TANDEM: A COMPONENT-BASED FRAMEWORK FOR INTERACTIVE, COLLABORATIVE VIRTUAL REALITY

BY

BENJAMIN A. GOLDSTEIN
B.CompSci., University of Concordia, Montreal, 1998

THESIS

Submitted as partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science in the Graduate College of the University of Illinois at Chicago, 2000

Chicago, Illinois
ACKNOWLEDGMENTS

I would like to thank everyone at EVL who have made this work possible. Specifically I would like to thank my committee, Andy, Jason and Tom M. who have supported me graciously throughout Tandem’s growing pains. I am also very grateful to Tom D. and Dan who have provided such an extensive laboratory, without which this work could never have been realized.

I would also like to thank everyone in the CAVERN group who have made contributions to this work having spent time thrashing out ideas when Tandem was not yet a toddler. Mike Lewis for burning the midnight oil with me during Tandem’s first demo. Yalu and Joe for being patient with my modeling demands and providing their artistic input.

I would to thank Chris and Alex for helping me put together Mars Explorer, the first application for Tandem that looks like it might evolve into something more.

Finally I would like to thank Alan Verlo for taming chaos and keeping the wheels of EVL turning when all hell is breaking loose.

BAG
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong> INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Statement of the Problem</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Thesis Overview</td>
<td>4</td>
</tr>
<tr>
<td><strong>2</strong> MOTIVATION AND BACKGROUND WORK</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Libraries, Toolkits and Frameworks</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 Libraries</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2 Toolkits</td>
<td>6</td>
</tr>
<tr>
<td>2.1.3 Frameworks</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Design Patterns</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 Definition</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2 Specification of a Design Pattern</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Development Systems for Virtual Reality Software</td>
<td>9</td>
</tr>
<tr>
<td>2.3.1 Graphics Libraries</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2 Networked Virtual Environment Toolkits and Frameworks</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Evaluation of Development Systems for Collaborative Virtual Reality</td>
<td>16</td>
</tr>
<tr>
<td>2.5 Collaborative Virtual Reality Applications</td>
<td>16</td>
</tr>
<tr>
<td>2.6 A Component-based Approach to Collaborative Virtual Reality</td>
<td>19</td>
</tr>
<tr>
<td><strong>3</strong> REQUIREMENTS AND ANALYSIS OF A FRAMEWORK FOR COLLABORATIVE VIRTUAL REALITY</td>
<td>21</td>
</tr>
<tr>
<td>3.1 System Properties</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Graphics</td>
<td>23</td>
</tr>
<tr>
<td>3.2.1 Rendering</td>
<td>23</td>
</tr>
<tr>
<td>3.2.2 Model Loading</td>
<td>23</td>
</tr>
<tr>
<td>3.2.3 Intersection Testing</td>
<td>24</td>
</tr>
<tr>
<td>3.2.4 Selection of a Graphics Library</td>
<td>24</td>
</tr>
<tr>
<td>3.2.5 State and Behavior</td>
<td>24</td>
</tr>
<tr>
<td>3.2.5.1 Autonomous Behavior</td>
<td>25</td>
</tr>
<tr>
<td>3.2.5.2 Properties</td>
<td>25</td>
</tr>
<tr>
<td>3.2.5.3 To act or not to act</td>
<td>25</td>
</tr>
<tr>
<td>3.2.5.4 Interactive Graphics Components</td>
<td>25</td>
</tr>
<tr>
<td>3.3 Interaction</td>
<td>26</td>
</tr>
<tr>
<td>3.3.1 Audio</td>
<td>26</td>
</tr>
<tr>
<td>3.3.2 Travel</td>
<td>27</td>
</tr>
<tr>
<td>3.3.3 Selection and Manipulation</td>
<td>30</td>
</tr>
<tr>
<td>3.4 Interaction Abstracted</td>
<td>33</td>
</tr>
<tr>
<td>3.4.1 Events</td>
<td>33</td>
</tr>
<tr>
<td>3.4.2 Actions</td>
<td>33</td>
</tr>
<tr>
<td>3.5 Networking and Collaboration</td>
<td>35</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (CONTINUED)

3.5.1 Distributed Shared Memory ................................................................. 35
3.5.2 Connections and Asynchronous State Updates ................................. 35
3.5.3 Serialization ....................................................................................... 36
3.6 Extensibility, Flexibility, and Maintainability ........................................... 36

4 THE FRAMEWORK ...................................................................................... 39

4.1 A Component-Based Approach to Collaborative Virtual Reality ............. 39
4.2 Rationale Behind a Component-Based Approach ................................. 39
4.3 The Role of Design Patterns in Tandem ............................................... 40
4.4 Introduction to Tandem Architecture .................................................... 42
4.5 Core ........................................................................................................ 43
  4.5.1 Interaction Manager ................................................................. 49
  4.5.2 Traveler .......................................................... 50
  4.5.3 Action Events ......................................................... 51
  4.5.4 Actions ................................................................................. 51
  4.5.5 User Components ................................................................. 53
  4.5.6 Audio Player ................................................................. 54
  4.5.7 Execution Control ................................................................. 54
  4.5.8 Core Loader ................................................................. 55
  4.5.9 Avatar Manager ................................................................. 55
4.6 Graphics ................................................................................................ 56
  4.6.1 The Graphics Manager ................................................................. 59
  4.6.2 Graphics Display Library ................................................................. 60
  4.6.3 The Graphics Component Hierarchy ......................................................... 60
  4.6.4 Display Library Component Adapters ......................................................... 63
  4.6.5 Background Loading ................................................................. 65
  4.6.6 Graphics Visitors ................................................................. 67
  4.6.7 Graphics Iterator ................................................................. 69
  4.6.8 Intersection Testing ................................................................. 70
4.7 Networking .............................................................................................. 70
  4.7.1 The Network Manager ................................................................. 71
  4.7.2 The Network Event Manager ................................................................. 72
  4.7.3 The Connection Manager ................................................................. 72
  4.7.4 Receiving Docks ................................................................. 74

5 DESIGN IMPLEMENTATION AND RATIONALE ........................................... 76

5.1 Tandem Core Display Library Components .............................................. 76
5.2 Tandem Core User Components ............................................................... 77
5.3 Tandem Development Model Controllers .................................................. 78
  5.3.1.1 The Visitor Pattern ................................................................. 79
  5.3.2 Tandem Development Actions and Action Handlers .............................. 79

6 MARS EXPLORER: A TANDEM APPLICATION .............................................. 81
TABLE OF CONTENTS (CONTINUED)

6.1 Application Overview ........................................................................................................81
6.2 Application Architecture ....................................................................................................81
  6.2.1 Mars Explorer Client .................................................................................................82
  6.2.2 Mars Builder Server ..................................................................................................85
6.3 Mars Explorer Interaction .................................................................................................87
6.4 Mars Networking .............................................................................................................91
6.5 Putting it all Together ........................................................................................................92

7 CONCLUSIONS AND LESSONS LEARNED .......................................................................93

8 GLOSSARY .........................................................................................................................95

APPENDICES ..........................................................................................................................96

  Appendix A .........................................................................................................................97
  Appendix B .........................................................................................................................98

CITED LITERATURE .................................................................................................................306

VITA .........................................................................................................................................308
LIST OF TABLES

TABLE I  SUMMARY OF GRAPHICS AND INTERACTION CAPABILITIES OF VIRTUAL REALITY DEVELOPMENT SYSTEMS ................................................................. 15
TABLE II  SUMMARY OF NETWORKING CAPABILITIES OF VIRTUAL REALITY DEVELOPMENT SYSTEMS 15
TABLE III  SUMMARY OF COLLABORATIVE VIRTUAL REALITY APPLICATIONS ........................................ 19
TABLE IV SUMMARY OF NON-FUNCTIONAL REQUIREMENTS ........................................................................... 38
LIST OF FIGURES

Figure 1 Taxonomy of Travel Techniques for Immersive Virtual Environments ...................................................... 28
Figure 2 Simplified Taxonomy for Travel Techniques .......................................................................................... 30
Figure 3 Taxonomy of Selection/Manipulation Techniques .................................................................................. 32
Figure 4 Taxonomy of Interaction Events and Actions .......................................................................................... 34
Figure 5 Tandem Module Diagram ........................................................................................................................ 43
Figure 6 The Application Aggregate Class ........................................................................................................... 45
Figure 7 Application Class Factory Method Interaction Diagram ........................................................................ 47
Figure 8 Tandem Application Class Composition ................................................................................................ 49
Figure 9 Action Sequence diagram ....................................................................................................................... 53
Figure 10 Execution Control Loop ........................................................................................................................ 55
Figure 11 Graphics Module Overview .................................................................................................................... 58
Figure 12 Graphics Module Components ............................................................................................................... 59
Figure 13 Tandem Graphics Component Hierarchy ............................................................................................... 62
Figure 14 Performer Component Adapter Hierarchy ............................................................................................... 64
Figure 15 Background Loader Class Diagram ........................................................................................................ 67
Figure 16 Grab Controller Interaction Diagram ................................................................................................... 68
Figure 17 Graphics Iterator Class Diagram ........................................................................................................... 69
Figure 18 Network Manager Aggregation ............................................................................................................... 71
Figure 19 Network Connection Manager ............................................................................................................... 73
Figure 20 Sending a Network Event Sequence Diagram ........................................................................................ 74
Figure 21 Receiving Dock Class Diagram ............................................................................................................... 75
Figure 22 Cavern Builder Client Object Diagram ................................................................................................ 84
Figure 23 Cavern Builder Client Application Diagram .......................................................................................... 85
Figure 24 Cavern Builder Server Diagram ............................................................................................................. 86
Figure 25 Scanning a Rock Sample ......................................................................................................................... 88
Figure 26 Mars Explorer Interaction Manager ........................................................................................................ 90
Figure 27 Summary of Unified Modeling Language Notation ................................................................................ 97
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CAVE</td>
<td>CAVE Automatic Virtual Environment</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>CVR</td>
<td>Collaborative Virtual Reality</td>
</tr>
<tr>
<td>DOF</td>
<td>Degrees of Freedom</td>
</tr>
<tr>
<td>DSM</td>
<td>Distributed Shared Memory</td>
</tr>
<tr>
<td>FOV</td>
<td>Field of View</td>
</tr>
<tr>
<td>GOF</td>
<td>Gang of Four</td>
</tr>
<tr>
<td>HMD</td>
<td>Head Mounted Display</td>
</tr>
<tr>
<td>SGI</td>
<td>Silicon Graphics Inc.</td>
</tr>
<tr>
<td>SVE</td>
<td>Shared Virtual Environment</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>VE</td>
<td>Virtual Environment</td>
</tr>
<tr>
<td>VR</td>
<td>Virtual Reality</td>
</tr>
</tbody>
</table>
SUMMARY

Virtual Reality (VR) has shown much progress in this last decade. This progress has culminated in the form of several commercially available libraries and toolkits that handle the lower-level details of the hardware and software that supports this domain. As the construction of a high-bandwidth network infrastructure is currently being put into place, much VR research has shifted its focus from such problems as rendering graphics in real-time, and the development of VR hardware devices, to Collaborative Virtual Reality (CVR). Advances in this domain point towards the birth of a new medium of communication. Participants in locations around the world can participate in a shared space, conducting activities such as design reviews and tele-conferencing. It is essential to the maturation of CVR that systems that facilitate this development are readily available.

Application development in the domain of CVR is still exceedingly difficult due to the level of expertise required in the numerous hardware and software systems that make up this domain. Few systems are currently available that facilitate the development of VR applications and in addition support the necessary interaction and networking aspects of CVR. As a result the challenge still remains to integrate suitable hardware and software into a system that easily adapts to the constant changes that are occurring in this rapidly evolving medium.

Design patterns are ideal for supporting a component-based architecture. They are context-free, object-oriented solutions to reoccurring design issues in the development of software. The use of design patterns extends towards the construction of a library of reusable components, maximizing development effort. In addition, they also facilitate the integration of new systems that support underlying layers (such as graphics and networking) as the domain undergoes change.

This thesis is an examination of the requirements of CVR and the design and implementation of a component-based framework that supports these requirements. Design patterns form the basis of the architecture presented herein, and promote a system that is directed towards the rapid development of CVR applications. In addition, an overview of an application developed using this framework is given. This will evaluate the usefulness of this framework with respect to a set of features common to the domain of CVR.
1 INTRODUCTION

Collaborative Virtual Reality (CVR) is setting the stage for a new medium of communication. This thesis is an investigation of a framework that supports the development of applications within this medium. In particular this work examines the use of design patterns to construct a component-based framework that maintains the flexibility and extensibility necessary for a system that is to encapsulate such a rapidly evolving domain. In addition to maintaining a flexible architecture this framework prioritizes the performance requirements of this domain.

Over the last decade many advances have been made to the domain of virtual reality (VR). It has matured from a sensationalistic vision of the future depicted by science fiction writers such as William Gibson and novels such as Snow Crash to a medium suitable for rapid virtual prototyping, scientific visualization and the integration of networked audio and video conferencing, also known as tele-immersion (1). Since its inauguration into the public eye in the early eighties, it still has yet to live up to the media’s bombardment of promise. In this vision VR offers an escape, an alternate world that is so real it offers a replacement for the daily doldrums of day-to-day existence. This remains a far-fetched and ludicrous ideal, hovering on the outer rim of pure fiction. We are no closer to this vision than HAL 9000, the thinking and feeling computer depicted in the film 2001: A Space Odyssey.

Nevertheless, the ability to display realistic stereo, head-tracked, graphics at interactive rates1, in combination with CD-quality2 sound has shown much promise in creating a virtual environment (VE) that enforces a great deal of presence3 and immersion4. Such advances have given rise to several toolkits and libraries aimed at capturing this development effort. Much of this work has focussed on libraries that provide a uniform interface for various VR hardware display and input devices as well as serving the needs

---

1 The generally accepted value is greater than 15hz to maintain presence and immersion (ref).
2 44 kHz
3 the sense of being there
4 the blending of boundaries between the physical and the virtual environment
of VR applications. This enables a quicker development time and allows an application to be written for a varying hardware configuration. Such effort can be seen in the likes of WorldToolKit (2) the CAVE library (3), and recently VR Juggler (4). Other research has focussed on specific problems within VR, such as scientific visualization (e.g. Limbo/VTK (1) and the Virtual Windtunnel (5)). Over the last few years, as VR hardware and the software that supports it has stabilized, researchers and developers have started to focus on a new set of problems, those of CVR. As an infrastructure for high bandwidth super-computing is being realized, distributed VR software technologies have begun to emerge. The single user VR application has evolved into shared virtual environments (SVEs).

In a SVE users from around the globe can participate in an experience that in the past would require co-location. Avatars, graphical renderings of the user, generally represent participants in a SVE, and enforce tele-presence, that is the sense that other humans share the environment (6). The most common type is the Puppet Avatar, which is generally represented by polygonal models. Video avatars are a variation of Puppet Avatars in their use of graphic models, but in addition, video of the user is brought into the VE and texture mapped onto the model’s polygons. Collaboration in a SVE is made possible because each user can witness the actions of others in the environment. In CVR users can interact with each other and the VE by discussing features of the environment and manipulating its state (e.g. bringing in new models and changing existing ones). This enables designers to conduct reviews of virtual prototypes with colleagues in other cities and scientists to examine visualizations of data sets that have just been recorded. To aid in the development of such applications several toolkits have been created for CVR (e.g. Bamboo (7), RAVEL (8), CAVERNsoft (1,9, 10), Avocado (11)).

1.1 Statement of the Problem

Despite these advances, one area that has suffered neglect is a complete VR development system that supports common interaction techniques within this medium as well as a means to make this interaction easily collaborative. The interaction component of such a system is homologous to the desktop graphic user interface (GUI) toolkits that support common user interface widgets and metaphors such as scrollbars, list boxes and mouse events. The difficulty in developing such a system is symptomatic of any new medium. One reason is the lack of any standard interaction techniques within VR. This stems from the fact that a
large body of researchers has only investigated VR over the last decade. In addition, there are many forms of devices supporting VR. The interaction techniques employed vary greatly depending on whether one is using a head-mounted display or a projection-based display device. Even within the realm of projection-based display devices the interaction techniques differ. For instance in a room sized device where the VE is projected on 3 walls and a floor, surrounding the user, it is appropriate to simply walk around an object to view it from various angles. A single screen device on the other hand does not afford such luxury, and a technique must be employed to rotate the object to achieve a similar view. Furthermore, unlike desktop interaction, VR is not reduced to 2D metaphors. This makes interaction within VR inherently more complex. Not only can VR model the way we interact with the real world, but entirely new interaction techniques can be created, such as the ability to fly, walk through solid objects and manipulate objects far beyond the user’s reach.

Until any VR interaction standards emerge, in addition to providing a set of pre-built metaphors, a VR toolkit must allow developers to extend these metaphors or build new ones from a set of base components. This type of functionality is reduced to the addition of new behavior and state that must be easily associated with graphical elements in the VE. An application that employs collaborative aspects also requires this state to be shared. As users interact with graphical objects in CVR, state must be propagated through the system on the sending end, over one or more network connection(s) and through the system of each receiving end. This requires maintaining the state of an object on one node and its mapping to the state(s) of other node(s).

Because this area of research is relatively new, it is important that any toolkits or frameworks directed towards this domain maintain an extremely flexible, extensible and maintainable architecture as the hardware and software systems supporting CVR are rapidly evolving. The framework cannot be permanently tied to a particular graphics or network library and hence its classes and their structure must maintain the highest level of extensibility. Software design patterns (12) provide many solutions in this regard, as they are a distillation of many years of software design experience.

The current packages geared towards CVR focus on only a few aspects of distributed computing. These include RPC-like and object serialization services such as the Common Object Request Broker Architecture (CORBA) networked callbacks and distributed shared attributes. No one package has made the attempt to
fuse together software technology that caters to the three main areas of CVR, VR graphics, interaction and collaboration, and within a flexible and extensible object-oriented framework that leverages the power of C++ and makes full use of design patterns. Many of the VR toolkits that enable collaboration support the networking aspect of CVR at the connection level. However, this does not specify a design for propagating interaction by a user in the environment through the application layers to the network connection. The interface between the graphics layer and the network layer must be defined by the application. SGI’s Performer library (13) provides a complete scene-hierarchy based graphics library that takes advantage of the performance of the underlying hardware, however it does not specify a software architecture for organizing the state and behavior of an application that caters to CVR.

Tandem is a framework for developing interactive applications within CVR. It was designed with two key driving forces. First, Tandem aims to facilitate the development of collaborative VR applications. Its key distinction as a framework is that it specifies an application design and architecture as opposed to a toolkit that provides a library of functions. Second, Tandem is an examination of the feasibility of building a component-based framework for CVR based on design patterns while maintaining the performance requirements of this medium.

1.2 Thesis Overview
This thesis aims to document the process of developing Tandem from its inception through to the implementation. First a chapter on the background work will survey the current literature and works in the domain of CVR. A summary of this work is then presented. This summary is further explored in the following chapter where a taxonomy of CVR is constructed. The next chapter will examine the requirements of Tandem, and how these requirements map to the taxonomy of CVR. Following that, will be a chapter on Tandem’s architecture. After an outline of the architecture is given, an examination of the motivation and rationale behind the design is seen in the following chapter. Finally a chapter will evaluate the usefulness and practical aspects of using Tandem in the development of Mars Explorer, an application that implements the basic requirements of CVR. The goal is to not only provide a case study in the development of a framework, but also to highlight how a process can be applied to the domain of Virtual Reality, bringing sound software engineering practice to the field.
2 MOTIVATION AND BACKGROUND WORK

There have been many contributions from the VR research community in recent years to the domain of CVR. Tandem is an amalgam of modern software engineering practice and the contributions of recent CVR research. It is beyond the scope of this chapter to provide an exhaustive survey of all works that have contributed to CVR. Rather, the purpose of this chapter is to summarize and evaluate those works that have influenced Tandem’s design and implementation.

First a brief coverage of the distinctions between frameworks, toolkits and libraries is given. Following that is a summary of design patterns. This treatment allows the reader to follow the classifications made throughout. In the next section, an overview of development systems for CVR is presented. An evaluation of these systems follows. Then a survey of CVR applications is presented. Following this survey of work a discussion of the necessity for component-based solution is presented. This will explain why design patterns are an ideal solution to this approach. At the end of this chapter the reader will understand the context in which Tandem has been built as well as the contributions of its supporting systems.

2.1 Libraries, Toolkits and Frameworks

Developing software is a costly and involved process that is difficult to manage, budget and monitor. Since the advent of the Object-Oriented (OO) programming paradigm which was first suggested in the late 1960’s there has been a move towards establishing a standardized development process as in other engineering disciplines. This move places an emphasis on a well-understood and defined process as well as a repository of established software designs. This first culminated in the 1970s with the structured method for software engineering. However, it was not until the early 1990s that the realization of this goal became feasible with the widespread industry acceptance of the OO paradigm. This approach has been embraced because of the potential for component re-use. This inevitably leads to quicker development times and software that is easier to maintain (14). In accord with component re-use are software systems that are not applications in and of themselves, but aid the development of applications in a particular domain by providing commonly occurring routines, modules and subsystems.
2.1.1 Libraries
Libraries are the oldest and most common of the three development systems mentioned in this section. Libraries have been around since the notion of a library archive. There are two major varieties of libraries, procedural and class libraries. Procedural libraries typically offer a collection of functions for a given domain. For example, OpenGL is a graphics library that provides a complete set of routines for rendering graphics to a display system. Class libraries are the OO equivalent of procedural libraries and will be described in the next section. Both OO and procedural libraries do not specify the architecture of an application and little if any collaboration is required between the components of the library.

2.1.2 Toolkits
Toolkits provide a set of related, reusable classes or procedures that provide functionality for a given domain. An example of a toolkit is the Standard Template Library, a freely distributed library of template collections, and generic algorithms originally developed at Hewlett Packard. Another example is GLUT, the OpenGL Utility Toolkit. GLUT provides a standardized windowing interface for OpenGL. As opposed to a library, which tends to be a flat hierarchy of procedures, toolkits often specify the way in which its various classes or procedures must interact with each other. A toolkit, nevertheless, does not enforce a design, it stipulates only how its components must collaborate in order to achieve the desired functionality (12).

2.1.3 Frameworks
Frameworks specify a design and architecture for a given application. It imposes the most constraints, but offers the most reuse in terms of its class structure, their interaction and the functionality that is provided. The authors of Design Patterns: Elements of Reusable Object-Oriented Software, also known as Gang of Four (GOF) put this succinctly in their definition:

“A framework is a set of cooperating classes that make up a reusable design for a specific class of software” (12).

The advantages of using a framework are an accelerated development time, a generalized structure that makes applications easier to understand, and the ability to adopt well-understood solutions to problems within a given domain. This enables the development of software with far fewer risks.

Developing a framework however, is the most difficult of the development systems outlined in this chapter. The author(s) must constantly make assumptions as to what most applications will require for a
given domain. This requires a careful examination of the advantages and disadvantages for a given design. It is generally impossible to make major changes to the design of framework and maintain compatibility between itself and applications written for a previous version. For this reason, the architecture of a framework must exhibit the extreme in flexibility, extensibility and maintainability. Hence, the development of a framework is often an iterative process. A sound approach to the development of a framework is to split the activity into three groups. The first group is responsible for the framework, and the other two are each responsible for an application built using the framework. In this fashion, the framework group designs, implements and teaches the framework. The application groups try and reuse as many features of the framework as possible and offer criticism as to how the framework can be improved.

A drawback of using a framework is that developers lose a certain degree of freedom since they no longer have control over the execution loop or the general structure of the application. This is known as inversion of control. The benefits however, generally far outweigh any negative aspects.

### 2.2 Design Patterns
#### 2.2.1 Definition
Design patterns specify a thoroughly tested OO design solution for a commonly occurring design problem. A design pattern does not specify a specific context, and hence is generally applicable to all software that is trying to solve a similar problem. Design patterns are the distillation of many years of development experience by both practitioner’s and researchers. They are often the best solution for a problem that frequently occurs in a software system. By using design patterns one tends to develop a more flexible architecture that provides the facility to add extensions to its existing components. Moreover, a framework built from design patterns enables clients to combine existing components or build new ones that are easily integrated. This gives clients a vast number of combinations to chose from in building their applications. Furthermore, design patterns provide a valuable way of documenting a framework. This permits the use of a common nomenclature for abstract ideas representing a complex design solution. In this fashion, those familiar with design patterns are able to more quickly grasp the functionality and behavior of the system without having to turn to the source code.
2.2.2 Specification of a Design Pattern

In their 1995 book on the topic (12), the GOF specify a template for describing a design pattern. This template is briefly described here in order to give the reader a clearer understanding of what a design pattern is, how they are presented and how one may select and use a particular design pattern. Each pattern is presented through the use of several sections describing its name, purpose and use. Below a list of each section is given in the order in which it is to be presented with a brief description of its purpose. For a treatment of how one may go about writing their own patterns, the readers are referred to (16).

1. **Pattern Name and Classification**: Each pattern begins with a title describing its name and classification. This presents a nomenclature that gives it a precise name, which implies its possible uses.

2. **Intent**: This section helps in selecting an appropriate pattern as it outlines its intended purpose.

3. **Also Known As**: This section lists other possible names for a given pattern.

4. **Motivation**: This gives the reader insight into the scenario that prompted the creation of this pattern.

5. **Applicability**: This describes the design issues addressed by this pattern, and when its use is most appropriate.

6. **Structure**: This section provides a class diagram describing the typical structure of the pattern.

7. **Participants**: This lists all of the entities that play a role in the structure of this pattern.

8. **Collaborations**: This section describes the dynamic behavior of the participants, that is how they interact with each other.

9. **Consequences**: This section helps in the selection of a pattern by describing the trade-offs that result from using a particular pattern.

10. **Implementation**: This describes several variations of how a particular pattern may be implemented.

11. **Sample code**: This illustrates several code examples in either C++ or Smalltalk.

12. **Known Uses**: This gives several gives examples in context of how this pattern has been used.

13. **Related Patterns**: This final section gives a treatment of how this pattern relates to other similar patterns and also how it may be used in combination with other patterns.

Throughout the chapters describing Tandem’s design, references will be made to the design patterns used. The reader is encouraged to consult (12) for a detailed explanation of the pattern. It is beyond the scope of this thesis to present each pattern. Instead, the purpose of making a reference to a pattern is to demonstrate how it solves a particular design issue within CVR, the criteria used in selecting this pattern, and potentially the discovery of any new patterns.
2.3 Development Systems for Virtual Reality Software

To date, many systems have been written that aid in the development of VR software. To gain an appreciation of this passed work this section divides the work into its major categories so that one can better assess the contribution each work has made. The first group caters to the basic requirement of VR, rendering graphics to a display device. The next group caters to libraries, toolkits and frameworks that aid the development of CVR applications. A summary of the capabilities of these systems is given at the end of this section. Table I provides a summary of the systems that offer graphics and interaction capabilities. Table II summarizes the networking functionality of those systems that include it. If a particular system is absent from a table, it is because it is not applicable to the category of features that appear in that table.

2.3.1 Graphics Libraries

Graphics libraries take care of the necessary routines that allow the rendering of graphics to a display device. Projection-based VR adds the necessity of rendering graphics to customized display devices that differ from typical desktop devices. As such, a VR graphics library will often adapt an established desktop graphics library to work with the required display devices. VR display devices may also alter the typical conventions that are used in desktop computer graphics. For instance, desktop VR does not afford the user the ability to provide a user-centered perspective when a head-tracked user looks beyond the narrow field of view (FOV).

OpenGL

OpenGL is a mature graphics API written for C. It has implementations on almost every platform and is generally the library of choice for the low-level graphics routines of many graphics intensive applications. Until very recently it was typically the library chosen for the development of VR applications. Its main weakness is the amount of coding effort required to render complex models to a display, since each model must be described as a low level set of primitives. In addition, OpenGL does not provide sufficient support for efficient collision detection, bounding volumes and selection of graphics components. Structures and algorithms must be custom coded by the developer in order to provide these features.

Open Inventor

Open Inventor is an OO version of OpenGL. It provides many high-level components that make graphics programming an easier task. Its graphics geometry descriptions are stored in an easily editable text format or in its own binary format. This format was used as the basis for the Virtual Reality Markup
Language 1.0 specification. Its major weakness is its lack of support for multi-processor systems and its generally sluggish performance in comparison to OpenGL or Performer based applications.

**IRIS Performer**

Performer is a multi-processed, pipelined, scene graph-based graphics library. A scene graph is a directed acyclic graph (DAG) that determines the sequence in which operations are performed on nodes in the graph. Performer is designed to get the best performance from the hardware architecture and was originally designed for the RISC-based SGI systems. The rendering pipeline is divided into three stages, APP, CULL and DRAW. In a Performer application, one can associate an application callback function with any node in the Performer hierarchy. This function is called once per frame, during the APP stage. The second stage involves culling graphics nodes that are not visible within the current clipping plane. The final stage involves rendering the graphics into the scene. Performer’s routines enable the most efficient use of the scene hierarchy. Performer provides efficient culling, use of bounding volumes, and provides an interface for efficient collision detection using ray casting techniques (13). In addition, Performer provides support for reading in many popular file formats from various 3D modeling applications. This greatly reduces the coding effort required to display detailed graphics models in a VE.

**CAVElib**

The CAVE library provides an application programming interface (API) for writing VR applications primarily for projection-based VR displays and a variety of VR input devices. It handles rendering of OpenGL or Performer-based graphics to projection-based displays as well as an API for various VR input devices such as the standard 3D mouse and numerous tracking devices. As such the developer is never concerned with low-level details such as calculating the view frustum or querying serial ports for tracker data bit streams. The CAVElib provides all the necessary routines to initialize the scene, change the viewpoint and obtain the state of input devices. Furthermore, the CAVElib is written in such a way that an application written for one type of projection display can easily be run on another type through text-based configuration files. The standard CAVElib configuration assumes a stereo, head tracked display as well as a tracked three button input device that includes a joystick (3). Two commonly used display devices are the CAVE (CAVE Automatic Virtual Environment) and the ImmersaDesk. A CAVE is a cube shaped room that provides the user with a completely immersive VR experience (17). The standard CAVE configuration includes four walls (front, left, right and the floor). An ImmersaDesk is a single screen, semi-immersive
display device. It looks similar to a large drafting table and is normally angled at a forty-five degree angle away from the user to give a partial feeling of depth. In this fashion the user can look straight ahead as well as down.

2.3.2 Networked Virtual Environment Toolkits and Frameworks

The systems covered in this section are mainly focussed on developing networked VEs. A networked virtual environment is any VR application that sends and retrieves data from a network. An example of a networked VE is a scientific-visualization application that retrieves large sets of data. A networked VE does not imply that it is shared or that it is collaborative, it simply means there is network communication. Some means of interaction with other users is necessary for both collaboration and the creation of a SVE.

The primary concern of the networking components of CVR development systems is creating a high-level interface for distributing data without having to deal with low-level networking issues such as sockets and networking protocols. Many of these systems also provide mechanisms for ensuring proper synchronization, state consistency and persistence.

