'_c} = \frac{\eta\left(\epsilon_{cf}/\epsilon'_c\right)}{\eta - 1 + \left(\epsilon_{cf}/\epsilon'_c\right)^{\eta\kappa}} \tag{6-4}$$

where,

f'_c = peak compressive stress (in psi)

ϵ'_c = strain when f_c reaches f'_c

$= \frac{f'_c}{E_c}\frac{\eta}{\eta - 1}$

η = curve-fitting factor

$= 0.80 + f'_c/2500$

E_c = tangent stiffness when ϵ_{cf} equals zero

$= 57{,}000\sqrt{f'_c}$ $\quad f'_c \leq 6000\,psi$

$= 40{,}000\sqrt{f'_c} + 10{,}000$ $\quad f'_c > 6000\,psi$

κ = factor to increase the post-peak decay in stress

$= 0.67 + f'_c/9{,}000$ $\quad < 1.0$

$= 1.0$ $\quad \geq 1.0$

Although it is typical to ignore concrete in tension for ultimate design in reinforced concrete, it is desirable to include some tension capacity in the analysis of prestressed girders. Otherwise, the girder will have a brittle failure in reverse curvature. For concrete in tension, linear behavior is assumed (using the compressive elastic modulus) until the stress reaches the modulus of rupture (ACI 318-08). This behavior is captured in Equation (6-5), Equation (6-6), and Equation (6-7).

$$f_r = 7.5\sqrt{f'_c} \qquad (6\text{-}5)$$

$$\varepsilon'_s = f_r/E_s \qquad (6\text{-}6)$$

$$f_c = E_s\varepsilon_{cf} \qquad \text{for } \varepsilon_{cf} \leq \varepsilon'_s \qquad (6\text{-}7)$$

Tension stiffening then dictates the behavior after rupture. Tension stiffening is the phenomenon that occurs after concrete cracks and the section is still able to transfer

stress around the prestressing or mild steel reinforcement. The presence of the reinforcement lets the concrete transfer tension around the cracks, as long as the reinforcement has not yielded. Tension stiffening only occurs 7.5 times the diameter away from the reinforcement (Collins and Mitchell, 1997). Equation (6-8) represents this behavior.

$$f_c = \frac{\alpha_1 \alpha_2 f_r}{1 + \sqrt{500\varepsilon_{cf}}} \qquad for\ \varepsilon_{cf} \geq \varepsilon'_s \tag{6-8}$$

where,

α_1	= 1.0 for mild deformed bars
	= 0.7 for prestressing strands
α_2	= 1.0 for short-term loading
	= 0.7 for long-term loading

Combining all the governing equations for concrete produces a continuous stress-strain diagram as shown in Figure 6.6.

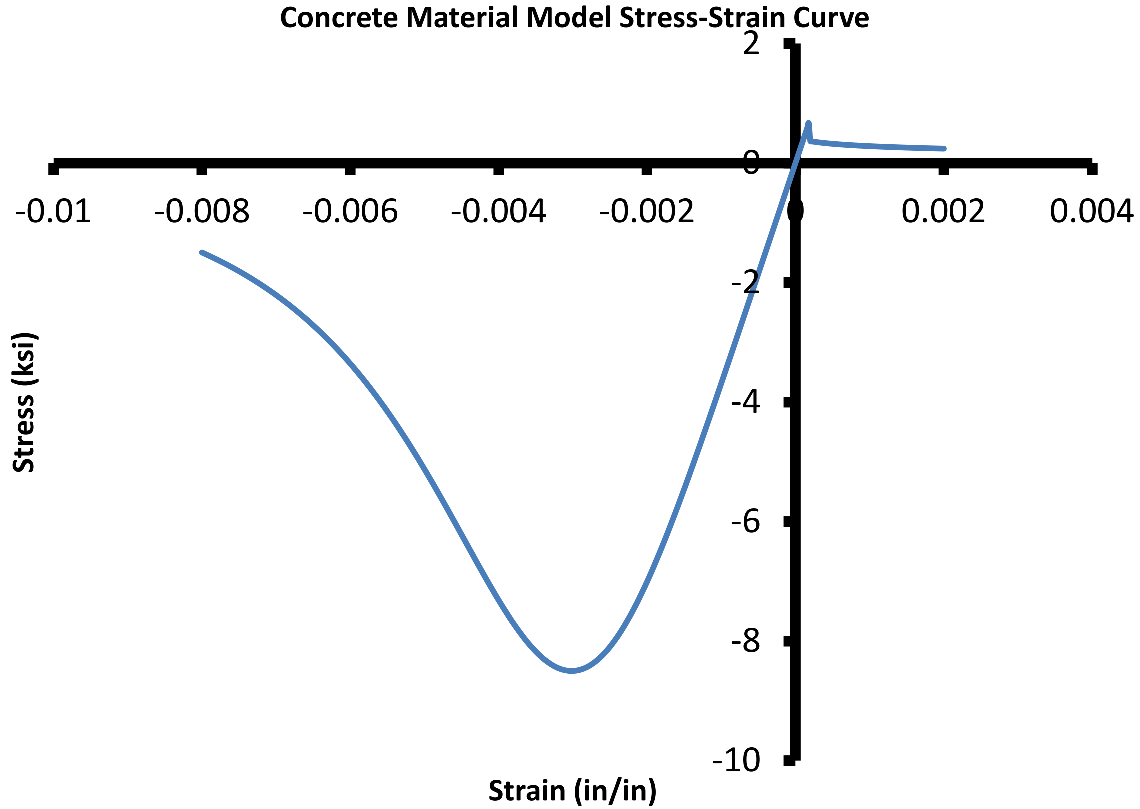

Figure 6.6: f'_c = 8,500 psi Concrete Model Stress-Strain Curve

Mild Steel Reinforcement Model

Mild steel reinforcement used in prestressed concrete creates a “partially prestressed” section, a terminology used in the industry. The addition of mild steel reinforcement increases the ductility of a girder compared to a “fully prestressed” girder, which is a girder without mild steel reinforcement.

The governing equations used for the mild steel reinforcement model is the standard bilinear curve up until the end of the steel yield plateau. Beyond this point, a

parabolic curve is fit from the yield plateau to the ultimate stress of the concrete. The mild steel reinforcement is then considered to be ruptured beyond the ultimate strain. This behavior is captured through Equation (6-9), Equation (6-10), Equation (6-11), and Equation (6-12).

$$f_s = E_s \varepsilon_{sf} \qquad \text{for } \varepsilon_s \leq \varepsilon_{sy} \tag{6-9}$$

$$f_s = f_{sy} \qquad \text{for } \varepsilon_{sy} \leq \varepsilon_s \leq \varepsilon_{sh} \tag{6-10}$$

$$f_s = \frac{1}{(\varepsilon_{su} - \varepsilon_{sh})^2} \left[\begin{array}{c} (f_y - f_u)\varepsilon_{sf}^2 + (2\varepsilon_{su} f_{su} - 2\varepsilon_{su} f_y)\varepsilon_{sf} \\ + \varepsilon_{su}^2 f_{sy} - 2\varepsilon_{su}\varepsilon_{sh} f_u + \varepsilon_{sh}^2 f_{su} \end{array} \right] \qquad \text{for } \varepsilon_{sh} \leq \varepsilon_s \leq \varepsilon_{sy} \tag{6-11}$$

$$f_s = 0 \qquad \text{for } \varepsilon_{su} \leq \varepsilon_s \tag{6-12}$$

where,

E_s = steel modulus of elasticity

f_y = yield strength

f_u = ultimate strength

ε_{sy} = yield strain = f_y/E_s

ε_{sh} = strain at the onset of strain hardening

ε_{su} = strain at ultimate stress

Figure 6.7 presents the mild-steel reinforcement behavior up until the ultimate strain. This behavior is equivalent for both tension and compression.

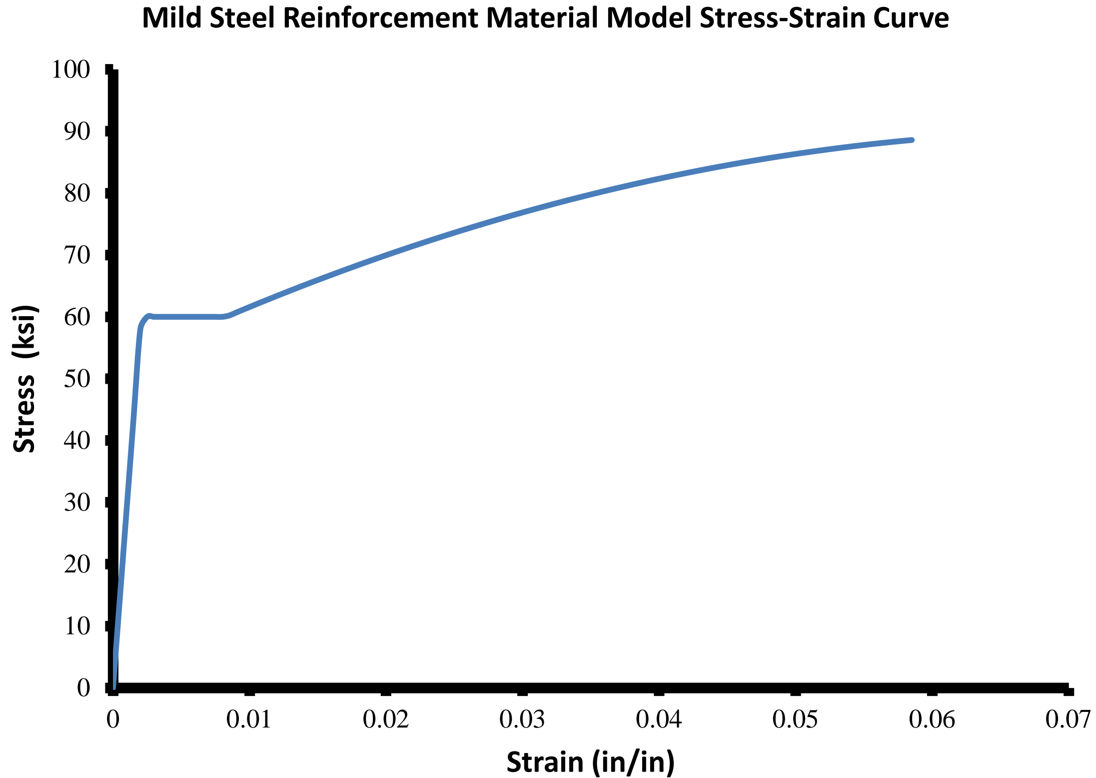

Figure 6.7: 60 ksi Reinforcement Material Model

Prestressing Strand Material Model

Modern-day prestressing utilizes thin wires or 7-wire strands, with the most common strength being 270 ksi low-relaxing strands. Unlike mild-steel reinforcement, prestressing strands do not have a well defined yield plateau. A common formula for the stress-strain response of prestressing strands is the modified Ramberg-Osgood function (Mattock, 1979), shown in Equation (6-13).

$$f_{ps} = \varepsilon_{ps} E \left[Q + \frac{1 - Q}{\left\{ 1 + \left(\frac{E\ \varepsilon_{ps}}{K\ f_{py}} \right)^{R} \right\}^{1/\mathrm{R}}} \right] \leq f_{pu} \qquad (6\text{-}13)$$

where,

f_{ps}	= prestressing stress of the strand
ε_{ps}	= prestressing strain of the strand
E	= modulus of elasticity
f_{py}	= yield stress of strand
f_{pu}	= ultimate stress of strand
$Q, K, and\ R$	= curve fitting parameters

For this research, the prestressing strand stress-strain model is based on Equation (6-14), which was developed by Devalapura and Maher (1992). This equation is an enhanced version of Equation (6-13), fit to experimental test data. One notable finding from their research is that the modulus of elasticity is typically higher than 28,000 ksi, so the constants were adjusted to a modulus of 28,500 ksi. Recommended constants for the different types of strands are given in Table 6-1.

$$f_{ps} = \varepsilon_{ps} \left[A + \frac{B}{\left\{ 1 + (C\ \varepsilon_{ps})^{D} \right\}^{\frac{1}{D}}} \right] \leq f_{pu} \qquad (6\text{-}14)$$

where,

f_{ps} = prestressing stress

ε_{ps} = prestressing strain

f_{pu} = ultimate stress of strand

$A, B, C, and\ D$ = curve fitting parameters

Table 6-1: Power Formula Constants for the Prestressing Stress-Strain Diagram

Steel Type	f_{ps}/f_{pu}	ε_{py}	A	B	C	D
270 ksi strand	0.90	0.010	887	2,7613	112.4	7.360
250 ksi strand	0.90	0.010	384	27,616	119.7	6.430
250 ksi wire	0.90	0.010	435	28,565	125.1	6.351
235 ksi wire	0.90	0.010	403	28,597	133.1	5.463
150 ksi bar	0.85	0.080	467	28533	225.2	4.991

Table is replicated from Devalapura and Maher (1992)

In the prestressing strand constitutive model, the strand has no capacity in compression. Figure 6.8 shows the stress-strain curve for 270 ksi low relaxation strand.

