Graph visualization using virtual environments

by

Frode Aarstad

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Carolina Cruz-Neira, Major Professor
Gary Leavens
Daniel Ashlock

Iowa State University

Ames, Iowa

2002

Graduate College
Iowa State University

This is to certify that the master's thesis of

Frode Aarstad

has met the thesis requirements of Iowa State University

Table of Contents

List of Figures

Acknowledgements

I would like to thank Dr. Carolina Cruz-Neira for her help and support during the creation of this work and for letting me work in great environment at the Virtual Reality Application Center.

I would also thank the VR Juggler development team for their efforts in creating a great piece of software. In particular, I would like to thank Patrick for his help and his swift and comprehensive responds to questions concerning VR Juggler.

Abstract

Virtual Reality (VR) is an emerging technology that has helped extend the boundaries of scientific visualization and many other different areas. This thesis focuses on applying virtual environments to a graph visualization problem and determining if the field of graph visualization can benefit from introducing virtual environments to aid the visualization. We present a list of goals for this work along with a background discussion of virtual reality and graph visualization. Then we cover the design of the virtual environment, the implementation details of the virtual environment and finally we perform a case study to evaluate our application. The case study is conducted with a graph of a metabolic pathway. Based on the implementation and the case study we can conclude that the development of this virtual environment has been successful. Our efforts have strengthened the hypothesis that virtual reality could provide useful insights into the field of graph visualization.

Chapter 1: Introduction

1.1 Motivation

Virtual Reality (VR) is a potent technology that has helped push the boundaries of scientific visualization and many other different areas. The immersivness of virtual reality among with intuitive user interfaces have created a more powerful computing platform compared to traditional computer screens and WIMP[1] user interfaces. Traditionally virtual reality has been restricted to large companies and research labs, but with the decreasing cost of powerful computer equipment and the increased popularity of open source software, virtual reality is becoming more and more popular and accessible.

Because of the natural three-dimensionality of immersive environments, one of the areas where they can be beneficial is visualization of graphs. Graph visualization is a sub-field of information visualization in which there is a relation between the data elements to be visualized. If such a relation exists then the data can be represented as the vertices of a graph, with the edges representing the relations.

According to [Herman00], applying virtual environments in graph visualization would be a new and interesting approach, and might provide exciting new insight in the field of graph visualization. It would also expand the application of virtual reality into other fields of study. Among the biggest current challenges in graph visualization is that an inherently three-dimensional problem is visualized using traditional two-dimensional computer screens and ultimately pen and paper. It is our belief that by implementing some of the traditional two-dimensional graph visualization methods on graphs in a virtual environment and designing new methods utilizing the visualization power of virtual environments we can

[1] WIMP Interfaces: Windows, Icons, Mouse, Pointers (or Pull-down menus)

provide a prototype of an immersive graph visualization space that demonstrates the usefulness of virtual reality in the field of graph visualization and thereby expand the applications of virtual reality into previously unfamiliar territory.

1.2 Scope of Work

The research presented in this thesis focuses on exploring the possibilities for graph visualization in the context of a virtual environment. It is our belief that using an extra dimension of visualization together with the interactive immersivness of virtual environments, we can develop a set of tools that will help application developers to deal with large information spaces. In particular we apply these tools to the specific field of graph visualization. We concentrate on developing tools and navigation methods for virtual environments but we will not address the problem of graph layout that according to [Hermman00] is an entirely different field in its own. Since there does not exist any universally used three-dimensional layout algorithm we use known two-dimensional algorithms for graph layouts and combine these with traditional visualization and navigation methods for virtual environments.

To achieve the goal, the research has been conducted in the following stages.

- Review related work on graph visualization.
- Define requirements for application.
- Design the virtual environment with Layered, Fisheye and Lens methods for graph visualization.
- Apply our methods to a set of laid-out graphs provided by our collaborators and obtain feedback from them after using our tool.

- Summarize results and findings.
- Recommend directions for future work.

Chapter 2: Virtual Reality Background

This chapter gives a brief introduction to the field of Virtual Reality focusing on the components needed for a useful Virtual Reality system. First we introduce and define the term Virtual Environment. Then we discuss user interfaces and input devices, components that are essential for user interaction with the virtual environment. Finally we cover software that simplifies the development of virtual environments.