WorldToolKit

WorldToolKit (WTK) is a cross-platform solution that combines both rendering capabilities and a library of C functions that enable the development of VR applications. The WTK provides drivers for a wide range of VR devices and integrates with other products from Sense8 such as a visual builder (WorldUp) and a client/server network architecture (World2World) for collaborative capabilities. It provides interactive capabilities through components such as Paths, Motion Links and Sensors. Like Performer, WTK is scene graph based. It also provides an API for 3D and stereo sound. In addition, WTK also has a multi-processor multi-pipe option. Though WTK is not an object-oriented library it does provide C++-wrappers (2;18).
**Avocado**

Avocado is an OO framework for building networked VEs. Avocado builds on top of Performer and Scheme to provide a high-level framework for building a VR application. It integrates the networking, graphics, interaction and behavior into one system. It also provides a distributed shared memory (DSM) model. In this framework objects can be instantiated either as local objects, or distributed objects. In the former the object is only visible in the local address space. In the latter case, a distributed object automatically results in the instantiation of objects in each client’s address space. Changes to a distributed object result in automatic updates to each of its copies. Avocado uses a process group model. Each group member is guaranteed to receive the delivery of network messages in exactly the same order. In addition when a new member joins the group, all communication is suspended until the new member’s state is updated. This guarantees state consistency. This framework was being used for a distributed visualization in the petroleum industry at the time the following paper was written (11).

**CAVERNsoft**

CAVERNsoft was designed at the Electronic Visualization Laboratory by Jason Leigh to address the requirements of Tele-Immersion and CVR. This software provides clients with a framework for developing networked VR applications. It provides a DSM model, a message passing system and a persistent store. It targets the problems of consistent shared state, persistence and performance concerns resulting from the various types of data transfer in a networked VR application. As opposed to most libraries that presume the type of transfer that is required, CAVERNsoft allows clients to chose the appropriate means of transfer based on the type of data being transferred. For example avatar tracker data is generally transferred over an unreliable (UDP) channel while important state information that cannot afford to be lost is transferred over a reliable (TCP) channel. The essential CAVERNsoft components are the Information Resource Broker (IRB), channels and keys. Each node on the network has its own IRB. Clients create a connection between themselves and another node by creating a channel and linking a local key with a key on a remote site that has a known string name. The IRB provides an interface for creating channels. The key names are part of a global name space maintained by the IRBs. Any client that places data onto a key will result in the automatic distribution of the data to all clients linked to that key name. Upon receipt of data for a given key, a client-specified callback is triggered. The IRB permits either a star, or chained topology. Hence,
CAVERNsoft does not dictate a particular network topology. CAVERNsoft also handles synchronization issues related to these updates, and provides developers with a choice of synchronization types. (1; 9; 10)

**World2World**

World2World (W2W) is a network solution developed by Sense8 that offers complete integration with their WTK toolkit and WorldUp visual simulation builder (18). It offers a client/server based network architecture. In a W2W application there are two types of servers, the Server Manager and the Simulation Servers. For each world application there is always one Server Manager. This component is responsible for providing a known access point to the application, and handles connection management, and login/logout notification. There is one or more Simulation server(s), responsible for the sharing of distributed data, through designated share groups. Distribution of data is done by the sharing properties associated with a share group. This creates a DSM model. In order to create the connection between two properties on remote client nodes, each client interested in receiving updates to the property must specify that the property is shared. Updates to this property are carried out by an event model, to which actions can be designated in response to this event. Because there may be more than one Simulation manager, distributed processing of shared data is made possible. W2W is written on top of the UDP networking protocol. In order to provide reliability, W2W implements its own acknowledgement protocol. It provides facilities for specifying connection-based update rates, as well as synchronization through a server controlled global simulation time. Persistence is handled by designating a particular share group and the desired properties as persistent. The state of these properties will be maintained even in the absence of any clients.

**Bamboo**

Bamboo is a networking framework supporting the development of SVEs. It differs from previous toolkits and frameworks in this domain in that modules may be linked at run-time, creating a dynamically extensible application. Its main features are an object database supporting type identification; an elaborate distributed callback system; an event handling mechanism; threading; and a run-time environment (7; 19). The event handler enables the triggering of callbacks that have registered interest in a particular event. Bamboo currently uses the ACE library for its networking, providing synchronization, threading and timing (20). However, ACE will soon be replaced by Netscape Portable Runtime (21). Bamboo also provides an architecture for persistence (22).
**Ravel**

The Remote Attribute Virtual Environment Library is yet another software development system geared towards providing networking capabilities to virtual environments (8). Its focus is to provide low-latency communication between tasks on remote nodes. It is also directed towards maintaining state consistency between nodes. Ravel divides its attributes into four categories. The first two are categorized as *out* attributes (to be modified) and *in* attributes (to be read). The third is a *shared* attribute that may be both modified and read. The last is a *local* attribute that it is not distributed. A name server process running on a known machine provides a central access point to the system. Clients connect to the name server and register interest with out attributes. When a client changes an out attribute the server provides the producing process with the list of clients interested in the changes. Out attributes are connected to in attributes on remote nodes in this fashion. Shared attributes provide a global name space whereby the in and out connection is not necessary.

**VR Juggler**

VR Juggler is an object-oriented framework written in C++ that is designed to quickly develop VR applications by abstracting the complex details of the hardware and software systems (4). It supports a variety of VR display devices including projection-based and head mounted displays (HMDs), as well as a number of input devices, such as Virtual Technologies CyberGlove and various tracking devices. Juggler aims to provide a common interface for varying devices so that an application doesn’t need to be adapted to run on a different set of hardware. For instance, the framework should handle the transition from a HMD to a projection-based display device. Performance is another key concern of Juggler and hence a GUI-based program called the Environment Manager provides the developer with the ability to monitor and graph performance data. The Environment Manager can also be used to reconfigure the application at run-time to adjust performance-related properties. This framework is also designed for cross platform development and supports the SGI, NT and HP platforms at the time this writing. Finally networking support is provided through an abstract Network manager. It is intended that the Network manager be extended to support the needs of CVR sometime in the future.
TABLE I
SUMMARY OF GRAPHICS AND INTERACTION CAPABILITIES OF VIRTUAL REALITY DEVELOPMENT SYSTEMS

<table>
<thead>
<tr>
<th>Applications</th>
<th>Integrated Rendering Capabilities</th>
<th>Support for Interaction Mechanisms$^a$</th>
<th>Support for HMDs</th>
<th>Support for projection-based display devices</th>
<th>Support for various input devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAVElib$^b$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IRIS Performer$^c$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>WorldToolKit</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Avocado$^d$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bamboo$^e$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VR Juggler$^f$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

This table shows the trends of features provided by VR development systems with respect to graphics and interaction. As indicated by the frequency of checks, integrated rendering capabilities and support for interaction mechanisms are the most common.

TABLE II
SUMMARY OF NETWORKING CAPABILITIES OF VIRTUAL REALITY DEVELOPMENT SYSTEMS

<table>
<thead>
<tr>
<th>Applications</th>
<th>Distributed Shared Memory Model</th>
<th>State Update Synchronization</th>
<th>State Consistency</th>
<th>Persistence</th>
<th>Client/Server, Star Topology</th>
<th>Distributed, Chained, or Pier to Pier Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAVElib$^g$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Avocado</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CAVERNsoft</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>World2World</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bamboo$^h$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ravel</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

This table summarizes the networking capabilities of VR development systems. The frequency of checks shows that state consistency is one of the more commonly supported networking features that appear here, followed closely by DSM and state update synchronization. Finally a star networking topology is most common.

$^a$ This involves programming components such as callback mechanisms, event classes, and built-in mechanisms provided for the development of common VR interaction techniques such as selecting an object.

$^b$ Support is provided through either OpenGL or Performer.

$^c$ Also supports OpenGL.

$^d$ Support is provided through Performer.

$^e$ Support is provided through Performer.

$^f$ Support is provided through Performer and OpenGL.

$^g$ Supports only rudimentary networking features, such as the ability to multicast tracker data on a given networking address and port.

$^h$ Bamboo currently uses the ACE library to support state update synchronization and consistency.
2.4 Evaluation of Development Systems for Collaborative Virtual Reality

Despite these advancements, many aspects of a CVR application are nevertheless left up to the developer. One consideration is how the various attributes that make up the state of an object and its behavior (determined by functions that operate on these attributes) are associated with a graphical component in a VE. Another consideration is how the impact of change can be reduced to as few modules as possible if a new behavior or attribute is to be added. In a CVR application a subset of these attributes is often part of a global shared state. Performing an action on object will often result in some predetermined behavior that will alter the state of the object. Thus, when a user in a SVE picks up a graphical model, the application must transfer the position and rotational data for the given model over the network. The recipients must then determine what local model this data refers to. Moreover, how this data is propagated through components of the application on the sending end, over the connection and through components on the receiving end (and vice-versa) is also left to the developer. The choice of what data is to be transferred over a particular connection is another issue that the developer must consider. In summary, it has been demonstrated that, even with the existing toolkits and frameworks there are many design issues that make the development of CVR applications a particularly involved process. To compound this matter, expertise is required in almost all aspects of computing including graphics, databases, distributed systems and system architecture.

2.5 Collaborative Virtual Reality Applications

In this section a representative sample of CVR applications is presented. The features of these applications are summarized at the end of this section (see Table III). This will provide the basis for the topology of CVR presented in the next chapter.

LIMBO

Limbo is an application designed to demonstrate the basic features of a CVR application. By exploring Limbo, developers can quickly get an understanding of how to develop applications for CVR. Its main features are the ability to load models into a VE at run-time, the collaborative manipulation of these models, co-presence through avatars, and persistence of the VE through CAVERNsoft and a designated server. It has been adapted to group of applications and has been successful in disseminating CVR development expertise (9).
NICE
The Narrative Immersive Collaborative Environment permits the exploration of CVR as a learning environment for young children. In this environment the participants can collaborate on the design, planting and harvest of a virtual garden. Similar to CALVIN, a central server provides persistence for the garden, and controls its growth over time. This allows users to return to the VE and see effects of other users as well as the passage of time. A web-enable interface also permits a desktop view of the garden. This explores a heterogeneous distributed system composed of both projection-based VR and desktop systems (23).

Vmail
This application explores the possibility of extending desktop-based email into VR. In this system users can leave voice messages, combined with avatar gestures and positional information. This permits the annotation of a virtual environment, and allows users to work together asynchronously. When the recipient plays back a Vmail, the user sees the reenactment of the sender’s avatar with the recorded audio message. As in the previous applications a central repository is used to achieve persistence (24).

CALVIN
The Collaborative Architecture via Immersive Navigation application enables users to collaboratively design an architectural space. It is one of the older applications in this section and hence concepts such as virtual co-presence (the perception of other users in the SVE) and persistence of the environment were relatively new areas of research at the time of its creation. Virtual co-presence is realized through avatars, while a central repository of design spaces enables persistence. CALVIN also explores the ability to simultaneously view the environment from multiple perspectives. In this application both an outside in, and inside out perspective of the world is displayed. (25; 26)

Motorola Drop Test Visualization
This application is a visualization of the consequences of impact a pager undergoes when it is drop-tested. Motorola typically drop-tests its products to study how their external housing may be improved to protect the internal circuitry. By using an immersive environment to display simulation data, designers are able to place a cutting plane through the pager and visualize the impact frame by frame. This gives them a much better understanding of what components fail at the point of impact as opposed to typical observations, which are generally reduced to pass/fail qualifications. The unique aspect of this application
is the ability to synchronously play a frame-based animation so that participants in the SVE are simultaneously viewing the same frame. This permits designers to interactively discuss the visualization at various points of interest during the drop-test (9).

**NPSNET**

NPSNET is a test bed for large-scale CVR that was developed by the Naval Postgraduate School in the spring of 1990. Since that time it has undergone several iterations, NPSNET-IV being the last, which was developed in 1996. It is a landmark CVR application environment because it was one of the first and largest of its type. It uses Performer for its rendering needs, offers 3D spatialized sounds, integrates many types of user interfaces, supports an assortment of world databases, and employs various types of Avatars. In this VE Avatars are not only represented by graphical representations of humans, but also as an assortment of military vehicles. NPSNET was designed to support anywhere from a hundred to a thousand remote participants. For this reason, it supports four levels of detail (LOD) for the display of models in order to maintain frame rates between 10 and 12 Hz. The choice of LOD involves displaying a lower quality graphics model when it is greater than a set distance from the user (6).

**Creating a Live Broadcast from a Virtual Environment**

This application describes an approach to CVR that is quite different from the SVEs presented in this section. It involves collaborating users from both a head tracked VE that use HMDs as well as conventional desktop workstations. This work (27) focuses on what the authors describe as “inhabited TV”, in which a TV broadcast is combined with a SVE. This application was tested in a live broadcast at ISEA in September of 1998. In this event participants competed in various games formatted for the broadcast of a live game show. The theme of these games involved two teams competing to escape from a doomed space station. Each team had a tracked (head and two hands) leader who was using a HMD. Each team consisted of four members who participated from desktop workstations. Desktop participants navigated through the VE with a conventional PC joystick. Most of the games required the collaboration of the desktop participants with their team leader to achieve some goal. For instance, one game required the desktop participants to guide their leader through the VE so that they could catch “space fish”.

In addition to the team leaders and desktop participants, there were also four human controlled virtual cameras. These exhibited a unique set of viewpoint control features that differed in their increased user
interface control complexity as compared to the joystick-controlled travel techniques utilized by the other applications mentioned in this section. Also unique to this application was the use of constrained travel techniques that were enforced on the desktop participants. This permitted another participant, the “World Manager”, to enforce “crowd” control on desktop participants in which bounding boxes constrained travel, as well as herding participants to new locations ensuring that everyone remained together and did not get lost in the VE.

### TABLE III

**SUMMARY OF COLLABORATIVE VIRTUAL REALITY APPLICATIONS**

<table>
<thead>
<tr>
<th>Applications</th>
<th>Runtime Loading of models</th>
<th>Travel</th>
<th>Synchronized Animation Display</th>
<th>Collision Detection</th>
<th>Object Manipulation</th>
<th>Avatars</th>
<th>Viewpoint control outside of Travel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limbo</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NICE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Vmail</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CALVIN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Motorola Demo</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NPSNET</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Live Virtual</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Environment Broadcast</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

This table summarizes the features of a sample of CVR applications. The trends show that Travel, Object Manipulation and Avatars are the most commonly supported features of CVR.

### 2.6 A Component-based Approach to Collaborative Virtual Reality

After the survey of VR systems and applications presented in this chapter it is evident that this domain is in constant flux. This affects nearly all aspects of CVR including the hardware systems and their drivers, the underlying graphics libraries, and the networking systems. In addition there does yet exist a standard set of interaction techniques, and new ones are actively being investigated. It is a well-known fact that development is expensive, both in terms of human resources, equipment and time. It is true that equipment costs have been steadily dropping, however the high-end still remains well beyond the budget of many research departments. A framework that focuses on CVR must adapt to these factors.
For these reasons, a component-based solution is necessary for a framework that is to capitalize on previous development effort while remaining flexible enough to keep pace with an ever-changing environment. It is the transition from the early research stages of rapid-prototyping and throwaway architectures. A component-based solution is ideal because it allows the underlying graphics and networking layers to remain transient, and later be replaced with minimal impact to the rest of the framework. Because software development is inherently expensive, encapsulating interaction techniques into a single component lends to the creation of a repository that encourages reuse. A component-based architecture allows for the seamless integration of existing components by a simple selection process from this repository. In addition, new interaction techniques can be created by extending existing components or by creating new ones. In the next chapter the requirements of CVR are presented. An analysis of these requirements will result in a set of high level components that specify the basis for a framework in this area.
3 REQUIREMENTS AND ANALYSIS OF A FRAMEWORK FOR COLLABORATIVE VIRTUAL REALITY

In this chapter the domain of CVR is analyzed. An examination of the current applications and systems for CVR leads to the specification of a set of requirements for a framework in this domain. This is achieved by a careful identification of the patterns and similarities that exist within this work, which leads to a list of the required features. Once this is achieved the features must be classified and prioritized. This extends to the requirements that are later satisfied by the design presented in the next chapter. The classifications of the requirements for CVR are covered in this chapter. It is beyond the scope of this thesis to present an exhaustive account of the analysis process, this treatment is better left to a text on OO analysis; Instead each of the requirements outlined in the chapter are to be considered the result of an analysis process. As such, they are crucial to the framework and of high priority. Lower priority requirements that are not implemented are not discussed in this thesis.

Before this analysis can begin, it is necessary to first examine the non-functional requirements. These types of requirements specify issues related to the properties and constraints imposed on the development of this framework. In many cases it is essential to first establish these constraints before specification of the functional requirements. To illustrate this point, consider the constraint of maintaining frame rates of at least 15 Hz. Even with today’s state-of-the-art, this eliminates the possibility of specifying the use of conventional ray-tracing techniques as a rendering mechanism the desired performance would not be achieved. In many cases non-functional requirements result in the selection of an existing software system. Where appropriate the rationale behind the choice of this system will be explained. These requirements are presented in the first section on the system properties for this framework. A summary of the non-functional requirements is seen at the end of the chapter in Table IV.

The next sections focus on the system requirements of the framework. They are divided into three categories: graphics, interaction and networking. The graphics requirements focus on such items as loading of models into the environment and the ability to associate various state and behavior with graphic components. The interaction requirements focus on the types of interaction techniques the framework must enable. This extends to a discussion of the provisions necessary to allow users to interact with one another
as well as with the environment. The networking requirements will establish what is needed to achieve collaborative interaction. Finally, the networking requirements will also cover issues of achieving state synchronization, consistency and persistence.

Further analysis of the requirements within each of these groups will lead to the construction of a high-level classification system. This will build upon methods presented by Bowman in his dissertation on interaction in VEs (28). In this work the author outlines a method for creating a taxonomy, which he describes as “the science of classification” and a “specific classification”. The author divides interaction in VEs into two separate categories, Travel, and Selection and Manipulation. A taxonomy is provided for each of these categories. These taxonomies will be extended to the domain of CVR.

3.1 System Properties

Over the last decade many definitions have been given to the concept of Virtual Reality. Some would argue that simply displaying 3D models on a desktop PC connected to the Internet constitutes VR. However, besides simply enabling the development of CVR applications, an underlying goal of Tandem is the ability to produce highly immersive, interactive and compelling VEs. It is difficult to achieve this through a desktop PC environment. The desktop PC suffers from a narrow FOV and does not afford the user a space that one can physically move about in. Furthermore the low-end desktop system cannot compete with the performance of high-end machines in areas like texture mapping and rendering ability, thus limiting the visual detail. All of these deficits produce a desktop experience that pales in comparison to a state of the art high-end system for VR. The type of VR that Tandem is geared towards, is one that achieves the highest degree of presence and immersion. There are several key factors to achieving this. First a user-centered perspective must be provided through a 6 degrees of freedom (DOF) head-tracked display. Next a sense of depth is to be provided by stereo shutter glasses. It is generally accepted that a wide FOV greater than approximately 100 degrees increases immersion. In line with maximizing the immersive experience, projection-based technology is to be considered. For the purposes of interaction an input device is required. This is generally satisfied by a six DOF tracked 3D-mouse device, which offers 3 buttons and a joystick. A device known as the “wand” satisfies this requirement. Finally, a frame rate of at least 15 Hz, in stereo, is to be maintained in order to preserve immersion and presence. Informal testing in a
four wall CAVE displaying stereo graphics has shown that a dual pipe, 8 processor Onyx 2 with IR graphics permits a scene to be rendered in the range of 60,000 polygons. Beyond this range, the display rates deteriorate below 15 Hz, detracting from the sense of presence and immersion. Hence from this point onward when VR is mentioned, this description is the type of VR experience that is being referred to.

### 3.2 Graphics

#### 3.2.1 Rendering

The most basic requirement of VR is the ability to render graphics to some display device. Without this ability, none of the other categories can even be considered. This requirement is not specific to the domain of VR, but any form of electronic visualization. The distinction between the rendering requirements of VR and those of other electronic visual mediums is that instead of displaying static rasterized images, in VR three dimensional models must be rendered in stereo, in real-time. In addition, they must also lend to an environment that is both compelling and immersive, leading to the notion of a VE. Part of this involves the rendering of complex scenes with convincing lighting models. In order to achieve the performance required, VR requires efficient culling of primitives from the scene. For this reason a careful balance must be struck between visual detail, and the performance of the display loop.

#### 3.2.2 Model Loading

Because of the visual detail required it is often far too time consuming to specify models in terms of low level routines that draw primitives. Often, the best route is instead to model graphics objects in a 3rd party modeling package such as Alias/Wavefront or SoftImage and export them to a format that is importable by the graphics library used. Model loading may occur at various points during the execution of an application. At the start of an application an initial set of models are loaded, forming the entrance point of the VE. Models may be loaded when a new participant joins the environment in the form of an avatar. In addition participants may chose to share new models or user interaction may result in a model being added to the environment. In a distributed environment, remote nodes must dynamically load models at run-time, in response to asynchronous network events. In these instances, the loading of a model cannot interfere with the display and interaction of the application. Thus, in order to maintain an interactive display rate of 15 Hz, the system cannot stop responding to user input until a model is finished loading.
3.2.3 Intersection Testing
In order to provide interactive graphics another important requirement is the ability to efficiently calculate intersections. This is often done by ray casting methods that specify the start position, direction and length of an invisible ray of light. The graphics system must then determine what primitives this ray intersects. An alternative approach is to use bounding volumes. In this method the system must determine whether the bounding volume of one object intersects with that of another.

3.2.4 Selection of a Graphics Library
Since it is far beyond the scope of this work to construct a graphics library, the requirements of this section are meant to guide the selection of an existing graphics library API. For these requirements IRIS Performer is the solution of choice, because it utilizes the hardware to its fullest potential, with little intervention from the developer. This permits the application to attain the required frame rates. In addition there are facilities provided to maintain these rates, by dropping levels of detail if the performance declines. It also provides efficient culling, bounding volumes and intersection testing. There are also many Performer loaders for a wide range of formats used by various modeling packages. There are also facilities for implementing background, asynchronous loading of models. In addition, Performer provides a conversion to a Performer binary format for each of the accepted format types that optimizes the rendering performance.

The experience of the author is that the most successful model format tends to be Inventor/VRML 1.0. It is a text format that is easily edited by hand, allowing small adjustments outside of a modeling package or the importing graphics library. It also preserves the proper scale when loaded into Performer and allows the specification of texture file path names from within the text file.

3.2.5 State and Behavior
These requirements involve not the graphics themselves but data and behavior about the graphics. Loading a model, rendering it into the scene and displaying it on the desired VR display device is only part of the desired functionality of a CVR application. The next set of considerations involves the capabilities of a graphic object.
3.2.5.1 Autonomous Behavior
Autonomous behavior describes the capabilities of an object independent of being manipulated. For instance, a fan blade may spin on an axis whenever it is turned on. The act of turning on the fan may trigger the spinning action, however it will continue to spin independent of user interaction until it is turned off. An object must be able to exhibit one or more autonomous behaviors, and the application should be able to determine the selection of a particular behavior dynamically, associating a given behavior with a particular scenario.

3.2.5.2 Properties
Another consideration is the ability to associate various properties with a particular graphic object. In a given VE there may be several classes of objects, each with their own distinct properties. For instance, an application that simulates mechanics may associate a mass, material, physical dimensions and possibly aerodynamics with each object in the scene. These properties should be modifiable, extended and deleted during the execution of an application.

3.2.5.3 To act or not to act
It also important to consider what actions a user can perform on a graphic object, what events the object responds to and how the actions performed on the object can be carried out. For instance certain objects may be picked up, while others may be securely fastened, and hence not respond to the action of being grabbed. Also, when thrown some objects may possess properties that enable it to glide, while others may simply drop like a rock. Some objects may even carry out their own behavior based on a user’s proximity.

3.2.5.4 Interactive Graphics Components
All of these require careful consideration if a framework is to support them. Of course the goal is not to presuppose the actions, events and behavior of graphics objects, instead the requirement is to provide components that enable the user to build their own. Furthermore, these components must be easily extended, and combined with one another. They are meant to be the “clay” from which developers can mold their interactive graphics behavior. A further constraint is that the application must be able to change and manipulate these components at run-time. Moreover, the behavior of an object should be modifiable. It should also be possible to copy a given behavior from one object and paste it onto another object, such that both objects would possess the same behavior.
3.3 Interaction

The interaction requirements of this section are not entirely independent from some of the requirements already mentioned. The requirements of Graphics State and Behavior certainly fall into this category. However, this section intends to provide a complete picture of the interaction requirements. The goal of providing interaction within this framework is not to implement specific interaction techniques. The goal is also not to investigate all of the types of interaction within VR. Based on previous work that investigates the domain of interaction within VE, and the representative set of sample applications in the previous chapter, the objective is to specify the requirements for a group of high-level, abstract components that enable the construction of specific interaction techniques. This is similar to a GUI toolkit that provides such components as scroll bars and list boxes. In such a toolkit, seldom does one see an implementation of a specific GUI. Instead a set of basic components are provided from which a complete GUI is constructed. These high level components are not meant to be refined to the extent that 2D interfaces have been, since no standards yet exist for interaction within a 3D environment. Hence, they will remain more abstract than widget sets for 2D interfaces. In order to develop these abstractions the interaction requirements are first divided into the following groups:

1. Audio feedback within a VE
2. Travel in a VE. This involves changing the user's viewpoint.
3. Selection and manipulation tasks

Based on Bowman’s (28) work a taxonomy for groups two and three is presented. An analysis of these requirements leads to a to a high level set of components suitable for building a robust set of reusable interaction techniques.

3.3.1 Audio

In almost any modern multi-media environment, audio generally plays a role in interaction. Audio is often used as a feedback mechanism, or as component of the application itself. Within a VE audio plays several roles. In terms of immersion, it can be used to accentuate the degree to which the user feels part of the VE. That is, proper use of sound can compliment the visual element providing a multi-modal experience that plays on the user’s emotions, imagination and intrigue. Audio can also enhance the amount of information the user can collect, comprehend and synthesize from the VE. As a frequently used HCI mechanism it can be used to provide feedback for a given task. Audio feedback can be used to indicate to
the user that a task has commenced or as an indication of task completion. The favorable property of audio feedback is that it requires relatively little cognitive load, does not require the user to be looking in a particular direction, and in general if not competing with other sound in the user’s environment is difficult to be missed.

### 3.3.2 Travel

Travel as defined by Bowman (28) involves “viewpoint motion control” and is one of the principle requirements of interaction within almost any VE. Travel is to be distinguished from wayfinding, which involves the cognitive process of using visual and/or other cues within the environment to travel to a desired destination. Travel combined with wayfinding defines the task of navigation. In this section only the travel component of navigation is considered since other aspects are covered in other sections. For instance, the requirements of wayfinding may use such devices as maps, or other visual aids. This type of requirement is satisfied by the state and behavior requirements of the graphics section. Bowman provides two taxonomies of travel techniques in his work. The first is seen below in Figure 1. This taxonomy divides travel into three main task groups. The first group involves direction/target selection. This task set determines the facility employed by the user to travel to a desired location. For instance, in a head-tracked system, one technique is to use the direction of the user’s head, to determine the direction of travel. Another method involves using a tracked input device to determine the direction. And yet another technique that is employed by cosmo3D is to provide a discrete list of viewpoints from which the user can select. A similar method is to allow the user to select specific objects in the world, which will initiate travel to that location. Finally, the last approach considered, is to ignore the dimension of depth and treat the scene like a 2D plane. In this scheme whatever selection is made, is based on the intersection of the user’s cursor with an object in the scene, projected onto a 2D plane, or vertical slice of the VE. This technique is most easily envisioned on a desktop where the entire scene is rendered on to a two dimensional plane by the projection matrix.
Bowman further simplifies these techniques in Figure 2. This simplified version provides the basis for the travel requirements since Figure 2 covers all of the techniques seen in Figure 1. This second, simplified taxonomy is divided into four tasks: “Start to Move”, “Indicate Position”, “Indicate Orientation” and “Stop Moving”. In this particular model, the start of travel can be viewed as an event initiated mechanism. In general this event occurs as a result of some user triggered interaction, such as the press of a button,
movement of the joystick, or physically walking in the space. Of course an automated, continuous travel scheme would eliminate the need for this task. The next task is the indication of the position to travel to. This is further subdivided into the selection of the destination, the velocity and the acceleration. The selection of the destination can be a discrete selection from a list or an object of the environment. Alternatively a route-based approach that controls curvature and distance (similar to manipulating control vertices of a bezier-curve) may be used. Finally, the position may be selected by continuous specification invoking techniques described in the first taxonomy such as gaze-directed, and pointing. In general the velocity is often constant with no acceleration, though these characteristics have been successfully employed in combination with an object selection technique whereby the user selects a target, travel to that target accelerates to a maximum velocity and then decelerates to zero upon reaching the destination.

A requirement that is seen throughout this chapter is the necessity for extension. Since this framework is not going to provide every conceivable travel technique, mechanisms must be in place to allow the development of customized techniques, similar to those appearing in the above taxonomies. Finally, like the requirement of dynamic behavior of graphic components, the ability to change the travel technique at run-time is also necessary. That is, an application may require the use of a specific technique in a particular area of the VE and may chose to change this technique in another area. The framework must provide this facility.
### 3.3.3 Selection and Manipulation

Selection and manipulation covers the bulk of interaction within the virtual environment after travel.

Selection is the task of choosing the graphical object the user wishes to interact with. Almost every user interaction begins with a selection task. What follows after this task defines how the user interacts with the VE, and is defined as manipulation. Finally after the desired manipulation, the task of releasing the object occurs. These tasks are depicted in Figure 3, and form the three main groupings.

In this diagram the selection task is further subdivided into three subtasks: feedback, indication of object and indication to select. Feedback is the type of indication given by the system that informs the user of the various states of selection. The second subtask of selection, is the indication of object. That is the means by which the user determines the object they wish to select. As an example of indication of object,
graphical feedback could be given by changing the color of a virtual hand or pointer when an invisible ray extending from the pointer intersects an object. Finally the indication to select, is the mechanism used to actually perform the selection. In the CAVE environment a wand button is usually reserved for selection.

Manipulation is divided into four subtasks. Object attachment refers to the change in position of the object at the start of manipulation. For instance the “object moves to hand” task means that the selected object is translated to the position of the virtual hand at the start of manipulation. Object position refers to how the object is moved with respect to the user during manipulation. To illustrate this task, an indirect control could mean the use of a translate tool that moves an object at fixed intervals. Object Orientation refers to how the object is oriented, and is similar to the idea of position, except this task deals with rotation angles instead of translation coordinates. Finally the feedback task determines the type of feedback given during all of the various states encountered during the manipulation.

The release task is the most straightforward of the three, and simply refers to how the user terminates the manipulation of the object. This involves two subtasks, the indication to drop and the object final location. The former indicates that the user has chosen to terminate manipulation. The latter indicates the resulting position and orientation of the object when the manipulation task ends.

An example taken from the CAVE illustrates a common method for selection, manipulation and release in this environment. One way to grab objects in the CAVE environment is to use a skewer metaphor. That is, first the user selects the object by pointing at it with the virtual pointer and pressing and holding a wand button. The object is skewered by the virtual pointer while the wand button is pressed. The object is then transformed (translated and rotated), by the movement of the user’s tracked hand until the button is released. The object remains at the position and orientation it is in at the time the button is released.
This diagram shows the three main components of selection and manipulation tasks in VR. Each of these tasks is further subdivided into the classification of activities that make up the task.
3.3.4  Interaction Abstracted

Now that travel, selection and manipulation tasks have been examined it is possible to create an abstraction level that encapsulates the common properties of these two categories. The two main abstractions are *Events* and *Actions*. An event is the occurrence of a particular activity. It tells the system that an activity has commenced. An action is what occurs as the result of this event. That is, actions express events. The following two sections explain how these constructs can be used to cover the requirements for interaction. These abstractions are illustrated in Figure 4. In this diagram three tasks related to actions and events are presented. For each of these tasks, the possible methods of invocation are displayed. A detailed discussion follows.