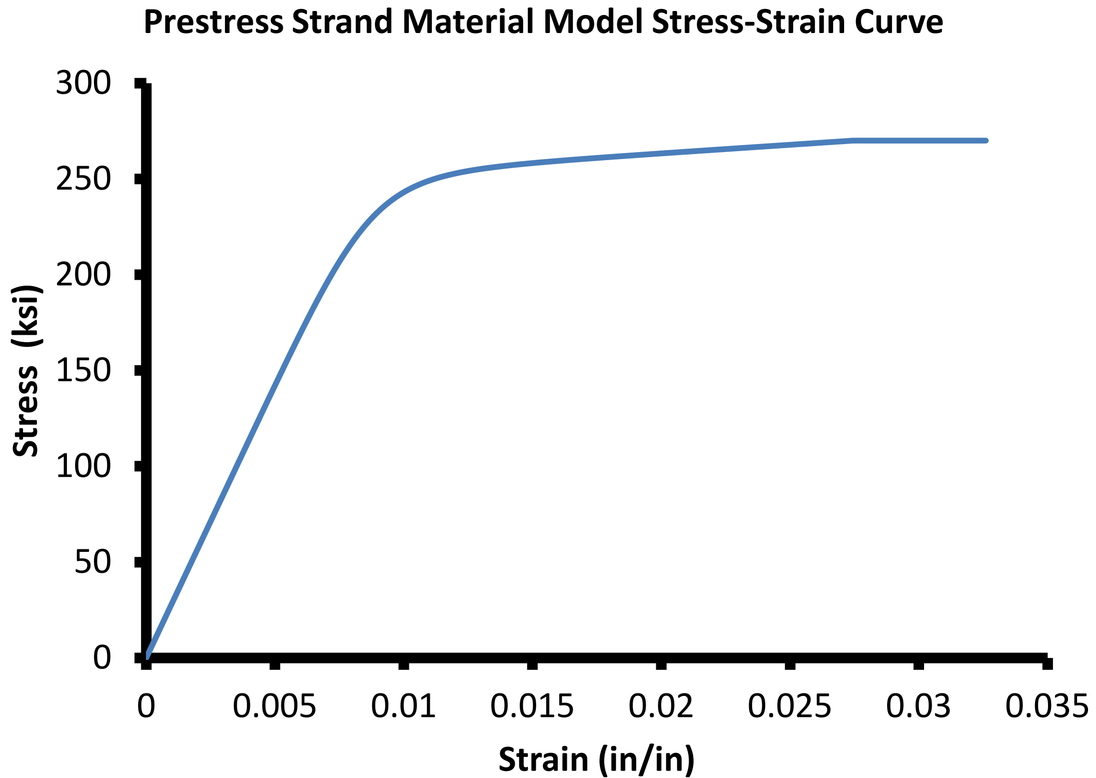

Figure 6.8: 270 ksi Low-Relaxation Prestressing Strand Material Model

Dynamic Increase Factor (DIF)

Past research (e.g., Malvar (1998)) has shown that concrete and mild-steel reinforcement have increased capacity at high strain rates. To account for strain rate effects, the analyses for this research use the *dynamic increase factors* (DIF) given in Table 6-2. DIFs give the ratio of dynamic strength to static strength and are frequently used for blast-resistant design in SDOF models (Dunseberry, 2010). For concrete design

in the "far range" or low pressure range, concrete exhibits an increase in the ultimate strength but not in yielding (Figure 6.9). With mild-steel reinforcement, the yield strength and ultimate strength are increased (Figure 6.10). Because prestressing strand is highly stressed, these elements are not known to exhibit DIFs (DOD, 2008).

Table 6-2: Dynamic Increase Factor for Far Range

	Yield Stress	**Ultimate Stress**
	f_{dy}/f_y	f_{du}/f_u
Concrete[1]	—	1.12
Mild-Steel Reinforcement[1]	1.17	1.05
Prestressing Strand[2]	—	1.00

1 – UFC 3-340-02; page 1068 (Department of Defense, 2008)

2 – UFC 3-340-02; page 1651 (Department of Defense, 2008)

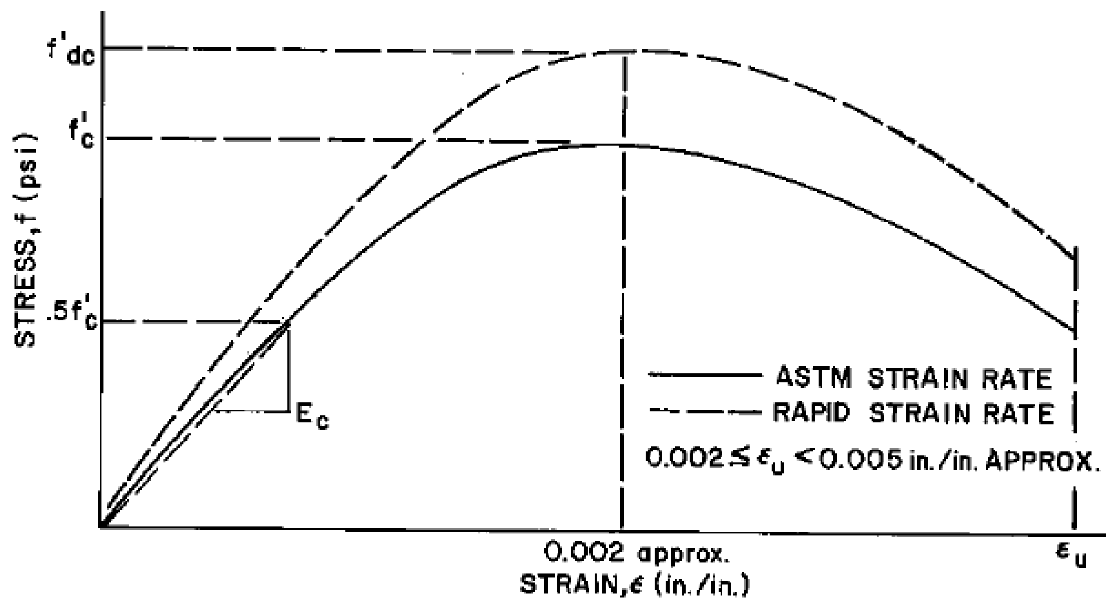

Figure 6.9: Dynamically Adjusted Concrete Stress-Strain Curve (Department of Defense, 2008)

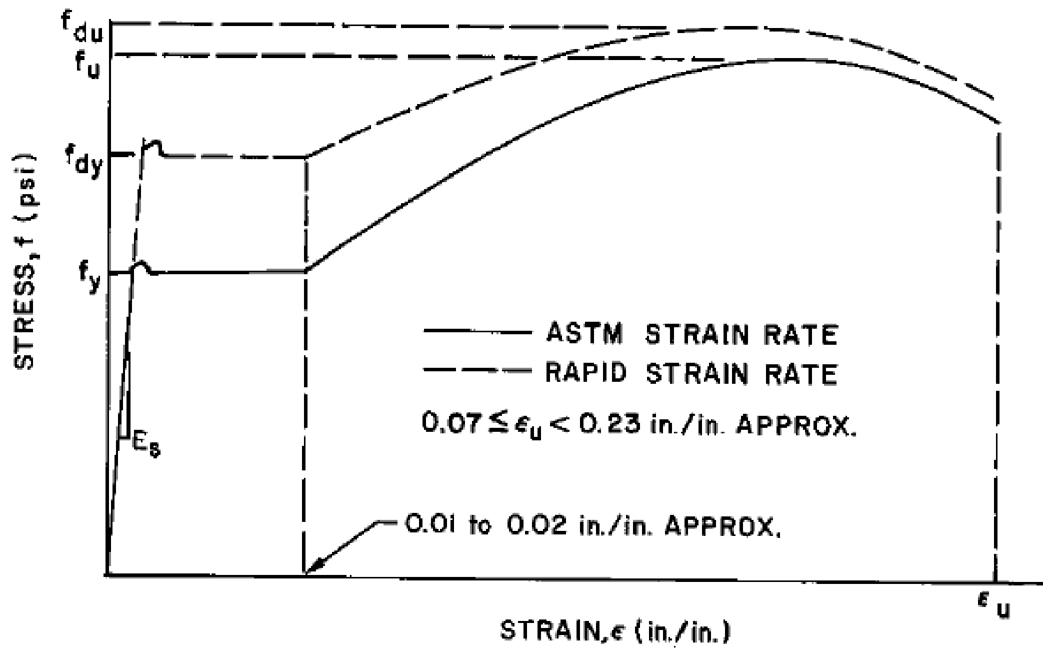

Figure 6.10: Dynamically Adjusted Mild-Steel Reinforcement Stress-Strain Curve (Department of Defense, 2008)

Layer-by-Layer Moment-Curvature Analysis

Utilizing the material models described in the previous sections, the moment-curvature relationship for the Colorado Bulb-Tee is shown in Figure 6.11 using a layer-by-layer analysis procedure. Observing Figure 6.11, the girder has a much larger capacity when bending in positive curvature than negative curvature. The section is able to avoid brittle failure because of the mild-steel reinforcement. The maximum positive moment capacity is approximately 4680 kip-ft. For negative curvature, the maximum

capacity is approximately 440 kip-ft, nearly 10 times less than the positive bending moment capacity.

To validate the accuracy of the layer-by-layer analysis, the same section was analyzed using RESPONSE 2000. RESPONSE 2000 is a program created by Bentz and Collins (2001) that is able to determine the complete load-deformation response curve for a prestressed or a reinforced concrete section. The program was verified against experimental data during its development. One of the features in RESPONSE 2000 is the ability to create a moment-curvature diagram.

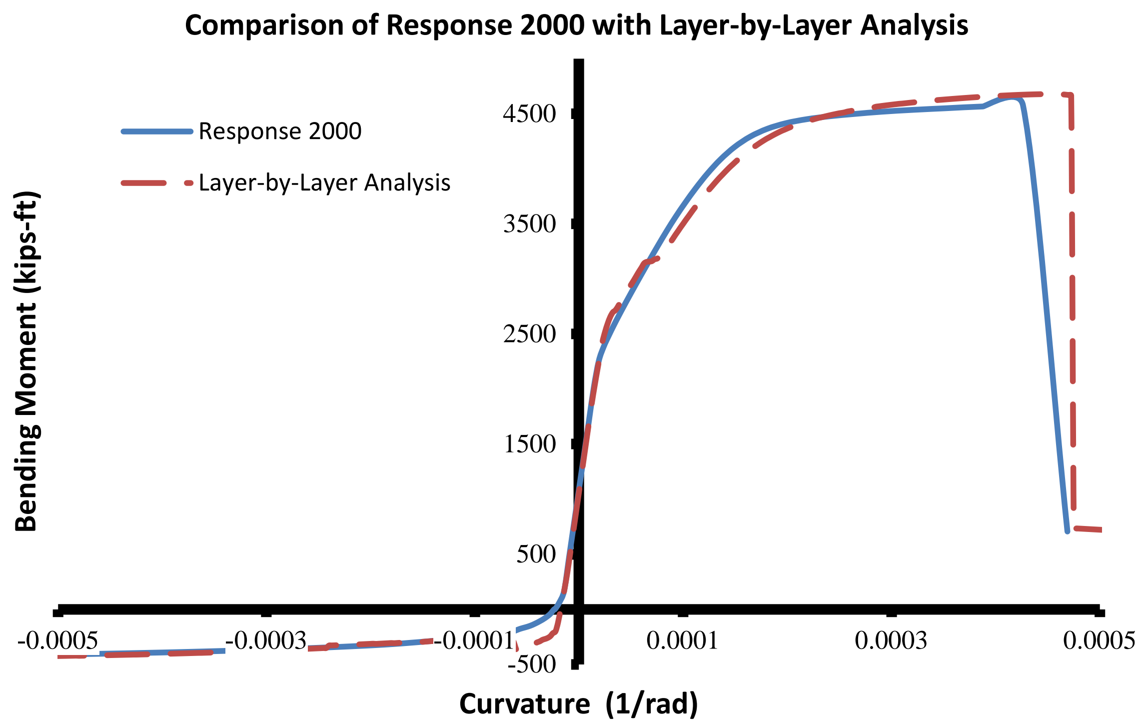

Figure 6.11: Colorado Bulb-Tee Moment-Curvature

Looking at the two sets of data, the layer-by-layer analysis developed for this research can be seen to compute the response of Colorado bulb-tee accurately with respect to RESPONSE 2000. There are minor differences in the moment-curvature response when the girder starts to behave non-linearly, but the differences are within reasonable limits.

Bilinear Moment-Curvature Diagram

For computational efficiency, after running the layer-by-layer analysis, the moment-curvature relationship needs to be reduced to a simplified bilinear expression so it can be used to develop a resistance function for SDOF analyses of blast-loaded girders. The critical points on the bilinear moment-curvature relationship are the yield curvature and the ultimate curvature. To maintain the same slope for the elastic range between positive and negative loading, the initial capacity is interpolated between the positive and negative yielding points.

The second point on the bilinear moment-curvature plot is achieved when the girder section yields. Yielding in a non-prestressed section occurs when the bottom layer of mild-steel reinforcement reaches the yield strain (for positive curvature). With the addition of prestressing strand, however, an analysis must be performed to determine which of the two reinforcement types yield first. In addition, for an over-reinforced section, it is likely that concrete crushes before either of the reinforcement types can reach its yield strain. As noted by Paulay and Priestly (1992), for a section in which the yielding curvature is controlled by the concrete strain, the compressive strain at the

extreme fiber can be assumed to be $\varepsilon_c = 0.0015$. To determine which scenario controls, the numerical procedure described in the previous section iterates through the fibers to check for yielding. When found, it saves the corresponding curvature.

The third point on the bilinear moment-curvature diagram is the ultimate curvature of the section. Ultimate curvature occurs when the mild-steel reinforcement goes beyond its fracture strain or the prestressing strand reaches its ultimate strain. For concrete, the ultimate strain is taken to be the well established value of $\varepsilon_{cu} = 0.003$ from ACI 318-08. Like the yield curvature, the program iterates through and saves the ultimate curvature and corresponding moment when one of the fibers reaches the ultimate strain.

The results of the bilinear extraction are plotted alongside the layer-by-layer moment-curvature diagram in Figure 6.12. As seen in the figure, the yield curvature of the bilinear moment-curvature relationship is the linear extrapolation of the value found in the layer-by-layer analysis up to the ultimate moment capacity. This simplified bilinear extrapolation can then be used to generate a resistance function, which is described in detail in the next section.

The bilinear moment-curvature shown in Figure 6.12 plots the additional moment capacity of the girder after the initial moments were applied. If the initial moments were included, the layer-by-layer moment-curvature would cross through the origin at zero-curvature, with the entire plot translating accordingly. Due to the simplifications made in the bilinear extraction, there will be some residual bending moments remaining at zero

curvature. For the non-linear static analysis presented later, plastic hinging is checked using a modified bilinear moment-curvature considering the undeformed girder, which includes the initial moments in the plot.