2.1 Virtual Environment

Virtual Reality (VR) is a term that many people are familiar with, but according to [Cruz93] it is also a term that is widely misused. So in order to avoid confusion, we will use the term virtual environment to refer to our test-bed for graph visualization. According to [Stuart01] the term virtual environment is a more accurate term than virtual reality; since virtual environments do not necessary try to replicate reality. From Stuart we get the following definition of virtual environments:

> A Virtual Environment system is a human-computer interface that that provides interactive immersive multisensory 3-dimensional synthetic environments, it uses position tracking and real-time update of visual, auditory and other displays in response to the user's motions to give the user a sense of being 'in' the system, and it could be either a single- or a multi-user system. [Stuart01]

There exist several different types of such systems, each with a different degree of immersivness and different degrees of interaction. Systems also range from affordable desktop boxes to fully immersive projection systems. The system used by our work is described in detail below.

2.2 User Interfaces

A user interface is an interface between the user and the computer. For any computer system to be useful it requires a good user interface, because it is through the user interface the user interacts with the computer. According to [Norman90], any user interface designed for use in a computer system should mimic similar real world interfaces as much as possible. By designing the interfaces this way we would reduce the overall confusion of the user. But as [Bowman95] explains, creating user interfaces for virtual environments that are intuitive and resemble real world interfaces could increase the complexity of the interactions instead of reducing it.

A great deal of research has been done in constructing user interfaces for virtual environments, some researchers have called for revolutionary interfaces, while others have explored more traditional approaches, like using well proven two-dimensional user interface methods [Lindeman99]. There seems to be no consensus as to which type of user interface is more effective for use in a virtual environment, but according to [Lindeman99] and [Norman90], using interfaces that people are already familiar with is of great help in reducing the overall complexity of any system.

2.3 Input Devices

From our definition of virtual environment systems we know that such systems are highly interactive. Because of the high degree of interactions, input devices are an important part of any virtual environment system. Input devices range from various gloves, 3D mice, and voice recognition to simple wands. They allow the user to interact with and navigate in

the environment. Research has been done to establish effective and user-friendly input devices and the main challenge is to create an input device that is intuitive to the environment while still being powerful enough to use [Stuart01].

In a virtual environment that is designed with user navigation and exploring in mind, the developers might utilize other input devices than in an environment that is designed for the user to manipulate and interact with the environment.

It is also possible to utilize more than one input device at the same time. In a virtual environment where the emphasis is on being able to both explore the environment and manipulate the objects in the environment it might be advantageous to use a combination of devices. This way it is possible to combine an input device that is specialized for navigation and one that is specialized for object manipulation. It is however desirable to minimize the number of input devices to make the virtual environment more accessible to the user.

2.4 Navigation

Navigation in virtual environments is closely related to the traditional notion of navigation, finding a path from point A to point B. In this thesis navigation can be thought of as way finding in a large graph represented in the virtual environment. In all instances of navigation we have to address the problem of getting lost and disoriented, and not knowing how to proceed. So it is very important to design a navigation scheme that is simple and intuitive, while not confusing the user.

Several methods have been proposed to simplify navigation, some applications might benefit from letting the user be totally free to navigate in any direction, resembling flying. This navigation scheme allows the user to freely explore every part of the virtual

environment, but this requires the virtual environment to define proper boundaries and implement collision detection in order to prevent the user from getting lost and disoriented. This navigation scheme could also be restricted to only allowing navigation in the xz-plane, resembling walking. This might be more intuitive for the users and useful for virtual environments that do not require movement along the y-axis.

We also might just allow the user to follow a predefined path in the virtual environment. This is perhaps the simplest form of navigation and has been used successfully in [Bowman98]. By this navigation method we can direct the user to the interesting areas of the virtual environment and makes navigation easy for the users. On the other hand, this method reduces the users' ability to interact with the environment and reduces the users' ability to explore the environment freely.

Another method is to allow the user to change back and forth between different navigation modes, this way we can easily adjust the navigation scheme to fit the user and provide a more satisfying experience for the user. In our system we will use this technique and create a general navigation class that can be sub-classed by more particular navigation schemes.

Other schemes have been proposed to simplify navigation. In particular we implemented Worlds in Miniature [Stokley95], which approaches the navigation problem by creating a world in miniature of the large virtual world. We discuss the WIM as a navigation aid as well as an aid for interpreting the data.