3.3.4.1 Events

The first abstraction comes from the fact that almost every interaction is inherently event-driven, or derived from some event that is triggered. This is common to both travel and selection. In the case of travel, Figure 2 illustrates this by the task “Start to Move”. Similarly, this follows for the selection task, as both the indication of object and indication to select can be classified as events. The feedback is also generally event driven, usually resulting from some condition holding true. Even in the case of “no explicit command”, some condition has to be met, which in turn triggers an event. The occurrence of an event can result in several possible outcomes. In a closed-loop situation, the user triggers an event that the system responds to, and the interaction terminates. A cascading situation is one whereby the occurrence of an event triggers one or more events, which may in turn trigger other events. The conclusion of this interaction occurs when all cascaded events terminate. Finally the last situation is where the occurrence of an event triggers a condition that results in the repeated occurrence of some activity until another events terminates this behavior. Whereas events signal the occurrence of some activity, they also specify the termination. This is seen in the travel taxonomy by the “Stop Moving” task and in the selection/manipulation taxonomy by the “Release” task.

3.3.4.2 Actions

Actions are the expression of events. Whereas an event is the occurrence of some activity, an action defines how an activity is carried out. For instance, in a 3D-widget system, the selection of a particular menu item is the event, but the operation of some device, or performing some activity, is an action. In the travel taxonomy an action refers to the act of moving through the VE. This may be used in combination
with events for the purposes of selecting a direction, but the actual change of viewpoint can be considered as the action of travel. Going back to our selection and manipulation tasks, a user may trigger a select event by pointing at an object and pressing a button. The manipulation of the object is determined by an action that specifies how this manipulation is performed. This action is carried out until the user terminates it by some event.

**Figure 4 Taxonomy of Interaction Events and Actions**

This diagram illustrates an analysis of travel, selection and manipulation tasks presented in Figures 1-3. From a requirements analysis level, the tasks presented in the first three diagrams can be broken down to the activities presented here. In this diagram each activity is further subdivided into its types of invocation.
3.4 Selection of a VR System Library

Given the interwoven requirements of the system, the graphics and the interaction, there still exists a supporting layer necessary to tie all of these factors together. That is a VR system library capable of displaying Performer graphics to projection-based displays as well as providing access to the various tracker and user input devices. A suitable library that provides the required features is the CAVE library. It is flexible enough as to allow the configuration of an application for various display, tracking and input devices without rewriting the application code. It is also available on the IRIX, Linux x86 and HP-UX platforms. For this reason it is a good fit for an underlying layer of the framework’s graphics and interaction requirements.

3.5 Networking and Collaboration

The final set of requirements specifies what should be provided in terms of networking, and what is required to provide collaborative facilities. In addition to the taxonomies presented above, CVR dictates that nearly all of the travel, selection, manipulation, event and action categories should have a networked counterpart. In addition to this, the ability to share a model is also necessary. To elaborate, once a user loads a model into the environment, the user should be able to share it with others so that any manipulation to this model is seen globally by all participants. The requirements of a networking system that allows for these tasks to be carried out in a shared environment are presented below.

3.5.1 Distributed Shared Memory

The fundamental principle of the networking component is the ability to share state, that is the creation of a DSM model. In this model a change to a DSM attribute affects all other nodes connected to that attribute. The system should encapsulate the DSM module in a way such that the system as whole is not responsible for determining what nodes must be updated.

3.5.2 Connections and Asynchronous State Updates

Typically providing a DSM involves a connection of some sort to other node(s) on the network. For a given connection, the developer should have the choice of what type of update is sent across this connection. That is, a connection can be specified as a reliable or unreliable connection. In the former, every message sent across a connection is guaranteed to be handled, and in the order that it is sent. In the latter, messages may be lost and received out of order, but this requires much less system overhead and
hence can achieve a lower latency. A reliable connection is one method of ensuring state consistency.

Synchronization is achieved by determining what protocol is used in handling attribute updates. As an example consider two nodes “A” and “B” and a shared state $\alpha$. The synchronization protocol determines how $\alpha$ is kept in sync after both A and B perform updates to $\alpha$. One method is to use a global time, such that a state is synchronized by a chronological order of updates. Another method is to specify a particular node as the reference node and perform content and time comparisons against that node.

These updates occur at any time. That results from the fact that a user can perform an action at any time, and the fact that networks cannot currently guarantee a fixed time of delivery. For this reason the system must be able to handle asynchronous updates. Careful measures must be taken so that they do not interfere with what is currently being processed, and race conditions are avoided.

### 3.5.3 Serialization

Since this framework is implicitly object-based, a convenient method for updating state is to simply send an object across the network. This is known as serialization. For this framework it is not required to provide an entire set of CORBA-like facilities. However a scaled-down interface that enables some degree of serialization is necessary. This will provide a system protocol for sending object data across the network, and specify an interface for implementing the send and receive methods of serialization. This will allow the development of network events that can be sent and received with relative ease through the use of the serialization protocol, the connections and DSM.

### 3.6 Extensibility, Flexibility, and Maintainability

Throughout this work these terms have been mentioned. They are ubiquitous throughout the OO literature and especially where frameworks are concerned. Nevertheless they are essential to the development of a successful system that resists an ever-changing environment.

An extensible system is one in which clients can easily add new features. Generally this is done by inheritance, in which a client class overrides or extends existing functionality. This need arises throughout the graphics and interaction sections. New attributes and properties must be added to describe graphical elements of a VE. Furthermore new behavior must be easily incorporated into these elements. This enables the development of new techniques for interaction and Travel. The intent of this work is not to provide a
library of every conceivable interaction technique. The goal is to allow clients to develop, catalogue and
deploy their own techniques. In this manner, components can be placed into repositories. Components can
then be extended and combined with new ones. This permits reuse in applications with similar needs, and
prevents wasted development time spent on repeatedly creating the same components. As repositories
grow, behavior and interaction can become increasingly for complex. It is intended that a component-based
solution will thus lead to the creation of rich VEs. These needs dictate a system that must be extensible.

Flexibility of a system describes the degree to which the system can adapt to new configurations. This
means combining its components in new ways, changing underlying layers, or porting and adapting new
systems to the framework. The more flexible a system, the more resilient it is to a changing environment.
This system requires flexibility in its choice graphics and networking subsystems. It also must be flexible in
the way in which new state and behavior is added to the system. Extensibility and flexibility lend to
maintainability. Maintainability describes the relative ease in which a system is able to adapt to a changing
hardware and software configuration, in essence its ability to evolve.

The ability to add new features, port the system to new environments and allow for client
customization, is essential to a framework that is to serve any purpose to a domain that is rapidly changing.
A framework for CVR must be designed for change. This change could take place anywhere from its
rendering engine to its networking capabilities.

These needs are addressed by design patterns. They play an important role in providing an extensible,
flexible and maintainable solution. The catalogue of design patterns from which Tandem is built caters
specifically to these needs. In the next chapter the framework is examined in detail. This chapter will
illustrate how Tandem meets the needs of CVR outlined in this chapter. In particular it will also investigate
what patterns are most appropriate to the domain of CVR, why they have been chosen, and how they are
used.
TABLE IV
SUMMARY OF NON-FUNCTIONAL REQUIREMENTS

<table>
<thead>
<tr>
<th>Non-Functional Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Support for VEs with high degree of presence and immersion</td>
</tr>
<tr>
<td>• User centered perspective through a 6 DOF head-tracked display</td>
</tr>
<tr>
<td>• Depth cues through stereo shutter glasses</td>
</tr>
<tr>
<td>• FOV greater than 100 degrees</td>
</tr>
<tr>
<td>• A 6 DOF input device providing at least 3 buttons and a joystick</td>
</tr>
<tr>
<td>• A stereo frame-rate greater than or equal to 15 Hz</td>
</tr>
<tr>
<td>• Efficient culling, bounding volumes and intersection testing</td>
</tr>
<tr>
<td>• Extensible, flexible and maintainable architecture</td>
</tr>
</tbody>
</table>

This table summarizes the non-functional requirements presented through this chapter. These requirements deal with the performance and system constraints.
4 THE FRAMEWORK

Building Collaborative Virtual Reality applications is a challenging and involved process. The intrepid developer must be skilled in many aspects of systems programming, and attain mastery of the necessary hardware and software. To make matters worse, the entire landscape of VR is constantly shifting. At one moment the graphics library of choice is OpenGL and the next it is Performer. Today SGI Onyx 2s with IR graphics offer a suitable platform for the VR system described in the system properties of the previous chapter, tomorrow no one knows what this platform may be. For this reason Tandem employs a component-based approach to its architecture.

4.1 A Component-Based Approach to Collaborative Virtual Reality

A component-based approach emphasizes a “pluggable” architecture in which its various components are easily replaced with client customized versions. This lends to the construction of repositories that catalogue past development effort. In essence, a component-based approach is a well architected OO system that places customization and flexibility as top priorities in its design. This approach holds the assumption that almost any major component may be change by a client in the future. Consequently, many of the algorithms that make up the execution loop of the framework must be designed to account for these factors. This results in careful planning of the component interfaces. The key to customization is to abstract and separate those parts that are likely to change from those that are more likely to remain the same. This is the driving force behind design patterns and is in fact a major criterion for determining that a particular design pattern is a better solution for a given design issue over a more restrictive solution.

4.2 Rationale Behind a Component-Based Approach

It’s not entirely certain that a framework is a viable approach to developing CVR software. Undoubtedly it aims to make the task easier, and in turn lessen the development time. However, one of the reasons that CVR, and VR in general, remains an active area of research is that no consensus has yet been reached on what systems are best suited to its development. Furthermore it remains an open question to the research community to determine what applications of art and science are best suited to the domain of CVR. For this reason, much of the current CVR software development remains in the area of exploratory programming.
and rapid prototyping. As of yet, there does not exist the “killer application” that justifies the existence of VR as a true sign of progress, and solidifies its contribution.

An entirely different approach to developing a complete domain framework for CVR is to create small reusable modules that can be retrofitted to existing applications. This certainly is much more difficult than developing an application for the domain of CVR from the onset (10). However, this approach enables the application of CVR technology to many domains that remain untested in this area. An illustration of this approach is Cavernsoft. This system is designed to allow for the addition of networking capabilities to applications that were not originally intended for this purpose. Cavernsoft enables tele-immersion by maintaining a distinct separation between the graphics and networking requirements of this domain. As such, it caters only to the networking aspects of CVR. This approach is different from that of Tandem, in which many layers of software operating in parallel are integrated into one system.

Nevertheless, Tandem strives to succeed as a framework, a system for the development of CVR, and as a set of reusable components that may be employed outside of Tandem’s architecture. As a framework Tandem is an investigation of known software patterns that are most applicable to CVR. It offers a solution for the problem space outlined in the previous chapter. As a set of reusable components, many of Tandem’s features can be extracted and use outside of its architecture. This is strongly encouraged by the use of design patterns. By satisfying these goals Tandem enables the investigation of other areas of VR research outside of system development. This involves such questions as:

“What are the principle differences between traditional forms of collaboration and those achieved in CVR?”

“What constitutes a complete set of tools necessary for the development of CVR”

“How can CVR improve aspects of our day to day life that could not be achieved in other mediums”.

4.3 The Role of Design Patterns in Tandem

Before delving into the details of the architecture it is first helpful to see an overview of how design patterns have contributed to Tandem’s design. This architecture relies heavily on design patterns to satisfy the need for a component-based approach. A list of the patterns that make major contributions to Tandem
are presented here. In this overview the name of each pattern accompanied by a brief description of their contribution is given. All of the patterns presented below, with the exception of Multicast/Typed message (16), are presented in the GOF text (12). The reader is encouraged to consult these texts for a more in depth treatment of each. The major patterns will be described later with respect to the components that employ them.

**Factory Method:** Provides an interchangeable architecture for the networking, interaction and application components.

**Template Method:** Enables the customization of algorithms throughout the system. This includes the interaction control loops as well as the networking application protocols for serialization, and the sending and receiving of networked events.

**Strategy:** Permits the construction of a repository of travel techniques as well as a system for interchanging these techniques at run-time.

**Command:** Enables a loosely coupled system between interactions that trigger events and the entities that must respond to the event. This also permits the interchanging of encapsulated behavior and event components between graphics components at run-time.

**Composite:** Provides a mechanism for recursive decomposition that is responsible for the hierarchical tree structure of graphics components. This allows graphic components to be treated uniformly.

**Multicast/Typed Message:** Sets up a notification mechanism that allows a broadcast of the occurrence of a particular action to any number of handlers. Each action type is presented to the handler without the need for downcasting a base class type. A handler may respond to multiple actions.

**Adapter:** Facilitates the integration of other graphics and networking subsystems.

**Visitor:** Provides a mechanism that handles the lower-level details of the graphics interaction emphasizing a separation of concerns. This permits a framework that allows clients to deal with different levels of granularity for graphics interaction, and allows the underlying algorithms and structures to change independently of high-level client code. This also allows for a repository of interaction components that do not require an understanding of their underlying details.
**Iterator**: Enables a convenient mechanism for traversing the graphics component hierarchy and other structures within the framework.

**Façade**: Provides a high-level interface that hides the details of the underlying networking and graphics modules. This pattern is the motivating factor behind the manager classes that are seen throughout the framework.

Throughout the class diagrams presented in this chapter, the reader frequently encounters annotations enclosed by “<<” and “>>”. This is known as a stereotype in the Unified Modeling Language (UML), which is used in these diagrams. A stereotype allows the incorporation of customized semantic information into the language. In this case it is used to label a class with the corresponding participant name defined in the design pattern catalogue definition. In this fashion the reader can more easily map the pattern implementation presented in this thesis to what appears in the GOF text.

### 4.4 Introduction to Tandem Architecture

Tandem divides its functionality into three modules, **Core**, **Graphics** and **Networking**. The Core forms the foundation of Tandem’s architecture and includes the basic application and interaction components. The Graphics module provides everything necessary to render graphics, load models, and specify interaction and behavior for the graphical components. The Graphics module depends on Performer and the CAVE library. Performer provides the underlying graphics rendering subsystem, while the CAVE library services Tandem with the necessary routines to display graphics to projection based displays, and provides an API for tracker and wand devices. The Core combined with the Graphics module enable the construction of a VR application. The Networking satisfies the final set of requirements for CVR. It satisfies two main categories. The first provides a set of cooperating classes necessary to send and receive network events, serialize data and manage shared state. The second specifies a protocol that allows an existing networking system to be interfaced with Tandem’s architecture. As such, it is dependent on CAVERNsoft, which provides DSM, data synchronization and consistency, and persistence. Finally, throughout Tandem the Standard Template Library (STL) is used for the needs of its underlying data structures. STL provides a robust set of container and algorithm classes, together with a published set of run-time analyses, making it an ideal toolkit for the construction of efficient data structures. The relationships between these modules
are best visualized in Figure 5. In this diagram Tandem’s modules are drawn with a heavier outline for the purposes of clarity. It is beyond the scope of this thesis to discuss every minute detail of each Tandem class, however an overview of the important classes within each module is presented in the remainder of this chapter. Appendix B contains a complete API for Tandem’s classes.

![Tandem Module Diagram](image)

**Figure 5 Tandem Module Diagram**
This diagram shows the Tandem modules and their supporting subsystems. Tandem modules are presented with a thicker outline. As is shown here Tandem Networking is supported by CAVERNsoft while Tandem Graphics is supported by Performer and the CAVElib. Tandem’s low-level data structures and access algorithms are selected from the Standard Template Library.

### 4.5 Core

The Core provides the overall structure for the framework. It consists of an application class that is an aggregation of core, graphics and networking components. In addition the core provides several independent interaction classes for use within development applications. The application class and its aggregated components are presented below, accompanied by a brief description.

**Application Class**: This class is the foundation for all Tandem applications. It is an aggregation of the components that follow. Though many components are optional, this is the bare minimum from which a Tandem application is constructed.

**Audio Player**: This component provides basic audio capabilities for use within a Tandem application.
**Core Loader:** This component provides a very basic interface for initializing Tandem graphics components.

**Avatar Manager:** This provides a common base class for managing the participants and their corresponding representations within the SVE.

**Graphics Manager:** This is a façade for the graphics module and provides a common interface that hides the details of the many underlying components.

**Network Manager:** This is a façade for the network module and provides a common point of entry for application classes that need to send network events.

**Execution Control:** This class encapsulates the main control loop that drives a Tandem application.

**Interaction Manager:** This is the heart of interaction within the framework. It manages all of the core interactions that take place each iteration of the main control loop and contains hooks for client extensions by means of Template methods.

The aggregate Application class and its aggregated components are seen in Figure 6. The Application class is also responsible for creating each of its aggregated members (listed above). Because the Application class is an aggregate class, a derived client application class is able to access each of these components. This in turn allows derived client application classes to share this access with its own customized data members. By default the Application class shares access to the Graphics Manager with the Interaction Manager. Client classes that derive from the Core Interaction Manager also share this access enabling clients to define interaction with graphics components. Client classes can customize what access each component has with respect to the other components.
Figure 6 The Application Aggregate Class

This diagram shows each of the aggregated parts that make up the Application class. In addition to the core classes, the Application class also interfaces with the graphics and networking modules, as shown by the Graphics Manager and Network Manager respectively.

With the presentation of the above components, a more detailed view of the core module components and their relationships can be presented. The heart of this module is the TcApplication (Tandem Core Application) class. Throughout this module all classes are prefixed with “Tc” (Tandem Core) followed by the class name. TcApplication provides the bare minimum for creating a Tandem application. It is from this class that all other components are created. That is, it is responsible for initializing the Core as well as the Graphics and Networking modules. Developers create an application by inheriting from this class and overriding various factory methods. A factory method is a method that creates a given component type, and returns a pointer to the newly created object. In this manner clients are able to extend existing Tandem classes. For instance, one of the principle classes necessary for extending basic interaction is the InteractionManager. An application supplies an InteractionManager to the Core by following several steps.
First a class derived from TcInteractionManager is defined, for the purposes of illustration it will be called DevelInteractionManager (development interaction manager). Next the application class derived from TcApplication, overrides the createInteractionManager() method. The client implementation of this factory method returns an instance of DevelInteractionManager. A working example of this is seen in chapter 6, which presents an overview of Tandem application. Before any of the aggregated members may be accessed an initial call to the config() method must be made. This triggers each of the factory methods, and sets up the application class for use. This mechanism as employed by the core Application class is depicted in the interaction diagram seen in Figure 7. In this diagram the order in which each component is created is shown. This order must be carefully chosen to accommodate any interdependencies between components where certain components require access to others (as in the case of the Interaction Manager).
Figure 7 Application Class Factory Method Interaction Diagram

This diagram shows the order in which each of the factory methods is triggered by an initial call to the Application config() method. It also shows that the Execution Control class is instantiated last so that it may receive a reference to the other components, which were created previous to itself.

The TcApplication class is an aggregate of several entities. As already mentioned in the above example, it contains a TcInteractionManager. The other key classes are TcExControl, TcNetManager, TcLoader and TcGfxManager. This is best depicted in the following class diagram using the UML notation (see Figure 8). It is beyond the scope of this thesis to give a detailed discussion of this notation, however a quick overview is given in Appendix A, and the reader is referred to the following references for more information (29;
30). The TcNetManager and the TcGfxManager classes seen in this diagram are covered in detail in the sections on the networking and graphics modules respectively. However, briefly, TcNetManager (Core Network Manager) controls Tandem’s networking module and as such provides access to Tandem’s major networking components. The TcGfxManager (Graphics Manager) is responsible for the graphics module. Its chief responsibilities are managing the graphics components that make up the VE. The TcInteractionManager provides a protocol for updating many of Tandem’s interaction components. As such it also interacts with the Graphics module. Finally, the TcExControl (Execution Control) class encapsulates Tandem’s main thread of control. Tandem Applications extend this algorithm at various places through the Template Method pattern (described in the next section).
This diagram shows the relationships between the core Application class and all of its sub-components, which combined make up the Application class. The aggregated Interaction Manager is itself an aggregate class, made up of a set of components. Many of the aggregated classes are created by factory methods. These allow clients to customize each of the products that result from the invocation of a factory method. A factory method is invoked from a client derived class that customizes what is returned from this virtual function. These methods are triggered when a config() call is made to the instance of the creator class.

Each <<Product>> results from an object returned by the implementation of the factory method seen in the <<Creator>>.

The Interaction Manager is both a <<Creator>> and a <<Product>>.

### 4.5.1 Interaction Manager

The Interaction Manager is the key component for handling interaction within Tandem. As seen in Figure 8, it is also an aggregate class. It is further decomposed into the following components:
50

- Graphics Intersection Manager
- Traveler
- Tracker Manager
- Input Manager

Together these classes allow the Interaction Manager to satisfy the interaction requirements of a CVR framework. Core interaction consists of updating trackers, updating intersections and reading the state of a user input device (wand). This is handled by the TcTrackerManager, TcGfxIntersectionManager and TcInputManager classes respectively. The TcTraveler class handles the control of the viewpoint. It is updated after the core interactions.

Developers are able to implement their own interaction behavior by inheriting from TcInteractionManager. Derived Interaction Manager classes also have access to the above list of components because they are aggregated as part of the base class Interaction Manager. Clients can extend, override or create new methods within a derived Interaction Manager. This permits core interaction updates to be handled by the base class. A template method called updateInteractions(), is used to extend the behavior of this class. The idea behind a template method is that it defines an algorithm in a base class consisting of primitive operations. A derived class implements these operations. In this case the primitive operations are doTdInteractions() and doTdTravel(). The “doTd” means “do Tandem Development”. It signifies two things. The “do” prefix signifies that is a primitive operation, part of a template method. The “Td” prefix means that default behavior is provided and the developer does not necessarily have to override this method. To further aid the developer in the implementation of interaction techniques several components intended for this purpose are included as part of the Core.

### 4.5.2 Traveler

No VE would be complete if the user was not able to move around the space in some fashion or another. Furthermore, it is often necessary to change the type of travel a user performs depending on a given situation. Tandem provides both of these features with the TcTraveler class. Based on the Strategy pattern, this class encapsulates the travel algorithm in another class called TcTravelScheme. Clients implement the travel method by deriving a customized travel scheme from the TcTravelScheme class. At run-time the TcTraveler class is called upon by the InteractionManager to update the viewpoint. This allows for an interchangeable component-based approach to travel techniques. By default, a simple
walk/fly travel scheme that provides collision detection is built-in to the framework. Developers are encouraged to implement as many travel schemes as is appropriate for their application. They can then easily be swapped at run-time. Furthermore a library of such travel schemes can be collected over time and reused in whatever applications require them.

4.5.3 Action Events

The Action Event component is based on the Command design pattern. It is a fairly simple mechanism that directly satisfies the event abstraction of the previous chapter. TcActionEvent classes may be associated with graphics components (described later). The purpose of an ActionEvent is to encapsulate a particular behavior mechanism without its clients having to know any details of the mechanism. This is achieved by deriving from the abstract base TcActionEvent and implementing the doEvent() method. Developers are decide when a particular ActionEvent is instantiated and when its doEvent() method is called. In this fashion, a client only knows that calling the doEvent() method triggers the behavior of an ActionEvent. The doEvent() method is overridden by classes derived from TcActionEvent, thus implementing the desired client behavior. ActionEvents are useful when a developer wants to associate specific behavior with a very small group of graphic components. They are not as successful in implementing behavior that is to be applied to a larger set of graphic components, because of the overhead of associating an ActionEvent will each component in the set. In this scenario, it is perhaps better to add the desired functionality to the InteractionManager through an aggregated component or function member. A convenient feature of TcActionEvents, is that they may easily be easily transferred, copied or deleted from graphic components at run-time, enabling the implementation of dynamic interaction.

4.5.4 Actions

Complex interactions that involve a stimulus and one or more responses are best implemented with the TcAction class. The first step in creating a TcAction class is to derive a class from TcAction. Of course this class may include whatever methods and attributes are required of the particular action. TcAction is a direct implementation of the Multicast/Typed Message pattern appearing in (16). This implementation provides several unique features. To illustrate these features, consider an action called DevelAction that inherits from TcAction. Any other class can now be designated as a Handler for DevelAction simply by inheriting from DevelAction::Handler (an automatically generated nested template class) and overriding a handleAction(const DevelAction &action) method. The reason why it is called Multicast/Typed Message is
twofold. As multicast, one simply creates an instance of DevelAction, and calls the broadcast() method. This in turn notifies all DevelAction Handler classes of the occurrence of this action, via the handleAction() method. The source of the typed message label is that the message type is sent as a unique operation signature, hence handlers do not require any form of switch statement to decipher various action types. This enables handling many actions with ease. Thus TcAction allows both a 1-to-N or N-to-1 mapping of actions to handler. To add further flexibility, Handler classes can turn on and off their interest in an action type. This mechanism is depicted in the sequence diagram presented in Figure 9. A sequence diagram depicts the dynamic behavior that exists between objects in a trace of method calls. In this diagram each class instance is depicted by a rectangle on the top of the diagram, attached to a vertical line. The vertical bars on the line denote the control scope. Time elapses from the top to the bottom of the diagram. Each horizontal line between class instances is associated with a method call.
4.5.5 User Components

Throughout the requirements chapter a reoccurring theme has been flexibility and extensibility. That is, in some way, many of the components should be easily added to, changed and manipulated by developers. In accordance with this goal, the TcUserComponent class allows any client class to be associated with graphic components simply by inheriting from TcUserComponent. This allows graphic components to have attributes and behavior easily extended to them during development, and then queried, manipulated and changed at run-time. Similar to that of TcActionEvents, TcUserComponents can also be deleted and swapped between graphic components. They are especially useful when one wants to store client data about a particular graphics component. Scenarios where this might be useful would be in cases that require

---

Figure 9 Action Sequence diagram

This diagram shows the general method invocation sequence during the use of a Tandem Core Action component. This diagram shows a client class instantiating an action, setting its state and invoking its behavior. Following this sequence of method calls, a handler is notified of the occurrence of this action through the broadcast() method. This results in the notification of each designated handler for that action through the handleAction() method. This last method invocation transmits action type information, through a unique function signature consisting of the action type as its argument.
querying an object. For instance, this could be in response to an intersection, a network event, or some user action.

4.5.6 Audio Player

Audio plays an important role in many multi-media applications. For this reason, it is essential that a framework for CVR provide a means to incorporate audio into the VE. This class provides a very simple interface, similar to that of a Compact Disc Player. Developers can add, play, forward, rewind, pause, stop and delete sounds. It is made accessible to the InteractionManager, so that audio can be combined with various interaction techniques.

4.5.7 Execution Control

The Core Execution Control class maintains the main thread of control for a Tandem application. It is not intended for client use, but is an integral component of any Tandem application. During each iteration it is responsible for ensuring that the graphics are updated, followed by interactions, travel, and finally that incoming networking events are handled. As such the Execution Control class has access to the Graphics Manager, Interaction Manager and Network Manager. The control loop is depicted in Figure 10.
Figure 10 Execution Control Loop

This diagram shows the decomposition of the execution control loop into its discrete steps. Each of these method calls is further decomposed into a sequence of steps carried out by a manager class. The drawScene() call abstracts the details of the underlying graphics library, while the updateScene() takes care of updating the graphics properties related to graphics objects (such as position for instance). The updateInteractions(), updateTravel() and handleNetworkEvents() calls are template methods that outline an algorithm made up of primitive operations. Clients can customize these primitive operations through inheritance allowing them to determine how the application is executed and what occurs during that execution.

4.5.8 Core Loader

The TcLoader class is a very simple base class that provides an interface for initializing the graphics scene both from compiled code, and from a text-file that is read at application start-up. Clients are not required to provide this class, and its use is left up to the requirements of a given application.

4.5.9 Avatar Manager

This class is the base class for Avatar Manager classes. Since every application handles avatars in their own particular way, currently this class only provides a means for clients to pass an Avatar Manager to the TcApplication class during initialization. In the future, it is intended that as the networking and collaborative aspects of CVR stabilize, this class can provide the features of a generalized Avatar Manager.
4.6 Graphics

The Graphics module is the largest and most complex of Tandem’s modules. This is due to the many requirements related to graphics that Tandem must satisfy. The most basic of these is rendering graphics to a display device. This not a simple task and requires a careful integration of a VR System Library (CAVElib) and a graphics library (Performer) into an architecture that promotes the development of CVR. Once the need for graphics rendering has been satisfied, the ability to load models from various commercial modeling applications must be addressed. For the purposes of interaction it is necessary to facilitate the selection of graphics objects within the VE. This is achieved through an intersection manager. In addition several components must allow for the client definition of state and behavior that can be associated with various graphical objects. The key components of this module are:

Graphics Manager: This is the façade that provides an interface for accessing the major features of Tandem’s graphics module. It manages the rest of the components described here.

Graphics Display Library: This is a base class for the underlying graphics system. It handles the initialization and update of Tandem’s underlying graphics subsystem. In this instance it adapts Performer and the CAVE library to Tandem’s architecture.

Background Loader: This provides asynchronous loading capabilities. It provides features such as an observer-notification mechanism that updates interested clients when a graphics model has completed background loading.

Core Loader: This component is part of the core, in order to facilitate client customization through a factory method. As mentioned in that section, it provides a protocol for initializing the graphics component hierarchy.

Graphics Component Hierarchy: This a tree structure that uses recursive composition to permit the uniform management of graphics components.

Graphics Library Component adapters: The library adapters enable the functionality of Tandem’s Graphics Component Hierarchy by adapting the components of the underlying graphics library. An overview of Performer adapters is presented in this chapter.
The Graphics Module is initialized by the Graphics Manager when the Application Class is first instantiated and configured. This module is made accessible to the Execution Control and Interaction Manager classes so that they may interact with various graphics components. Specifically the Execution Control class must be able to update the graphics module through the Graphics Manager class. In addition core and client Interactions rely upon access to the Graphics Manager to carry out their behavior. Some of the components seen above operate transparently to developers. For instance the Graphics Display Library Class is an abstract base class for a derived Graphics Library Implementation (in this case Performer). Clients do not normally operate on this class directly, rather it is used internally to initialize the graphics library, abstracting this level of detail away from developers. Of course, customization is available to those who need to change the way initialization occurs. Similarly the Background loader is not used directly by client code. It is used internally be the Graphics Component Hierarchy to provide asynchronous loading of models. The Graphics Components Hierarchy is the container for visual components in Tandem. This structure provides a uniform method for access to graphics components as well as for associating dynamic state and behavior. Though it is often used as a container for graphics components it can also be used to store components that may not have visual representations. Objects of the Graphics Component Hierarchy are seen throughout the module. In order for the Graphics Component Hierarchy to complete its role it uses a separate hierarchy of Graphics Library Component Adapters to adapt the interfaces of a particular graphics library to Tandem. The relationships between these components are seen in Figure 11.
Figure 11 Graphics Module Overview

This diagram shows the main components of the Graphics Module. As is shown here, the Graphics Manager provides the central access point for the Graphics Module as it is an aggregation of the other components. The Core Loader component is shared with the Core module.

Throughout this module the prefix “TcGfx” designates a class as being part of the Core Graphics Module. In addition, some classes are prefixed with “TdGfx”. This designates those classes as being Tandem Development Graphics classes. They are derived from TcGfx classes and are provided for use within Tandem applications. The relationships between the major components of the graphics modules are seen in Figure 12. It is to be noted that the TcGfxComponent class in an abstract base of Tandem’s graphics hierarchy. This permits graphics components to be treated uniformly and as such is seen as an aggregated member throughout this diagram. Tandem’s current implementation provides an adaptation of the Performer CAVE library. As such, the TcGfxPfLibAdapter implements this adaptation. This is done by inheriting from the TcGfxDisplayLibrary abstract base class and implementing its protocol. The derived class stores the low-level Performer structures that are encapsulated in TcGfxComponents and passed to the rest of the Graphics module in this form.
4.6.1 The Graphics Manager

The TcGfxManager (Graphics Manager) class is the controlling factor of the Graphics module. It is called upon from the Core to initialize this module. It is also responsible for maintaining the bulk of components that support the functionality of Tandem’s graphics features. Many of these components can be used independently of Tandem’s architecture, which makes them ideal candidates for reuse in other
applications. As seen in the Figure 12 the Graphics Manager is composed of several classes. These are: TcGfxDisplayLibrary, TcGfxPfLibAdapter (Performer Library Adapter), TcLoader, and TcGfxPfBackgroundLoader (Performer Background Loader). These enable the graphics manager to fulfill its various responsibilities. These include rendering the scene, updating the state of graphics components, updating the viewpoint, and providing access to Tandem’s Graphics Component Hierarchy.