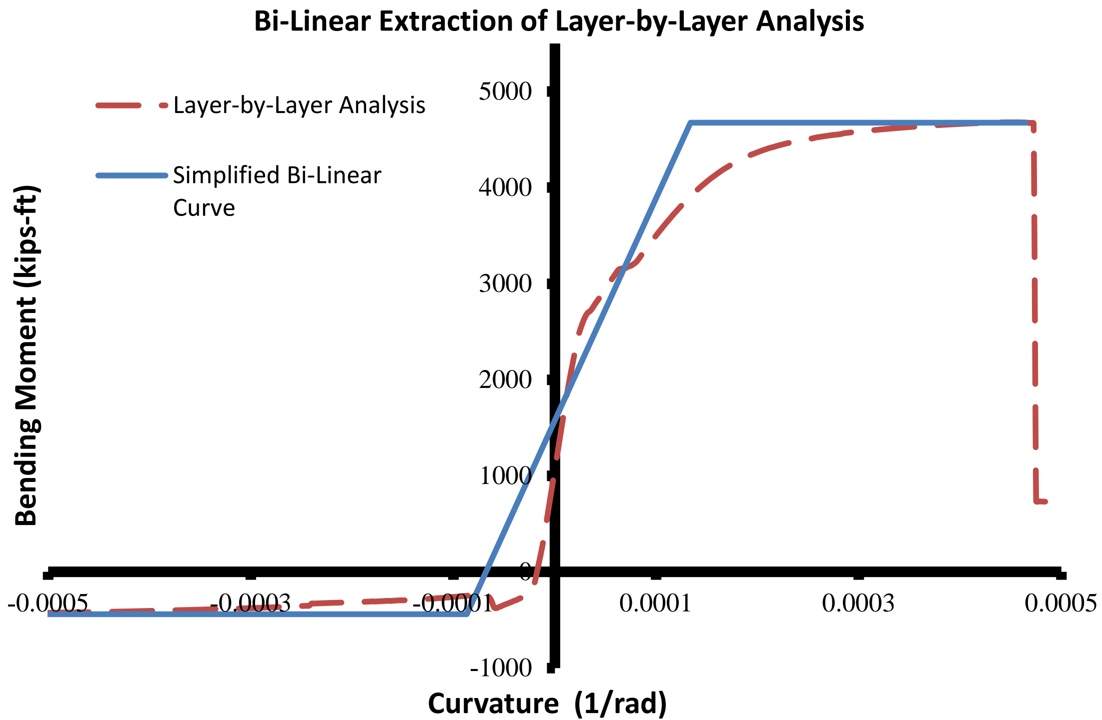

Figure 6.12: Bilinear Moment Curvature

Resistance Function

The next phase in the model development is creating the resistance function. Described by Biggs (1964) as "the internal force tending to restore the element to its unloaded position", the resistance function is used in the SDOF model as the resisting force in the dynamic analysis. In the model, the resistance function is generated by

stepping through a non-linear static analysis accounting for both geometric and material non-linearity. Unlike the previous discussion which focused on the response of an individual cross-section, the resistance function is determined by considering the overall response of a girder, accounting for load magnitude and distribution, support conditions, and internal resistance.

Static Analysis

In the numerical procedure used to determine the resistances function for a specific girder, a non-linear static analysis is performed to compute the load-deflection response. The analysis solves for displacements using Equation (6-15), where $[K_t]$ is the tangent stiffness matrix and $[P_t]$ is the global load vector for unconstrained degrees-of-freedom.

$$[K_t][\Delta_t] = [P_t] \tag{6-15}$$

The tangent stiffness matrix is the combination of the elastic stiffness $[K_e]$ and the geometric stiffness $[K_g]$, shown in Equation (6-16). The elastic stiffness matrix accounts for the linear strength of a beam element due to its physical properties. The geometric stiffness accounts for second-order P-Δ effects and depends only on the internal axial load and the length of a given beam element.

$$[K_t] = [K_e] + [K_g] \tag{6-16}$$

The global load vector is the combination of the nodal forces $[P_j]$ and the member fixed-end forces $[P_m]$. Nodal forces are concentrated loads acting at the node. Member

forces act along a beam element and are distributed to the nodes as "equivalent forces". Equation (6-17) presents the global load vector in condensed form.

$$[P_t] = [P_m] + [P_j] \tag{6-17}$$

Appendix B presents the complete elastic and tangent stiffness matrices for the beam element used in this research as well as the load vectors for a uniformly distributed load and a triangular load.

Load Spatial Distribution

The loading distribution caused by an explosive depends upon the explosive location and the charge weight as shown in Figure 6.13. Before the dynamic response of a prestressed girder begins, static forces are first applied to determine the resistance function. The dynamic analysis uses the resistance function along with the inertia of the system to compute a time-varying response. In determining the resistance function, the numerical procedure used in this study applies initial prestressing forces 'P' and the initial moment 'M'. The prestressing puts the beam elements in compression and into negative curvature. In addition to the prestressing force, the user is given the option to apply a uniform dead load 'ω_1' to account for self-weight load such as the concrete deck or rails.

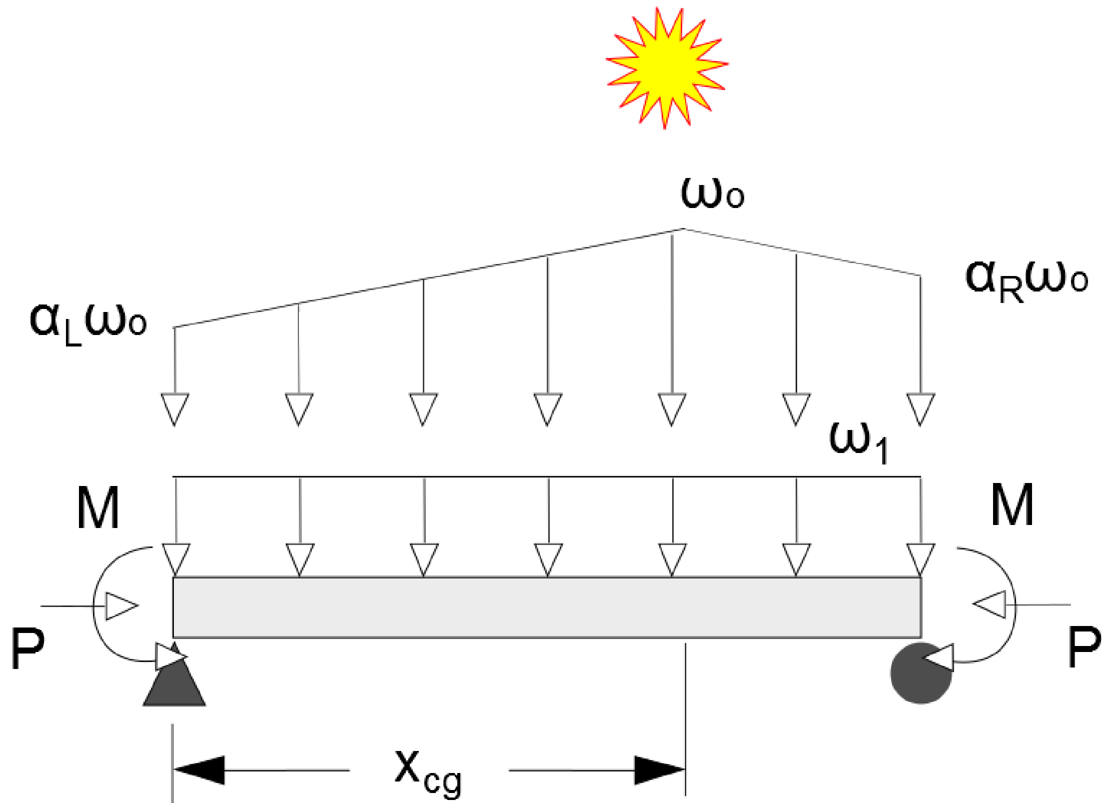

Figure 6.13: Load Distribution

The load being incremented in the model is the blast load generated from the explosive as specified in the threat scenario defined by the user. Explosives are high-rate chemical reactions that have a sudden release of energy that propagates through air as a shock wave. The shock waves create a sudden over-pressure with respect to the ambient pressure at a point in space. After the shock front passes a given point, the pressure decays rapidly until it becomes less than the ambient pressure, creating low-magnitude suction. Figure 6.14 illustrates the pressure-time history for a typical shock front.

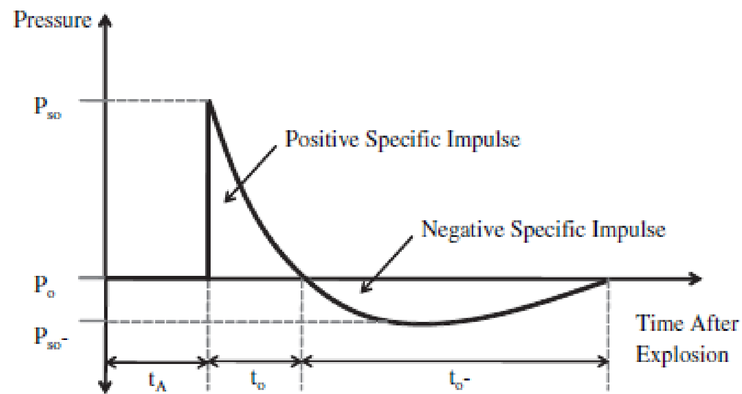

Figure 6.14: Pressure-Time Curve for Free-Air Explosion (Department of Defense, 2008)

As the shock front propagates from the center of the explosive, the pressure reduces the further it travels without any reflections. Therefore, depending on the standoff of the explosive from the target, the pressure distribution along a girder could vary dramatically for a small standoff threat or be nearly uniform for a large standoff threat. Figure 6.15 shows three distinct scenarios: scenario (a) shows a 'contact' detonation that creates a large local spike around the explosive, scenario (b) shows a 'close-in' detonation that has a varying distribution along the blast-loaded face, and scenario (c) shows a 'far-range' detonation with a 'near' uniform distribution along the blast-loaded face.

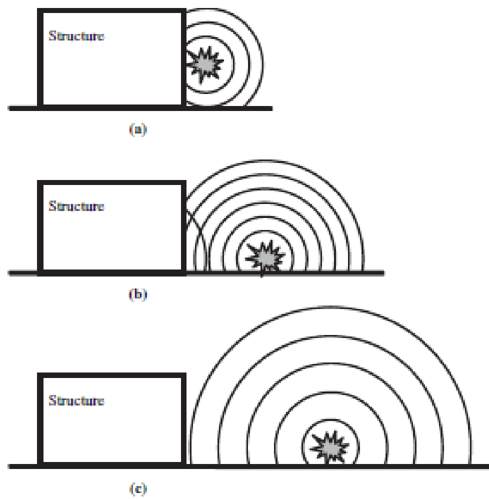

Figure 6.15: Blast Distribution Variation with Respect to Standoff (Department of Defense, 2008)

For the analysis procedure adopted for this research, the distribution along a girder is idealized as varying linearly, with the peak pressure corresponding to the center of gravity of the explosive. This distribution is able to cover both the 'close-in' detonation as well as the 'far-range' detonation. As illustrated in Figure 6.13, the end pressure is a function of the peak pressure 'ω_o'. To determine the end pressures, the positive impulse needs to be determined for the left and right edge as well at the peak location. The positive impulses are determined by summing the positive pressure time-history, generated by BEL, before the negative phase. The end distribution factors are then calculated based on the ratio of the specific impulse at the ends with respect to the peak impulse (Equation (6-18) and Equation (6-19)). This model retains the peak pressure as the remaining independent quantity to increment.

$$\alpha_L = \frac{I_{SP.Left}}{I_{SP.Peak}} \tag{6-18}$$

$$\alpha_R = \frac{I_{SP.Right}}{I_{SP.Peak}} \tag{6-19}$$

Incremental-Iterative Method

With the inclusion of geometric and material non-linearity, an incremental load stepping scheme is needed to solve for the load-deflection curve for a given girder. Instead of a single-step scheme such as the 'Forward Euler' method, an incremental-iterative scheme is utilized. At each incremental load step $\{dP_i\}$, the system response is solved iteratively to determine the equilibrium position. Figure 6.16 illustrates the method, and Equation (6-20) is solved at each increment.

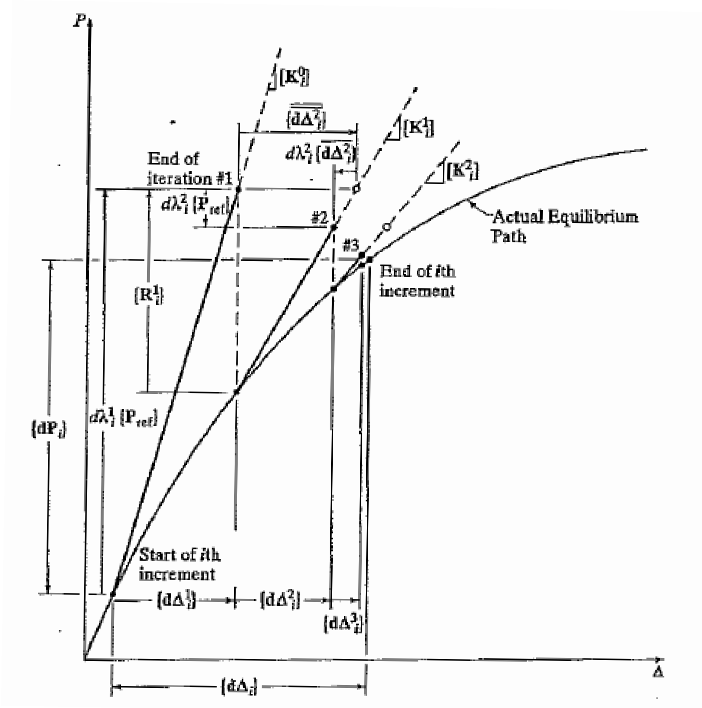

Figure 6.16: Incremental Iteration (McGuire, Gallagher, and Ziemian, 2000)

$$[\Delta_i] = [\Delta_{i-1}] + \sum_{j=1}^{m_i} \{d\Delta_i^j\} \tag{6-20}$$

The algorithm used to solve the non-linear static analysis is described in the book by McGuire, Gallaher, and Ziemian (2000). The two parameters that the method is solving for at each load increment are the load ratio factor $d\lambda_i^j$ and an incremental displacement vector $d\Delta_i^j$. The load ratio factor is used with respect to a reference load $\{P_{ref}\}$. The 'i' subscript notation on the variables denotes the load increment, whereas the superscript 'j' denotes the iterative step.