2.5 VR Juggler

VR Juggler [Juggler01], [Bierbaum00] is an object-oriented development environment for efficient development of time-critical, interactive immersive applications independently of the underlying virtual reality technologies. Among the many features of the VR Juggler platform, it abstracts the complexities of the underlying VR system and allows the developer to use any of the many supported graphics API for developing virtual environments. By developing applications using the VR Juggler platform, a developer can write an application on a local VR system and run it on any other VR system without much effort.

The VR Juggler virtual platform (JVP) is the base for the VR Juggler system. The basic JVP system is composed of an application object, a draw manager, and the VR Juggler kernel. The interface between the application object and the JVP consists of the kernel interface that provides the hardware abstraction for the virtual platform, and the draw manager that provides the abstraction for the graphics API. The JVP kernel interface provides all application accessible functionality. The kernel is responsible for controlling all of the components in the VR Juggler system. The kernel does not depend upon any graphics API specific details, instead it captures all of these in the draw manager, which is an external manager of the VR Juggler kernel. Applications use the draw manager portion of the virtual platform interface to access any API specific details that are needed.

In VR Juggler all the user applications are objects. The VR Juggler system uses the application object to create the VR environment in which the user interacts. The application objects inherits from based application objects that define an interface that needs to be implemented by the application object. Then VR Juggler kernel maintains control over the

environment an calls the appropriate methods defined in the application interface. When the kernel calls the application's methods, it gives up control to the application object, allowing the application to execute the code needed to create the virtual environment.

The first step in writing an application object is to derive from the base classes that define the kernel and draw manager interfaces the application needs to implement. There is a base class for the interface that the kernel expects and a base class for interfaces needed by each of the available draw managers. For example in the `userOglApp` class in figure 2.1 the interface for the Kernel is `vjApp` and the interface needed by the draw manager is `vjGlApp`. Since all applications must interact with the kernel, all applications are required to implement the `vjApp` interface. Only OpenGL application objects need to implement the `vjGlApp` interface.

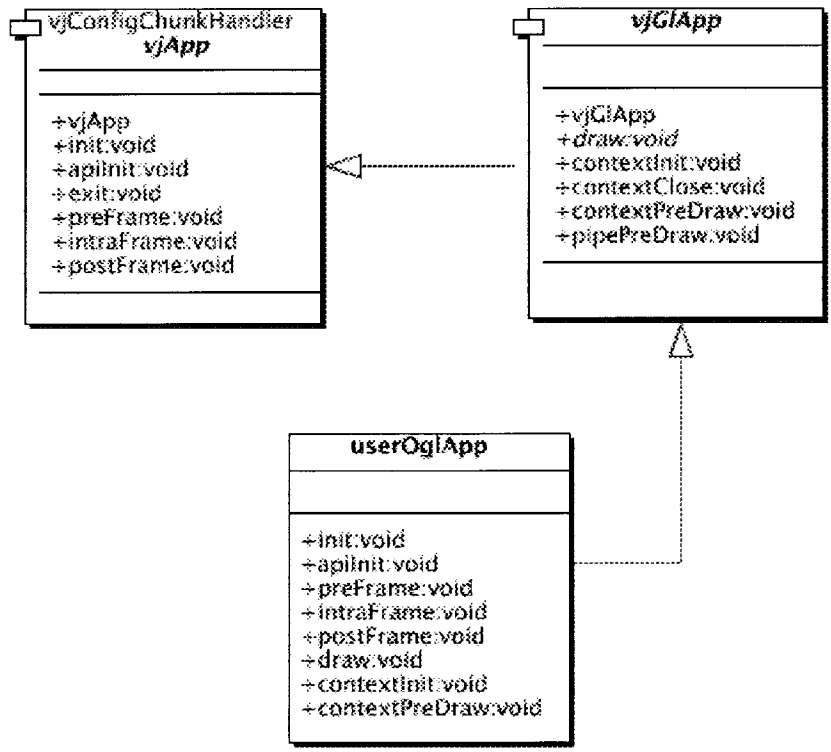

Figure 2.1 `userOglApp` hierarchy

This work uses the OpenGL [OpenGL01], which is a widely used and powerful two- and three-dimensional, graphics API. VR Juggler allows us to create OpenGL-based applications by deriving our application from the `vjGlApp` class, the derived class serves as the draw manager in the VR Juggler system.