### 4.6.2 Graphics Display Library

The TcGfxDisplayLibrary class is an essential component that adapts a given graphics display library for use within Tandem’s architecture. Currently Tandem uses Performer and the CAVE library as its underlying graphics layer. This requires an adaptation of their respective APIs. As such the TcGfxPfLibAdapter class inherits from TcGfxDisplayLibrary as seen in Figure 12. The Graphics Manager is able to initialize the Performer CAVE library by simply creating an instance of TcGfxPfLibAdapter. This abstracts the library details from developers, allowing them to exert effort on their application and not on the underlying VR subsystems. The TcGfxPfLibAdapter is called upon each iteration to draw one Performer graphics frame. This also enables the rendering of OpenGL graphics primitives. This feature is provided through a hierarchy of TcGfxGLGroups and TcGfxGLObjects described later on.

The structure of the Graphics module enables Tandem’s architecture to integrate the graphics capabilities of libraries that were not considered at that the time Tandem was first implemented. If Tandem were to be ported to other graphics libraries, the Graphics Manager would follow the Factory Method pattern mentioned in the Core section. This would require clients to implement a createGfxLibrary() method. This method would return a new class derived from TcGfxDisplayLibrary. This class would need to encapsulate the necessary behavior for initializing and maintaining that particular display library.

### 4.6.3 The Graphics Component Hierarchy

The Graphics module arranges its graphics components into a hierarchical structure based on the Composite pattern. This enables developers to treat graphics components in a uniform fashion. The root of this class hierarchy is the TcGfxComponent class. This is the base class for each member of Tandem’s graphics component hierarchy. It provides a uniform interface for the rest of the class hierarchy. This specifies a protocol for loading, unloading, hiding and unhiding graphics, as well as adding and removing children from its composite classes. The TcGfxComponent class can store any of the following data:
• geometry File Name
• position, transformation matrix
• two types of event lists associated with each graphics component
• a TcGfxDspLibComponent
• a set of TcUserComponents

The hierarchy is divided into two main branches. The first leads to leaf nodes. These components do not have children. The second branch refers to composite components. These use recursive composition to hold a set of child TcGfxComponents. Any child may either be a leaf or a composite component. This allows the interface of TcGfxComponent to operate uniformly on either a single leaf object or an entire subtree. In addition the TcGfxComponent class maintains the necessary associations for extendible Graphics State and Behavior. This is achieved by a set of TcActionEvent instances (described in the Core module) and TcUserComponent instances (also described in the Core module). These components are seen below in Figure 13.
Figure 13 Tandem Graphics Component Hierarchy

This diagram shows the structure behind Tandem’s Graphics Component Class hierarchy. Instances of classes within this hierarchy can be treated uniformly as the root type (TcGfxComponent). This structure provides such features as recursive loading, hiding and deleting of nodes. Client code simply makes the appropriate call on a TcGfxComponent object. This call will be carried out accordingly regardless of whether the actual object is a leaf or group node (possibly consisting of a set of other group nodes). This structure also shows how customized state and behavior is associated with Graphics components through the TcUserComponent (described in the Core section) and TcActionEvent (also described as part of the Core). Finally, a separate class hierarchy rooted at the TcGfxDspLibComponent abstract base class provides a means for adapting Graphics Library components to Tandem’s architecture besides what is already provided for Performer (see Figure 14).
4.6.4 Display Library Component Adapters

The graphics hierarchy requires the services of the underlying graphics library to support its features. This is achieved by several components. First a leaf and composite component is created for the desired graphics library. In the case of the Performer, this leads to the TdGfxPfObject and TdGfxPfGroup classes, which inherit from the TcGfxObject and TcGfxGroup classes respectively. Performer components are adapted to Tandem’s architecture in an entirely separate hierarchy. This is done for two reasons. First, if the adaptation code were left in the component hierarchy developers would need to change the existing code base of TdGfxPfObject and TdGfxPfGroup whenever new features were added. Second, it would result in a large amount of code being lumped into the graphics component hierarchy, detracting from its cohesion and making it unwieldy. Hence, TcGfxComponent classes contain an aggregated TcGfxDspLibComponent instance (see Figure 13). Adapter classes inherit from TcGfxDspLibComponent and implement an interface that adapts Tandem’s protocols to a given library component. This allows for the integration of a graphics library component into the structure of TcGfxComponent. For instance, the TdGfxPfDCSAdapter class adapts Performer Dynamic Coordinate System nodes. In this fashion, Tandem utility vectors can now be used to instantiate and manipulate TdGfxPfDCSAdapter instances. Of course the underlying Performer structure can always be extracted if a developer simply wants to operate directly on the structure. Currently a subset of Performer classes are adapted and provided for development purposes. An overview of these Performer adapters is seen in Figure 14. This protocol can be used to enable the integration of any number of graphics libraries into Tandem by deriving classes from TcGfxDspLibComponent.

A similar treatment is given to OpenGL classes. TcGfxGLObject and TdGfxGLGroup inherit from TcGfxObject and TcGfxGroup achieving OpenGL integration. Because Performer directly supports OpenGL, the adaptation is much more straightforward than Performer. All that is required to implement an OpenGL class is to inherit from TcGfxGLObject. The draw() and update() methods provide a familiar protocol for OpenGL developers. OpenGL objects are added into a tree of OpenGL TcGfxComponents. Tandem will automatically traverse this structure and update and draw each component. The Core Loader provides an initGLComponents() method that allows developers to specify which OpenGL classes to instantiate. By placing any shared memory initialization into the constructors of OpenGL classes, Tandem
ensures that this memory is initialized at the appropriate time. Of course OpenGL components inherit all of the features of TcGfxComponent, enabling event, behavior and attribute extensions (see Figure 13)

Figure 14 Performer Component Adapter Hierarchy

This diagram shows the adapter classes that are provided for Performer. As is seen here each adapter must inherit at some point from the TcGfxDspLibComponent abstract base class. This class is used to associate the adapter classes with TcGfxComponent instances. In this diagram TcGfxPfComponentAdapter creates a base for all Performer adapters. Classes beneath this map to corresponding Performer classes.
4.6.5 Background Loading

Since the loading of models may occur at any point during the execution of a CVR application, part of the reason why Performer is an ideal choice for the underlying graphics display library is because of the functionality it offers in this area. However Performer does not offer a complete solution. What it does offer is an optional DBASE process that may be forked during Performer initialization and a set of procedures that may be invoked from a user-specified callback. These procedures allow an application to append and remove Performer graphics components from its scene graph at run-time. The issues that are not covered by Performer are those of a typical producer-consumer concurrency problem. The scenario is as follows. First the MAIN application process produces a request to load a model that is consumed by the DBASE process via the user specified callback. This involves communication through a shared memory region, and hence appropriate locks must be in place to avoid race conditions. Once the DBASE process has completed its load request it produces a completed request that is consumed by the MAIN application process. This signifies that the DBASE process is ready for another request, and that the application can update its structures in response to the completion of this request. To further complicate this scenario, the MAIN process can never block on the shared memory mutual exclusion lock since the completion of a load request in the DBASE process is dependent upon the continuous execution of MAIN. This is due to Performer draw routines that render the scene-graph and synchronize the appending and deletion of scene-graph nodes performed in the DBASE process. If MAIN were to block, a deadlock situation occurs.

The principal components of the background loader are:

- TcGfxPfBackgroundLoader
- TcGfxLoadRequestPtr (Load Request Pointer),
- TcGfxBgLoaderObserver (Background Loader Observer)
- TcGfxPfSharedLoaderData

The Background Loader class is responsible for coordinating load requests sent to the DBASE process, and completed requests that are returned. The Graphics Manager triggers this sequence of events by calling the handleRequests() method from the Background Loader once per iteration of the Core Execution Control loop. The Load Request Pointer implements a smart pointer, by aggregating a nested TcGfxLoadRequest
class from The Background Loader. The use of the smart pointer allows developers to instantiate a load request on the stack without being responsible for memory management. The reason for the class nesting is that it allows the TcGfxLoadRequest class to access private members of the Background Loader. A load request is initiated by setting members of a Load Request Pointer, and then calling the registerRequest() method with a pointer to a Background Loader Observer instance as argument. The Observer inherits from the TcGfxBgLoaderObserver and implements the gfxLoadUpdate() request. In this fashion the Observer is notified when the load request has been completed. Finally the communication between the MAIN and DBASE processes is done through the TcGfxPfSharedLoaderData structure which is allocated in shared memory region. These components are depicted in Figure 15.

Generally the developer does not use the Background Loader directly, as it is employed by the Graphics Module to achieve asynchronous loading and unloading of Graphics Components. However, it has been used outside of Tandem’s architecture.
This diagram shows the components of the background loader. An instance of TcGfxLoadRequest is held in a TcGfxLoadRequestPtr. This smart pointer allows Tandem to use the stack without having to deal with dynamic memory management of TcGfxLoadRequest instances. The Background Loader class communicates with the Performer DBASE process through its aggregated TcGfxSharedLoaderData structure which is allocated in shared memory. TcGfxComponents are notified of a completed load request because they inherit from TcGfxBgLoaderObserver class and send pointers of themselves as argument to the registerRequest() method of the TcGfxLoadRequest class.

4.6.6 Graphics Visitors
Graphics visitors are based on the visitor pattern and are designed to extend the behavior of the Tandem Graphics Component hierarchy. They can also be seen as controllers because they are a mechanism used to carry out an action on a graphics component. An example is the TdGfxPfGrabController. This controller is used to transform (rotate and translate) a moveable graphics component. Clients may also develop their own controllers by inheriting from the graphics visitor base class. The way a visitor mechanism works is that a client class first instantiates or gains access to the desired controller and the graphics component that the action is to be performed on. Next the client sends the graphics component the visitor instance via the
accept(TcGfxVisitor *v) method. Finally the graphics component instance is wired to call the right visitor method from the visitor class. That is the component will call the v->visit<Type>() method, where type corresponds to the type of graphics component. There is a visit method for each type in the graphics component hierarchy. The visitor component is able to encapsulate the behavior and algorithms necessary to operate on the underlying structures of graphics components, separating these details from other client code. This mechanism is illustrated in the interaction diagram presented in Figure 16. In this diagram the final method getDspLibComponent() demonstrates the use of the controller class to operate on the underlying display library component adapters that hold the Performer graphics structures.

Figure 16 Grab Controller Interaction Diagram
This diagram shows the dynamic behavior of a TcGfxVisitor. The TdGfxPfGrabController is a descendent of this class. It is used to translate and rotate objects in a VE. When a particular wand button is pressed while the user points at a moveable graphics object the Interaction Manager sends a TdGfxPfGrabController instance to the intersected graphics object with the accept() method. The graphics object responds with a visitGfxPfGroup() call. Finally the grab controller operates on the low-level structures.
4.6.7 Graphics Iterator

The graphics iterator provides a mechanism for simplifying client traversals through the graphics component hierarchy. As seen in Figure 17 below, this component inherits from the Core Iterator class. The Core Iterator class defines a protocol for all of Tandem’s iterators. The graphics iterator class implements a breadth first traversal of the hierarchy and maintains a placeholder at the current position. This permits operations to be performed on sub-trees, or the entire hierarchy. Clients instantiate an iterator and initialize it with the desired root node. The hierarchy can then be stepped through. The currentItem() method returns the instance of the graphics component that the iterator is currently at. Clients are responsible for checking that the iterator has not reached the end of the hierarchy via the isDone() method. Because the graphics iterator is a template class, the structure of the graphics component hierarchy can be changed independently of the iterators.

![Diagram of TcIterator and TcGfxIterator classes]

**Figure 17 Graphics Iterator Class Diagram**

This diagram shows the interface for Tandem’s template iterators. The ‘Item’ box at the upper right of TcIterator denotes the template argument (class type). An iterator is first instantiated and initialized with a root node. It can then be advanced with the next() method. The item the iterator is currently pointing to is extracted with the currentItem() method. The first() method rewinds the iterator to the beginning. The isDone() method determines if the iterator has reached the end of the structure that is being iterated.
4.6.8 Intersection Testing

Intersection testing is facilitated by two main components:

Intersection Manager

Intersection Test

Clients instantiate and initialize an intersection test component. They can then pass this object to the Intersection Manager, which will then take care of updating it each iteration for a possible intersection. This permits clients to delegate the responsibility of updating the intersection test to the Intersection manager and can then simply query the manager for the results of a given intersection test whenever necessary. Alternatively clients can update and query the intersection test manually.

4.7 Networking

This module is responsible for making Tandem applications collaborative and distributed. It is composed of three main components:

Network Manager

Network Event Manager

Network Connection Manager

The principle class is the Network Manager, which is an aggregation of a Network Event Manager and a Network Connection Manager class. The Network Event Manager is the processor of network events passed between a client class and the Network Connection Manager. It packages client send requests into Network Events that are to be sent over the network. It also unpacks incoming network events and distributes them to the appropriate modules. The Connection Manager is the shipper/receiver. It knows nothing about the semantics of a Network event, or its data. It knows simply how to send and receive events over a connection. This design is similar to the Broker pattern, particularly with respect to the client and client proxy as well as the server and server proxy which map to the Network Event Manager and Network Connection Manager. These two classes can be used in both client and server applications. As such, Tandem’s networking module can be used independently of the others, providing another example of component reuse. In order to integrate networking features into an application client classes must inherit
and customize each of these three components. The relationships between these components are depicted in Figure 18.

**Figure 18 Network Manager Aggregation**

This diagram shows decomposition of the Network Manager into its two components, a Connection Manager and an Event Manager. Similar to the Application class of the Core module, the Network Manager uses factory methods to customize the creation of the Connection and Event managers. The Execution Control class updates the Network module via the handleNetworkEvents() method that is called once per iteration.

### 4.7.1 The Network Manager

The Network manager provides an interface to the network module. Similar to the Core Application class, the Network Manager uses Factory methods to enable client customization of networking components. Since there is currently no standard for the networking of SVEs, there are no default implementations for these classes and clients must provide their own. As such, the client class derived from TcNetEventManager would contain the necessary methods for creating appropriate network events for a particular client application. The Network Manager is accessible to descendents of the Application class.
Since the Network Manager contains the other networking components, outgoing events are often sent by the client class to the Network Manager and then to the desired network component.

### 4.7.2 The Network Event Manager

The Network Event manager is responsible for managing Network Events. Incoming events are directed from the network connection to this class. Information about which network events are to be routed to which application class is stored here. The Network Event Manager typically employs the use of *receiving docks* (described below) to handle the receipt of incoming asynchronous network events. The Tandem core execution loop calls upon the Network Event Manager class once per iteration to handle network events. This results in the processing of events waiting in the receiving docks. This mechanism replaces locking strategies that would otherwise be required if asynchronous events were to be routed directly to their destination classes as they came in from the network.

### 4.7.3 The Connection Manager

The Connection Manager handles the sending and receiving of network events across the network connections. CAVERNsoft currently satisfies the needs of the underlying connection, synchronization and persistence mechanisms. The CAVERNsoft API is adapted to Tandem’s architecture by extending and customizing the base networking classes discussed above. As such there is a CAVERN Connection Manager, as well as CAVERN Network Event class. These components implement the required mechanisms to adapt the CAVERNsoft API to Tandem’s architecture. This permits client classes to utilize a high level protocol that does not require knowledge of underlying network details. Client application code calls a method from the customized network manager that results in the creation of a NetworkEvent. This NetworkEvent is sent as an argument to the tdSendNetworkEvent() method (depicted in Figures 19; 20). The CAVERN Connection manager implements the required functionality to send this network event across a network connection. Should the network layer be changed, the application code that interacts with the network manager would not be impacted. Changes will be restricted only to this module, especially the Network Events, which are responsible for implementing the underlying packing and unpacking code that interacts directly with the network connection API. This provides a serialization protocol through the sendToCavernKey() and getFromCavernKey() methods (see the TcNetCavEvent class below).
Figure 19 Network Connection Manager

This diagram shows the structure of the Network Connection Manager and associated Network Events. Both of these classes have been adapted to CAVERNsoft as denoted by the ‘Cav’ in their class names. The Cavern Connection Manager contains a table of network connections (links) across which network events are sent and received. The Connection Manager also contains a table of Keys that implement DSM in CAVERNsoft. In addition The Connection Manager stores CAVERNsoft’s Information Request Broker (IRB). Network Cavern Events (TcNetCavEvents) provide an abstract base class protocol for serialization via the `getFromCavernKey()` and `sendToCavernKey()` methods. Developers define their network events by inheriting from this class and implementing these methods.
Figure 20 Sending a Network Event Sequence Diagram

This diagram illustrates the runtime sequence of events that occur when a network event is instantiated as a result of some action. In this case a model is translated in a SVE and a network event is sent to notify other clients to update their local copies. In reality the network event would be a derived class instance of TcNetCavEvent, however for clarity the base class name is used here. Essentially, one can see that the call from the Interaction Manager results in a call to the Network Manager and the creation of a network event. This event is then sent to the Connection Manager to be sent over the network.

4.7.4 Receiving Docks

Receiving docks provide a queuing mechanism for incoming network events. As described in the section on the Connection Manager, incoming events are placed into a receiving dock. This class provides locking mechanisms to ensure that race conditions do not occur. The protocol is straightforward. Each time an event is to be received, the lockDock() method is called, followed by a receiveItem() call. Finally an unlockDock() method releases the lock. When events are being processed the dock is locked through the lockDock() method. It is determined if there are waiting events to be processed by a call to the empty() method. The processItem() method will then return an event at the front of the queue. Finally an unlockDock() call will release the lock. CAVERNsoft has a DSM mechanism called a key. Each key is encapsulated in a Tandem class called a TdNetCavKey. Whenever a network event is sent on a given key a callback mechanism is triggered in each client that is connected to this key. The callback mechanism accesses a pointer to the Receiving Dock instance that is associated with a given key. Refer to Figure 21 for an illustration of these classes.
Figure 21 Receiving Dock Class Diagram

This diagram shows the TdNetReceiving dock and an associated TdNetCavKey class. Events triggered by incoming network events result in the event being added to a designated receiving dock.
5 DESIGN IMPLEMENTATION AND RATIONALE

This section covers a few of the design details that are not mentioned in the previous chapter. In order to understand this material it is first necessary to have read the architecture overview that is presented in chapter 4. With this overview in mind, the design rationale is revisited. This discussion will provide insight into why certain components have evolved into their current form.

5.1 Tandem Core Display Library Components

In order for Tandem to maintain its flexibility as a framework, an important design consideration is the ability to adapt to various graphics libraries. In this way the user is not tied to any one graphics library implementation. In addition this makes it easier to port Tandem to other platforms where libraries such as SGI’s Performer may not be available. The Adapter pattern provides the desired structure. An abstract base class called a TcGfxDspLibComponent is defined with a minimal interface and plays the role of Target in the Adapter pattern. (see Figures 13 & 14)

Clients of Tandem inherit from this class in order to implement concrete display library component types. For instance, the TdGfxPfNodeAdapter, derived from TcGfxPfComponentAdapter provides a ready to use development class for Performer nodes. The TdGfxPfNodeAdapter class stores a reference to a pfNode and provides get and set methods. If desired, clients can further refine this adapter class by defining derived classes. In order to access the derived class methods a downcast seems necessary. This is traditionally a poor design, leading to complications in the client code. However there are two alternatives that avoid this unwanted behavior.

One alternative to simply downcasting is to define a Visitor class with visit methods for each of the component types. This pattern provides a means for recovering type information. This pattern breaks down however, when the hierarchy is not stable, since it must resort to case statements. The other problem with this approach is that Visitor classes must be defined for a known role or operation (For instance, a graph display visitor could iterate through a hierarchy in order to build a visual graph with icons that represent each type within the hierarchy). In general it is not possible to anticipate what type of Visitors may be needed since more often than not, they are specific to a given application. Therefore Visitor classes are best left for clients to define on their own.
Another approach that is inspired by the *Memento* pattern (also called Type Laundering (16)) is to allow visitor classes, called graphics controllers, to perform a downcast on a *TcGfxDspLibComponent* instance. In accordance with the Memento pattern the TcGfxComponent class serves as the caretaker responsible for the safekeeping of classes derived from TcGfxDspLibComponent. The role of *memento* is fulfilled by TdGfxPfDCSAAdapter. An example of the *originator* is the TdGfxPfGrabController class. A TdGfxPfGroup instance stores a *TdGfxPfDCSAAdapter* in its base class *TcGfxDspLibComponent* member. The *TcGfxComponent* class is responsible for maintaining an association to display library components.

The memento-based approach provides several benefits. Type safety is aided when downcasting a TcGfxDspLibComponent to a TdGfxPfDCSAAdapter by predetermined application protocols. For example, in the Mars Explorer application described in the next chapter, a Grab Controller visitor class generally operates on TdGfxPfGroup instances. The controller class can provide warning and error behavior if an application developer sends it to the wrong TcGfxComponent type. More importantly however, naïve clients can treat TcGfxComponents uniformly without having to know anything about their underlying structure. Clients avoid the complicated tag-and-case-statement style of programming by delegating originator classes to access TcGfxDspLibComponents. Clients that require changes to the underlying graphics library components can carry out this behavior by originator classes such as the grab controller that know how to handle the type information of a particular TcGfxDspLibComponent. Finally, this design makes it easier to adapt a given application to another graphics library without having to change existing code in the *TcGfxComponent* hierarchy. It also enables an application to code cross platform capabilities by configuring it with a given TcGfxDspLibComponent instance at run time.

### 5.2 Tandem Core User Components

In almost every application it is necessary to associate domain-specific state and behavior information with elements of TcGfxComponent hierarchy. Clients may also want to further refine or add elements to this hierarchy. One alternative is to simply extend the current hierarchy through inheritance. This breaks the current hierarchy’s cohesion by tacking on additional information that does not necessarily apply to its role of logically grouping together graphics components in a hierarchical structure. It is also undesirable to require clients to modify Tandem source code in order to extend its functionality. Hence, another
abstraction that solves this issue is the TcUserComponent class. TcUserComponents are stored in a table indexed by a user specified component ID of type TcUtString. Tandem developers can inherit from this class and configure a TcGfxComponent with a client defined class. In this way, clients are able to associate an arbitrary number of client defined types with TcGfxComponents.

5.3 Tandem Development Model Controllers

The interaction requirements of chapter three define a taxonomy of actions that a user can perform on graphical models in CVR. An example of such actions could be:

- Grab Model
- Transform Model
- Share Model
- Throw Model
- Add/Delete Model
- Highlight Model
- Select model

One way of providing these capabilities is to add these methods as an interface to the current TcGfxComponent hierarchy.

So we would have:

```cpp
class TcGfxComponent {
public:
  virtual Grab Model
  virtual Transform Model
  virtual Share Model
  virtual Intersect Model
  virtual Add/Delete Model
  virtual Highlight Model
}
```

This requires modification to the current interface and implementing code in the existing hierarchy. As already mentioned, this is not an acceptable means of extending the behavior of Tandem components. The current hierarchy is cohesive, it caters to a hierarchical structure that maintains a core set of operations that operate on graphical elements such as hiding and un hiding or loading and unloading geometry data. It doesn’t make sense to break this cohesion by polluting it with added behavior that pertains to application specific requirements. Clients of Tandem can not be expected to modify the Tandem Component hierarchy.
5.3.1.1 The Visitor Pattern

This pattern creates a separate hierarchy that operates on the existing Tandem hierarchy. This allows one to add behavior without modification to existing code within the Tandem hierarchy. The mechanism behind this pattern is that TcGfxComponent must define an accept(visitor *v) interface. Then each class within the hierarchy must implement this method to call the appropriate visitor for its class type. For instance:

```cpp
TcGfxPfGfxObject::accept(visitor *v)
{
    v->visitPfGfxObject(this);
}
```

In this manner, the visitor pattern extracts class type information so that different actions can be performed on groups and objects without an unwieldy switch/case statement. The idea is that the Visitor base class defines an interface with a visit method for each class type within the TcGfxComponent hierarchy. Clients inherit from this abstract class and create a specific type of visitor that defines behavior for each type of class within the Tandem hierarchy. Also a mechanism adopted from the Memento pattern, permits the visitor to maintain knowledge about specific adapter types and can thus perform a downcast giving the visitor a wider interface with which to operate on a particular graphics component. Tandem visitors are known as controllers and gives rise to the following definition:

**Tandem Development Graphics Controller**: An entity that directs how a particular action performed on a graphics component is carried out.

5.3.2 Tandem Development Actions and Action Handlers

Before a call is made to instantiate a particular visitor and invoke this visitor upon a TcGfxComponent the user must initiate this behavior from the GUI. This potentially creates a situation where it is necessary for various graphics interface components to invoke actions. Since we don’t want the GUI to know any details about the underlying mechanism the best way to achieve this is through a Multicast pattern. In this pattern one or more Receivers (observers or listeners) are notified of the occurrence of one or more actions (also known as messages or subjects). The receiver implements a separate handler for each message type that it registers an interest in. This is different from the observer pattern in which typically one subject is
observed by many observers. Another situation where this mechanism is useful is in the case of incoming network events.

The idea is that every time a user triggers an event from the GUI an *action* is instantiated and all registered handlers of this action are notified. Registered handlers respond to the action in a manner that is appropriate. For instance, an action handler could instantiate a corresponding controller to carry out the action. This pattern also allows clients to create their own action types without being restricted to passing a single abstract base class type as an argument to handler classes. Instead the actual action type is sent, which in terms of complex behavior is desirable over calling a virtual update() method with no arguments (as in the observer pattern).

---

1. An entity that has the power to direct or regulate. A device for regulating a mechanism.
6 MARS EXPLORER: A TANDEM APPLICATION

6.1 Application Overview

Mars Explorer is an interactive and collaborative learning environment that aims to teach children about collective sample spaces. It is a Tandem application based on Mars Builder designed for the CAVE or IDesk Environments. In this application the user will be assigned the task of collecting a representative sample of geological data from the virtual environment. Students are motivated to explore the Martian surface while collecting data about the rocks they encounter. The goal is to teach children about the process of scientific inquiry by providing a rich virtual environment to explore. The user will have to determine categories of sample data in order to determine what types of data may require further exploration and recording. After the first phase of data collection in the Mars virtual environment, students are asked to perform an analysis in a traditional classroom setting. This analysis prompts the students to observe clustering relationships that exist within the sample space. Much the data to be collected is derived from information collected and published by NASA's explorations of the planet.

Mars Builder provides a starting point for developing Tandem applications. It is an existing application shell, implemented with the Tandem framework. That is, it includes pre-configured application, interaction and networking classes that implement features such as script-based loading of models, travelling through the space and collaborative interaction.

6.2 Application Architecture

Mars Explorer is an application that is written by adapting the code base of Mars Builder. Mars Builder is a fully collaborative environment that provides integrated Avatars in a distributed application that is divided into a server and a client. The centralized server keeps track of the users in the environment and provides a known point of entry for new participants. It also manages several synchronization issues necessary for client initialization. In addition the server architecture permits users to share single models or entire virtual environments consisting of a set of models with users connected to the server. Users connect to a known world name, and users may only interact with those in the same world name. This permits one
physical machine to manage multiple worlds. Mars Explorer has not made any modification to the Mars Builder Server code base.

The Mars Builder client provides the bulk of the application processing and takes advantage of the majority of Tandem’s features. Mars Explorer extends this code base by adding customized interaction.

6.2.1 Mars Explorer Client

As described in the previous chapter building an application with Tandem involves deriving customized classes and overriding various Template methods. Mars Builder provides customization for many of Tandem’s classes and Mars Explorer adds to this code base where appropriate. The first step in building a Tandem application is to construct the Application class. The Tandem Core Application class is the only class that must be extended. However, to get the full set of interaction and collaboration features further customization is required. In order to provide a fully functional application, the Mars Explorer client implements the following customized classes:

CAVERN_builder_Application: This is the main component in the application. It is responsible for providing implementations of the Factory methods of its base TcApplication class.

CAVERN_builder_Loader: This provides an initialization of the Tandem Component Hierarchy and a text file parser that allows Mars Explorer to initialize the scene with various terrain models without the need for recompiling. This text format also provides several fields for setting attributes about the models to be loaded such as their scale, orientation and position as well as whether they are initially shared at application start-up. It has been customized for the Mars Explorer application to handle the placement of rocks throughout the terrain. This class inherits from TcLoader.

CAVERN_builder_NetManager: This class manages the networking features within Mars Builder. It is used as a front-end by the rest of the application classes that require access to the network. This class inherits from TcNetManager and provides implementations of its factory methods. As such there are two other classes provided for the networking:

CAVERN_builder_NetEventManager: This class manages the receiving docks. All incoming events are queued in designated receiving docks and handled once per iteration of the execution control loop.
CAVERN_builder_NetConnectionManager: This class manages the connections to the server. Outgoing network events are sent to the connection manager to be shipped across a specified connection. This class also takes care of establishing the connections upon initialization of the application on start-up.

CAVERN_builder_AvatarManager: This class manages the arrival and departure of new participants in the environment by updating the graphics components associated with their avatars. In addition, this class manages their local state (such as position and orientation) while they are in the SVE.

CAVERN_builder_InteractionManager: This class controls interaction within the Mars Explorer application. It provides customizations for Mars Explorer such as a scanner module and bar chart display. It also manages interaction such as the transformation, loading, sharing and deleting of models in the VE.

In order to customize the communication between classes, access to various components is shared by the CAVERN_builder_Application class. Because it inherits from TcApplication it has access to all of the base class aggregated members. In Figure 7, it was shown how the config() method of TcApplication triggers all of the factory methods in a specific order. The implementations of these factory methods that appear in the CAVERN_builder_Application class customize the component’s access to each other. As such access to the Network Manager is shared with the Avatar Manager. This allows the Avatar manager to initialize various avatar related connections with the server. Because the Interaction Manager requires access to the graphics, network, audio and avatar components it is given access to the Network Manager, Graphics Manager, Audio Player and Avatar Manager. In addition the CAVERN_builder_Loader is given access to the Network Manager so that it can share new graphic objects with others in the SVE. An illustration of this shared access is seen in the following object diagram, in Figure 22.
Figure 22 Cavern Builder Client Object Diagram

This diagram illustrates how the components are able to communicate with each other. The aggregation relationships depict access of a component to another via a shared reference. The side with the diamond is the class that aggregates the object on the opposite end of the relationship. For example, the Loader shares a reference to the Network manager with the Avatar and Interaction managers.

Each of the Cavern Builder classes depicted in Figure 22 inherit from Tandem’s classes. This allows them to extend the behavior of the Tandem core classes as well as participate in the framework that Tandem provides. This enables customized primitive operations that are defined as part of Tandem’s internal algorithms to be customized. The relationships between each of these customized classes to their Tandem counterparts are depicted in Figure 23.
Figure 23 Cavern Builder Client Application Diagram

This diagram shows how each of the Cavern Builder classes is related to Tandem classes through inheritance as well as each other. The CAVERN_builder_Application class is an aggregation of the other classes in this diagram. Because the Application class keeps the derived types as data members, no downcasting is required to access them, at the cost of storing an extra pointer.