Geometric Non-linearity

At the beginning of each load increment, the geometry of the previous converged state $\left[K_i^{j-1}\right]$ is used to solve the system response at the next iteration. Equation (6-21) is used to solve for the reference displacement vector $\left\{\overline{d\Delta_i^j}\right\}$ applying the reference load to the previous converged state. Equation (6-22) determines the residual load vector $\left\{R_i^{j-1}\right\}$, which is the difference between the external load vector $\left\{P_i^{j-1}\right\}$ and the internal load vector $\left\{F_i^{j-1}\right\}$. For the first iteration, the residual load vector is set to '0'. With the residual load vector, the residual displacement vector $\left\{\overline{\overline{d\Delta_i^j}}\right\}$ can then be found using Equation (6-23).

$$\left\{P_{ref}\right\} = \left[K_i^{j-1}\right]\left\{\overline{d\Delta_i^j}\right\} \tag{6-21}$$

$$\left\{R_i^{j-1}\right\} = \left\{P_i^{j-1}\right\} - \left\{F_i^{j-1}\right\} \tag{6-22}$$

$$\left\{R_i^{j-1}\right\} = \left[K_i^{j-1}\right]\left\{\overline{\overline{d\Delta_i^j}}\right\} \tag{6-23}$$

The next step in the analysis is determining the load ratio factor. The initial load ratio factor $d\lambda_i^1$ is determined by using Equation (6-24), which is an automated approach described in the text by McGuire, Gallagher, and Ziemians (2000). That approach uses a scaled parameter that is multiplied with the previous initial load ratio $d\lambda_{i-1}^1$. The scale parameter is the ratio of the desired number of increment 'N_d' over the previous number of increment 'N_{i-1}' raised to the exponential parameter 'γ'. The exponential parameter is set at 0.5.

$$d\lambda_i^1 = \pm d\lambda_{i-1}^1 \left(\frac{N_d}{N_{i-1}}\right)^{\gamma} \tag{6-24}$$

For $j \geq 2$, the load ratio factor is determined using Equation (6-20)—the 'Constant Arc Length' method. The method is implemented because it is able to account for limit points as well as for sharp changes in the load-deflection response that may occur due to plastic hinging.

$$d\lambda_i^j = -\frac{\{d\Delta_i^1\}^T \left\{\overline{\overline{d\Delta_i^j}}\right\}}{\{d\Delta_i^1\}^T \left\{\overline{d\Delta_i^j}\right\} + d\lambda_i^1} \tag{6-25}$$

Once the load ratio is determined, the displacement for the iteration is calculated using Equation (6-26). With the displacement vector calculated, the system is then checked for convergence.

$$\left\{d\Delta_i^j\right\} = d\lambda_i^j \left\{\overline{d\Delta_i^j}\right\} + \left\{\overline{\overline{d\Delta_i^j}}\right\} \tag{6-26}$$

The convergence criterion is checked by calculating the Modified Euclidean norm, shown in Equation (6-27). This value is computed using a normalized ratio with the kth iterative displacement '$d\Delta_k$', the largest total translational displacement 'Δ_{ref}', and the number of unknown displacements 'N'. The acceptable tolerance used is 10^{-4}.

$$\|\varepsilon\| = \sqrt{\frac{1}{N}\sum_{k=1}^{N}\left(\frac{d\Delta_k}{\Delta_{ref}}\right)^2} \tag{6-27}$$

Material Non-Linearity

Once the load ratio is determined, displacements are calculated at the beam element nodes. Internal forces are checked to determine whether yielding has occurred. If the load ratio indicates the plastic moment capacity has been exceeded, the load ratio is proportioned downward and the increment is then recalculated with the new load ratio. If the load ratio is determined to fall within the tolerance, a plastic hinge is placed in the appropriate beam element. In the next increment, the structure will be analyzed with the new configuration.

GENERALIZED SINGLE-DEGREE-OF-FREEDOM SYSTEM

After the development of the resistance function, the model is now able to solve a generalized SDOF system. A single-degree-of-freedom model is chosen over a multi-degree-of-freedom model because it is computationally more expedient while still maintaining an acceptable level of accuracy. As stated in Conrath, et al. (1990), as long as the mode of response is well understood, a SDOF model is an "effective and efficient method of accounting for the transient nature of the blast load."

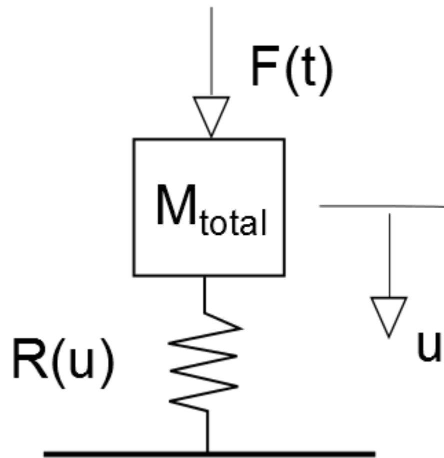

Figure 6.17: Equivalent Single-Degree-of-Freedom System

Equation (6-28) is the governing equation of motion as described by Biggs (1964) to compute the displacement of the system as a function of time. In the differential equation, the two terms that need to be solved are the acceleration term $\frac{d^2}{dt^2}u(t)$ and the displacement term embedded in the resistance function $R(u)$. The terms modifying the acceleration are the load-mass factor $K_{LM}(u)$ (presented in the next paragraph) and the total mass of the system M_{total}. As stated previously, the resistance function is the restoring force and depends non-linearly on the deformation of the system. The damping term is neglected in this model because of its negligible contribution (Conrath, et al., 1990). On the right-hand side of the equation is the forcing function $F(t)$, explained in the next section.

$$K_{LM}(u)\, M_{total} \frac{d^2}{dt^2} u(t) + R(u) = F(t) \tag{6-28}$$

The load-mass factor shown in Equation (6-29) is the ratio of the mass factor $K_M(u)$ relative to the load factor $K_L(u)$. These factors are used to transform the real system into an equivalent SDOF system. The mass factor relates the actual distributed inertial resistance to the idealized mass, by equating the kinetic energy of the two systems. Likewise, the load factor relates the total loading distribution in the actual system with the idealized load, by equating the work done by the two systems. Additional details for computing these factors can be found in Biggs (1964).

Both factors use the shape function $\phi(x, u)$, which is the deformation mode shape from a static application of the applied blast load distribution. Equation (6-30) and Equation (6-31) transform the mass and load into a single parameter. The load-mass factor is calculated at each load step during the development of the resistance function. More information can be found from Biggs (1964).

$$K_{LM}(u) = \frac{K_M(u)}{K_L(u)} \tag{6-29}$$

$$K_M(u) = \frac{\int_0^L m(x,u)\phi(x,u)^2\, dx}{\int_0^L m(x,u)\, dx} \tag{6-30}$$

$$K_L(u) = \frac{\int_0^L f(x,u)\phi(x,u)\, dx}{\int_0^L f(x,u)\, dx} \tag{6-31}$$

Forcing Function

Due to the short-duration blast loading that is being considered, it acts over a much shorter time than the period of the girders typically found in practice. As such, the loading is impulsive, and the model forcing function seeks to preserve the total impulse acting on the structure. Equation (6-32) defines the forcing function as the product of maximum force F_{max} and an exponentially decaying function. The maximum force is determined by using the peak pressure at the center of gravity with the loading distribution utilized for the non-linear static analysis. An exponentially decaying function is used to capture the loading with respect to time for a typical blast load, and it conservatively ignores the negative phase of the blast load.

$$F(t) = F_{max}\left[e^{-t/t_0}\right] = \int_0^L A(x)\, p_{max}(x) dx \left[e^{-t/t_0}\right]$$
$$= b_w p_{CG.max}\left(l_L\left[\frac{(1+\alpha_L)}{2}\right] + l_R\left[\frac{(1-\alpha_R)}{2}\right]\right)\left[e^{-t/t_0}\right] \tag{6-32}$$

where,

b_w = width of the girder

$p_{CG.max}$ = maximum pressure at C.G.

l_L = length of the girder to the left of the explosive

l_R = length of the girder to the right of the explosive

The only parameter that is not defined yet in Equation (6-32) is t_0, the equivalent time duration. This variable is found by preserving the impulse, a measure of loading

intensity. Equation (6-33) calculates the equivalent uniform impulse EUI_{sp} by normalizing the specific impulse with the elastic shape function $\phi(x)$. Equation (6-34) calculates the equivalent uniform pressure similarly to the equivalent uniform impulse. The impulse and pressure are determined at the end points and explosive center-of-gravity by summing the pressure time-history from BEL. Afterwards, the impulses for the remainder of the girder nodal locations are interpolated from those points. Finally, with the equivalent uniform impulse and the equivalent peak pressure, t_0 can be calculated using Equation (6-35).

$$EUI_{SP} = \frac{\int_0^L I_{SP}(x)\phi(x)\,dx}{\int_0^L \phi(x)\,dx} = EUP\; t_0 \tag{6-33}$$

$$EUP = \frac{\int_0^L P_R(x)\phi(x)\,dx}{\int_0^L \phi(x)\,dx} \tag{6-34}$$

$$t_0 = \frac{EUI_{sp}}{EUP} \tag{6-35}$$

Solving the Equation of Motion with Newmark-beta Method

Equation (6-28) cannot be solved with a closed form solution because of the irregular loading and the non-linear behavior of the resisting function. Therefore, the model for this research utilizes the Newmark-Beta average acceleration scheme to solve the equation of motion (Chopra, 2006). The first step at each load increment is to determine the resisting force and the instantaneous load-mass factor from the resistance function. The second step is to determine the total applied force on the system at that

time step. The next step is to solve for displacement at the next time step. The last step is to solve for velocity and acceleration at the current time step. Appendix E presents the detailed steps needed to solve the differential equation of motion using the Newmark-beta method. Additional information on this numerical procedure can be found in the book by Chopra (2006) or Tedesco, McDougal, and Ross (1990).

EXAMPLE OF GIRDER MODEL

To demonstrate the SDOF model, the response of an explosive placed at mid-span is explored for the Washington State Bulb-Tee girder. The explosive is located at a scaled standoff distance of $Z = X.XX\ ft/lb^{1/3}$ above the deck. The physical standoff and the charge weight were chosen to ensure that inelastic flexural response will occur without causing local failure. For comparison, a finite element model was developed by Hendryx (2012) to analyze the girder with the same assumed loading.

Girder Single-Degree-of-Freedom Model

With the physical geometry of the girder established, pressure-time histories are developed from BEL for points at the center-of-gravity location and at the girder end (Figure 6.18). The solver then calculates the peak pressures, peak specific impulse, and load distribution factor used in establishing the loading distribution. The values are given in Table 6-3.

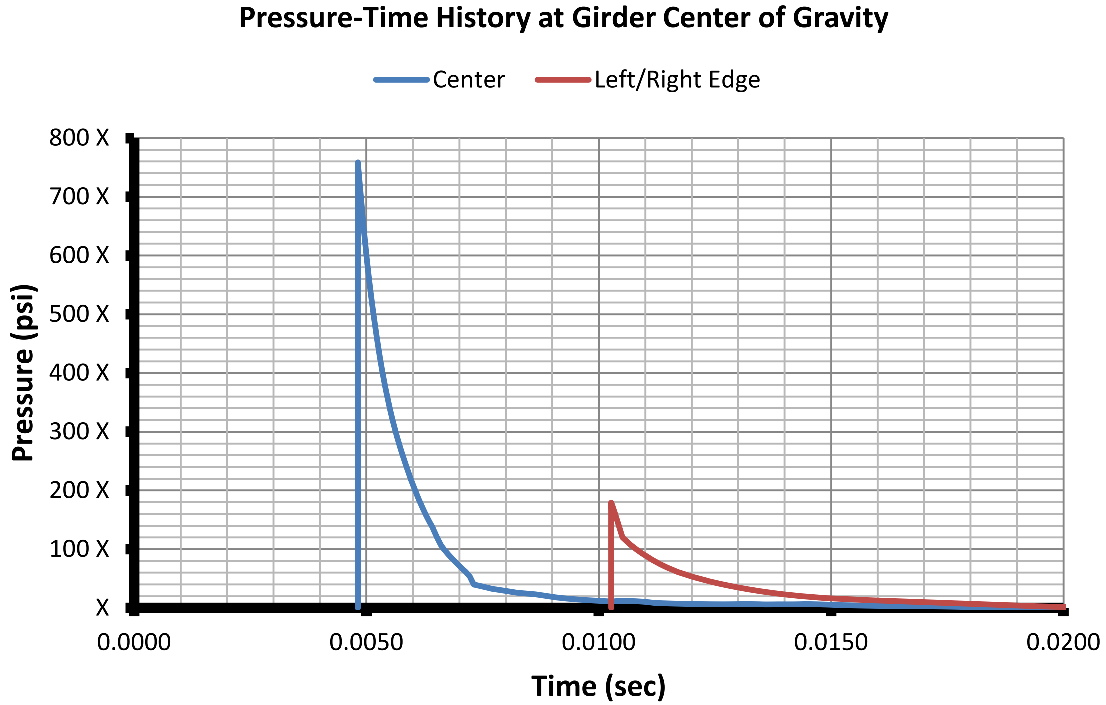

Figure 6.18: Pressure-Time History at Girder Center and Left/Right Edge

Table 6-3: Loading Parameters

Peak Pressure at CG	$P_{Peak.CG} = XXX.X\ psi$
Peak Pressure at Girder End	$P_{Peak.End} = XXX.X\ psi$
Specific Impulse at CG	$I_{SP.CG} = 0.XXX\ psi\text{–}s$
Specific Impulse at Girder End	$I_{SP.Ends} = 0.XXX\ psi\text{–}s$
Load Distribution Factor at Girder End	$\alpha_{end} = 0.389$
Equivalent Uniform Pressure	$EUP = XXX.0\ psi$
Equivalent Uniform Specific Impulse	$EUI_{SP} = 0.XXX\ psi\text{–}s$
Time Constant	$t_o = 0.000765\ s$
Maximum Force	$F_{max} = XXXXX.X\ kips$

With the loading distribution determined, a non-linear static analysis is then performed for the girder. The girder is sub-divided into 30 beam elements with all elements having the sectional properties shown in Figure 6.19. Before the loading increment begins, the girder is first loaded statically with an initial axial compressive force $P = 724.6\,\text{kips}$ and an initial moment $M = -13{,}267$ kip-in. An additional uniformly distributed load $\omega = 0.056$ kip/in is applied to account for self-weight.

With the combined loading, the girder deflects in reverse curvature with a camber at mid-span of 2.4 in. The girder is then incrementally loaded until it fails. The resistance function is then determined by storing the mid-span deflection and total applied incremental load at each load step. Figure 6.19 shows the mid-span resistance function. As expected, when the resistance function deflects in negative curvature, it has substantially less capacity then when it deflects in positive curvature. Once the girder hinges, it behaves plastically until it fails.