By inheriting from the `vjGlApp` class we are assured that VR Juggler renders the OpenGL application correctly. This also gives us access to data from external from input devices through the `vjInputManager` class. And while the application overrides the functions in the `vjKernel` class from figure 2.1 correctly the developer is assured that the

application runs smoothly on anything from a Windows based PC to an immersive projection system.

2.6 Tweek

Tweek [Juggler01] is a module designed for VR Juggler that allows for communication between Java clients and a VR Juggler application. The communication is performed using the CORBA [CORBA02] standard for communication between software written in any language on any operating system on any hardware platform. In particular, Tweek can be divided into two parts, a Java API and a C++ API. The C++ API is used by a potentially complex VR Juggler application to define objects that the Java API can access and modify. This pattern is often called a subject/observer pattern and is essentially what the Tweek module implements. This pattern defines a relationship between the Java Graphical User Interface (observer) and the C++ application (subject).

Tweek uses Swing and Java Beans to create a generalized graphics user interface (GUI) framework. This allows the developers to plug in components (Beans) into this framework at runtime to extend its functionality.

This work will use Tweek run on a palm computer iPAQ [Compaq02] or a tablet computer [Intermec02] for the Java GUI and this will in turn interact with the graph visualization application.

2.7 Our System

This work has been developed using VR Juggler on both Linux and IRIX platforms, the portability and scalability of VR Juggler allows us to develop using Linux machines and then transform the code to IRIX with only minor problems.

The Virtual Reality system used by this work consisted of an immersive surround screen projection [Cruz95] system, using six projectors to fully immerse the user. The system provides wireless tracking using Ascension Technologies wireless MotionStar [Ascension01] magnetic tracker. User interaction with the virtual environment is achieved by using a RemoteRF tracked wand [Interlink01] and a tablet computer [Intermec02].

The tracked wand is used for navigation and interacting with the environment, while the palm computer is used for displaying textual information about the user's interactions and query information about the environment. Figure 2.2 shows the input devices utilized by this work.

Figure 2.2 Input devices.

Chapter 3: Graph Visualization Background

This chapter introduces some important terms relating to graphs and graph visualization. First we give an introduction to the field of graph visualization and discuss some of the problems encountered in graph visualization and how introducing virtual environments into the field could possibly provide useful new insight and new ideas to graph visualization. Then we discuss metabolic pathways, which is an area that graph visualization techniques has been applied to. Finally, we discuss some existing visualization systems.

3.1 Graph Visualization

Graph visualization deals with visualization methods for many different types of data represented in the form of a graph. Some types of data have elements that are unstructured and with no inherent relation, while other types of data might be structured with an inherent relation between the elements in the dataset. In the case when there is no apparent relationship between the data, the goal of the visualization system is to help discover possible relations along the data using visual means. On the other hand, if the data elements are somehow related to each other, the data can be represented by vertices in a graph, with the edges representing the relation between the vertices.

Usually, the term "Graph Drawing" elicits an image of the classic textbook style drawings of graphs. But there are many approaches to viewing such data including less obvious methods such as 3D scenes, cityscapes, nested boxes and cubes, and alternative geometries [di Battista99]. Graph visualization has many areas of application. File hierarchies can be represented as a tree, which is a special type of a graph. The layout of the

World Wide Web is also a graph with each computer being a vertex and the connection between them coded in the edge. Other applications of graphs are compiler data structures, Parallel computer architecture, VLSI circuit semantics, scene graphs and Network architecture. These are just a few of the applications of graphs in the field of computer science. Other fields like biology, chemistry and math also rely heavily on graphs to store, relate and visualize information.

The size of the graph to be viewed is a key issue in graph visualization. Large graphs from large datasets raise several difficult problems when designing a visualization system. According to [Herman00], only a few systems can claim to handle datasets with several thousand elements effectively, even though most of the interesting problems and applications are of this magnitude. The size of a graph also puts restrictions on the visualization methods and algorithms that can be applied to the graph. There is also no guarantee that an algorithm that works great for one hundred elements, will scale and work well for one thousand elements. In most cases algorithms do not scale well at all. A computer screen can only display a certain amount of data before the view becomes cluttered and occlusions occur, making navigation and interaction with the particular vertices impossible. Consequently, a first step in the visualization process is often to cut down the size of the graph to view. As a result, classical layout algorithms still remain usable tools for visualization, but only when combined with other methods.