6.2.2 Mars Builder Server

The server for this application demonstrates the use of Tandem’s network module independent of the Core and Graphics modules. This is an example of how Tandem’s components may be used outside of its architecture. The server consists of three classes common to Tandem networking. In addition, information about a participant is stored in a client profile. This result in the following classes:

CAVERN_builder_ServerNetManager

CAVERN_builder_ServerNetEventManager
CAVERN_builder_ServerNetConnectionManager

These classes serve the same role as their client counterparts, however their implementation provides server functionality. To illustrate this, consider an incoming connection request. This will result in an initialization procedure on the server that differs from a client response. The client response would result in the addition of a new avatar to be managed, while the server would add a new profile to be managed and update the new client with the current state of the VE. These classes are depicted in Figure 24.

Further component reuse exists by the sharing of network events. The same classes that contain the implementations of the sendToCavernKey() and getFromCavernKey() methods for a particular event (e.g. Avatar Arrived Network Event) are used without modification both on the client and the server. Furthermore the mechanism provided by the Action and Handler components provides different handler configurations on both the client and the server.

Figure 24 Cavern Builder Server Diagram

This diagram shows the server network components. This shows how the Tandem networking framework is suitable both to client and server applications, and how it may be used independently of the graphics and core components. In this diagram the Server Net Manager is seen to be an aggregation of a ServerNetEventManager and a ServerNetConnectionManager. It is also shown that the ServerNetEventManager stores a list of client profiles for each client that is present in the SVE.
6.3 Mars Explorer Interaction

Mars Builder provides manipulation of graphics components by allowing the transforming, sharing, and deletion of models. Transformation involves translating and rotating an object through the use of selection and manipulation. This involves the use of the grab controller described in the previous chapter as well as the intersection capabilities of Tandem. Sharing a model means that it is sent to all participants in the VE. In this state any manipulation to this model is propagated to everyone in the VE, allowing for collaboration. Deletion of a model removes it from the VE, and results in network activity if the model is shared. Mars Explorer allows participants to “scan” rock samples in the VE and outputs their analysis in a bar chart graphical display component. The bar display demonstrates how a TdGfxPfObject may be customized through inheritance. The update to this display also demonstrates a new interaction technique that is specific to this application. This display is shown in Figure 25. In this diagram the wand pointer is shown scanning a rock. This is achieved simply by pointing at the desired target. The analysis is displayed in the lower right corner of the screen.
The Interaction Manager makes use of the Action/Handler mechanism to respond to incoming network events. Instead of the Network module having to keep track of the mappings between each incoming network event and their destination classes, handlers for each network event are defined. As such the Interaction Manager is a handler for the following network events:

Network Model Updates (transformations by remote users)

Removal of a shared Model
New Models added to the VE

Each of the network events listed above inherits from the TcAction template class. The Interaction Manager then inherits from the handler class of the network events that it is interested in. This automatically sets up notification so that whenever a network event is broadcast the appropriate handleAction() method of the Interaction Manager is called. The Interaction Manager also maintains pointers to components that are involved in interaction such as the bar graph display and sound events. The various classes associated with the Mars Interaction Manager are shown in Figure 26. In this diagram the three handler inheritance relationships are shown for the above list of network events. Also shown in this diagram are the other components that enable the Mars Explorer client to complete processing of interaction events during the execution control loop.
Figure 26 Mars Explorer Interaction Manager

This diagram shows a detailed view of the Mars Explorer Interaction manager. As seen by the class name, Mars Explorer reuses and modifies the code of the Mars Builder Interaction manager to customize it for the Explorer application. In this diagram the Interaction Manager is seen to inherit from three Handler classes. This allows it to respond to notification from three separate network events (model transformation, model deletion, and the addition of a new model). In addition the network manager shares access to the Avatar Manager, Network Manager, as well as components customized for Mars Explorer Interaction such as the Bar Display and Sound Events. Mars Explorer uses the GrabController class to allow users to pick up and move objects within the Mars VE. Finally ModelAttributes stores information about the currently intersected model (see Figure 25).
As mentioned in an earlier chapter, developers extend functionality in Tandem by implementing primitive operations. In the case of the Interaction Manager, the execution control loop calls the primitive operation doTdInteractions() once per iteration from the updateInteractions() template method (see Figure 9). This allows developers to customize what occurs after the core interactions have been updated. In Mars Explorer this consists of the following activities:

1. Updating tracker data
2. Checking for Intersections
3. Updating wand display
4. Updating model interactions
5. Updating bar display

### 6.4 Mars Networking

Mars Builder provides the foundation for the networking in Mars Explorer. Each of the network events defined for Mars Builder makes use of the Action/Handler mechanism to route incoming network events.

A summary of these network events is as follows:

**Avatar Left Event**: Notification that a user has left the SVE

**Avatar Arrived Event**: Notification that a user has joined the SVE

**Model Update Event**: Update of a model position, orientation or scale

**Avatar Update Event**: Update of an avatar position and orientation

**Remove Shared Model Event**: Notification that a model has been deleted

**Geometry Transfer Event**: Notification of the addition of a new model in the SVE

**New Connection Event**: Confirmation of a connection request to the server, upon start-up

Each of these Network event types is stored in its own receiving dock. Once per iteration the execution control loop calls the handleNetworkEvents() template method. This results in the Network Manager forwarding this update to the NetworkEventManager, where it will check its receiving dock for newly arrived events. This results in the following calls in the NetworkEventManager:

handlePublicSessionEvents()

dhandleAvatarEvents()
handlePublicAvatarUpdates()

handleModelEvents()

handlePublicModelUpdates()

Each of these calls will check the designated receiving dock for newly arrived network events. In order to minimize the amount of references the NetworkEventManager must maintain to other components, NetworkEvents use the TcAction/Handler mechanism to maintain the mapping of which network events are to be routed to which components. As such the NetworkEventManager simply calls the broadcast() method on a received network event and this will automatically result in its delivery to the appropriate components.

6.5 Putting it all Together

Once all of the components have been written and configured, the Mars Explorer client is run from a main loop by instantiating a Mars_Builder_Application class and calling runTcApp(). The instantiation results in a call to the config() method which will trigger all of the factory methods(). Finally a call to the runTcApp() starts the Tandem execution control loop.

To start the application, one must start a server and a client. Once the client is started, providing a server is already running, the client will connect to the server establish all the connections and initializations. This includes receiving a list of users currently participating on the server, and sending those participants any models, which are designated as shared. This information is read from a script that the user configures in a text editor. The execution control loop ensures communication between each of the components occurs and is updated at each iteration, until the application is quit.
7 CONCLUSIONS AND LESSONS LEARNED

Tandem in combination with Performer, the CAVElib and CAVERNsoft provide a complete development solution for Collaborative Virtual Reality. Tandem places an emphasis on a component-based approach that has resulted in an extremely flexible architecture and a set of reusable components. Throughout this thesis it has been demonstrated how design patterns lead to the development of a component-based framework that emphasizes flexibility, extensibility and maintainability. This framework also provides features that expedite the development of interaction techniques in a SVE. In addition the developer is given the ability to catalogue the components created in an application for use in the future. This provides a much more efficient development model that increases productivity and promotes research in other areas of CVR.

Interaction within CVR is inherently complex due to the added dimension and the distributed computing requirements. As such, Tandem provides many components that enable complex interaction techniques within a SVE such as Action Events, Actions, Handlers and Network Events. In addition pre-built interaction techniques such a Grab controller provide developers with a ready to use component. This component can be extended, customized or used as a template for new interaction techniques.

Because Tandem is created for a rapidly changing domain it provides many features that enable it to quickly adapt and integrate the features of new systems. In this fashion it is not permanently fixed to its networking or graphics layers. Tandem provides a system that integrates graphics, interaction and networking. This facilitates the development of CVR applications.

The construction of Mars Builder and Mars Explorer has provided a preliminary evaluation of this framework. This evaluation has shown that Tandem is indeed a viable tool for the development of CVR applications. Throughout the development of both Mars Builder and Mars Explorer experience has culminated in several observations and lessons learnt. These are outlined below:

Action/Handlers are as of yet best used by network events

It’s not certain that the Network Event Manager is best used to construct Network Events since the Network Manager could just as easily perform the same task. Further evaluation of this is necessary.
No Interaction technique has as of yet required the use of an ActionEvent an Action/Handler and a Visitor/Controller for the same interaction sequence. Further evaluation is required to determine if these components are sufficient for more complex interaction.

Intersection requires further evolution due to the complexities of Local and World Coordinates

Model and Avatar state updates must be placed on timers to avoid having faster machines flood slower ones.

Preliminary testing has indicated that the Receiving Docks are best emptied at each frame instead of processing one event per iteration. The latter method results in flooding, in which the queue continues to grow.

Never overestimate the complexity of system initialization and component instantiation. Due to dependencies this has required a fair amount of design effort.

With a fairly complete application design, one person can construct a rough application prototype similar to Cavern Builder in about a one-week full-time development effort.

Despite the measures taken to maintain flexibility, Tandem has yet to be adapted to other graphics libraries and networking systems. Further evaluation is required to determine if it meets the promises outlined in this work.

Finally, Tandem is the start of a new direction for CVR. It is among the first systems in this domain that implement a truly OO approach to application development in VR. Further evaluation of this framework is necessary to truly appreciate the gains of a component-based approach. Certainly this will see the evolution of its current systems. Ideally this will result in the creation of a visual language for CVR. This is typically the final stage of evolution for a framework. This would permit the creation of CVR applications either from within a CVR environment itself, or in combination with a desktop integrated development environment.

A true sign of progress will be the development of many CVR applications that highlight the benefits of this medium by improving our daily life. This will not occur until existing frameworks and toolkits easily facilitate the development of such applications. Tandem is a first step in this direction.
8 GLOSSARY

**Action**: A component that performs some operation or activity.

**ActionEvent**: A component that is used to trigger a particular action, as in the event that results in the invocation of an action.

**Aggregate Class**: This is the owning class, which is made of aggregated classes, forming a part-whole relationship. The parts are typically maintained by pointers.

**Aggregated Class**: This is the component part, which makes up an aggregate class.

**Composite Class**: This is the owning class, which is made of composited classes, forming a part-whole relationship. The parts are typically maintained by value.

**Composited Class**: This is the component part that makes up a composited class.

**Consistency**: Ensuring that the value of shared state on one node is consistent with its value on a remote node.

**Distributed Shared Memory**: A memory model that provides a method for a shared state across machines connected together over a network.

**Data Synchronization**: A protocol that specifies how local and remote updates to shared state are to occur, as in what order and what nodes are to be used as reference points (if any).

**Derived [Class]**: Any descendent class in an inheritance hierarchy. Inheritance is a typical feature of OO languages.

**Factory method**: A factory method is a design pattern in which a method is responsible for creating a given component type, and returning a pointer to the newly created object.

**Handler**: A component that responds to the occurrence of some activity. Usually in reference to an action.

**Persistence**: A mechanism for preserving the state of a system so that it can be restored at some later point.

**Primitive Operation**: A method called from a *template method* that implements a part of the algorithm defined by template method.

**Template Method**: This is a design pattern that involves defining a high level algorithm in the base class (the template method) which calls upon *primitive operations* to carry out key parts of the algorithm.
APPENDICES
Appendix A

SUMMARY OF UNIFIED MODELING LANGUAGE NOTATION

The notation used throughout this work is summarized in the following diagram:

Figure 27 Summary of Unified Modeling Language Notation
Appendix B

TANDEM APPLICATION PROGRAMMER’S INTERFACE

Contents

1 TcAction — Abstract base template class for all Action classes ............... 104
1.1 Handler — Nested class Handler ........................................... 105

2 TcActionEvent — Abstract base class for TcActionEvents .................. 108

3 TcActionEventObserver — ActionEvent observer class ....................... 110

4 TcApplication — Tandem core application base class ....................... 111

5 TcAudioPlayer — AudioPlayer Class ........................................... 118

6 TcAvatarManager — Base Avatar Manager class ............................... 123

7 TcExControl — Core Execution Control Class ................................. 124

8 TcInputManager — Core InputManager class .................................. 125

9 TcInteractionManager — Core InteractionManager Class ................... 127

10 TcIterator — Abstract base template Iterator Class ........................ 132
Appendix B (Continued)

11 TcLoader — Class TcLoader ............................................. 134
12 TrackerData — Tracker Data Class ..................................... 136
13 TcTrackerManager — Core TrackerManager Class ................. 137
14 TcTravelScheme — TcTravelScheme class .............................. 139
15 TcTraveler — Core Traveler Class .................................... 141
16 TcUserComponent — Abstract Base User Component Class ........ 144
17 TcGfxPbBackgroundLoader — Background loader class ............. 145
  17.1 TcGfxLoadRequest — Nested Class TcGfxPbBackground-
       Loader::TcGfxLoadRequest ........................................... 146
  17.1.1 TcGfxBgLoaderObserver — Nested Class TcGfxLoad-
       Request::TcGfxBgLoaderObserver ................................. 148
18 TcGfxCompList — Class: TcGfxCompList .................................. 151
19 TcGfxIterator — Template iterator class for Gfx classes ......... 154
20 TcGfxComponent — Abstract Base Class TcGfxComponent .......... 156
  20.1 LoadType — LoadType enumeration ................................ 159
  20.2 LoadState — LoadState enumeration .............................. 160
21 TcGfxDisplayLibrary — Class TcGfxDisplayLibrary .................. 162
Appendix B (Continued)

22 TcGfxDspLibComponent — Abstract base class for Display Library Component Adapters

23 TcGfxGObject — Class is used to hold GL Gfx objects that implement the draw and update methods

24 TcGfxGroup — Abstract base class for GfxGroups

25 TcGfxIntersectionManager — Intersection Manager Class

26 TcGfxIntersectionTest — Abstract base class for Intersection Tests

26.1 IsectState — enum IsectState

26.2 CoordType — enum CoordType

27 TcGfxLoadRequestPtr — Smart pointer class TcGfxLoadRequestPtr This class is instantiated on the stack in order to send a load request

28 TcGfxManager — Class: TcGfxManager Role: This class manages Tandem’s graphics subsystem

29 TcGfxObject — Abstract Class TcGfxObject

30 TcGfxPfComponentAdapter — Base class for all performer component adapters

31 TcGfxPfLibAdapter — Core Graphics Performer Library Adapter

32 TcGfxVisitor — Abstract base class for visitors and controllers
Appendix B (Continued)

33 TdGfxGLGroup — This class is used to setup a tree of GL components

34 TdGfxPfDCSAdapter — pfDCS adapter class

35 TdGfxPfGrabController — Performer grab controller class

36 TdGfxPfGroup — Class TdGfxPfGroup

37 TdGfxPfGroupAdapter — pfGroup adapter class

38 TdGfxPfIntersectionTest — Performer Cavelib Intersection Test

38.1 Sensor — Sensor enum

39 TdGfxPfNodeAdapter — Class TdGfxPfNodeAdapter

40 TdGfxPfObject — Class TdGfxPfObject

41 TdGfxPfSCSAdapter — pfSCS adapter class

42 TdGfxPfTravAttributes — Performer traversal attributes class

43 TcNetCavConnectionManager — Role: This class plays the role of Tc-NetEvent shipper and receiver

44 TcNetCavEvent — CAVERNsoft Network Event

45 TcNetConnectionManager — Abstract Base Class for Network Connection Manager

46 TcNetEvent — Abstract Base Class for Network Events.
Appendix B (Continued)

47 TcNetEventManager — Abstract base class for Event Manager ........ 257
48 TcNetManager — Abstract base class for Network Manager .......... 258
49 TdNetCavConnection — Class used to encapsulate a CAVERNsoft Connection ................................................................. 261
49.1 connectionType — Connection type enumeration ...................... 262
50 TdNetCavKey — Class used to encapsulate a CAVERNsoft key. ....... 264
51 TdNetReceivingDock — Template class for queuing asynchronous network events. ............................................................... 268
52 TcUtDebug — Debug utility class used throughout Tandem ............ 270
53 TcUtMutex — Base class for Tandem mutex .................................. 271
53.1 lockResult_t — enum lock Results ............................................. 272
54 TcUtSemMutex — Mutex class built on posix semaphore for single process, threaded apps. ............................................................... 273
55 TcUtSemMutex_mp — Mutex class built on posix semaphore for multi process apps. ................................................................. 275
56 TcUtObserver — Abstract Base class for observers ....................... 277
57 TcUtString — Tandem String Class ............................................... 278
58 operator<< — Output ................................................................. 279
Appendix B (Continued)

59  operator>> — Input ......................................................... 280
60  getline — Read line ....................................................... 281
61  operator== — Compare == .................................................... 282
62  operator!= — Compare != ..................................................... 283
63  operator< — Compare < ..................................................... 284
64  operator<= — Compare <= ................................................... 285
65  operator> — Compare > ..................................................... 286
66  operator> = — Compare >= .................................................. 287
67  TcUtStringHashTable — Template String Hash Table ................. 288
67.1  iterator — Nested Iterator class ..................................... 289
68  TcUtVector3f — Tandem 3 float utility vector ........................ 293

Class Graph ................................................................. 295
template <class T> class TcAction

Abstract base template class for all Action classes

B.0.0.0.1 Public Members

1.1 class Handler Nested class Handler ................. 105
virtual ~TcAction () Destructor
1.2 static void addHandler (Handler* aHandler)
   This method will register an action handler for notification ............... 106
1.3 static void removeHandler (Handler* aHandler)
   This method will unregister an action handler ......................... 107
void broadcast () This method will notify each registered action handler.

static void ts_addHandler (Handler* aHandler)
   Thread safe version of addHandler.
static void ts_removeHandler (Handler* aHandler)
   Thread safe version of removeHandler.
void ts_broadcast () Thread safe version of broadcast.

B.0.0.0.2 Protected Members

1.4 TcAction () Constructor ......................... 107
static void ts_notify (TcAction<T>* t)
   Thread safe version of notify.
Appendix B (Continued)

Abstract base template class for all Action classes. Clients must inherit from this class in order to define an Action.

### 1.1

**class** `Handler`

**Nested class Handler**

### B.0.0.3 Public Members

1.1.1 **Handler** (bool autoRegister = true)

*Default Constructor* .......................... 105

1.1.2 virtual **Handler** ()

*Destructor*

1.1.2 **virtual int** handleAction (const T& t)

*Pure virtual member function* ............ 106

Nested class Handler. This class is a nested member class of each action class derived from TcAction. Action handlers are defined by inheriting from this nested class.

**1.1.1**

**Handler** (bool autoRegister = true)

*Default Constructor*

Default Constructor. User can turn off automatic action handling. Default value sets up any class inheriting from Handler as an implicit action handler. Clients must implement int
handleAction<const T t>; To turn off automatic action handling you would have code similar to the following in the constructor for an action handler: MyActionHandler::MyActionHandler() : MyAction::Handler(false)

```
1.1.2
virtual int handleAction (const T& t)
```

Pure virtual member function. This method must be implemented by each Action Handler. It is called whenever a broadcast occurs for a given action.

```
1.2
static void addHandler (Handler* aHandler)
```

This method will register an action handler for notification. To turn on action handling you would call the static method MyAction::addHandler(myActionHandler) where myActionHandler is an instance of MyActionHandler.
This method will unregister an action handler. To turn off action handling you would call the static method `MyAction::removeHandler(myActionHandler)` where `myActionHandler` is an instance of `MyActionHandler`.

1.4

TcAction()

Constructor. This makes TcAction an abstract base class.


Appendix B (Continued)

2

```cpp
class TcActionEvent

Abstract base class for TcActionEvents
```

### B.0.0.0.4 Public Members

- **TcActionEvent()**  
  *Default Constructor.*

- **TcActionEvent(const TcUtString &name)**  
  *Constructor*  
  

- **virtual ~TcActionEvent()**  
  *Destructor.*

- **void setName(const TcUtString &name)**  
  *Sets the name of the ActionEvent.*

- **const TcUtString &getName()**  
  *Returns the name of this ActionEvent.*

- **virtual void doEvent()**  
  *This method must be overridden by derived classes.*

Abstract base class for TcActionEvents. Clients must inherit from this class in order to define an ActionEvent. Objects of this Class can be associated with TcGfxComponents. See TcGfxComponent::addEvent

2.1

```cpp
TcActionEvent(const TcUtString &name)
```

*Constructor*
Appendix B (Continued)

**Parameters:**

- name — An optional string name can be used to identify an ActionEvent. This must be set either here or by the setName method before adding to a TcGfxComponent.

2.2

```cpp
void setName (const TcUtString& name)
```

*Sets the name of the ActionEvent*

Sets the name of the ActionEvent.

**Parameters:**

- name — This is the name used by the TcGfxComponent::getEvent method.

2.3

```cpp
virtual void doEvent ()
```

*This method must be overridden by derived classes*

This method must be overridden by derived classes. This implements the behavior of the event. It is called by TcGfxComponent::triggerEvent.
class TcActionEventObserver

**ActionEvent observer class**

B.0.0.0.5  **Public Members**

```cpp
TcActionEventObserver ()
    Constructor.

virtual ~TcActionEventObserver ()
    Destructor.

dvirtual void handleActionEvent (TcActionEvent *event)
    All ActionEventObserver classes must
    implement this method.
```

ActionEvent observer class. Clients may wish to specify observers of a given ActionEvent. They may do so by inheriting from this class and implementing the handleActionEvent method. The management of observer to ActionEvent mappings is left to the client. Clients must store a list of observers for a given ActionEvent. Upon triggering that ActionEvent via the doEvent call, they must iterate through the observer list for that event, and call handleActionEvent. The event instance is passed as argument to this method. This may be handled by the ActionEvent itself in the future.
class TcApplication

Tandem core application base class

B.0.0.0.6 Public Members

TcApplication (int argc, char *argv[])  Constructor.

virtual ~TcApplication ()  Destructor.

4.1 void tdAppUpdate ()  Not yet implemented

4.2 void runTcApp ()  This is called after the call to config() to

4.1 void runTcApp ()  start the application execution loop.

int config ()  This must be called in the constructor

of derived classes to trigger the factory

methods.

B.0.0.0.7 Protected Members

4.2 virtual TcNetManager*
Appendix B (Continued)

```
createNetworkManager (int argc, char* argv[],
    long& clientLocalPort,
    long& clientRemotePort,
    bool& clientRemoteConnect,
    TcUtfString& clientUserName,
    TcUtfString& clientWorldServer,
    TcUtfString& clientWorldName)

Factory method
```

```
4.3 virtual TcInteractionManager*
createInteractionManager (TcGfxManager &gfxManager)

Factory Method
```

```
4.4 virtual TcGfxManager*
createGfxManager (int argc, char *argv[],
    TcLoader *clientLoader)

Factory Method
```

```
4.5 virtual TcAvatarManager*
createAvatarManager ()

Factory Method
```

```
4.6 virtual TcLoader*
createLoader ()

Factory Method
```

```
4.7 virtual TcAudioPlayer*
createAudioPlayer ()

Factory Method
```

```
const TcNetManager*
getNetManager ()

This provides access to Tandem’s networking manager class.
```

```
TcGfxManager*
sceneManager

The scene manager provides access to Tandem’s graphics subsystem.
```

```
TcAudioPlayer*
```
Appendix B (Continued)

audioPlayer

The audioPlayer provides access to Tandem’s audio client.

Tandem core application base class. Every tandem application must inherit from this class. TcApplication is based on the Factory Method design pattern. Clients override the create methods to get the desired modules.

4.1

void tdAppUpdate ()

Not yet implemented

Not yet implemented. If need be this method will allow clients to add functionality outside of the interaction methods.
Factory method. This method will return a client network manager, and NULL if one is not provided.
Appendix B (Continued)

Parameters:

clientLocalPort — the port the client will receive on.

clientRemotePort — the port on the server that the client will connect to.

clientRemoteConnect — a flag indicating whether to connect to a server or not.

clientUserName — The client’s user name

clientWorldServer — The url of the host.

clientWorldName — The name of the world to connect to.

4.3

virtual TcInteractionManager* createInteractionManager (TcGfxManager &gfxManager)

Factory Method

Factory Method. This method will return a default interactionManager if not overridden by the client.

Parameters: gfxManager — This is the scene manager.
Factory Method. Generally this method is not overridden and the Tandem core graphics manager is used.

Factory Method. The application base class will hold a bare bones TcAvatarManager. Eventually Tandem will include a default avatar implementation. For now look at the MarsBuilder application for an example.
Appendix B (Continued)

Factory Method. This method will return a client loader and NULL if one is not provided.

```cpp
virtual TcAudioPlayer* createAudioPlayer ()
```

Factory Method. This method will return the default Tandem audio client class. It is available from within the InteractionManager, set by this class on startup.
class TcAudioPlayer

AudioPlayer Class

B.0.0.0.8 Public Members

5.1 TcAudioPlayer (const TcUtString &soundDir) Constructor ...................... 119

5.2 ~TcAudioPlayer () Destructor ...................... 120

5.3 void playSound (const TcUtString &name, int isLooped) Starts play of a sound clip ............ 120

5.4 void forwardSound () Stops the currently playing sound clip and plays the next clip ............ 120

5.5 void backSound () Stops the currently playing sound clip and plays the previous clip ............ 121

5.6 void addSound (const TcUtString &path,
        const TcUtString &filename) Adds a new playable clip to the list of sound files .............. 121

      void playFirstSound () Start playing the first sound in the list as determined by an ascending alphabetical sorting of the sound file names

5.7 void stopSound (const TcUtString &name) Stops the sound from playing ............ 122

      void stopCurrentSound ()
Stops the last sound clip in which play was initiated.

5.8    void    rewindSound (const TcUtString &name)
  Rewind the sound file

AudioPlayer Class. This class is currently implemented using Dave Pape’s Bergen Sound Server. A directory of .aiff files is read at class instantiation. Offers similar functionality to a CD player. Client’s can play, loop, stop/pause, rewind, add and delete sound clips. The server is assumed to be running on the machine on which this class is instantiated.

5.1

TcAudioPlayer (const TcUtString &soundDir)

Constructor

Parameters:    soundDir — This is the full path name of the directory containing the .aiff files.

5.2

~TcAudioPlayer ()

Destructor
Destructor. This will kill all currently playing sounds.

5.3

```c
void playSound (const TcUtString &name, int isLooped)
```

*Starts play of a sound clip*

Starts play of a sound clip. This works similar to the function of a CD player in that will resume play from the point the clip was paused, or start from the beginning if it was not previously playing. If a sound has already been played it must be rewound before it can be heard again.

**Parameters:**

- `name` — name of the soundfile to be played. This must have been read in at class instantiation or by an explicit call to add sound.
- `isLooped` — If the value is 0 the sound will be played once. Otherwise it will be looped.

5.4

```c
void forwardSound ()
```

*Stops the currently playing sound clip and plays the next clip*

Stops the currently playing sound clip and plays the next clip. The order is sorted alphabetically determined by ascii values.
5.5

void backSound ()

*Stops the currently playing sound clip and plays the previous clip*

Stops the currently playing sound clip and plays the previous clip. The order is sorted alphabetically determined by ascii values.

5.6

void addSound (const TcUtString &path, const TcUtString &filename)

_Adds a new playable clip to the list of sound files_

Adds a new playable clip to the list of sound files. This sound file be inserted alphabetically into the list determined by ascii value. String literals are accepted as arguments.

**Parameters:**

- path — This is the absolute path not including the file name of the file to be added.

- filename — This is the file name to be added.
void **stopSound** \((\text{const} \ TcUtString &\text{name})\)

*Stops the sound from playing*

Stops the sound from playing. If the sound is currently playing it serves as a pause. If the sound has reached the end it serves as a rewind.

**Parameters:**

- **name** — The name of the sound file that one wishes to stop.

---

void **rewindSound** \((\text{const} \ TcUtString &\text{name})\)

*Rewind the sound file*

Rewind the sound file. Exactly the same behavior as **stopSound**. Provided only as a naming convenience, to maintain coherency. Whenever a sound is played it is a good idea to call **rewindSound** to ensure that is not already at the end from a previous play.
class TcAvatarManager

Base Avatar Manager class

B.0.0.0.9 Public Members

TcAvatarManager () Constructor.
virtual ~TcAvatarManager () Destructor.

Base Avatar Manager class. This class is intended to provide Core avatar management features. Currently this is implemented in the MarsBuilderApplication. Eventually it will be moved up into this class.
class TcExControl

Core Execution Control Class

B.0.0.0.10 Public Members

TcExControl (int argc, char *argv[]),
TcNetManager* clientNetManager,
TcGfxManager& clientSceneManager,
TcInteractionManager&
clientInteractionManager)

Constructor

virtual ~TcExControl () Destructor

void startApp () Starts the execution control loop.

Core Execution Control Class. This class encapsulates the core execution control loop. It is not intended to be used by clients.
class TcInputManager

Core InputManager class

B.0.0.0.11 Public Members

enum buttonChange Possible values returned by getButtonChange

enum buttonState Possible values returned by getButtonState

void tdUpdateInputDevices ()
   Called each frame by the Core Interaction manager to update device state.

8.1 buttonChange
   getButtonChange (const int &buttonNum) const
   Returns whether there has been a change
   in button state since the last frame .... 126

8.2 buttonState getButtonState (const int &buttonNum) const
   Returns the current button state ....... 126

Core InputManager class. This provides an interface to the obtaining the state of wand buttons. Clients can access this class through the Interaction manager.
Returns whether there has been a change in button state since the last frame

Parameters:

buttonNum — Determines what button number to query.

buttonState getButtonState (const int &buttonNum) const

Returns the current button state

Parameters:

buttonNum — Determines what button number to query.
class TcInteractionManager

Core InteractionManager Class

B.0.0.0.12 Public Members

TcInteractionManager (TcGfxManager &gfxM)
Constructor.

virtual ~TcInteractionManager ()
Destructor.

9.1 int config ()
This must be called in the Constructor of
a derived class before any of the aggregate
members can be accessed

bool isConfigured ()
This determines if config has been called.

void updateInteractions ()
This is called from the Core Execution
Control loop.

9.2 virtual TcTrackerManager*
createTrackerManager ()
Factory Method

9.3 virtual TcGfxIntersectionManager*
createIntersectionManager ()
Clients must override this if they wish to
provide a custom IntersectionManager

9.4 virtual TcInputManager*
createInputManager ()
Factory Method

const TcInputManager&
Appendix B (Continued)

getInputManager () Returns a reference to the inputManager.

virtual void doTdInteractions () This must be overridden by derived
classes to implement customized interactions.

virtual void doTdTravel () This must be overridden by derived class
to implement customized travel.

const TcGfxManager&
getGfxManager () Returns the graphicsManager.

9.5 void setTreeDebug (const bool &b)
Sets a debugging mode attached to buttons 2 3 Button 2 will toggleLoad the
current TcGfxComponent .......... 131

B.0.0.0.13 Protected Members

TcTrackerManager*
trackerManager TrackerManager
TcGfxIntersectionManager*
intersectionManager IntersectionManager
TcInputManager*
inputManager InputManager
TcGfxManager &
sceneManager GraphicsManager

Core InteractionManager Class. Role: This class performs the core interaction updates. It is
an aggregate consisting of a TrackerManager, a GraphicsIntersectionManager, an InputManager,
a Traveler, a GraphicsManager and a NetworkManager.
Appendix B (Continued)

Responsibilities: It is responsible for instantiating the Input Manager, the TrackerManager and the GraphicsIntersectionManager. These are the components updated during core interaction updates.

9.1

```c
int config()
```

*This must be called in the Constructor of a derived class before any of the aggregate members can be accessed*

This must be called in the Constructor of a derived class before any of the aggregate members can be accessed. This includes the InputManager, TrackerManager and GraphicsIntersectionManager.