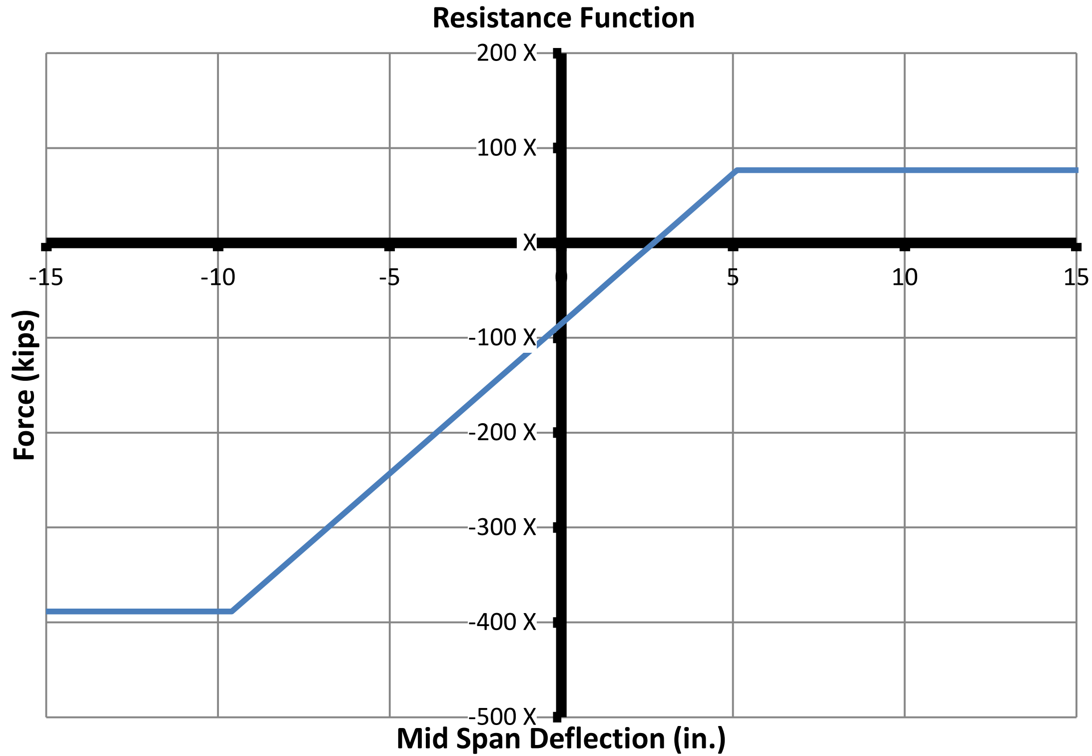

Figure 6.19: Resistance Function

In Figure 6.19, the negative applied load—the resultant force from the distributed blast load—corresponds to load applied over the deck with the loads acting downward. The deflection therefore will also be downward and in positive curvature. For the positive applied load, the reverse scenario exists.

After the non-linear static analysis, the deflection at each load step is used to generate the shape functions and the load-mass factors. To solve for the equivalent uniform specific impulse and the equivalent uniform pressure, a shape function at an early load increment is used. Using Equations (6-32), the forcing function can be seen in

Figure 6.20 with all the forcing parameters presented in Table 6-3. The forcing function shows an applied downward load on the structural system that decays at approximately 0.004 seconds.

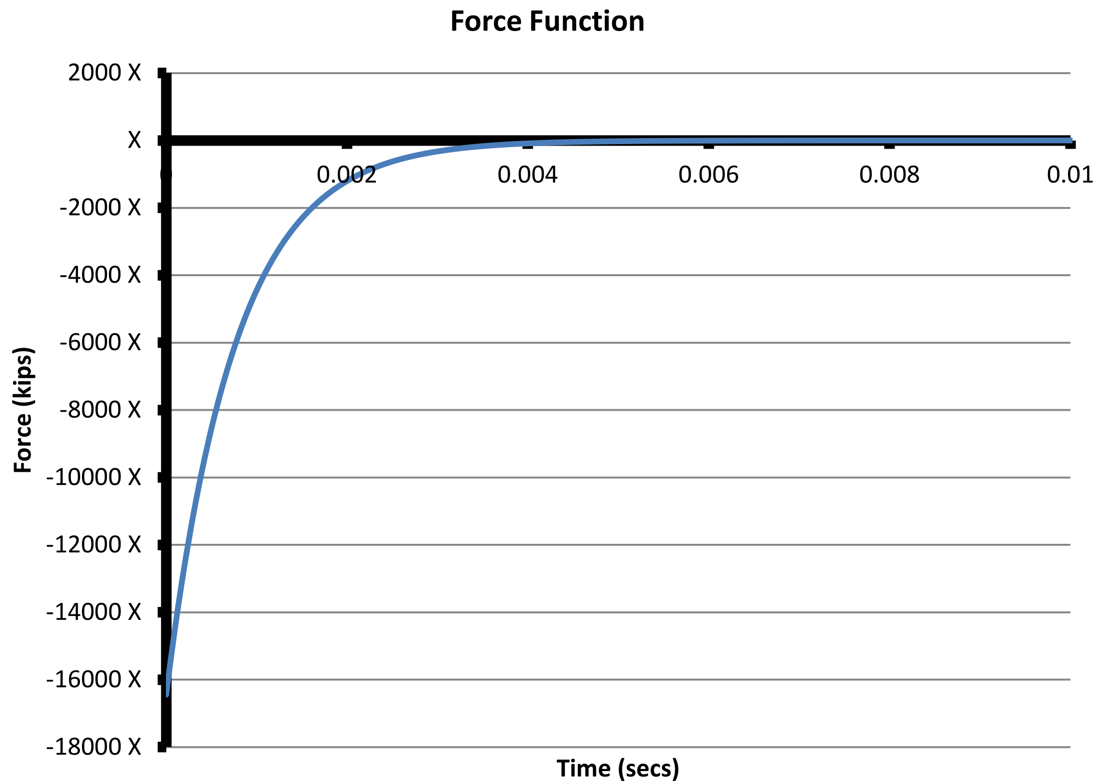

Figure 6.20: Forcing Function

With the resistance function, the load-mass factor, and the forcing function, the SDOF model is then solved. Figure 6.21 shows the displacement, velocity, and acceleration time-history of the girder model. During the early stage of the loading, the girder deflects downward up until it reaches approximately -5.0 in. When it hits its peak

negative deflection, the girder still remains elastic. Upon rebounding, the girder reaches plastic deformation at approximately 5 in. and deforms plastically until it reaches 13.3 in.

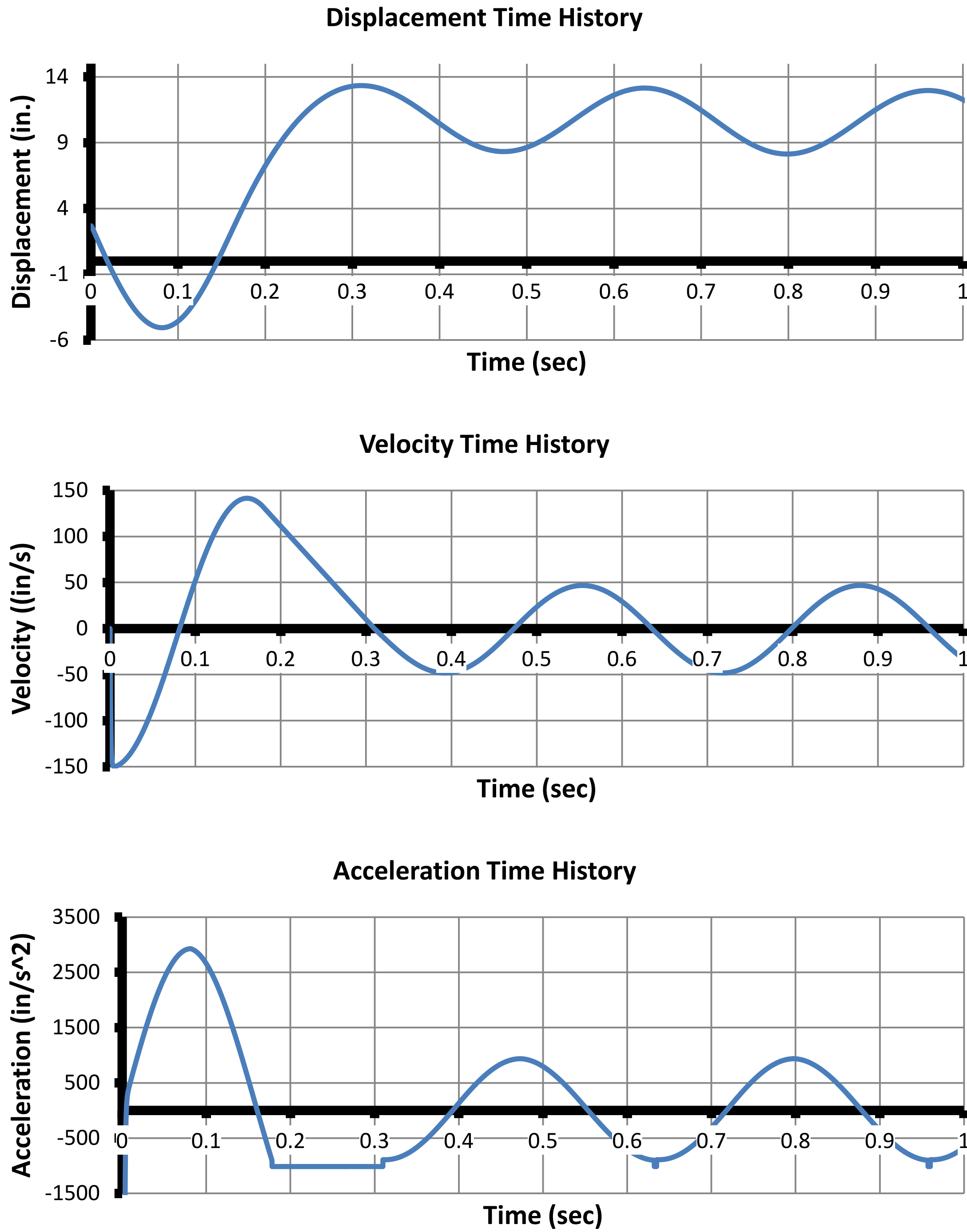

Figure 6.21: Displacement/Velocity/Acceleration Time History at the Mid-Span.

Comparison of SDOF Model with FEM Model

The FEM model used for validation was constructed in LS-DYNA running an explicit analysis. The details of the model are reported in Hendryx (2012). Figure 6.22 shows FEM results of the mid-span deflection with respect to time. Prestressing force was applied initially, and then the blast load was applied 1.0 second later (after equilibrium from the prestressing force was achieved). The peak negative deflection was computed to be approximately 5.0 in., while the girder rebounds up to a deflection of 4.14 in.

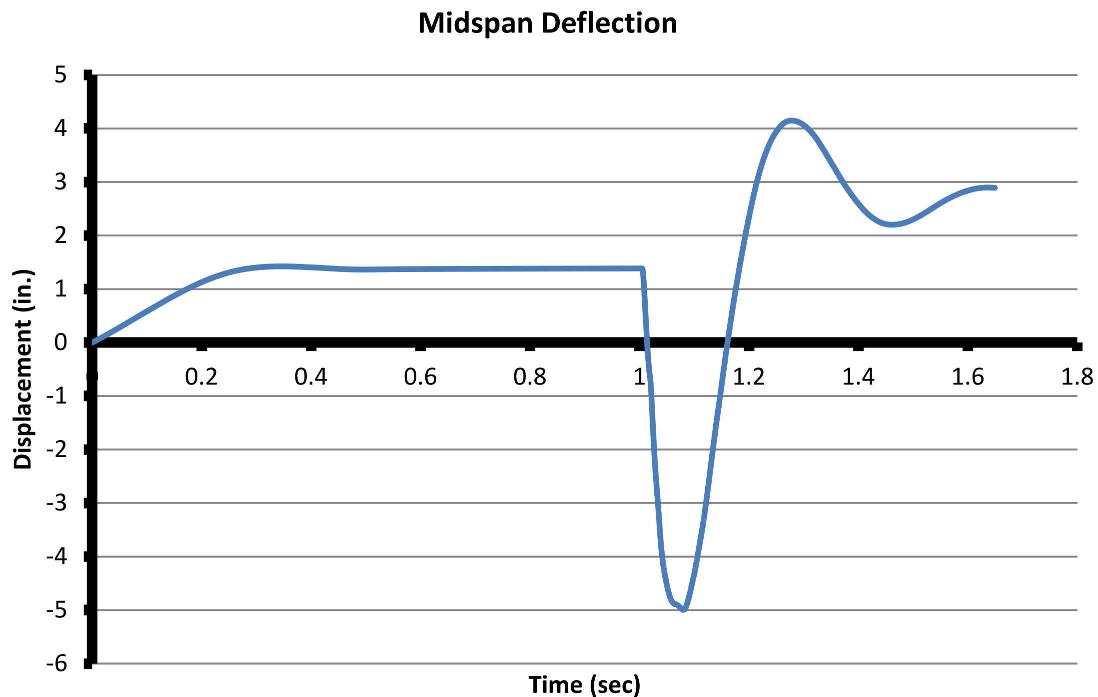

Figure 6.22: Girder FEM Displacement-Time History at Mid-Span

Compared with the FEM model, the SDOF model predicts the peak negative displacement accurately but grossly overestimates the rebound displacement. The camber determined for the FEM model returns approximately 1.4 in., as opposed to the 2.4 in. determined by the SDOF model. The difference between the two models is because the SDOF model uses a reduced stiffness generated from the moment-curvature diagram, which leads to over-predicting the camber—which in actuality remains in the elastic range.

At this time, a great amount of uncertainty still exists for the two models. One interesting area of concern is that the SDOF model remains elastic during the early phase of response, whereas the FEM model shows some plasticity. This difference might be attributed to the simplification of the sectional property of the model, where the reduced stiffness allows the SDOF model to remain elastic for a larger deflection.

For the rebound displacement, the SDOF model vastly over-predicts the displacement because the capacity of the girder in negative curvature is so weak; once the girder hits the plastic region it starts deforming plastically. These large differences between the two models indicate that the SDOF model and the FEM model have some discrepancies that require further study. Refinement of the models is expected to occur outside of the work presented in this thesis.

SUMMARY

This chapter includes an example of a solver: the prestressed girder model. The girder model determines a moment-curvature response curve and a corresponding resistance function for use in dynamic analyses of an SDOF system. For comparison, an FEM analysis with the same load case was modeled. Although the SDOF model trends with the FEM results, further refinement is necessary to provide a more reliable model. The next chapter summarizes this thesis and gives recommendations for future research.

Chapter 7: Summary, Recommendations, and Conclusions

During the last decade following September 11, 2001, government officials and the engineering community have devoted time and resources to protect the country from such attacks again. Because the highway infrastructure plays such a critical role in the public's daily life, research was conducted on various bridge components to determine their resiliency against explosive attacks. While more tests are needed, it is now time to transfer the research into tools to be used by the design community.

SUMMARY OF RESEARCH PROGRAM

The US Department of Homeland Security sponsored the research described in this thesis with the primary goal of creating a user-friendly PC software that analyzes the effects of explosives on bridge components. The software is designed to be a clearinghouse of previous research, incorporating numerical models validated against experimental data. The program is intended to be fast-running an easy to use. The target audience is design engineers, but it can still be used by emergency responders in planning for such attacks (Sammarco, 2012).