9.2

```c
virtual TcTrackerManager* createTrackerManager()
```

*Factory Method*

Factory Method. Clients must override this if they wish to provide a custom TrackerManager. They will have to implement the interface of the default tracker manager. Currently one is provided that is implemented on top of the CAVElib.
Appendix B (Continued)

9.3

virtual TcGfxIntersectionManager* createIntersectionManager()

Clients must override this if they wish to provide a custom IntersectionManager.

Clients must override this if they wish to provide a custom IntersectionManager. They will have to implement the interface of the default intersectionManager. Currently one is provided.

9.4

virtual TcInputManager* createInputManager()

Factory Method

Factory Method. Clients must override this if they wish to provide a custom InputManager. They will have to implement the interface of the default inputManager. Currently one is provided that is implemented on top of the CAVElib.

9.5

void setTreeDebug (const bool &b)

Sets a debugging mode attached to buttons 2 3 Button 2 will toggleLoad the current TcGfxComponent
Appendix B (Continued)

Sets a debugging mode attached to buttons 2 3 Button 2 will toggleLoad the current TeGfx-Component. Button will iterate through the tree. Results are printed to standard out.
Appendix B (Continued)

```cpp
template <class Item> class TcIterator
```

Abstract base template Iterator Class

B.0.0.0.14 Inheritance

B.0.0.0.15 Public Members

```cpp
virtual void first () // Set the iterator to the first element.
virtual void next () // Advance the iterator to the next element.
virtual bool isDone () const // Returns true if the end of the set has been reached.
virtual Item currentItem () const // Returns the item the iterator is currently at.
```
Abstract base template Iterator Class. This class is used to provide an interface for Tandem Iterators. In particular it is used by the TcGfxIterator.
class TcLoader

Class TcLoader

B.0.0.0.16 Public Members

TcLoader () Constructor.
virtual ~TcLoader () Destructor.
virtual TcGfxComponent* initTanComponents ()
    This method must be overridden to initialize the scene from compiled code.
virtual void initGLComponents ()
    This must be overridden to initialize openGL components.
11.1 virtual void initSharedMem ()
    This method is optionally overloaded to initialize shared memory

void setSceneComponent (TcGfxComponent *sceneNode)
    Used by the Core classes.
void setSoundPlayer (TcAudioPlayer *sndPlayer)
    Used by the Core classes.
void setNavComponent (TcGfxComponent *navNode)
    Used by the Core classes.
void setGLComponent (TcGfxComponent *glComp)
    Used by the Core classes.

B.0.0.0.17 Protected Members
Appendix B (Continued)

TcGfxComponent* navNode

The navigation node.

TcGfxComponent* sceneNode

The root scene node.

TcGfxComponent* glRoot

The GL node.

TcAudioPlayer* soundPlayer

The Core Audio Player.

ifstream gfxFileIn

Used to read a scene file.

Class TcLoader. This class is core the TcGfxComponent loader. Clients must inherit from this class and override the interface.

11.1

virtual void initSharedMem ()

This method is optionally overloaded to initialize shared memory.

This method is optionally overloaded to initialize shared memory. This call is made just before Performer forks off its app, cul, draw processes.
class TrackerData

Tracker Data Class

B.0.0.0.18 Public Members

TrackerData () Constructor.
~TrackerData () Destructor.

const TcUtVector3f&
getPosition () const Returns the position vector.

const TcUtVector3f&
getOrientation () const Returns the orientation vector.

const TrackerData&
TrackerData::operator Assignment operator.

Tracker Data Class. This class encapsulates a rotation and position vector for a given tracker sensor.
Appendix B (Continued)

class TcTrackerManager

Core TrackerManager Class

B.0.0.0.19 Public Members

13.1 TcTrackerManager (const int numT =

DEFAULT_NUM_TRACKERS)

Constructor .............................. 138

virtual ~TcTrackerManager ()

Destructor.

TrackerData* getTrackerData (int sensorNum)

Returns a pointer to tracker data for a
given sensor number.

const TcUtilVector3f&

getSensorPosition (int sensorNum)

Returns a vector containing the position
of a given sensor number in world coor-
dinate system.

const TcUtilVector3f&

getSensorOrientation (int sensorNum)

Returns a vector containing 3 rotation
angles (x, y, z) for a given sensor number
in world coordinate system.

13.2 void tdUpdateTrackers () Called by the Core Interaction manager
during core interaction updates ....... 138
Core TrackerManager Class. This class takes care of updating the set of tracker sensors for a VR system.

13.1

**TcTrackerManager** (const int numT = DEFAULT_NUM_TRACKERS)

*Constructor*

**Parameters:**

- `numT` — Determines how many sensors will be updated. Default is 2. Typically sensor 0 is the head tracker and sensor 1 is the wand tracker.

13.2

**void tdUpdateTrackers ()**

*Called by the Core Interaction manager during core interaction updates*

Called by the Core Interaction manager during core interaction updates. It is marked td to indicate that a default implementation is provided.
Appendix B (Continued)

class TcTravelScheme

TcTravelScheme class

B.0.0.0.20 Public Members

TcTravelScheme (TcGfxComponent *navComp) Constructor.

virtual ~TcTravelScheme () Destructor.

virtual void travel () Perform a travel update.

float compute_nav_height () Computes the navigation height.

int checkForWall (float xvec, float yvec, float zvec) Ensures the user isn’t walking through a wall.

int checkCaveWalk () Moves the world back if a user physically tried to walk through a wall.

void setMaxHeight (const float &height) Sets the maximum height the user can jump.

B.0.0.0.21 Protected Members

TcTravelScheme () Default constructor called by derived classes.

TcTravelScheme class. This class encapsulates the algorithm used to define travel in a VE.
Appendix B (Continued)

Clients can inherit from this class and override the travel method to define their own algorithms. By default a simple walk scheme with collision detection is provided as the default travel scheme. This scheme is automatically instantiated by default.
Core Traveler Class

This class is the user’s traveler object. It is responsible for updating the users viewpoint by the handleTravel method. The TravelScheme encapsulates the algorithm used. It may be changed at run time.
15.1

TcTraveler (TcTravelScheme *scheme)

Constructor

Parameters: scheme — A travelScheme object must passed as argument.

15.2

TcTraveler (TcGfxComponent *nav)

Constructor

Parameters: nav — This is a TcGfxComponent containing the navigation node.

15.3

void setTravelScheme (TcTravelScheme *scheme)

This method allows clients to change the travel scheme at run-time
Appendix B (Continued)

This method allows clients to change the travel scheme at run-time. The client is responsible for deleting the previous scheme.
Abstract Base User Component Class

class TcUserComponent

B.0.0.0.24 Public Members

TcUserComponent ()
    Constructor.

~TcUserComponent ()
    Destructor.

const TcUtString&
getComponentId () const
    Returns the component id given to the instance.

B.0.0.0.25 Protected Members

void setComponentId (const TcUtString &value)
    Used by derived classes to set the component id.

Abstract Base User Component Class. This class is used to associate user classes with TcGfxComponents. Any class that is derived from this class can be added as a user class to a graphics component. The class can then be retrieved by its string component id.
class TcGfxPfBackgroundLoader

Background loader class

B.0.0.0.26 Inheritance

B.0.0.0.27 Public Members

17.1 class TcGfxLoadRequest Nested Class TcGfxPfBackgroundLoader

TcGfxPfBackgroundLoader ()
Constructor

~TcGfxPfBackgroundLoader ()
Destructor

static void addRequest (TcGfxLoadRequest *req)  
Add a request to the request queue.

void initSharedMem ()  
Initialize shared memory.

17.2 void handleRequest ()  
Handle load requests in the application

main process  

void setPfScene (pfScene *scene)
Appendix B (Continued)

Used in initialization to place the scene node pointer into shared memory.

```c
void setPfNav (pfDCS *nav)

Used to place the navigation node pointer into shared memory.
```

```c
static void backgroundLoadCB (void *data)

This is the callback triggered by the DBASE process.
```

```c
static pfNode* manualPfLookup (pfGroup *, const TcUtString &)

This performs a manual search for a node in performer when pfLookup fails.
```

B.0.0.0.28 **Protected Members**

```c
struct TcGfxPfSharedLoaderData

Nested Structure TcGfxPfBackgroundLoader::TcGfxPfSharedLoaderData
```

Background loader class. This class takes care of asynchronous loading of performer models. It must be properly initialized with the performer library and clients must inherit from the nested observer class to complete load requests. The classes are nested to avoid having to declare everything in the public interface or declare each class as a friend.

```c
17.1

class TcGfxLoadRequest

Nested Class TcGfxPfBackgroundLoader::TcGfxLoadRequest
Appendix B (Continued)

B.0.0.0.29 Public Members

17.1.1 class TcGfxBgLoaderObserver

Nested Class TcGfxLoadRequest::TcGfxBgLoaderObserver.

TcGfxLoadRequest ()
Constructor

~TcGfxLoadRequest ()
Destructor

17.1.2 void setCompName (const TcUtString& name)
Set the name to be set in the pfNode

void setFileName (const TcUtString& name)
Set the file name that the loader will search for in the path set by the PFPATH environment variable

17.1.3 void setParentName (const TcUtString& name)
Set the name of the parent node

const TcUtString&

getCompName () const
Returns the component name.

const TcUtString&

getFileName () const
Returns the file name.

const TcUtString&

getParentName () const
Returns the parent name.

void registerRequest (TcGfxBgLoaderObserver *obs)
Registers the observer and sends the request.

void sendAnonymousRequest ()
Appendix B (Continued)

Sends an anonymous load request with no observer.

bool isAccepted ()

Returns true if the load request has been accepted.

int completeRequest ()

Informs observers that the request has been completed.

Nested Class TcGfxPfBackgroundLoader::TcGfxLoadRequest. This class is used internally by the TcGfxLoadRequestPtr and should not be instantiated by clients. Calls to operator-> sent to the TcGfxLoadRequestPtr class are redirected here.

17.1.1

class TcGfxBgLoaderObserver

Nested Class TcGfxLoadRequest::TcGfxBgLoaderObserver.

B.0.0.0.30 Public Members

TcGfxBgLoaderObserver (bool observeClass = false)

Constructor.

virtual ~TcGfxBgLoaderObserver ()

Destructor.

virtual int gfxLoadUpdate (const TcGfxPfBackgroundLoader::TcGfxLoadRequest& request)
Clients must inherit from this class and override this function if they wish to be informed when the loader has completed the request.

17.1.2

```cpp
void setCompName (const TcUtString& name)
```

_set the name to be set in the pfNode_

Set the name to be set in the pfNode. This will be the same name given to the TcGfxComponent.

17.1.3

```cpp
void setParentName (const TcUtString& name)
```

_set the name of the parent node_

Set the name of the parent node. This node must already exist in the performer hierarchy prior to sending an ASYNC load request.
17.2

void handleRequest ()

Handle load requests in the application main process

Handle load requests in the application main process. This method completes processing of handled request and sends a new request to the background DBASE process.
class TcGfxCompList

B.0.0.0.31 Public Members

template <class Item> friend class
   TcGfxChildIterator Friend declaration.
   TcGfxCompList () Constructor
   ~TcGfxCompList () Destructor

const TcGfxCompList&
   operator= (const TcGfxCompList &right) Assignment Operation

18.1 int addListItem (TcGfxComponent *item)
   Add a TcGfxComponent to the list .... 152

18.2 int removeListItem (TcGfxComponent *item)
   Remove a TcGfxComponent from the list

void printDebug () Print the items in the list to standard out
   by CompName.

18.3 const TcGfxComponent*
   getCompByName (const TcUtString &name)
   Return a pointer to the item with the
   string name, name ..................... 153

   bool isEmpty () Returns true if the list is empty, and
   false otherwise.

Class: TcGfxCompList.
Appendix B (Continued)

Role: This class provides an interface to a list of TcGfxComponents. Each child can be leaf nodes or other groups containing lists.

Responsibilities: It adds and removes TcGfxComponents. Returns dynamically allocated iterators to its children.

18.1

```c
int addListItem (TcGfxComponent *item)
```

*Add a TcGfxComponent to the list*

Add a TcGfxComponent to the list. The compName must be defined in the item, and it must be unique in the list. A return value of 1 indicates a successful insertion, 0 means that the name already exists.

18.2

```c
int removeListItem (TcGfxComponent *item)
```

*Remove a TcGfxComponent from the list*

Remove a TcGfxComponent from the list. A return value of 1 indicates a successful deletion, 0 means that the name already exists.
const TcGfxComponent* getCompByName (const TcUtString &name)

Return a pointer to the item with the string name, name

Return a pointer to the item with the string name, name. Returns NULL if not found.

Parameters: name — String name of a TcGfxComponent.
Appendix B (Continued)

```
template <class Item> class TcGfxIterator : public TcIterator<Item>

Template iterator class for Gfx classes
```

B.0.0.0.32 Inheritance

![Inheritance Diagram]

B.0.0.0.33 Public Members

19.1 `TcGfxIterator ()` Constructor .................................. 155
19.2 `TcGfxIterator (const Item root)` Constructor ................. 155

virtual `~TcGfxIterator ()` Destructor.

virtual void `first ()` 
`Sets the iterator to the root set by the constructor.`

virtual void `next ()` Iterates to the next item.

virtual bool `isDone () const` Returns true if iterator has reached the end of the set.

virtual `Item currentItem () const` Returns the item the iterator is currently pointing to.
Template iterator class for Gfx classes. Clients can instantiate an iterator by simply calling TcGfxComponent::iterator i(comp). This class can then be used to iterate through gfx components in a breadth first fashion.

19.1

TcGfxIterator ()

Constructor

Constructor. Returns an uninitialized iterator.

19.2

TcGfxIterator (const Item root)

Constructor

Constructor. This is the most common constructor to be used.

Parameters: root — This is the node from which iteration starts. The iterator is set to this node whenever first is called.


Appendix B (Continued)

class TcGfxComponent

Abstract Base Class TcGfxComponent

B.0.0.34 Inheritance

B.0.0.35 Public Members

20.1 enum LoadType

LoadType enumeration .................. 159

20.2 enum LoadState

LoadState enumeration .................. 160

virtual int addChild (TcGfxComponent *gfxNode)

Add a child to this component.

virtual int removeChild (TcGfxComponent *gfxNode)

Remove a child from this component.

20.3 virtual void loadGeometry (const LoadType &ldType = ASYNC)

Loads the geometry ..................... 160

virtual void unLoadGeometry () Unload the geometry.

virtual void hideGeometry () Hide the geometry.

virtual void unHideGeometry () Unhide the geometry.
virtual void toggleLoadGeometry ()
    Toggles load/unload geometry depending
    on the current state.

virtual void toggleHideGeometry ()
    Toggles hide/unhide geometry depending
    on the current state.

virtual void accept (TcGfxVisitor &)
    This is a visitor design pattern method.

virtual const TcGfxComponent*
    getCompByName (const TcUtString &)
    Searches for a given component in the
    tree rooted by this node.

virtual void triggerEvent (const TcUtString &eventName)
    Triggers the event with the given name
    associated with this component.

virtual TcActionEvent*
    getEvent (const TcUtString &eventName)
    Returns the event with the given name
    associated with this component.

virtual TcActionEvent*
    getIterativeEvent (const TcUtString &eventName)
    Returns the iterative event with the given
    name associated with this component.

virtual void handleIterativeEvents ()
    Iterates through the list of iterative events
    and triggers each one.

const TcUtString&
    getCompName ()
    Returns the name of this component.

void addEvent (TcActionEvent *)
    Associate an event with this component.

void removeEvent (TcActionEvent *)
Appendix B (Continued)

void addIterativeEvent (TcActionEvent *)
Associates an iterative event to this component.

void removeIterativeEvent (TcActionEvent *)
Removes an iterative event from this component.

void addUserComponent (const TcUtString &name,
TcUserComponent *component)
Add a user component with the given name to the UserClass table

TcUserComponent*
getUserComponent (const TcUtString &name)
Return the component with the given name and NULL if it is not found.

void removeUserComponent (const TcUtString &)
remove from table but doesn’t delete component

const vector <TcActionEvent *> *
getEventList ()
Returns a pointer to the entire list of events associated with this component.

TcUtString getGeoFileName ()
Returns the file name of the geometry associated with this component.

void setGeoLoaded (bool) Sets the geometry loaded flag.
void setGeoHidden (bool) Sets the geometry hidden flag.
void setCompName (const TcUtString &name)
Sets the component name.

bool geoLoaded ()
Returns true if the geometry is loaded.

bool geoHidden ()
Returns true if the geometry is hidden.

TcGfxComponent*
Appendix B (Continued)

\begin{verbatim}
getParent () \hspace{1cm} \textit{Returns the parent of this component.}
void setParent (TcGfxComponent *)\hspace{1cm} \textit{Sets the parent of this component.}
void setDspLibComponent (TcGfxDspLibComponent *)\hspace{1cm} \textit{Sets the display library component.}
TcGfxDspLibComponent* getDspLibComponent () const \hspace{1cm} \textit{Returns the display library component.}
\end{verbatim}

\textbf{B.0.0.0.36 Protected Members}

\begin{verbatim}
20.5 TcUtString \textit{geometryFileName Attribute}: \textit{geometryFileName Description: This attribute stores the file location of the corresponding geometry file}
\end{verbatim}

Abstract Base Class TcGfxComponent.

This is the root class for all Tandem Graphics Components.

Role: This abstract base class forms the composite pattern by letting clients treat TcGfxObjects and TcGfxGroups uniformly.

Responsibilities: It provides an interface to TcGfxObjects and TcGfxGroups

\begin{verbatim}
enum LoadType
\end{verbatim}

\textit{LoadType enumeration}
Appendix B (Continued)

Parameters:

- **ASYN**C — Asynchronous load type
- **SYNC** — Synchronous load type

### 20.2

```cpp
enum LoadState
```

*LoadState enumeration*

Parameters:

- **LOADED** — Loads the component during instantiation if the Load-Type is set to **SYNC** otherwise it is ignored.
- **UNLOADED** — The component is not loaded. Can be used in combination with **SYNC** or **ASYN**C

### 20.3

```cpp
virtual void loadGeometry (const LoadType &ldType = ASYN)
```

*Loads the geometry*
Appendix B (Continued)

Parameters:  
ldType — Determines whether the load will be asynchronous or synchronous.

```c
void addUserComponent (const TcUtString &name, TcUserComponent *component)
```

Add a user component with the given name to the UserClass table

Add a user component with the given name to the UserClass table. This is one way clients can add additional features/attributes to a component without subclassing. It is especially useful for adding application specific properties and behavior. WARNING - Because it uses SGI's STL hash map, `name` must be properly allocated and exist for the lifetime of the `userComponent`. The preferred way of doing this is to use the `getName()` method as an argument to `name`.

20.5

```c
TcUtString geometryFileName
```

Attribute: geometryFileName Description: This attribute stores the file location of the corresponding geometry file

Attribute: geometryFileName Description: This attribute stores the file location of the corresponding geometry file. For instance it could be a performer binary (.pfb) or an inventor file (.iv)
Appendix B (Continued)

```cpp
class TcGfxDisplayLibrary

Class TcGfxDisplayLibrary

B.0.0.0.37 Inheritance

```

B.0.0.0.38 Public Members

```cpp
TcGfxDisplayLibrary ()
    Constructor.

TcGfxDisplayLibrary (TcLoader* tanLoader)
    Constructor.

TcGfxDisplayLibrary (TcGfxComponent *scene,
                        TcGfxComponent *nav,
                        TcGfxComponent *Av)
    Constructor.

virtual ~TcGfxDisplayLibrary ()
    Destructor.

TcGfxComponent *
    getSceneComponent ()
    Returns Scene component.

TcGfxComponent *
```
Appendix B (Continued)

```
getNavComponent ()
Returns Navigation component.
TcGfxComponent *
getAvComponent () Returns Avatar component.
virtual void setViewPoint (const TcUtVector3f *rot,
                           const TcUtVector3f *trans)
Sets the view point.
virtual void setIsectManager (TcGfxIntersectionManager *)
Sets Intersection manager.
```

Class TcGfxDisplayLibrary. This is the base class for all core graphics library classes. The current implementation provides support for IRIS Performer through the derived TcGfxPfLibAdapter. Future support for other libraries will be implemented by deriving from this class.
class TcGfxDspLibComponent

Abstract base class for Display Library Component Adapters

B.0.0.39 Inheritance

B.0.0.40 Public Members

virtual const TcUtString&
getComponentId () const
Returns the component id associated with this component adapter.

virtual const char*
getCompId_c_str () const
Returns a C style string pointer to the component id.

virtual void hideGeometry ()
Virtual function used to hide geometry from the scene.

virtual void unhideGeometry ()
Virtual function used to unhide geometry from the scene.
Appendix B (Continued)

virtual void loadGeometry ()  
Virtual function used to load a geometry component into the scene.

virtual void unloadGeometry ()  
Virtual function used to unload a geometry component from the scene.

virtual bool isActive ()  
Returns true if the geometry component is loaded and false otherwise.

virtual ~TcGfxDspLibComponent ()  
Destructor.

void setComponentId (const TcUtString &value)  
Set the name of the component id.

B.0.0.0.41  Protected Members

TcGfxDspLibComponent ()  
Default constructor.

22.1 TcGfxDspLibComponent (const TcUtString &name)  
Constructor  

Abstract base class for Display Library Component Adapters. This class is used to adapt underlying display library components to the TcGfxComponent hierarchy. Each component adapter inherits from this base class and can then be associated with a TcGfxComponent.

22.1 TcGfxDspLibComponent (const TcUtString &name)

Constructor
Appendix B (Continued)

**Parameters:**

name — Component id to be set in the component.
class TcGfxGLObject : public TcGfxObject

Class is used to hold GL Gfx objects that implement the draw and update methods

B.0.0.0.42 Inheritance

TcGfxComponent

\( \vee \)

TcGfxObject

\( \vee \)

TcGfxGLObject

B.0.0.0.43 Public Members

TcGfxGLObject () Default constructor.
TcGfxGLObject (const TcUtString &name)

Allows the compName to be set by the constructor.

virtual ~TcGfxGLObject () Destructor.

void loadGeometry (const LoadType& lType)

Loads the openGL object into the scene

void unLoadGeometry () Unloads the openGL from the scene.
virtual void hideGeometry ()  Hide the geometry so that it is still loaded, but not drawn.

virtual void unHideGeometry () Make geometry that has been hidden visible.

virtual void draw () Implemented by descendents and called once per iteration by Tandem Execution.

virtual void update () Implemented by descendents and called once per iteration by Tandem Execution.

virtual TcUtVector3f getPosition () Returns the position of the object.

virtual int setPosition (TcUtVector3f) Sets the position of the object.

virtual TcUtVector3f getOrientation () Returns the orientation of the object.

virtual int setOrientation (TcUtVector3f) Sets the orientation of the object.

virtual void accept (TcGfxVisitor &v) Used to accept TcGfxVisitors.

### B.0.0.0.44  Protected Members

bool* draw_gl  Boolean flag that must be allocated from shared memory to be seen by the draw process in multi-processor and multi-processed systems
Appendix B (Continued)

Class is used to hold GL Gfx objects that implement the draw and update methods. Shared memory allocation must occur in constructor initialization.

23.1

void loadGeometry (const LoadType & lType)

*Loads the openGL object into the scene*

Parameters: 

1Type — Generally not used by GLObjects, but can be ASYNC or SYNC
Appendix B (Continued)

```cpp
class TcGfxGroup : public TcGfxComponent

Abstract base class for GfxGroups
```

B.0.0.0.45 **Inheritance**

```
TcGfxComponent
  \\)
  V
  \\)
  \\)
  TcGfxGroup
```

B.0.0.0.46 **Public Members**

- `TcGfxGroup () Constructor.
- `TcGfxGroup (TcUtString name) Constructor.
- `TcGfxGroup (const TcUtString &name,
  TcGfxPfComponentAdapter *dspLibComp)`
Appendix B (Continued)

24.2  
**TcGfxGroup** (const TcGfxGroup & right)  
*Constructor*  

**Copy Constructor**  

171

```
virtual TcGfxGroup()  
*Destructor*
```

```
const TcGfxGroup&  
operator=(const TcGfxGroup & right)  
Assignment Operation Not yet implemented
```

```
virtual int addChild(TcGfxComponent * gfxNode)  
Add a child to the child list.
```

```
virtual int removeChild(TcGfxComponent * gfxNode)  
Remove a child from the child list.
```

```
TcIterator <TcGfxComponent *> *  
getChildIterator()  
Returns a dynamically instantiated iterator to a child component 172
```

```
virtual const TcGfxComponent*  
getCompByName(const TcUtString & name)  
Returns a pointer to component in the sub-tree rooted at this node and NULL if not found.
```

24.1  

**TcGfxGroup** (const TcUtString & name, TcGfxPfComponentAdapter * dspLibComp)  
*Constructor*
Appendix B (Continued)

Parameters:

- name — Name given to the component.
- dspLibComp — Display Library Component set in the base class.

24.2

TcGfxGroup (const TcGfxGroup &right)

Copy Constructor

Copy Constructor. Not yet implemented.

24.3

TcIterator <TcGfxComponent *> * getChildIterator ()

Returns a dynamically instantiated iterator to a child component

Returns a dynamically instantiated iterator to a child component. WARNING - The client is responsible for deleting this iterator.
Appendix B (Continued)

25

class TcGfxIntersectionManager

Intersection Manager Class

B.0.0.0.47 Public Members

TcGfxIntersectionManager ()
Constructor.

TcGfxIntersectionManager ()
Destructor.

virtual void tdUpdateIntersections ()
Update intersections. Can be overridden by descendents if necessary.

void addIntersectionTest (TcGfxIntersectionTest *test)
Add an intersection test to the manager.

doTcIntersectionTest will be called each iteration.

void deleteIntersectionTest (const TcUtString &id)
Remove an intersection with the given id.

TcGfxIntersectionTest*
getIntersectionTest (const TcUtString &id)
Returns an intersection test with the given id, and NULL otherwise.

Intersection Manager Class. This class handles the automatic update of intersectionTests that are added to it.
class TcGfxIntersectionTest

Abstract base class for Intersection Tests

B.0.0.0.48 Inheritance

26

TcGfxIntersectionTest

38

TdGfxPfIntersectionTest

B.0.0.0.49 Public Members

26.1 enum IsectState

26.2 enum CoordType

26.3 TcGfxIntersectionTest (const TcUtString &isectId)

virtual ~TcGfxIntersectionTest ()

virtual IsectState doTcIntersectionTest ()

const TcUtString&
Appendix B (Continued)

getIsectId () const  This method returns the string identification of the intersection test.

void setIsectId (const TcUtString &)  Sets the string identification for the intersection test.

const IsectState&
getLastUpdateState () const  Determines if an intersection was found in the last call to doTcIntersection test.

const TcUtString&
getLastIsectName () const  Returns the graphics component name of the last intersection.

B.0.0.0.50  Protected Members

void setIsectState (const IsectState &T)  Sets the intersection state.

void setIsectName (const TcUtString &T)  Sets the name of the graphics component intersected.

Abstract base class for Intersection Tests. This class specifies the interface for all Tandem core intersections.

26.1

enum IsectState
Appendix B (Continued)

Parameters:

| ISECT_NOTFOUND — No intersection was found. |
| ISECT_FOUND — An intersection was found. |
| ERROR — An error occurred. |

26.2

```c
enum CoordType
```

Parameters:

| WORLD — World coordinate system. |
| LOCAL — Local coordinate system. |

26.3

```c
TcGfxIntersectionTest (const TcUtString &isectId)
```

Constructor
Appendix B (Continued)

Parameters: isectId — String identification given to the intersection test.
class TcGfxLoadRequestPtr

Smart pointer class TcGfxLoadRequestPtr This class is instantiated on the stack in order to send a load request.

B.0.0.0.51 Public Members

TcGfxLoadRequestPtr ()

Constructor.

~TcGfxLoadRequestPtr ()

Destructor.

TcGfxPfBackgroundLoader::TcGfxLoadRequest*
operator-> ()
Forwards requests to the underlying LoadRequest.

TcGfxPfBackgroundLoader::TcGfxLoadRequest&
operator* ()
Forwards requests to the underlying LoadRequest.

Smart pointer class TcGfxLoadRequestPtr This class is instantiated on the stack in order to send a load request. It encapsulated a TcGfxLoadRequest.
class TcGfxManager

Class: TcGfxManager Role: This class manages Tandem’s graphics subsystem

B.0.0.0.52 Public Members

28.1 TcGfxManager (int argc, char *argv[], TcLoader *clientLoader)
    Constructor .............................. 180

virtual ~TcGfxManager () Destructor.

bool drawScene () Updates draw routines handled by the TcGfxLibrary.

void updateScene () Updates GL components and interactive events.

bool updateTravel () Updates the travel scheme.

TcGfxComponent*
    initTcGfxScene () Initialize the graphics scene.

void removeTcGfxComponent (TcGfxComponent *) Remove node from the TcGfxComponent hierarchy.

TcGfxComponent*
    getAvComponent () Returns the Avatar Group.

28.2 TcGfxComponent*
    getGfxCompByName (const TcUtString &name) const
    Returns the TcGfxComponent with the given name in the tree and NULL if it does not exist .......................... 181

28.3 TcGfxComponent*
getGLCompByName (const TcUtString &name) const
Returns the TcGfxComponent with the
given name in the GL tree and NULL if
it does not exist ...................... 181

TcTraveler* getTcTraveler () const
Returns a pointer to the TcTraveler.

void setIntersectionManager (TcGfxIntersectionManager &)
Sets the TcGfxIntersectionManager ...

void setViewPoint (const TcUtVector3f *rot,
const TcUtVector3f *trans)
Sets the position and orientation of the
viewpoint in the world .................. 182

Class: TcGfxManager
Role: This class manages Tandem’s graphics subsystem. This includes
drawing, searching, updating and maintaining its various objects.

Responsibilities: It provides an interface for searching TcGfxComponents. This class also
updates the TcGfxLibrary’s draw routines. In addition it traverses the TcGfxComponent hierarchy
each frame and updates iterative events.

28.1

TcGfxManager (int argc, char *argv[], TcLoader *clientLoader)

Constructor

Constructor. This class is instantiated internally by TcApplication.
Appendix B (Continued)

Parameters:  
clientLoader — Class returned by createLoader in TcApplication.

28.2

TcGfxComponent* getGfxCompByName (const TcUtString &name) const

Returns the TcGfxComponent with the given name in the tree and NULL if it does not exist

Parameters:  
name — This can be a TcUtString or a string literal.

28.3

TcGfxComponent* getGLCompByName (const TcUtString &name) const

Returns the TcGfxComponent with the given name in the GL tree and NULL if it does not exist

Parameters:  
name — This can be a TcUtString or a string literal.
28.4

```c
void setIntersectionManager (TcGfxIntersectionManager &)
```

Sets the `TcGfxIntersectionManager`

Sets the `TcGfxIntersectionManager`. This should not be done at runtime unless the client wants to change the manager.

28.5

```c
void setViewPoint (const TcUtVector3f *rot, const TcUtVector3f *trans)
```

Sets the position and orientation of the viewpoint in the world

Sets the position and orientation of the viewpoint in the world. Usually called by the interaction manager at startup.
Appendix B (Continued)

```cpp
class TcGfxObject : public TcGfxComponent
```

**Abstract Class TcGfxObject**

B.0.0.0.53 Inheritance

```
TcGfxComponent

\( \vee \)

TcGfxObject

\( \rightarrow \)

40

TdGfxPfObject

23

TcGfxGLObject
```

B.0.0.0.54 Public Members

- `TcGfxObject () Constructor.
- `TcGfxObject (TcUtString name) Constructor.
- `TcGfxObject (TcUtString nodeName, TcUtString fileName) Constructor.
- `TcGfxObject (TcUtString name,
  
  TcGfxDspLibComponent *dspLibComp) Constructor.
```
Abstract Class TcGfxObject. This is the abstract class for all GfxObjects. It takes care of methods that involve traversing the component tree as well as issuing warnings for clients that try to add or remove children everywhere else remains abstract.