This thesis explains in detail how ATP-Bridge was developed to address the primary project objective. It describes how to develop software that is both user-friendly and expedient, yet still able to account for different bridge components with different modes of failure. The challenge was addressed by using object-oriented programming principles—specifically *inheritance*—to normalize common features while still giving

developers enough flexibility to modify their data structures and graphical components to their specific bridge components.

To create this multi-component analysis software, ATP-Bridge requires each bridge component to have separate data structures, graphics components, and solvers. These three parts embody different functions of the bridge component: the data structure is used to store the information from the user, the graphics component is used to render the information, and the solver analyzes the information.

There are three major data structures: the ***Structural Component***, ***Load***, and ***Nexus***. The ***Structural Component*** stores all the physical attributes of the bridge component, such as geometries, materials, and boundary conditions data. The ***Load*** stores all the loading data, both explosive loading and external static loading. The ***Nexus*** processes the data from the previous two data structures and connects it with the solver to analyze.

The graphical user interface (GUI) connects the user with the back-end of the program: the data structures and solvers. The GUI has multiple components that are used to communicate with the user: ***Menu Item Control***, ***Quick Icon Control***, ***Navigation Control***, and ***3D Rendering Viewer***. The first three components are different ways to trigger the user's commands through various forms of Windows-based controls. The last component, ***3D Rendering Viewer***, is used to render the user's commands and display graphics in 3D.

Finally, an example of a solver (prestressed girder with advanced SDOF analysis model) is presented to illustrate a fast-running algorithm and attempts at its validation. The SDOF model incorporates the development of a moment-curvature response curve created by a layer-by-layer analysis, a non-linear static analysis accounting for both geometric non-linearity as well as material non-linearity, and a Newmark-beta-based SDOF analysis. The model is then compared with an FEM model developed by Hendryx (2012). While the initial response between both models was shown to be in good agreement, the overall differences between the two cases require further study.

SNAPSHOTS OF THE ANTI-TERRORIST BRIDGE PLANNER

Taking the concepts developed and illustrated in the thesis, the development team at UT Austin has been working on many overall features of the software, including geometry and loading forms, the graphics components, and post-processing the results. Some snapshots are provided below for the prestressed girder.

Figure 7.1 shows a snapshot of the geometry form with the cross-section displayed. Geometry forms are designed to have all the necessary information pertaining to the physical parameters such as dimensions, material properties, and boundary conditions. The form uses *tab pages* at the top to switch between major categories; for the prestressed girder the categories are 'Section', 'Elevation', and 'Material'. By defining all the geometry information in one form, error checking is simplified. If the user inputs any incorrect value or values beyond the model's limitation, the form will be able to prevent the user from storing to the ***Structural Component*** data structure. Under

such conditions, the program gives the user instant feedback where the error occurred. This programming approach adds a layer of data security inside the program.

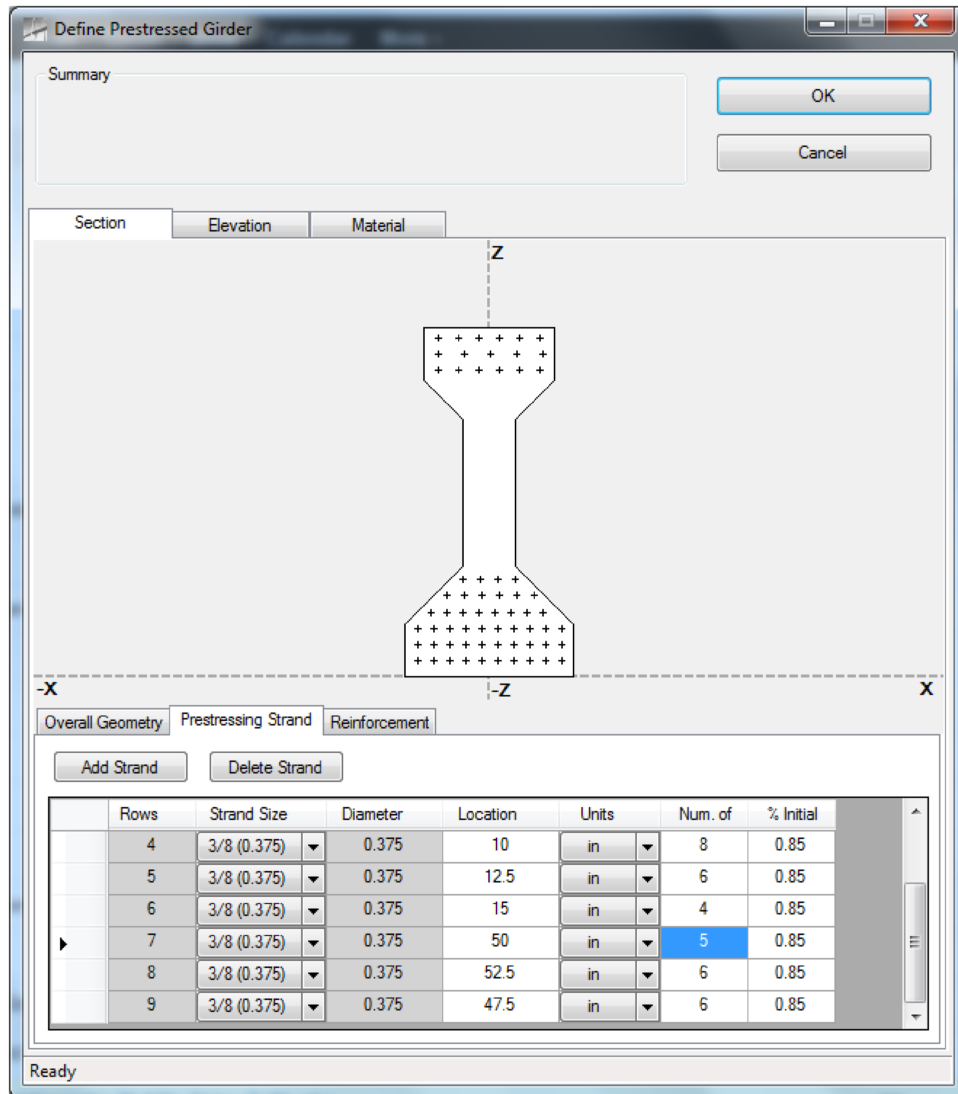

Figure 7.1: Geometry Form for Prestressed Girder

Figure 7.2 shows a snapshot of the load form. The form is used to define parameters containing the loading conditions including not only the explosive threat scenario but also the static loading associated with dead loads. The center of the form shows the explosives relative to the girder model, scaling both the explosives and the girder graphics. The parameters used to define the explosive threat scenario are the charge weight, shape, and explosive type (populated with options allowed in BEL).

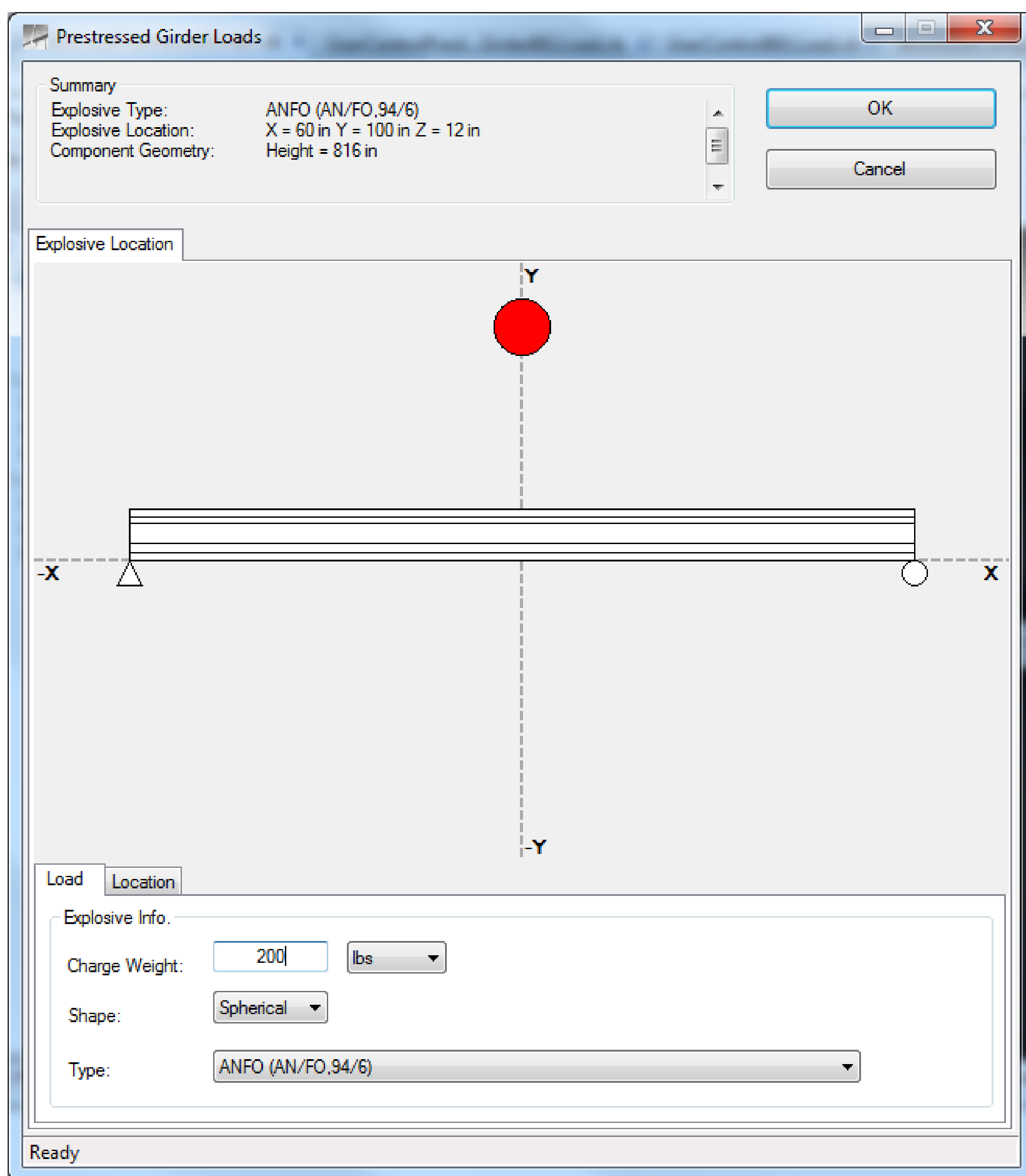

Figure 7.2: Load Form for Prestressed Girder

Figure 7.3 is a snapshot of the ATP-Bridge main form. At the center of the form is the ***3D Rendering Viewer***, showing an isometric view of the prestressed girder being analyzed. At the top of the form is the traditional ***Menu Item*** control with the ***Quick Icon*** control just below it. At the far left-hand side of the form is the ***Navigation Control***. Inside of it is the ***Tree-View*** control, showing the different types of bridge components inside the project.

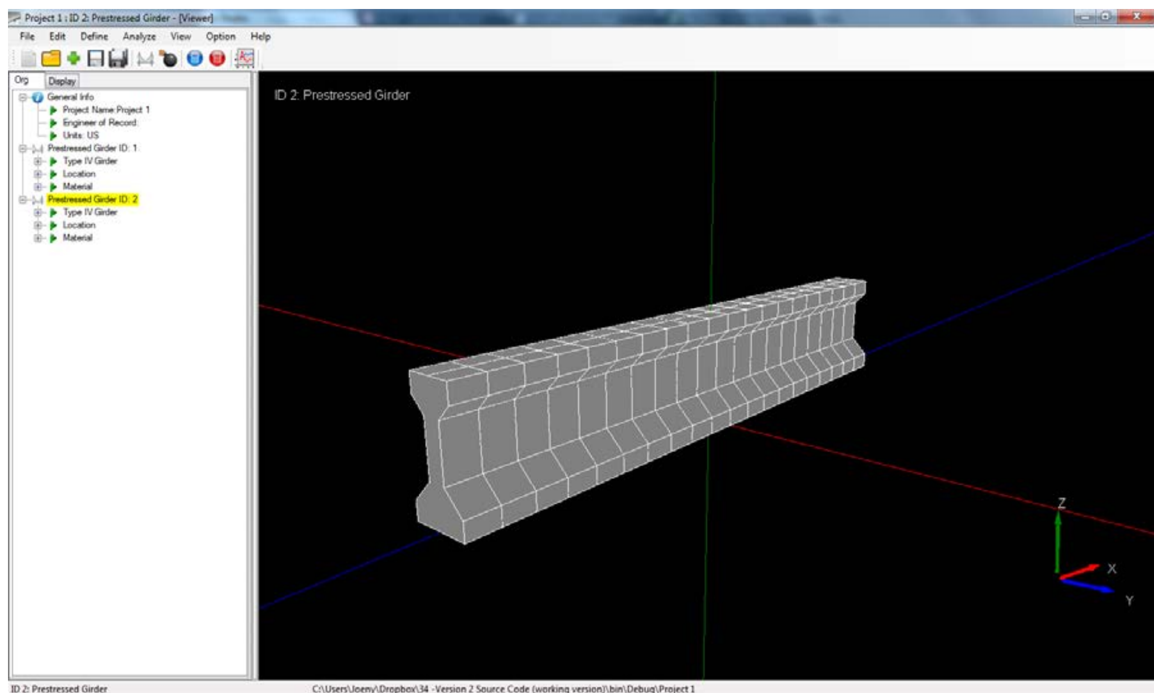

Figure 7.3: Graphics Engine Rendering of Prestressed Girder

RECOMMENDATIONS FOR FUTURE WORK

As ATP-Bridge continues to progress, there are some areas that are worth exploring to create a more efficient program. One area that could expedite the design

cycle for design engineers is allowing ATP-Bridge to be able to look at an entire bridge system and not exclusively a single component. For example, currently in the software when analyzing a prestressed girder bridge the user is required to define separately multiple reinforced concrete columns and prestressed girders as well as multiple loads for each component. Another example of this is the steel tower, requiring the user to define multiple interior cells and corner cells when analyzing a single tower.

If the program is capable of defining the bridge system globally, this allows the user to more efficiently utilize their time and also minimize user-input errors. To achieve this in ATP-Bridge, it is recommended that an additional layer to the software be developed. This top layer would define the global geometry of the bridge system that will then automate single bridge components and explosives for analysis.

Another area of concern in ATP-Bridge is the file output. Currently, the software generates multiple files in 'txt' extension format. Although this method is adequate, there are a series of issues concerning program robustness and security. First, the files are output to an external file from the solver and then read back into the software. This creates vulnerability in the software, where the file can be corrupted or where content can be changed outside the software before the files are input back into the program. Another concern is that 'txt' files are in ASCII format, which can be opened from any notepad program and read by unintended users.

The recommended solution to address these issues is the implementation of a database. A database can store the data into one file and maintain a single file throughout

a project. Also, with some databases, it is possible to encrypt the file with a password to keep it from intrusion.

RECOMMENDATIONS ON PRESTRESSED GIRDER SOLVER FOR FUTURE WORK

With the prestressed girder model, there are a series of simplifications that were made with respect to Sammarco's reinforced concrete column model that need to be explored further. The first omission is the effects of dynamic shear. Most of the research currently for dynamic shear looks at traditional reinforced concrete, not prestressed concrete. There is not enough information in the open literature at this time, and it is recommended that testing be done to study this potential response mode in prestressed girders.

Another behavior that is of interest is confinement effects, where concrete experiences an increase in strength and ductility when it is confined with transverse reinforcement. Confinement effects in traditional reinforced concrete have been well established. The most widely used model is explained in Mander's landmark paper (1989), which considers a wide range of parameters including effective lateral confining stress and transverse reinforcement spacing. Recently, Ross et al. (2012) observed an increase in ductility with an increase in confinement steel for prestressed girders. Another test done by Patzlaff, et al. (2012) led to similar conclusions about the increase in ductility; however, the researchers observed no noticeable increase in flexural capacity. It is recommended that future work consider confinement effects in the development of the moment-curvature response curve for prestressed girders.

Although the girder specimen used in the Washington State research had strands positioned at a constant elevation, it is common in practice to find prestressed girders that have draped or harped strands for long spans. Draping or harping is used to distribute the eccentric internal prestressing along the length of the girder with near zero eccentric moment at the ends and maximum eccentric loading at the mid-span. One advantage delivered from varying the eccentricity is eliminating tensile stress at the top flange near the supports. Another advantage is the reduction in the number of strands required because of vertical forces produced from the prestressing (Nawy, 2003). It is recommended that in constructing the resistance function, different section properties are used along the length of the girder to account for draping or harping.

In ATP-Bridge, the prestressed girder excludes the strength of the deck when calculating the strength because the specimen tested by Washington State was a bare girder. For typical design, the deck is accounted for in the flexural response calculations of prestressed girders. Therefore, it is recommended that the deck be included in the analyses and that the predicted response be validated against either new experimental data or detailed FEA models.

In BEL, pressure-time histories do not account for non-plane shapes. Because of this limitation, shape factors were proposed by Williams (2009) for analyzing blast-loaded bridge columns that take into account the effects of clearing and column engulfment. For prestressed girders, clearing effects will depend upon the loading scenario being above-deck or below-deck. In the above-deck scenario, the blast pressure

will be distributed across the bridge deck, which will then induce load into the girders. In the below-deck scenario, the effects of the blast loading will first interact with the I-shape girder before it engages the deck. In addition, the effect of blast reflection off the deck and girders is not well understood. Therefore, more work needs to be done to understand the local effects of the blast loads around prestressed girders.

Finally, a more refined assumption is needed to characterize the load distribution along the girder length. In the girder model, a simplification of a trapezoidal loading is used to account for the explosive, as proposed by Sammarco et al. (2012) for his column model. But the trapezoidal loading assumption might be overly conservative for long girder spans, where loading along the girder could be more localized as reported by Gannon et al. (2006). One recommendation is to track more pressures along the length, therefore getting a more refined load distribution.

CONCLUSION

In conclusion, the development of ATP-Bridge, a program intended to be used by bridge engineers and planners to investigate terrorist threats against bridges, is explained in this thesis. The overall project goal was to build a program that can incorporate multiple bridge components while still maintaining a simple, user-friendly interface. This goal was achieved by balancing three core areas: constraining the graphical user interface to similar themes across the program, allowing flexibility in the creation of the numerical models, and designing the data structures using object-oriented programming concepts to connect the GUI with the numerical models.

ATP-Bridge is the first software developed that incorporates multiple bridge components into one user-friendly engineering tool for protecting bridge structures against terrorist threats. The software is intended to serve as a synthesis of state-of-the-art knowledge, with future updates made to the program as more research becomes available. In contrast to physical testing and high-fidelity finite element simulations, ATP-Bridge uses less time-consuming, more cost effective numerical models to generate dynamic response parameters and damage estimates. With this tool, engineers and planners will be able to safeguard the nation's bridge inventory and, in turn, reinforce the public's trust.

Appendix A: Programming Glossary

Boolean – a binary variable, having two possible values of either true or false.

Cast Type –explicitly converting an expression to a specified data type, object, structure, class, or interface.

Class – collection of data types and methods that prescribes to object-oriented principles of encapsulation, polymorphism, inheritance, etc.

Class-Tree – collection of super-class and sub-class as one family.

Double – a floating value with double precision.

Function – declares the name, parameters, and code that defines a procedure.

Inherits – causes the current class or interface to inherit the attributes, variables, properties, procedures, and events from another class or set of interfaces.

Integer – a whole number; a number that is not a fraction.

List – contains any number of elements that are accessed sequentially.

Method – a subroutine, function, or property inside a class.

Must Inherit – specifies that a class can be used only as a base class and cannot create a new object directly from it.

Must Override – specifies that a property or procedure is not implemented in this class and must be overridden in a derived class before it can be used.

New – create a new object instance, or specifies a constructor constraint on a type parameter.

Object – a generic data type that could be character, string, integer, float, or boolean. The object has the ability to hold different data types in an array.

Overridable – specifies that a property or procedure can be overridden by an identically named property or procedure in a derived class. (vb.net keyword)

Parameter list – the list of data types and objects passed to a method by the caller.

Private – specifies that one or more declared programming elements are accessible only from within their declaration context, including from within any contained types.

Protected – specifies that one or more declared programming elements are accessible only from within their own class or from a derived class.

Property – store and retrieve a value.

Properties – data type used inside the class body that is global inside the class but private outside.

Public – specifies that one or more declared programming elements have no access restrictions.

Read Only – specifies that a variable or property can be read but not written.

Reference Variable – a variable that stores the address of a data type or object. By default, all class objects are 'passed by reference' in Visual Basic.

Signature – at the beginning of a method, the signature includes the method's name, private/public/protected, and the parameter list.

Signature – name and arguments of a method.

Single – floating point data type with single precision. (vb.net keword)

String – sequence of character or an array of characters.

Structure – composed of data types and methods but is not able to fully implement object-oriented programming. (vb.net keyword)

Sub-class – class that inherits the super-class and inherits all the properties and methods. (also known as 'child class', 'inherited class' or 'derived class').

Super-class – class that is inherited by another class (also known as 'parent class' or 'base class').

Appendix B: 3D Mathematics

The way to represent 3D space inside a 2D medium is through vector and matrix mathematics. The following section introduces vectors, matrices, and their manipulation.

Vector Algebra

Vectors are elements that have both magnitude and direction. They are used in Direct3D for a variety of different purposes, such as giving instruction on where light should be pointing, what direction a plane is facing, and the how the camera is oriented. Vector manipulation within Direct3D operates the same way as in vector algebra, such as vector addition, subtraction, multiplication, dot product, and cross product.

Vectors in Direct3D are defined with the tail being at the origin of the local coordinate axis (Figure B.1). Points are defined by a position vector, where the location of the point is at the top of the vector (Figure B.1). Four-tuples are used in Direct3D to represent vectors Equation (B-1) and points Equation (B-2). The difference between the two is that a vector has '0' in the last column, whereas points have '1'.

$$v = [x \quad y \quad z \quad 0] \tag{B-1}$$

$$w = [x \quad y \quad z \quad 1] \tag{B-2}$$

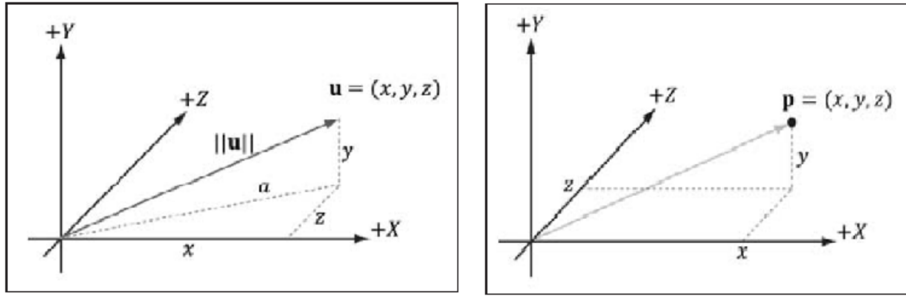

Figure B.1: Normal Vector (Left) and Position Vector (Right) (Luna, 2010)

Matrix Algebra

Matrices are used to perform all the geometric translation, rotation, and scaling of vectors in 3D space. All common matrix operations can be used to manipulate vectors and matrices, such as matrix addition, multiplication, transpose, and inverse.

Scaling

Scaling in Direct3D utilizes a 4×4 matrix as given in Equation (B-3). Scaling is done by multiplying the vectors or points by the scale matrix, modifying the column of the vector with the corresponding row in the matrix. S_x scales along the *x-axis*, S_y along the *y-axis*, and S_z along the *z-axis*.

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{B-3}$$

Rotation

Rotation in Direct3D is performed relative to the origin of the *world space*; world space is the rendering space with a third (z) dimension. Equations (B-4), Equation (B-5), and Equation (B-6) below are used to rotate a vector about the *x-axis*, *y-axis*, and *z-axis*, respectively. The rotation angle θ must be specified in radians.

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{B-4}$$

$$R_y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{B-5}$$

$$R_z = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{B-6}$$

Translation

Translations are performed by using a 4×4 matrix with the first three columns of the last row having a magnitude that adds to the distance in the vector. As shown in Equation (B-7), element [4, 1], [4, 2], and [4, 3] will change the magnitude of the vector a relative distance b_x, b_y, and b_z with respect the *x-axis*, *y-axis*, and *z-axis*.

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ b_x & b_y & b_z & 1 \end{bmatrix} \tag{B-7}$$

Appendix C: Frame Element in ATP-Bridge

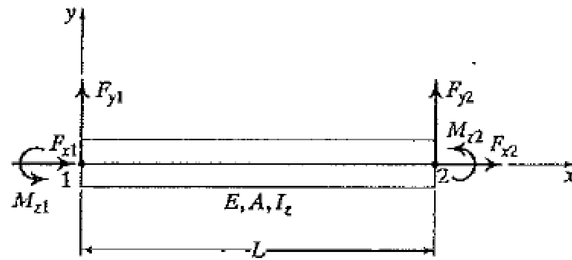

Figure C.1: Frame Element Degrees-of-Freedom (McGuire, Gallagher, and Ziemian, 2000)

The following variables are used throughout Appendix C.

E	= modulus of elasticity
A	= area of a frame element
L	= length of a frame element
I_x	= moment of inertia
P	= internal axial load along the section

ELASTIC AND GEOMETRIC STIFFNESS MATRIX (MCGUIRE ET AL, 2002) :

$$K_e = \begin{bmatrix} EA/L & 0 & 0 & -EA/L & 0 & 0 \\ 0 & 12EI_z/L^3 & 6EI_z/L^2 & 0 & -12EI_z/L^3 & 6EI_z/L^2 \\ 0 & 6EI_z/L^2 & 4EI_z/L & 0 & -6EI_z/L^2 & 2EI_z/L \\ -EA/L & 0 & 0 & EA/L & 0 & 0 \\ 0 & -12EI_z/L^3 & -6EI_z/L^2 & 0 & 12EI_z/L^3 & -6EI_z/L^2 \\ 0 & 6EI_z/L^2 & 2EI_z/L & 0 & -6EI_z/L^2 & 4EI_z/L \end{bmatrix} \quad \text{(C-1)}$$

$$K_g = \frac{P}{L}\begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 6/5 & L/10 & 0 & -6/5 & L/10 \\ 0 & L/10 & 2L^2/15 & 0 & -L/10 & -L^2/30 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -6/5 & -L/10 & 0 & 6/5 & -L/10 \\ 0 & L/10 & -L^2/30 & 0 & -L/10 & 2L^2/15 \end{bmatrix} \quad \text{(C-2)}$$

MEMBER LOAD VECTOR:

$$P_m = \begin{bmatrix} F_{x1} \\ F_{y1} \\ M_{z1} \\ F_{x2} \\ F_{y2} \\ M_{z2} \end{bmatrix} \quad \text{(C-3)}$$

Equations for Triangular Load Fixed-End Moments and Shear (Kassimali, 1999):

$$F_{y1} = \frac{w_1(L-l_1)^3}{20L^3}\left\{(7L+8l_1) - \frac{l_2(3L+2l_1)}{(L-l_1)}\left[1+\frac{l_2}{L-l_1}+\frac{l_2^2}{(L-l_1)^2}\right] + \frac{2l_2^4}{(L-l_1)^2}\right\} + \frac{w_2(L-l_1)^3}{20L^2}\left\{(3L+2l_1)\left[1+\frac{l_2}{L-l_1}+\frac{l_2^2}{(L-l_1)^2}\right] - \frac{l_2^3}{(L-l_1)^2}\left[2+\frac{15L-8l_2}{L-l_1}\right]\right\} \tag{C-4}$$

$$M_{z1} = \frac{w_1(L-l_1)^3}{60L^2}\left\{3(L+4l_1) - \frac{l_2(2L+3l_1)}{L-l_1}\left[1+\frac{l_2}{L-l_1}+\frac{l_2^2}{(L-l_1)^2}\right] + \frac{3l_2^4}{(L-l_1)^3}\right\} + \frac{w_2(L-l_2)^3}{60L^2}\left\{(2L+3l_1)\left[1+\frac{l_2}{L-l_1}+\frac{l_2^2}{(L-l_1)^2}\right] - \frac{3l_2^3}{(L-l_1)^2}\left[1+\frac{5L-4l_2}{L-l_1}\right]\right\} \tag{C-5}$$

$$F_{y2} = \left(\frac{w_1+w_2}{2}\right)(L-l_1-l_2) - F_{y1} \tag{C-6}$$

$$M_{z2} = \frac{L-l_1-l_2}{6}\left[w_1(-2L+2l_1-l_2) - w_2(L-l_1+2l_2)\right] + F_{y2}(L) - M_{z1} \tag{C-7}$$