29.1

TcGfxObject (const TcGfxObject &right)

Copy constructor

Copy constructor. Not yet implemented.

29.2

const TcGfxObject & operator= (const TcGfxObject &right)

Assignment

Assignment. Not yet implemented.
Appendix B (Continued)

class TcGfxPfComponentAdapter : public TcGfxDspLibComponent

Base class for all performer component adapters

B.0.0.0.55  Inheritance

B.0.0.0.56  Public Members

virtual ~TcGfxPfComponentAdapter ()
    Destructor.

pfNode* getPerformerNode () const
    Returns the performer node.

void setPerformerNode (pfNode *node)
    Sets the performer node.

virtual void hideGeometry ()
    Hides the geometry from the scene.
virtual void unhideGeometry () \hspace{1cm} \textit{Restores the geometry from the hidden state.}

virtual void loadGeometry () \hspace{1cm} \textit{Load geometry.}

virtual void saveMasks () \hspace{1cm} \textit{Saves the current APP, DRAW, ISECT masks.}

virtual void unloadGeometry () \hspace{1cm} \textit{Unloads the geometry from the scene.}

virtual const char* getCompId_c_str () \hspace{1cm} \textit{Return the char name pointer contained by the performer node.}

void deletePerformerNode () \hspace{1cm} \textit{Performs a pfDelete on the performer node.}

void setCompName (const TcUtString &node) \hspace{1cm} \textit{Sets the name of the pfNode.}

void setTravAttributes (const TdGfxPfTravAttributes &atts) \hspace{1cm} \textit{Sets a traversal mask for a given mask type.}

\textbf{B.0.0.0.57 Protected Members}

\begin{itemize}
\item \texttt{TcGfxPfComponentAdapter} () \hspace{1cm} \textit{Constructor.}
\item \texttt{TcGfxPfComponentAdapter} (pfNode *node) \hspace{1cm} \textit{Constructor \hspace{1cm} 187}
\item \texttt{TcGfxPfComponentAdapter} (const TcUtString &name) \hspace{1cm} \textit{Constructor}
\item void setPfName (const TcUtString &name) \hspace{1cm} \textit{Set the name of the performer node.}
\item pfNode* perfNode \hspace{1cm} \textit{Performer node.}
\end{itemize}
30.1

TcGfxPfComponentAdapter (pfNode *node)

Constructor

Parameters: node — A pre-instantiated performer node.
class TcGfxPfLibAdapter : public TcGfxDisplayLibrary

Core Graphics Performer Library Adapter

B.0.0.58 Inheritance

B.0.0.59 Public Members

31.1 TcGfxPfLibAdapter (int argc, char *argv[], TcLoader*ldr,
TcGfxPfBackgroundLoader *bgldr)
Constructor

~TcGfxPfLibAdapter ()
Destructor.

void doOneGfxCycle () Perform one performer draw cycle.

TcGfxComponent*
getGLComponent () Returns the GL component initialized in
this class.

virtual void setViewPoint (const TcUtVector3f *rot,
const TcUtVector3f *trans)
Appendix B (Continued)

Sets the viewpoint orientation and translation.

Core Graphics Performer Library Adapter. This class adapts the Performer CAVE library. It takes care of initialization and draw updates.

```
31.1

TcGfxPfLibAdapter (int argc, char *argv[], TcLoader*ldr, TcGfxPfBackgroundLoader *bgldr)
```

Constructor

**Parameters:**

- `argc` — argument passed from main() procedure.
- `argv` — argument passed from main() procedure.
- `ldr` — Pointer to the core loader.
- `bgldr` — Pointer to the background loader.
class TcGfxVisitor

Abstract base class for visitors and controllers

B.0.0.0.60 Inheritance

B.0.0.0.61 Public Members

TcGfxVisitor () Default constructor

virtual ~TcGfxVisitor () Destructor

virtual void visitGfxComponent (TcGfxComponent *c) Root base class visitor method ........ 191

virtual void visitGfxObject (TcGfxObject *o) Common base class TcGfxObject visit method ......................... 192

virtual void visitGfxGroup (TcGfxGroup *g) Common base class TcGfxGroup method

virtual void visitGfxPfObject (TdGfxPfObject *o) TdGfxPfObject visit method.

virtual void visitGfxPfGroup (TdGfxPfGroup *g)
Appendix B (Continued)

Abstract base class for visitors and controllers. Description: This abstract base class provides an interface for all actions that must distinguish the type of TcGfxComponent in order to carry out their behavior. It also allows clients to extend the behavior of the TcGfxComponent hierarchy.

Collaborators: ConcreteVisitors inherit from this class and then implement each of these interface methods. A client who holds a TcGfxComponent instance calls accept(visitor v) and passes a visitor instance as argument. See visitGfxComponent() for further detail.

32.1

virtual void visitGfxComponent (TcGfxComponent *c)

Root base class visitor method

Root base class visitor method. Description: This method provides an interface to specify default behavior for a Concrete Visitor.

Comments: To specify default behavior, a concrete visitor would place it into this method: ConcreteVisitor::visitGfxComponent(TcGfxComponent c) ...

Then when implementing any of the class specific visitors the default behavior could be extended.

eg. ConcreteVisitor::visitPfGfxObject(TcGfxObject *o) visitGfxComponent(o); <...> <extended implementation code>

In addition this is a catchall for extensions to the TcGfx hierarchy. See visitGfxObject for an example.
32.2

virtual void visitGfxObject (TcGfxObject *o)

Common base class TcGfxObject visit method

Common base class TcGfxObject visit method. Description: This is an interface for default TcGfxObject visitor behavior, see visitGfxComponent.

Comments: Like visitGfxComponent, this is a catchall for extensions to the TcGfxComponent hierarchy.

For example consider a new class called MyClassObject which inherits from TcGfxObject. MyClassObject::accept(visitor v) Because the visitor interface is not able to include new classes without recoding we call default behavior v->visitGfxComponent(this) v->visitTcGfxObject(this); New visitors could implement visitTcGfxObject as follows:

void myNewVisitor::visitGfxObject(TcGfxObject *o) visitGfxComponent(o);
MyClassObject *obj = dynamic_cast<MyClassObject * >(o); if(obj)<MyClassObjectspecificvisitorcodegoeshere>else<TcGfxObjectgenericvisitorcodegoeshere>

32.3

virtual void visitGfxGroup (TcGfxGroup *g)

Common base class TcGfxGroup method

Common base class TcGfxGroup method. See TcGfxObject for an example.
class TdGfxGLGroup : public TcGfxGroup

This class is used to setup a tree of GL components

B.0.0.0.62 Inheritance

B.0.0.0.63 Public Members

TdGfxGLGroup () Constructor

virtual ~TdGfxGLGroup () Destructor.

virtual int addChild (TcGfxComponent *gfxNode)

    Adds a TcGfxComponent GL object.

virtual int removeChild (TcGfxComponent *gfxNode)
Appendix B (Continued)

Removes a TcGfxComponent GL object.

virtual void hideGeometry ()  
Hides the geometry of a group of GL objects.

virtual void unHideGeometry ()  
Unhides the geometry of a group of GL objects.

virtual void loadGeometry (const LoadType& ldType)  
Loads the geometry of a group of GL objects.

virtual void unLoadGeometry ()  
Unloads the geometry of a group of GL objects.

virtual void draw ()  
Draw a group of GL objects.

virtual void update ()  
Update a group of GL objects.

virtual TcUtVector3f getPosition ()  
Return the position of GL objects.

virtual int setPosition (TcUtVector3f)  
Set the position of GL objects.

virtual TcUtVector3f getOrientation ()  
Get the orientation of GL objects.

virtual int setOrientation (TcUtVector3f)  
Set the orientation of GL objects.

virtual void accept (TcGfxVisitor &v)  
Used by the TcGfxVisitor mechanism.

This class is used to setup a tree of GL components. It is similar to the role of TdGfxPfGroup
for Performer components.

33.1  

TdGfxGLGroup (const TcUtString &id)
Constructors

Parameters: id — Sets the compName
Appendix B (Continued)

```cpp
class TdGfxPfDCSAdapter : public TdGfxPfGroupAdapter
```

**pfDCS adapter class**

### Inheritance

- **TcGfxDspLibComponent**
- **TcGfxPfComponentAdapter**
- **TdGfxPfGroupAdapter**
- **TdGfxPfDCSAdapter**

### Public Members

1. **TdGfxPfDCSAdapter** ()
   
   *Default Constructor* .......... 198

2. **TdGfxPfDCSAdapter** (float matrix[])

---

34

class TdGfxPfDCSAdapter : public TdGfxPfGroupAdapter
virtual ~TdGfxPfDCSAdapter ()

Destructor.

void setRot (const float &x, const float &y, const float &z)

Sets rotation in x, y, z float degree angles.

void setRot (const TcUtVector3f &)

Sets the rotation in x, y, z float degree angles.

void setTrans (const float &, const float &, const float &)

Sets the absolute translation in float x, y, z values

void setTrans (const TcUtVector3f &)

Sets the absolute translation in float x, y, z values

const pfDCS* 

get_perfDCS () const

Returns the pfDCS

pfDCS adapter class. This wrapper class constructs a pfDCS using TcUtVector3f objects. An
instance of this class is stored in the base TcGfxComponent as a TcGfxDspLibComponent. It is usually passed as an argument during the construction of a TdGfxPfGroup.

34.1

TdGfxPfDCSA dapter ()

Default Constructor

Default Constructor. scale = (1,1,1) rotation = (0,0,0) translation = (0,0,0)

34.2

TdGfxPfDCSA dapter (float matrix[])

Constructor

Parameters:

matrix — A matrix of 16 floats in row order a00..a03, a10..a13, a20..a23, a30, a33
Appendix B (Continued)

34.3

\textbf{TdGfxPfDCSAdapter ( const TcUtVector3f &orientation, const TcUtVector3f &translation )}

\textit{Constructor}

Constructor. These transformations are performed in the following order:

\textbf{Parameters:}

- \texttt{orientation} — Specified as float degree angles \texttt{x,y,z} NOT \texttt{head,pitch,roll}
- \texttt{translation} — Specified as \texttt{x,y,z} float values.

34.4

\textbf{TdGfxPfDCSAdapter ( const TcUtVector3f &scale, const TcUtVector3f &orientation, const TcUtVector3f &translation )}

\textit{Constructor}

Constructor. These transformations are performed in the following order:
Parameters:

scale — Specified as x,y,z and hence can be a non-uniform scale.

orientation — Specified as float degree angles x,y,z NOT head,pitch,roll

translation — Specified as x,y,z float values.

3.4.5

TdGfxPfDCSA
dapter (pfDCS *dcs)

Constructor

Parameters:

dcs — A pre-instantiated pfDCS.

3.4.6

TdGfxPfDCSA
dapter (const TcUtString &compID)

Constructor

Constructor. scale = (1,1,1) rotation = (0,0,0) translation = (0,0,0) This saves having to make
the setCompName(..) call.

Parameters:

compID — a string name that the pfNode will be set to.
class TdGfxPfGrabController : public TcGfxVisitor

**Performer grab controller class**

### B.0.0.0.66 Inheritance

- TcGfxVisitor
- TdGfxPfGrabController

### B.0.0.0.67 Public Members

- **Default constructor**
  ```
  TdGfxPfGrabController ()
  ```
- **Destructor**
  ```
  virtual ~TdGfxPfGrabController ()
  ```
- **Root base class visitor method**
  ```
  virtual void visitGfxComponent (TcGfxComponent *c)
  ```
- **Common base class TcGfxObject visit method**
  ```
  virtual void visitGfxObject (TcGfxObject *o)
  ```
- **Common base class TcGfxGroup method**
  ```
  virtual void visitGfxGroup (TcGfxGroup *g)
  ```
- **visitGfxPfObject**
  ```
  void visitGfxPfObject (TdGfxPfObject *o)
  ```
Appendix B (Continued)

void visitGfxPfGroup (TdGfxPfGroup *g)  

TdGfxPfGroup visit method.

void visitGfxGLObject (TcGfxGLObject *o)  

TcGfxGLObject visit method.

void visitGfxGLGroup (TdGfxGLGroup *g)  

TdGfxGLGroup visit method.

35.1 void reset (const TcUtVector3f &isectLocation)

This method must be called with the intersection point of the grab ............

void initArrow (TcGfxComponent &navGrp)  

This must be called once at instantiation.

void hideArrow ()  

This is used to hide the grab intersection arrow.

void setIsectLocation (const TcUtVector3f &loc)  

This is used to set the position the arrow is to appear.

Performer grab controller class. This is a development interaction class. A user can grab an object and transform it in a virtual environment. This can only be done on TdGfxPfDC- SAdapter classes. Error messages are shown for all other class types. This saves clients from having to switch on class types to achieve the same level of run-time type safety. First reset() must be called. Then an instance of this class must be sent as argument to a TcGfxComponent instance accept() method (the object to be grabbed). Objects in World Coordinates must be a TdGfxPfGroup containing a DCSAdapter. Objects in Local Coordinates must be TdGfxPfObject containing a DCSAdapter.
Appendix B (Continued)

This method must be called with the intersection point of the grasp.

Parameters: isectLocation — the location the object is grabbed from in World coordinates.
Appendix B (Continued)

class TdGfxPfGroup : public TcGfxGroup

Class TdGfxPfGroup

B.0.0.0.68 Inheritance

B.0.0.0.69 Public Members

36.1 TdGfxPfGroup (const TcUtString &name,
    const LoadState &initialLoadState,
    TcGfxPfComponentAdapter *dspLibComp)

    Constructor .........................  206

36.2 virtual ~TdGfxPfGroup () Destructor .........................  207

36.3 virtual void unLoadGeometry () This function unloads all children of this

    node, removing them from memory ...  207
36.4 virtual void loadGeometry (const LoadType &ldType = ASYNC)  
This function loads all children of this node ............................. 207

virtual void hideGeometry ()  Sets the APP, DRAW and ISECT masks to 0

virtual void unHideGeometry ()  Restores the APP, DRAW, ISECT masks to their state at the time of the call to hideGeometry

36.5 virtual int addChild (TcGfxComponent *gfxNode)  
Add a TcGfxComponent to the child list

36.6 virtual int removeChild (TcGfxComponent *gfxNode)  
Remove a TcGfxComponent from the child list ............................. 208

virtual TcUtVector3f getPosition ()  Not yet implemented.

virtual int setPosition (TcUtVector3f)  Not yet implemented.

virtual TcUtVector3f getOrientation ()  Not yet implemented.

virtual int setOrientation (TcUtVector3f)  Not yet implemented.

36.7 virtual void accept (TcGfxVisitor &v)  
Visitor pattern method ............................. 209

Class TdGfxPfGroup. This class is used to encapsulate groups of performer graphics components. It provides recursive operations that traverse its children. Its TcGfxDspLibComponent is generally an instances of TdGfxPfGroupAdapter or any of its descendants.

Role: This is a Development Graphics Group. Clients can use this class as is or inherit from it and extend its functionality. The grouping in this class specifies the load hierarchy. The load hierarchy can be used to partition the environment (swap objects in and out of performer memory) or define priorities (load surrounding objects based on proximity. The view hierarchy specifies the structure of the scene graph.
Appendix B (Continued)

Responsibilities: This class must implement the load and unload methods that operate on a list of children.

```
36.1

TdGfxPfGroup (const TcUtfString &name, const LoadState &initialLoadState, TcGfxPfComponentAdapter *dspLibComp)
```

Constructor

Constructor.

Parameters:

- name — This name defines the name used for searches on the TcGfxComponent tree. It must be unique in the entire tree to which it is to be added.
- initialLoadState — Groups to be asynchronously loaded should be set to UNLOADED and loadGeometry(ASYNC) call should be made after all the children have been added to the group.

```
36.2

virtual ~TdGfxPfGroup ()
```

Destructor
Destructor. This will take care of deleting all its children.

### 36.3

```cpp
virtual void unLoadGeometry ()
```

This function unloads all children of this node, removing them from memory.

This function unloads all children of this node, removing them from memory. Performer’s caching may sabotage certain unload efforts.

### 36.4

```cpp
virtual void loadGeometry (const LoadType &ldType = ASYNC)
```

This function loads all children of this node.

**Parameters:**

- `ldType` — An argument of SYNC will perform a synchronous load causing the program to stop responding in the interim. An argument of ASYNC will perform an asynchronous load of all children.
Appendix B (Continued)

36.5

```cpp
virtual int addChild (TcGfxComponent *gfxNode)
```

Add a TcGfxComponent to the child list

Parameters:
gfxNode — If this node is loaded its pfNode will be added to the scene by attaching it to the underlying pfGroup.

36.6

```cpp
virtual int removeChild (TcGfxComponent *gfxNode)
```

Remove a TcGfxComponent from the child list

Remove a TcGfxComponent from the child list. This method must be called to remove a child from a parent before the child is deleted.

36.7

```cpp
virtual void accept (TcGfxVisitor &v)
```

Visitor pattern method
Visitor pattern method. This will call v->visitGfxPfGroup to any visitor resulting in type laundering and implements a special case of double dispatching.
class TdGfxPfGroupAdapter : public TcGfxPfComponentAdapter

pfGroup adapter class

B.0.0.0.70 Inheritance

37

TdGfxPfGroupAdapter

22

TcGfxDspLibComponent

×

30

TcGfxPfComponentAdapter

×

37

TcGfxPfComponentAdapter

41

TdGfxPfSCSAdapter

34

TdGfxPfDCSAdapter

B.0.0.0.71 Public Members

37.1 TdGfxPfGroupAdapter (const bool & instantiate = true)
Appendix B (Continued)

37.2 **TdgFxPfGroupAdapter** (pfGroup *group)  
*Constructor*  

virtual ~**TdgFxPfGroupAdapter** ()  
*Destructor.*  

void **removeChild** (TcGfxPfComponentAdapter &childNode)  
*Removes the pfNode contained in the*  

childNode from the pfGroup.  

void **addChild** (const TcGfxPfComponentAdapter &childNode)  
*Adds the pfNode contained in the*  

childNode to the pfGroup.  

int **completeAsyncLoad** (const TcUtString &,  

TcGfxPfComponentAdapter &child)  
*Used by TcGfxComponent, after an*  

async load is completed.  

virtual bool **isActive** ()  
*Returns true;*  

B.0.0.0.72 **Protected Members**

37.3 **TdgFxPfGroupAdapter** (const TcUtString &compID)  
*Constructor*  

pfGroup adapter class. This wrapper class constructs a pfGroup.
Appendix B (Continued)

Parameters: instantiate — Determines whether a pfGroup is instantiated or not. By default a pfGroup is instantiated. Derived classes must set this to false to prevent instantiating two pfGroup nodes (or derivations from pfGroup).

37.2

\textbf{TdGfxPfGroupAdapter} (pfGroup \*group)

\textit{Constructor}

Parameters: group — Allows clients to pre-instantiate a pfGroup.

37.3

\textbf{TdGfxPfGroupAdapter} (const TcUtString \&compID)

\textit{Constructor}
Parameters: compID — Used by derived classes to set the pfNode name.
Appendix B (Continued)

class TdGfxPfIntersectionTest : public TcGfxIntersectionTest

**Perform Cavelib Intersection Test**

B.0.0.0.73 **Inheritance**

![Inheritance Diagram]

B.0.0.0.74 **Public Members**

38.1 enum Sensor

Sensor enum .......................... 217
Appendix B (Continued)

38.2 \texttt{TdGfxPfIntersectionTest} (const TcUtString &isectName, 
\texttt{const TdGfxPfIntersectionTest::Sensor}
\texttt{sensor}, 
\texttt{TcGfxComponent \&rootNode}, 
\texttt{const unsigned int \&isectMask}, 
\texttt{const unsigned int \&activeMask} 
= 0x01, \texttt{const int \&modeMask} = 
\texttt{PFTRAVJS_PATH} — 
\texttt{PFTRAVJS_GSET} — 
\texttt{PFTRAVJS_CULL_BACK})

\textit{Constructor} ......................... 218

\texttt{~TdGfxPfIntersectionTest} ()

\textit{Destructor}.

38.3 \texttt{void setActiveSegments} (const int \&numSegs = 1)

\textit{This sets the number of active segments as consecutive bits in the active mask} . 219

\texttt{void setIsectMask} (const unsigned int \&isectMask)

\textit{Sets the mask to search for in the graphics hierarchy}.

\texttt{void setMode} (const int \&modeMask)

\textit{Determines the behavior of the intersection test}.

38.4 \texttt{void setRayLength} (const float \&rayLength,
\texttt{const int \&segNum} = 0)
Appendix B (Continued)

Sets the length of a segment .......... 220

38.5 virtual IsectState

doTcIntersectionTest ()
updates the intersection test based on its
current state ....................... 220

38.6 const TcUtVector3f&

getIsectNormal (const int &segNum = 0)
returns the normal of the last successful
intersection ....................... 221

38.7 const TcUtVector3f&

getLastIsectPoint (const CoordType &type = LOCAL,
                const int &segNum = 0)
returns the position of the last successful
intersection ....................... 221

38.8 void setRayStartPositionFromWand (const int &segNum = 0)
set the start position for a given segment
number from the wand sensor ........ 222

38.9 void setRayDirectionFromWand (const int &segNum = 0)
set the direction for a given segment
number from the wand sensor ........ 222

38.10 void setRayStartPositionFromHead (const int &segNum = 0)
set the start position for a given segment
from the head sensor .............. 223

38.11 void setRayDirectionFromHead (const int &segNum = 0)
set the start direction for a given segment
from the head sensor .............. 223

38.12 void setRayStartPosition (const TcUtVector3f &vector,
                const CoordType &type = WORLD,
                const int &segNum = 0)
Appendix B (Continued)

Set the start position for the segment number from the given vector position

38.13 void setRayDirection (const TcUtVector3f &vector,
const CoordType &type = WORLD,
const int &segNum = 0)
Set the start direction for the segment number from the given vector orientation

38.14 void setCurSeg (const int &num)
Sets the segment number do TcIntersectionTest will operate on

Performer Cavelib Intersection Test. This class provides an interface to Performer’s pFSegSet. It can be automatically updated by passing it to the intersection manager, or the client can chose to update the intersection.

38.1

enum Sensor

Sensor enum
Appendix B (Continued)

Parameters:

HEAD — Set the intersection test from the head sensor.

WAND — Set the intersection test from the wand sensor.

NONE — Manually set the intersection position and direction.

---

```cpp
38.2

\text{TdGfxPfIntersectionTest} \ (\text{const TcUtString} \ &\text{isectName}, \ \text{const TdGfxPfIntersectionTest::Sensor} \ \text{sensor}, \ \text{TcGfxComponent} \ &\text{rootNode}, \ \text{const unsigned int} \ &\text{isectMask}, \ \text{const unsigned int} \ &\text{activeMask} = 0x01, \ \text{const int} \ &\text{modeMask} = \text{PFTRAVJS_PATH} \rightarrow \text{PFTRAVJS_GSET} \rightarrow \text{PFTRAVJS_CULL_BACK})
```

Constructor
Appendix B (Continued)

Parameters:

- **params**
  - **isectName** — This string is used to retrieve an intersection test from the intersection manager.
  - **sensor** — Determines whether doTeIntersection automatically updates the intersection test from a sensor position and direction.
  - **rootNode** — Sets the subtree in the graphics hierarchy that the intersection is tested from.
  - **isectMask** — Sets a bit field mask to test for.
  - **activeMask** — Sets the initial number of segments that are to be tested. This is a bitfield that determines which intersection segments are active.
  - **modeMask** — This is a Performer mask that determines the behavior of the intersection test. See pfSegSet.

```c
void setActiveSegments (const int &numSegs = 1)
```

*This sets the number of active segments as consecutive bits in the active mask*

This sets the number of active segments as consecutive bits in the active mask. For instance 5 active segments would cause the first 5 bits to be set to 1 in the activeMask.
Appendix B (Continued)

Parameters:  
numSegs — Determines the number of consecutive bits that are active.

```c
void setRayLength (const float &rayLength, const int &segNum = 0)
```

Sets the length of a segment

```
Parameters:
rayLength — A float value determining ray length.
segNum — Determines the segment to apply this length to. This must be less than the number of active segments.
```

```c
virtual IsectState doTcIntersectionTest ()
```

updates the intersection test based on its current state.
Appendix B (Continued)

Parameters: Returns — ISECT_FOUNDFound, ISECT_NotFound

38.6

const TcUtVector3f& getIsectNormal (const int &segNum = 0)

returns the normal of the last successful intersection

returns the normal of the last successful intersection. This value is not cleared after an update in which no intersection is found.

Parameters: type — Returns the normal for a given line segment.

38.7

const TcUtVector3f& getLastIsectPoint (const CoordType &type = LO-CAL, const int &segNum = 0)

returns the position of the last successful intersection

returns the position of the last successful intersection. This value is not cleared after an update in which no intersection is found.
Appendix B (Continued)

Parameters:

- **type** — Returns the point in local or world coordinates.
- **segNum** — Returns the point for the given line segment. This is a number starting from 0 and must be within the range of the active segments.

```c
void setRayStartPositionFromWand (const int &segNum = 0)
```

*Set the start position for a given segment number from the wand sensor*

Parameters:

- **segNum** — This is a number starting from 0 and must be within the range of the active segments.

```c
void setRayDirectionFromWand (const int &segNum = 0)
```

*Set the direction for a given segment number from the wand sensor*
Appendix B (Continued)

Parameters:  
\( \text{segNum} \) — This is a number starting from 0 and must be within the range of the active segments.

38.10

```
void setRayStartPositionFromHead (const int &segNum = 0)
```

*Set the start position for a given segment from the head sensor*

Parameters:  
\( \text{segNum} \) — This is a number starting from 0 and must be within the range of the active segments.

38.11

```
void setRayDirectionFromHead (const int &segNum = 0)
```

*Set the start direction for a given segment from the head sensor*

Parameters:  
\( \text{segNum} \) — This is a number starting from 0 and must be within the range of the active segments.
Appendix B (Continued)

38.12

```cpp
void setRayStartPosition (const TcUtVector3f &vector, const CoordType &type = WORLD, const int &segNum = 0)
```

*Set the start position for the segment number from the given vector position*

**Parameters:**

- **vector** — a positional vector.

- **type** — Currently if type is WORLD it will be converted to the viewpoint (local) coord system (CAVEConvertWorldToNav). This assumes the intersection starts from pfScene node.

- **segNum** — This is a number starting from 0 and must be within the range of the active segments.

38.13

```cpp
void setRayDirection (const TcUtVector3f &vector, const CoordType &type = WORLD, const int &segNum = 0)
```

*Set the start direction for the segment number from the given vector orientation*
Appendix B (Continued)

Parameters:

- **vector** — a directional vector.
- **type** — Currently if type is WORLD it will be converted to
  the viewpoint (local) coord system (CAVEConvertVectorWorld-
  ToNav). This assumes the intersection starts from pfScene node.
- **segNum** — This is a number starting from 0 and must be within
  the range of the active segments.

```c
void setCurSeg (const int &num)
```

*Sets the segment number doTcIntersectionTest will operate on*

Sets the segment number doTcIntersectionTest will operate on. This must be greater than 0
and less than the number of active segments.
class TdGfxPfNodeAdapter : public TcGfxPfComponentAdapter

Class TdGfxPfNodeAdapter

B.0.0.0.75 Inheritance

B.0.0.0.76 Public Members

39.1 TdGfxPfNodeAdapter (const TcUtString &compId)
Constructor  .......................... 227

39.2 TdGfxPfNodeAdapter (pfNode *nd)
Constructor  .......................... 227

virtual ~TdGfxPfNodeAdapter ()
Destructor.

39.3 int loadFile (const TcUtString &fileName)
Class TdGfxPfNodeAdapter. This class is a generic wrapper for pfNode. It is generally used to hold pfNode’s returned from pfdLoad(). It can also be used to write custom performer classes that contain pfObjects such as pfGeodes.

39.1

TdGfxPfNodeAdapter (const TcUtString &compId)

Constructor

Parameters: compId — Name stored in pfNode.

39.2

TdGfxPfNodeAdapter (pfNode *nd)

Constructor
Appendix B (Continued)

Parameters: nd — Pre-instantiated pNode.

### 39.3

```c
int loadFile (const TCHAR &fileName)
```

*Loads a file*

Loads a file. Results from loadGeometry() calls made to a TcGfxComponent.

Parameters: fileName — filename as it appears in PFPATH.
Appendix B (Continued)

class TdGfxPfObject : public TcGfxObject, public TcGfxPfBackgroundColorLoader TcGfxLoadRequest TcGfxBgLoaderObserver

Class TdGfxPfObject

B.0.0.0.77 Inheritance
Appendix B (Continued)

B.0.0.0.78 Public Members

virtual ~TdGfxPfObject () Destructor.

40.1 TdGfxPfObject (const TcUtString &name,

         const LoadState &ldState,

         const LoadType &ldType,

         TcUtString fname)

        Constructor ......................... 232

40.2 TdGfxPfObject (const TcUtString &name,

         const LoadState &ldState,

         const LoadType &ldType,

         TcUtString fname,

         const TdGfxPfTravAttributes &travAtts)

        Constructor ......................... 233

40.3 TdGfxPfObject (const TcUtString &name,

         const LoadState &ldState,

         TdGfxPfNodeAdapter *ndAdapter)

        Constructor ......................... 234

40.4 TdGfxPfObject (TcUtString name)

        Constructor ........................ 235

40.5 void loadFromSceneFile ()

        Read in a model description from a text

        file ............................... 235

40.6 virtual void clientLoadGfxObject ()

        Allows client extension of the loadFrom-

        SceneFile ........................... 236

40.7 virtual void loadGeometry (const LoadType & ldType = ASYNC)
Appendix B (Continued)

40.8 virtual void unloadGeometry () // Load geometry into the scene .......... 236
virtual void hideGeometry () // Unload geometry from the scene .......... 237
virtual void unHideGeometry () // Set the APP, DRAW, ISECT traversal
masks to 0.
virtual void unHideGeometry () // Restore the APP, DRAW, ISECT
traversal masks to their value prior to being hidden

40.9 virtual TcUtVector3f
getPosition () // Returns a vector with the current position
of the object ............................. 237

40.10 virtual int setPosition (TcUtVector3f)
// Sets the position of the object .......... 237

40.11 virtual TcUtVector3f
getOrientation () // Returns a vector with the current orientation
of the object ............................. 238

40.12 virtual int setOrientation (TcUtVector3f)
// Sets the orientation of the object ...... 238

40.13 virtual int gfxLoadUpdate (const TcGfxPfBackground-
Loader::TcGfxLoadRequest &
request)
// TcGfxBgLoaderObserver method implement-
ation ................................. 238

virtual void accept (TcGfxVisitor &)
// Visitor pattern method.

B.0.0.0.79 Protected Members
Class TdGfxPfObject. It contains information about an object that can be drawn in the scene.

Role: This is the development Graphics class for Performer leaf nodes. Clients can use it as is or extend its functionality by deriving a custom class.

Responsibilities: This class implements methods that enable clients to physically load and unload the object out of performer memory (pfdLoad) as well as the ability to hide and unhide the object.
Appendix B (Continued)

Parameters:

- **name** — String name given to TcGfxComponent and pfNode.
- **ldState** — Determines whether an object is loaded at instantiation.
- **ldType** — Determines whether an object is synchronously or asynchronously loaded.
- **fname** — Specifies the file name that pfdLoad will be sent.