Appendix D: Incremental-Iteration Variable Definitions

$\left[K_i^{j-1}\right]$ = stiffness matrix using deformed geometry

P_{ref} = reference load

$d\lambda_i^j$ = load ratio for the current iteration

$[\Delta_i]$ = total displacement vector at increment *i*

$\left\{d\Delta_i^j\right\}$ = iteration displacement vector at iteration *j*

$\left\{P_i^{j-1}\right\}$ = total external applied force vector

$\left\{F_i^{j-1}\right\}$ = total internal forces vector element forces at global degree of freedom

$\left\{R_i^{j-1}\right\}$ = imbalance between external and internal force vector

$d\lambda_i^j$ = load ratio at the current iteration

$\left\{\overline{d\Delta_i^j}\right\}$ = displacement vector due to reference load at iteration *j*

$\left\{\overline{\overline{d\Delta_i^j}}\right\}$ = displacement vector due to residual force at iteration *j*

$\left\{d\Delta_i^j\right\}$ = displacement vector at iteration *j*

Appendix E: Newmark-beta Average Acceleration (Chopra, 2006)

Average Acceleration Constant Parameter

$$\gamma = \frac{1}{2}$$

$$\beta = \frac{1}{4}$$

1.0 Initial Calculation.

1.1 $\ddot{u}_0 = \frac{p_0 - c\dot{u}_0 - (f_s)_0}{m}$

1.2 Δt

1.3 $a = \frac{1}{\beta \Delta t} m + \frac{\gamma}{\beta} c$

1.4 $b = \frac{1}{2\beta} m + \Delta t \left(\frac{\gamma}{2\beta} - 1 \right) c$

2.0 Calculations for each time step *i*.

2.1 $\Delta \hat{p}_\mathrm{i} = \Delta p_\mathrm{i} + a\dot{u}_i + b\ddot{u}_i$

2.2 Determine the tangent stiffness k_i.

2.3 $\hat{k}_i = k_i + \frac{\gamma}{\beta \Delta t} c + \frac{1}{\beta (\Delta t)^2} m$

2.4 Solve for Δu_i from $\hat{k}_i$ and $\Delta \hat{p}_i$ using the iterative procedure.

2.5 $\Delta \dot{u}_i = \frac{\gamma}{\beta \Delta t} \Delta u_i - \frac{\gamma}{\beta} \dot{u}_i + \Delta t \left(1 - \frac{\gamma}{2\beta} \right) \ddot{u}_i$

2.6 $\Delta \ddot{u}_i = \frac{1}{\beta (\Delta t)^2} \Delta u_i - \frac{1}{\beta \Delta t} \dot{u}_i - \frac{1}{2\beta} \ddot{u}_i$

2.7 $\mathrm{u}_{i+1} = u_i + \Delta u_i$

$\dot{u}_{i+1} = \dot{u}_i + \Delta \dot{u}_i$

$$\ddot{u}_{i+1} = \ddot{u}_i + \Delta\ddot{u}_i$$

3.0 Repetition for the next step. Replace i by $i + 1$ and implement steps 2.1 to 2.7 for the next step.

References

1. al-Mokhtar, U. "Insurgents Destroy 2 Bridges in Anbar." *The Washington Post* on the Web. 18 Oct. 2009. 5 July 2012. <http://www.washingtonpost.com/wp-dyn/content/article/2009/10/17/AR2009101700690.html>

2. American Concrete Institute (ACI 318-08) (2008). *Building Code Requirement for Structural Concrete and Commentary*. Farmington Hills, MI.

3. American Association of State Highway and Transportation Officials (AASHTO). (2002). *A Guide to Highway Vulnerability Assessment for Critical Asset Identification and Protection*. Science Applications International Corporation, Washington, D.C.

4. Ayoub. A, and Fillppou, F., (2010). "Finite-Element Model for Pretensioned Prestressed Concrete Girders." *Journal of Structural Engineering*, ASCE, Volume 136, Issue 4, pg. 401-409, April 2010

5. Bentz, E. and Collins, M. P., (2001). *Response-2000, Shell-2000, Triax-2000, Membrane-2000 User Manual*. Mar. 2001, 26 August 2012. <http://www.ecf.utoronto.ca/~bentz/manual2/final.pdf>

6. Biggs, J. (1964). *Introduction to Structural Dynamics*. McGraw-Hill, Inc. US

7. Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California

8. Brooks, F. P. (1986). "No Silver Bullet – Essence and Accident in Software Engineering." *IEEE Computer*. Pg 10-19

9. Bruneau, M., Fujikura, S., and Lopez-Garcia, D. (2007). "Blast Resistant Bridge Piers." *Structural Magazine*, March, pg 19-21.

10. Chopra, A. K. (2006). *Dynamics of Structures Theory and Applications to Earthquake Engineer 3rd Edition*. Prentice Hall.

11. Cofer, W. F., Matthews, D. S., and McLean, D. I. (2012). "Effects of Blast Loading on Prestressed Girder Bridges." *Shock and Vibration*. Vol. 19. Num. 1

12. Collins. M., and Mitchell. D. (1997). *Prestressed Concrete Structures*. Response Publications. Ontario, Canada

13. Conrath, E. J., et. al.. (1990). *Structural Design for Physical Security State of the Practice*. American Society of Civil Engineering. Reston, Virginia.

14. Craig, I. (2000). *The Interpretation of Object-Oriented Programming Languages*. Springer, London, Great Britain

15. Department of Army. (1990). *Structures to Resist the Effects of Accidental Explosions (TM 5-1300)*. U.S. Government Printing Office, Washington, D. C. (approved for public release).

16. Department of Defense. (2008). *Unified Facilities Criterion – Structures to Resist the Effects of Accidental Explosions (UFC 3-340-02).*

17. Department of Homeland Security (DHS). (1993). "The World Trade Center Bombing: Report and Analysis." *U.S. Fire Administration/Technical Report Series*, USFA-TR-076, February 1993

18. Devalapura, R. and Tadros, M. K. "Stress-Strain Modeling of 270 ksi Low-Relaxation Prestressing Strands." *PCI Journal*, March-April, 1992

19. Dusenberry, D. O. (2010). *Handbook for Blast Resistant Design of Building*. John Wiley & Sons, Inc., Hoboken, New Jersey

20. Faison, T. (2006). *Event-Based Programming Taking Events to the Limit*. Apress, New York, NY

21. Federal Emergency Management Agency (FEMA 427). (2003). *Risk Management Series. Primer for Design of Commercial Buildings to Mitigate Terrorist Attacks*.

22. Federal Highway Administration (FHWA), (2003). "Recommendations for Bridge and Tunnel Security." *Rep. Prepared by the Blue Ribbon Panel on Bridge and Tunnel Security*, Washington, D.C.

23. Fujikura, S. and Bruneau, M. (2011). "Experimental Investigation of Seismically Resistant Bridge Piers under Blast Loading." *Journal of Bridge Engineering*, ASCE, Vol. 16, No. 1, February 2011, pp. 63-71

24. Gannon, J. C., Marchand, K. A., and Williamson, E. B., (2006). "Approximation of Blast Loading and Single-Degree-of-Freedom Modeling Parameters for Long Span Girders." *Structures Under Shock and Impact IX*, pg. 3-12

25. Garlan, D., and Perry, D. (2005). "Introduction to the Special Issue on Software Architecture." *IEEE Transactions of Software Engineering – Special issue on software architecture*, Vol. 21, Issue 4, April 2005

26. Hendryx, R. (2012). Thesis, The University of Texas at Austin. December 2012

27. Holland, C. (2008). *Blast-Resistant Design of Highway Bridge Columns.* Thesis, The University of Texas at Austin, August 2008

28. Jenkins, B. M. (1997). "Protecting Public Surface Transportation and Patrons from Terrorist Activities: Case Studies of Best Security Practices and a Chronology of Attacks." *MTI Report 97-4*. Mineta Transportation Institute, San Jose, CA.

29. Jenkins, B. M. and Gersten, L. N. (2001). "Protecting Public Surface Transportation Against Terrorism and Serious Crime: Continuing Research on Best Security Practices" *MTI Report 01-07*. Mineta Transportation Institute, San Jose, CA.

30. Jenkins, B. M. and Butterworth, B. R. (2010). "Explosives and Incendiaries used in Terrorist Attacks on Public Surface Transportation: A Preliminary Empirical Examination (MTI Report WP 09-02)." Mineta Transportation Institute , San Jose, CA

31. Kassimali, A. (1999). *Matrix Analysis of Structures*. Brooks/Cole Publishing Company, Pacific Grove, CA.

32. Lafore, R. (2003). *Data Structures & Algorithms in Java. Second Edition.* Sams Publishing, Indianapolis, Indiana

33. Luna, F. (2008). *Introduction to 3D Game Programming with DirectX 10*. Woodware Publishing, Inc., Plano, TX.

34. Patzlaff, Q., Morcous, G., Hanna, K., and Tadros, M. K. (2012). "Bottom Flange Confinement Reinforcement in Prestressed Concrete Bridge Girders." *Journal of Bridge Engineering*, 17(4), pg 607-616

35. Paulay, T. and Priestley, M. J. N. (1992). *Seismic Design of Reinforced Concrete and Masonry Buildings.* Wiley Interscience Publication

36. Malvar, L. J., and Crawford, J. E. (1998). "Dynamic Increase Factors for Concrete." *Twenty-Eighth DDESB Seminars*. Orlando, FL

37. Malvar, L. J., and Crawford, J. E. (1998). "Dynamic Increase Factors for Steel Reinforcing Bars." *Twenty-Eight DDESB Seminar*. Orlando, FL

38. Mander, J. B., Priestley, M. J. N., and Park, R. (1988). "Theoretical Stress-Strain Model for Confined Concrete." *Journal of Structural Engineering*, 114(8), pg 1804-1826

39. Matthews, D. S. (2008). "Blast Effects on Prestressed Concrete Bridges." *Thesis, Washington State University*

40. Mattock, A. H. (1979). "Flexural Strength of Prestressed Concrete Section by Programmable Calculator." *PCI Journal*, Vol. 24. No. 1, Jan.-Feb. 1979.

41. McGuire, W., Gallagher, R.H., and Ziemian, R.D. (2000). *Matrix Structural Analysis Second Edition*. John Wiley & Sons, Inc. Danvers, MA.

42. Microsoft (2009), "Microsoft Application Architecture Guide 2nd Edition." *Pattern and Practices*, Microsoft Corporation

43. Miller, T. (2004), "Managed DirectX 9 Graphics and Game Programming." Sams Publishing, Indianapolis, Indiana.

44. National Bridge Inventory (NBI). (2011). "Count of Bridges by Structure Type." Excel, Retrieve February 21, 2012 from the World Wide Web. http://www.fhwa.dot.gov/bridge/struct.cfm

45. National Research Council (NRC). (1995). *Protecting Buildings from Bomb Damage: Transfer of Blast-Effects Mitigation Technologies from Military to Civilian Applications*. National Academy Press. Washington, D.C.

46. Nawy, E. G. (2003). *Prestressed Concrete: A Fundamental Approach Fourth Edition*. Prentice Hall, Upper Saddle River, NJ

47. Ray, J. C. (2006). "Validation of Numerical Modeling and Analysis of Steel Bridge Towers Subjected to Blast Loadings." *Proc., 2006 Structures Congress*, ASCE, Reston, Va.

48. Ray, J. (2007). "Risk-Based Prioritization of Terrorist Threat Mitigation Measures on Bridges." *Journal of Bridge Engineering*, Vol. 12, Iss. 2, pp 140-146

49. Ross, B. E., Hamilton, H. R., and Consolazio, G. R. (2011). "Experimental and Analytical Evaluations of Confinement Reinforcement in Pretensioned Concrete Beams." *Transportation Research Record 2251*, Transportation Research Board, Washington, D.C.

50. Sammarco, E.L., et. al. (2012). "Development of a Novel Engineering Tool for Assessing Vulnerability of Critical Highway Bridge Component Subjected to Blast." *Munich Bridge Assessment Conference*. Munich, Germany

51. Scacchi, W. (2001). "Process Models in Software Engineering." *Encyclopedia of Software Engineering, 2nd Edition*, John Wiley and Sons, Inc. New York, December 2001.

52. Tedesco, J. W., McDougal, W. G., and Ross, A. R. (1990). *Structural Dynamics Theory and Application*. Addison-Wesley. Meno Park, California.

53. Thorn, A. (2005). *DirectX 9 Graphics The Definitive Guide to Direct3D.* Woodware Publishing, Inc., Plano, TX.

54. Turner, J. (1980). "The Structure of Modular Programs." *Communication of the ACM, Vol. 23*, Num. 5, pg 272-277, May 1980

55. U.S. Army Corps of Engineers' Engineer Research and Development Center (USACE-ERDC). (2004). *Bridge Explosive Loading (BEL) version 1.1.0.3.* Vicksburg, MS. (distribution limited to U.S. Government agencies and their contractors).

56. Van Roy, P. (2009). "Programming Paradigms for Dummies: What Every Programmer Should Know." *New Computational Paradigms for Computer Music*

57. Weiser, B. and Baker, A. (2011). "A Bridge Under Scrutiny, by Plotters and the Police." The New York Times. Retrieve February 21, 2012 from the World Wide Web: http://www.nytimes.com/2011/04/27/nyregion/brooklyn-bridge-was-terror-plot-target-documents-reveal.html?ref=brooklynbridge

58. Williams, G. D. III. (2009), *Analysis and Response Mechanisms of Blast-Loaded Reinforced Concrete Columns.* Dissertation, The University of Texas at Austin, May 2009

59. Williamson, E. B., Bayrak, O., Williams, Marchand, K. A., Kulicki, J., and et al. (NCHRP 645) (2010), “Blast-Resistant Highway Bridges: Design and Detailing Guidelines.” *National Cooperative Highway Research Program Report 645*, Transportation Research Board

60. Winget, D.G, Marchand, K. A., and Williamson, E. B. (2005). “Analysis and Design of Critical Bridges Subjected to Blast Loads”, *Journal of Structural Engineering*, ASCE, Vol. 131, No. 8, pg 1243-1255

Vita

Joeny Bui was born and raised in Houston, Texas from two loving parents who immigrated to the US from Vietnam. He graduated from South Houston High School and received his Bachelor of Science in Architectural Engineering from the University of Texas at Austin in the fall of 2008. In August 2010, he returned to school to pursue research at the University of Texas at Austin working as a research assistant in the Mechanics, Uncertainty, and Simulation in Engineering (MUSE) laboratory.

Permanent e-mail: joeny.bui@utexas.edu

This thesis was typed by the author.