---

**40.2**

```
TdGfxPfObject (const TcUtString &name, const LoadState &ldState, const
LoadType &ldType, TcUtString fname, const TdGfxPfTrawAttributes & travAtts)
```

*Constructor*

Constructor. An object cannot be instantiated with ldState == LOADED and ldType == ASYNC since asynchronous loading requires the TdGfxPfObject to be attached to a parent. This request will be ignored.
Parameters:

- **name** — String name given to TcGfxComponent and pfNode.
- **ldState** — Determines whether an object is loaded at instantiation.
- **ldType** — Determines whether an object is synchronously or asynchronously loaded.
- **fname** — Specifies the file name that pfLoad will be sent.
- **travAtts** — Specifies traversal attributes for one of DRAW, ISECT or CULL. Additional masks can be set by a setTravMask call on the pfComponentAdapter.

---

**TdGfxPfObject** (const TcUtString &name, const LoadState &ldState, TdGfxPfNodeAdapter *ndAdapter)

*Constructor*
Appendix B (Continued)

Parameters:

- **name** — String name given to TcGfxComponent and pfNode.

- **ldState** — Determines whether an object is loaded at instantiation.

- **ndAdapter** — A pre-instantiated pfNodeAdapter that implements loadFile()

---

40.4

**TdGfxPfObject** (TcUtString name)

*Constructor*

Constructor. This must have its TcGfxPfComponentAdapter explicitly set.

Parameters:

- **name** — String name given to TcGfxComponent and pfNode.

---

40.5

**void** loadFromSceneFile ()

*Read in a model description from a text file*

Read in a model description from a text file. Not yet implemented.
Appendix B (Continued)

40.6

virtual void clientLoadGfxObject ()

Allows client extension of the loadFromSceneFile

Allows client extension of the loadFromSceneFile. Not yet implemented.

40.7

virtual void loadGeometry (const LoadType& ldType = ASYNC)

Load geometry into the scene

Load geometry into the scene. This method will invoke the performer loader either in the main process or via the DBASE process.

Parameters:  

ldType — Determines whether a synchronous or asynchronous (using DBASE process) is performed.

40.8

virtual void unLoadGeometry ()

Unload geometry from the scene
Unload geometry from the scene. This method will remove the pfNode from the parent and pfDelete it, removing it from memory. An error will be displayed to the console if pfDelete is not successful.

40.9

virtual TcUtVector3f getPosition ()

*Returns a vector with the current position of the object*

Returns a vector with the current position of the object. Not yet implemented.

40.10

virtual int setPosition (TcUtVector3f)

*Sets the position of the object*

Sets the position of the object. Not yet implemented.
Appendix B (Continued)

virtual TcUtVector3f getOrientation ()

*Returns a vector with the current orientation of the object.*

Returns a vector with the current orientation of the object. Not yet implemented.

virtual int setOrientation (TcUtVector3f)

*Sets the orientation of the object.*

Sets the orientation of the object. Not yet implemented.

virtual int gfxLoadUpdate (const TcGfxPfBackgroundLoader::TcGfxLoadRequest& request)

*TcGfxBgLoaderObserver method implementation.*

TcGfxBgLoaderObserver method implementation. This method is invoked when a load request is completed.
Appendix B (Continued)

40.14

**TdGfxPfObject** (const TcUtString &name, const LoadState &ldState, TcGfxPfComponentAdapter *ndAdapter)

*Constructor*

Constructor. This constructor enables extension to the TdGfxPfObject class. It is useful when you want to add TdGfxPfObjects that have differing load and unload behavior than a typical TdGfxPfObject which is loaded from a pfLoad() call and expects the geoFileName member to be set. The client is responsible for attaching and unattaching the pfNode to the parent as a result of calls to loadGeometry or unLoadGeometry.

**Parameters:**

- **name** — String name given to TcGfxComponent and pfNode.
- **ldState** — Determines whether an object is loaded at instantiation.
- **ndAdapter** — A pre-instantiated pfNodeAdapter that implements loadFile()
class TdGfxPfSCSAdapter : public TdGfxPfGroupAdapter

**pfSCS adapter class**

**B.0.0.0.80 Inheritance**

**B.0.0.0.81 Public Members**

41.1 TdGfxPfSCSAdapter (const TcUtVector3f &orientation,

const TcUtVector3f &translation)
pfSCS adapter class. This wrapper class constructs a pfSCS using TcUtVector3f objects. An instance of this class is stored in the base TcGfxComponent as a TcGfxDspLibComponent. It is usually passed as an argument during the construction of a TdGfxPfGroup.

Constructor. These transformations are performed in the following order:

Parameters:

- orientation — Specified as float angles x,y,z NOT head,pitch,roll
- translation — Specified as x,y,z float values.
Constructor. These transformations are performed in the following order:

**Parameters:**
- **scale** — Specified as x,y,z and hence can be a non-uniform scale.
- **orientation** — Specified as float angles x,y,z. NOT head,pitch,roll
- **translation** — Specified as x,y,z float values.
class TdGfxPfTravAttributes

**Performer traversal attributes class**

### B.0.0.0.82 Public Members

#### TdGfxPfTravAttributes ()
*Constructor.*

#### TdGfxPfTravAttributes (const int &maskType,
const unsigned int &mask,
const int &mode,
const int &bitop)
*Constructor*

#### ~TdGfxPfTravAttributes ()
*Destructor.*

const TdGfxPfTravAttributes &
**operator=** (const TdGfxPfTravAttributes &right)
*Assignment operator.*

const int &
**getMaskType** () const
*Returns the type of mask. See the constructor.*

void
**setMaskType** (const int &value)
*Sets the type of mask.*

const unsigned int &
**getMask** () const
*Gets the value of the mask.*

void
**setMask** (unsigned const int &value)
*Sets the value of the mask.*

const int &
**getMode** () const
*Returns the mode of the mask.*

void
**setMode** (const int &value)
Appendix B (Continued)

Sets the mode of the mask.

const int& getBitop () const

Returns the bit operation to be performed.

void setBitop (const int &value)

Sets the bit operation to be performed.

Performer traversal attributes class. This class is used to set performer traversal properties for the APP, DRAW, ISECT traversals. It is generally used to set the TcGfxPfComponentAdapter classes or as an argument to the constructor of a TdGfxPfObject.

42.1

TdGfxPfTravAttributes (const int &maskType, const unsigned int& mask,
const int &mode, const int &bitop)

Constructor

Parameters:

maskType — Determines which mask to set: either ISECT — APP
— DRAW

mask — This is the value of the mask itself, interpreted as a bit-field.

mode — This determines how the mask is set and whether to propagate to its children.

bitOp — either PFOR, PFSET or PFAND
class TcNetCavConnectionManager : public TcNetConnectionManager

Role: This class plays the role of TcNetEvent shipper and receiver

B.0.0.83 Inheritance

B.0.0.84 Public Members
Appendix B (Continued)

43.1  

\textbf{TcNetCavConnectionManager} (int argc, char* argv[]),

\begin{verbatim}
    const long&
    clientLocalPort = 7000,
    const long&
    clientRemotePort = 7001,
    const bool&
    clientRemoteConnect =
    false, const TcUtString&
    clientUserName =
    TcUtString("Demo"),
    const TcUtString&
    clientWorldServer =
    TcUtString("localhost"),
    const TcUtString&
    clientWorldName =
    TcUtString("Demo")
\end{verbatim}

\textit{Constructor}  \hspace{10cm} 249

virtual \textbf{~TcNetCavConnectionManager} ()
\textit{Destructor.}

43.2  virtual TdNetCavConnection*
Appendix B (Continued)

```cpp
createConnection (const TcUtString &keyName, const TdNetCavConnection::connectionType &type)

Creates a Cavern Connection

void addConnection (TdNetCavConnection *connection)

Add a connection to the connection table.

TdNetCavConnection*

getConnection (const TcUtString &name)

Retrieve a connection from the connection table.

void deleteConnection (const TcUtString &name)

Remove a connection from the connection table.

void addKey (TdNetCavKey *)

Add a key to the key table.

TdNetCavKey*

lookUpCavKey (const TcUtString &name)

Returns a key with the given name.

void closeLink ()

Close a link.

virtual int config (long& clientLocalPort, long& clientRemotePort,

TcUtString& clientUserName,

TcUtString& clientWorldServer,

TcUtString& clientWorldName)

Must be called to initialize the class.

const TcUtString&

getWorldServer () const

Return the server name.

const TcUtString&
```
Appendix B (Continued)

getWorldName() const

Return the world name.

const TcUtString&

getUserName() const

Return the user name.

static const CAVERN_irb&

getcavernirb() const

Return the CAVERN irb.

const long&

getLocalPort() const

Return the local port.

const long&

getRemotePort() const

Return the remote port.

const bool&

getRemoteConnect() const

Return the value of the remoteConnect flag.

Role: This class plays the role of TcNetEvent shipper and receiver. It adapts the CAVERNsoft API to Tandem.

43.1

TcNetCavConnectionManager (int argc, char* argv[]), const long&

clientLocalPort = 7000, const long&

clientRemotePort = 7001, const bool&

clientRemoteConnect = false, const

TcUtString& clientUserName = TcUtString("Demo"), const TcUtString& client-

WorldServer = TcUtString("localhost"),

const TcUtString& clientWorldName =

TcUtString("Demo"))
Appendix B (Continued)

Constructor

Parameters:

argc — argument from main()
argv — argument from main()
clientLocalPort — port number of the client
clientRemotePort — port number of the server
clientRemoteConnect — determines whether connection to server is needed.
clientUserName — name requested for connection to server.
clientWorldServer — url of the server to connect to.
clientWorldName — name of the world to connect to on the server.

virtual TdNetCavConnection* createConnection (const TcUtString &keyName, const TdNetCavConnection::connectionType &type)

Creates a Cavern Connection
Appendix B (Continued)

Parameters:

keyName — The name of the Connection. Used to retrieve from the connection table.

type — Either UDP or TCP.
class TcNetCavEvent : public TcNetEvent

CAVERNsoft Network Event

B.0.0.0.85 Inheritance

44

TcNetEvent

\lor

44

TcNetCavEvent

B.0.0.0.86 Public Members

virtual TcNetCavEvent() ~ TcNetCavEvent() destructor.

virtual int sendToCavernKey(CAVERNirbKey_c &key)
Description: This virtual function is implemented by derived classes .......... 252

virtual int getFromCavernKey(CAVERNirbKey_c &key)
Description: This virtual function is implemented by derived classes .......... 253

const TcUtString&
getDestination() const
Returns the destination key.

void setDestination(const TcUtString &)
Sets the destination key.

const TcUtString&
Appendix B (Continued)

```cpp
getOrigin () const Used to get the sender.
void setOrigin (const TcUtString &) Used to set the sender.
```

### B.0.0.0.87  Protected Members

```cpp
class TcNetCavEvent () Constructor.
    CAVERN.irbKey.c::status_t &
    getKeyStatus () Returns the CAVERNsoft key status after a send or receive.
    char* eventBuffer Used for event data.
    char* metaBuffer Used for event meta data.
    CAVERN.irbKey.c::status_t keyStatus Used for the key status.
    TcUtString destination Used to set the destination key.
    TcUtString origin Used to set the sender.
```

CAVERNsoft Network Event. This class provides serialization via CAVERNsoft.

```cpp
virtual int sendToCavernKey (CAVERN.irbKey.c &key)
```

Description: This virtual functions is implemented by derived classes.

Description: This virtual functions is implemented by derived classes. The intent is such that each class will implement its own CAVERN packing scheme and put the data onto the key passed to it.
virtual int getFromCavernKey (CAVERN::irbKey &key)

Description: This virtual function is implemented by derived classes. The intent is such that each class will implement its own CAVERN unpacking scheme and get the data from the key passed to it.
Appendix B (Continued)

class TcNetConnectionManager

Abstract Base Class for Network Connection Manager

B.0.0.0.88 Inheritance

TcNetConnectionManager

TcNetConnectionManager

B.0.0.0.89 Public Members

virtual ~TcNetConnectionManager ()
Destructor.

int tdSendNetworkEvent (TcNetEvent& event)
Overridden by clients to determine how a
network event is sent.

bool isConfigured () const
Determines if this class has been initial-
ized.

virtual int config (long& clientLocalPort, long& clientRemotePort,

TcUtString& clientUserName,

TcUtString& clientWorldServer,

TcUtString& clientWorldName)
Appendix B (Continued)

Used to initialize the class.

B.0.0.0.90  **Protected Members**

**TcNetConnectionManager** ()  
*Constructor.*

bool  **configureFlag**  
*Used to set the configuration state.*
Appendix B (Continued)

class TcNetEvent

Abstract Base Class for Network Events.

B.0.0.0.91 Inheritance

TcNetEvent

TcNetCavEvent

B.0.0.0.92 Public Members

TcNetEvent () Constructor.
virtual ~TcNetEvent () Destructor.
virtual void resolveOutgoingEvent ()
    Used to store mapping information for outgoing events.

virtual void resolveIncomingEvent ()
    Used to store mapping information for incoming events.
class TcNetEventManager

Abstract base class for Event Manager

B.0.0.0.93 Public Members

TcNetEventManager ()

Constructor.

virtual ~TcNetEventManager ()

Destructor.

virtual void handleNetworkEvents ()

Must be overridden by clients to handle

received network events.
class TcNetManager

Abstract base class for Network Manager

B 0 0 0 94 Public Members

48.1

TcNetManager (int argc, char* argv[],
long& clientLocalPort,
long& clientRemotePort,
bool& clientRemoteConnect,
TcUtString& clientUserName,
TcUtString& clientWorldServer,
TcUtString& clientWorldName)

Constructor ......................... 260

virtual ~TcNetManager () Destructor.

virtual TcNetConnectionManager* createNetConnectionManager ()
Factory Method

virtual TcNetEventManager* createNetEventManager ()
Factory Method

int config ()
Must be called prior to using the factory method products.

bool isConfigured () const
Determines if config has been called.

void handleNetworkEvents ()
Appendix B (Continued)

Called from the Tandem Execution Control loop once per iteration.

const long& getLocalPort () Returns the client port.
TcUtfString getLocalHost () Returns the client host name.

B.0.0.0.95 Protected Members

const TcNetEventManager*
getEventManager () Returns the Event Manager
const TcNetConnectionManager*
getConnectionManager () Returns the Connection Manager

int localArgC argc argument from main()
char** localArgV argv argument from main()
long localPort port number of the client
long remotePort port number of the server
bool remoteConnect determines whether connection to server is needed.

TcUtfString userName name requested for connection to server
TcUtfString worldServer url of the server to connect to.
TcUtfString worldName name of the world to connect to on the server.
TcNetManager (int argc, char* argv[], long& clientLocalPort, long& clientRemotePort, bool& clientRemoteConnect, TcUtString& clientUserName, TcUtString& clientWorldServer, TcUtString& clientWorldName)

Constructor

Parameters:

- argc — argument from main()
- argv — argument from main()
- clientLocalPort — port number of the client
- clientRemotePort — port number of the server
- clientRemoteConnect — determines whether connection to server is needed.
- clientUserName — name requested for connection to server.
- clientWorldServer — url of the server to connect to.
- clientWorldName — name of the world to connect to on the server.
class TdNetCavConnection

*Class used to encapsulate a CAVERNsoft Connection*

### B.0.0.0.96 Public Members

#### 49.1 `enum connectionType Connection type enumeration`  
262

#### 49.2 `TdNetCavConnection`  
(const TcUtString &connectID,  
const TcUtString &worldServer,  
const long &remotePort)  
Connection::connectionType type = TCP)  
Constructor  263

#### ~TdNetCavConnection()  
Destructor.

#### 49.3 `void addLink (TdNetCavKey &key)`  
Link this key to key on remote server  263

#### `void removeLink (TdNetCavKey &key)`  
Removes a link.

#### `void setCavLinkAttribute (CAVERN_linkAttrib &att)`  
Sets the attributes for the link.

#### `void setCB (void (*call-back)(CAVERN_irbChannel &::CAVERN_channelEvent &,`  
event, CAVERN_irbChannel &thisChannel,  
void *userData), void *userData)`  
Sets the callback for the channel.

CAVERN_irbLink &*
Appendix B (Continued)

```cpp
getLink (TdNetCavKey &key)  
    Returns a link for a given key by reference to the key.

CAVERN_cavLink_c*
getLink (const TcUtString &key)
    Returns a link for a given key by name.
```

Class used to encapsulate a CAVERNsoft Connection. This class maintains a table of CAVERNsoft links.

```cpp
enum connectionType

Connection type enumeration
```

**Parameters:**
- **UDP** — Unreliable connection type.
- **TCP** — Reliable connection type.

```cpp
TdNetCavConnection (const TcUtString &connectID, const TcUtString &worldServer, const long &remotePort, TdNetCavConnection::connectionType type = TCP)
```
Appendix B (Continued)

Constructor

Parameters:
- connectionID — Specifies the string ID of the connection.
- worldServer — Specifies the destination URL of the server for the connection.
- remotePort — Specifies the remote port for the connection.
- type — Specifies a reliable or unreliable connection type.

49.3

```c
void addLink (TdNetCavKey & key)
```

*Link this key to key on remote server*

Parameters:
- key — The name of this key must match the name on the remote server.
Appendix B (Continued)

50

class TdNetCavKey

*Class used to encapsulate a CAVERNsoft key.*

B.0.0.0.97 Public Members

50.1 TdNetCavKey (const TcUtString &kName,
const TcUtString &worldName)

Constructor .............................. 265

50.2 TdNetCavKey (const TcUtString &name, void (*call-
Back)(CAVERN_irbKey_c::CAVERN_irbKeyEvent_t
event, Cavern.irbKey_c *thisKey,
void* userData))

Constructor .............................. 266

50.3 TdNetCavKey (const TcUtString &name, void (*call-
Back)(CAVERN_irbKey_c::CAVERN_irbKeyEvent_t
event, Cavern.irbKey_c *thisKey,
void* userData),

TdNetReceivingDock<TcNetEvent *>

*rDock*

Constructor .............................. 266

virtual ~TdNetCavKey () Destructor.

const TcUtString&

getKeyName () const

Returns the key name.

virtual int sendNetCavEvent (TcNetCavEvent &event)
Send a network event on the key.

```c
void setCB (void (*callBack)(CAVERN_irbKey::CAVERN_irbKeyEvent *event, CAVERN_irbKey::*thisKey, void* userData))
  Set the callback for the key.

void setDock (TdNetReceivingDock<TcNetEvent *> *)
  Set the receiving dock for the key.

static void setIRB (CAVERN_irb::* irb)
  Set the IRB for all TdNetCavKey class instances.
```

50.1

```
TdNetCavKey (const TcUtString &kName, const TcUtString &worldName)
```

Constructor

**Parameters:**

- `kName` — Name of the key
- `worldName` — name of the world.
Appendix B (Continued)

50.2

```
TdNetCavKey (const TcUtString &name, void (*callBack)(CAVERN.irbKey::CAVERN.irbKeyEvent event,
CAVERN.irbKey::thisKey, void* userData))
```

Constructor

Parameters:

- `name` — Name of the key

- `callBack` — callback to be triggered when data arrives on the key.

50.3

```
TdNetCavKey (const TcUtString &name, void (*callBack)(CAVERN.irbKey::CAVERN.irbKeyEvent event,
CAVERN.irbKey::thisKey, void* userData), TdNetReceivingDock<TcNetEvent *> *rDock)
```

Constructor
Appendix B (Continued)

**Parameters:**

- **name** — Name of the key
- **callBack** — callback to be triggered when data arrives on the key.
- **rDock** — Receiving dock associated with the key.
template <class T> class TdNetReceivingDock

Template class for queuing asynchronous network events.

B.0.0.0.98  Public Members

TdNetReceivingDock ()
    Constructor.

~TdNetReceivingDock ()
    Destructor.

void receiveItem (T item)  Put an item into the dock.

T processItem ()  Remove an item from the dock.

void clearDock ()  Clear the dock of all items.

bool empty () const  Check if the dock is empty. Return true if it is empty, false otherwise.

void lockDock ()  Lock the dock prior to calling empty, receiving an item or process item

void unlockDock ()  Unlocks the dock mutex.

int size () const  Returns the number of items in the dock.

51.1

void lockDock ()

Lock the dock prior to calling empty, receiving an item or process item
Lock the dock prior to calling empty, receiving an item or process item. Locking scheme for processing an item: lockDock if !empty, processItem, unlock dock

Locking scheme for receive lockDock receive Item unlockDock
Appendix B (Continued)

52

class TcUtDebug

*Debug utility class used throughout Tandem*

### B.0.0.0.99 Public Members

static TcUtDebug
  
  **debug**

  *Public static instance of itself.*

void **debugMsg** (const TcUtString &file,

  const TcUtString &className,

  const TcUtString &method)

  *Prints a debug message to standard out.*
class TcUtMutex

Base class for Tandem mutex

B.0.0.0.100 Inheritance

B.0.0.0.101 Public Members

---

enum lockResult_t

virtual lockResult_t lock ()

virtual lockResult_t tryLock ()

virtual lockResult_t unlock ()

virtual ~TcUtMutex ()

---

enum lock Results

Blocking mutex lock.

Non blocking lock attempt

Unlock mutex.

Destructor.
enum lockResult_t

Parameters:

DEADLOCK — Deadlock has been detected.

FAILED — Lock failed.

GOTLOCK — lock was successful.

BUSY — lock returned busy.

UNLOCK_OK — unlock was successful.
class TcUtSemMutex : public TcUtMutex

Mutex class built on posix semaphore for single process, threaded apps.

B.0.0.0.102 Inheritance

B.0.0.0.103 Public Members

54.1

TcUtSemMutex (const char *name)  Constructor  .....................  274
virtual ~TcUtSemMutex ()  Destructor.
virtual lockResult_t lock ()  Blocking mutex lock.
virtual lockResult_t tryLock ()  Non blocking lock attempt.
virtual lockResult_t unlock ()  Unlock mutex.
54.1

TcUtSemMutex (const char *name)

Constructor

Parameters: name — Used for debugging purposes.
class TcUtSemMutex_mp : public TcUtMutex

Mutex class built on posix semaphor for multi process apps.

B.0.0.0.104 Inheritance

B.0.0.0.105 Public Members

TcUtSemMutex_mp (sem_t*s) Constructor ..........................

TcUtSemMutex_mp () Constructor.

void setSemMutex_mp (sem_t*)

virtual ~TcUtSemMutex_mp () Destructor.

virtual lockResult_t
lock () Blocking mutex lock.

virtual lockResult_t
tryLock () Non blocking lock attempt.

B.0.0.0.106 Public Functions
unlock ()  

Unlock mutex.

55.1

TcUtSemMutex_mp (sem_f *s)

Constructor

Parameters:  
s — sem structure allocated in shared memory and initialized (semunit).
Appendix B (Continued)

56

```
template <class T> class TcUtObserver
```

Abstract Base class for observers

B.0.0.0.106 Public Members

- `virtual ~TcUtObserver ()` Destructor.
- `virtual void update (T* subject)` Must be overridden by clients. Called on observer notification.

B.0.0.0.107 Protected Members

- `TcUtObserver ()` Constructor.
Appendix B (Continued)

57

class TcUtString

Tandem String Class

B.0.0.0.108 Public Members

TcUtString ( const char *cstring = "" )
Constructor

TcUtString ( const TcUtString & str )
Copy constructor

~TcUtString ( )
Destructor

const TcUtString&
operator= ( const TcUtString & rhs )
Copy

const TcUtString&
operator+= ( const TcUtString & rhs )
Append

char* c_str ( ) const
Return C-style string

int length ( ) const
Return string length

char operator[] ( int k ) const
Accessor operator[]

char& operator[] ( int k )
Mutator operator[]

enum Maximum length for input string
Appendix B (Continued)

ostream& operator<< ( ostream & out, const TcUtString & str )

Output
istream& operator>>(istream & in, TcUtString & str)
Appendix B (Continued)

getline (istream & in, TcUtString & str)

Read line
Appendix B (Continued)

```c
bool operator==(const TcUtString & lhs, const TcUtString & rhs)
```

Compare ==
bool operator!= (const TcUtString & lhs, const TcUtString & rhs)

Compare !=
Appendix B (Continued)

bool operator< ( const TcUtString & lhs, const TcUtString & rhs )

Compare <
bool operator<=(const TcUtString & lhs, const TcUtString & rhs)
bool operator> ( const TcUtString & lhs, const TcUtString & rhs )
Appendix B (Continued)

```c
bool operator>=( const TCHARString & lhs, const TCHARString & rhs )
```

*Compare >=*
Appendix B (Continued)

Template String Hash Table

B.0.0.0.109 Public Members

template <class Data> class TcUtStringHashTable

TcUtStringHashTable ()
Constructor.

~TcUtStringHashTable ()
Destructor.

67.1 class iterator : public TcIterator<Data>
Nested Iterator class ...................... 289

67.2 int insert (const TcUtString &key, Data item)
Insert an item into the table ............. 292

TcUtStringHashTable<Data>::iterator
first ()
Returns an iterator to the first item in
the table.

67.3 TcUtStringHashTable::iterator
find (const TcUtString& name)
Returns an iterator to an item with index
matching name ............................. 292

void erase (const TcUtString& key)
Remove an index entry from the table.

bool empty ()
Returns true if the table is empty and
false otherwise.
Appendix B (Continued)

67.1

```cpp
class iterator : public TcIterator<Data>
```

*Nested Iterator class*

B.0.0.0.110 Inheritance

![Inheritance diagram]

B.0.0.0.111 Public Members

- **iterator ()** *Constructor.*
- **iterator (TcUtStringHashTable<Data> &t)**
  *Constructor* .......................... 290
- **iterator (const TcUtStringHashTable<Data>::iterator & right)**
  *Copy constructor.*
- **Data getItem () const**
  *Returns the current item* ............ 291
- **void first ()**
  *Reset the iterator to the first element in*
  *the table.*
- **void next ()**
  *Forward the iterator one element.*
- **const TcUtString&**
Appendix B (Continued)

```cpp
getKey() const
Return the current key index ...........

bool end() const
Determines if the end of the table has
been reached. Deprecated. See isDone().

bool isDone() const
Determines if the end of the table has
been reached, same as end().

67.1.4 Data

const TcUtStringHashTable<Data> ::iterator&
operator=(const TcUtStringHashTable<Data>::iterator&
right)

Assignment operator.
```

67.1.1

```cpp
iterator (TcUtStringHashTable<Data> &t)
```

Constructor

Parameters:

- t — An instance of a TcUtStringHashTable to be traversed.

67.1.2

```cpp
Data getItem() const
```
Returns the current item. Deprecated. See `currentItem()` It is important to check that the iterator is not at the end before trying to get the current item. Behavior is undefined if this method is called when `end()` returns true.

```
67.1.3

const TcUtString& getKey () const
```

Return the current key index. It is important to check that the iterator is not at the end before trying to get the key index. Behavior is undefined if this method is called when `end()` returns true.

```
67.1.4

Data currentItem () const
```

Returns the current item. It is important to check that the iterator is not at the end before trying to get the current item. Behavior is undefined if this method is called when `end()` returns true.
Appendix B (Continued)

67.2

```cpp
int insert (const TcUtString &key, Data item)
```

*Insert an item into the table*

Insert an item into the table. The implementation is based on SGI's hash map. Returns 1 on successful insertion and 0 on failure.

**Parameters:**

- `key` — Unique string name for hash index. The key **MUST** be a reference to dynamically allocated data whose lifespan is equal to that of the item. — Data associated with the key.

67.3

```cpp
TcUtStringHashTable::iterator find (const TcUtString & name)
```

*Returns an iterator to an item with index matching name*

Returns an iterator to an item with index matching name. If the item is not found, the iterator will point to the end of the table and the `isDone()` method will return true.
Appendix B (Continued)

class TcUtVector3f

Tandem 3 float utility vector

B.0.0.0.112 Public Members

TcUtVector3f () Constructor.
TcUtVector3f (const float &x, const float &y, const float &z) Constructor.

TcUtVector3f (float array[]) Constructor ......................... 294

TcUtVector3f (const TcUtVector3f &right) Copy constructor.

~TcUtVector3f () Destructor.

const TcUtVector3f&
operator= (const TcUtVector3f &right) Assignment operator.

void setVector (const float &x, const float &y, const float &z) Set operation.

const float & x () const X component accessor.
const float & y () const Y component accessor.
const float & z () const Z component accessor.

const float & operator[] (int k) const Accessor operator[i], i must be between 0 and 2 otherwise an exception is thrown

float & operator[] (int k) Mutator operator[i], i must be between 0 and 2 otherwise an exception is thrown

const TcUtVector3f&
Appendix B (Continued)

**normalize()**  
Normalizes the vector

**TcUtVector3f**

**getNormal** (const TcUtVector3f &vec2)  
*Returns a vector normal as defined by this vector cross product vec2*

**void**  
**pack** (CAVERNplus::datapack &datapack)  
*Serialization pack method for CAVERN-soft*

**void**  
**unpack** (CAVERNplus::datapack &datapack)  
*Serialization unpack method for CAVERN-soft*

---

**68.1**

**TcUtVector3f** *(float array[])*

*Constructor*

**Parameters:**  
array — An array of 3 float values.
Appendix B (Continued)

Class Graph

1. TcAction ...................................................... 104

1.1. Handler ...................................................... 105

2. TcActionEvent ................................................. 108

3. TcActionEventObserver ...................................... 110
Appendix B (Continued)

4
TcApplication .............................................................. 111

5
TcAudioPlayer ............................................................... 118

6
TcAvatarManager ........................................................... 123

7
TcExControl ................................................................. 124

8
TcInputManager ............................................................ 125

9
TcInteractionManager ..................................................... 127
Appendix B (Continued)

14
TcTravelScheme

15
TcTraveler

16
TcUserComponent

17
TcGfxPfBackgroundLoader

40
TdGfxPfObject

17.1
TcGfxLoadRequest
Appendix B (Continued)

33
TdGfxGLGroup

21
TcGfxDisplayLibrary

31
TcGfxPfLibAdapter

22
TcGfxDepLibComponent

30
TcGfxPfComponentAdapter

39
TdGfxPfNodeAdapter

37
TdGfxPfGroupAdapter

41
TdGfxPfSCSAdapter

193
162
188
164
185
226
210
240
Appendix B (Continued)

47
TcNetEventManager

.............................. 257

48
TcNetManager

.............................. 258

49
TdNetCavConnection

.............................. 261

50
TdNetCavKey

.............................. 264

51
TdNetReceivingDock

.............................. 268

52
TcUtDebug

.............................. 270


Name
Benjamin Goldstein

Education
Master of Science, Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, IL, 2000
Bachelor of Computer Science, Software Systems Option, Concordia University, Montreal, Quebec, Canada, 1998

Professional Experience
- **Lab Instructor for Database Design**: (May 1998 – June 1998). Concordia University, Montreal, Quebec, Canada. Responsible for design and documentation of course project. Gave several lectures on object-oriented database programming in UniSql. Duties also included supervision of designated lab time, as well as grading of project demonstrations and reports.
- **Systems Architect**: (September 1995 – September 1996). Lorbec Metals Ltd., St. Hubert, Quebec, Canada. Responsible for the development of finished goods inventory system. Work included requirements engineering, analysis and design. Duties also included installation and deployment of system on Novell 4.1xx LAN.
- **Office Automation Specialist**: (November 1992 – October 1995). Les Services de Personnel Manpower, Montreal, Quebec, Canada. Responsible for the development of training software for Windows 3.1 and related applications. Also included training of clients on software applications.
using tutorial software. Subcontracted during summer of 1995 to assist Hewlett Packard in
Montreal area installation of WAN.