Only recently have researchers in graph visualization turned their attention towards developing three-dimensional layout algorithms and methods for graphs [di Battista99] and [Herman00]. The intuition is that the addition of an extra dimension will increase the available space to view graphs and therefore reduce clutter and ease interaction with the data.

According to [Herman00] the simple approach is to generalize the well-proven classical two-dimensional approaches to three-dimensions. However the addition of the extra dimension also adds to the complexity of the visualization. Objects in 3D can occlude one another, thus making it hard to find good and interesting views in space. As a consequence, many three-dimensional displays of graphs include additional visual cues, like transparency, depth queuing, etc. They also allow the user to interactively change the view by navigating the space. But the ability to change perspective adds another difficulty; common practices such as the minimization of edge-crossings is less rewarding if the user can change the perspective and see edge-crossings from another angle.

Another problem in graph visualization is that it is easy to lose spatial awareness and get information overload when exploring a graph. The graph visualization community has addressed these issues and several methods regarding navigation in large graphs and different layout techniques have been implemented and discussed. The biggest problem faced is that using two-dimension displays and input devices to interact with the three-dimensional problem of graph visualization creates an inherent discrepancy. According to [Herman00] three-dimensional interaction and use of virtual environments that immerse the user in the graph may have a profound effect in field of graph visualization.

3.2 Metabolic Pathways

There has been a lot of work done to utilize graph drawing methods to visualize metabolic pathways [Karp94]. To understand Metabolic Pathways it is necessary to explain what happens during the metabolism. Metabolism represents the chemical processes occurring within a living cell or organism that are necessary for the maintenance of life. In

metabolism some substances are broken down to provide energy for vital processes while other substances, necessary for life, are synthesized. This process is a complex one, and involves several different substances that interact with other to create new substances. Metabolic pathways are the maps that describe this process and the substances involved when a particular substance is synthesized. These maps contain vertices that represent specific biochemicals such as proteins, RNA and small molecules, or stimuli such as light heat or nutrients. Edges of the map capture regulatory and metabolic relationships found in the biological systems.

3.3 Existing Systems

The area of information visualization has reached a level of maturity in which large applications and application frameworks are being developed. However, many of these systems are pure research tools and have a short lifespan. We will comment on a few of the systems that are more interesting and relevant to our work. In particular, we will discuss the works of [Risch96], [Orimo99] and [Fairchild88]. These systems all have in common that they were designed to visualize various types of three-dimensional information using different visualizing techniques.

The Semnet [Fairchild88] system was a pioneering three-dimensional graph visualizing work. The system explored multi-dimensional scaling for laying out many simultaneous vertices. The developers concluded however, that due to the large number of vertices that were laid out in a three-dimensional space, visualization was very complicated and therefore hard for the user to draw and conclusions from the data set and explore any relations.

The Starlight [Risch96] system was designed for interactively visualizing exploratory intelligence data. Starlight also tried to solve the problem of establishing relationships between information in large datasets. This system created a linkage display system that allows for displaying information in multiple levels of abstraction. The developers also experimented with text and graphical-based interactions with the system. The system has a high degree of user interaction and allows the user to freely navigate in the information environment. This allows the user to inspect the data set from an exocentric viewpoint in order to get the best possible view of the data set. The system also allowed the user to select elements of the data set in order to further investigate particular data elements. The developers concluded that using several different simultaneous visualizations together with a high degree of user interaction would potentially be a very powerful approach for enabling effective interactive exploration of complex information spaces. The system was developed for a desk mounted binocular display with a conventional two-dimensional computer display, a six-degree of freedom input device and a voice recognition system.

The ZASH [Orimo99] system was designed for browsing and displaying multi-dimensional data stored in a database with information collected from a selection of movies. The system used three-dimensional space to improve visibility of links between the vertices in their data set. To aid in visualizing the data, the system incorporated fisheye views, multi dimensional scaling and multiple planes. Although this system were developed for traditional two-dimensional computer screen displays, the developers concluded that by using a three-dimensional space and viewing methods like fisheye, multidimensional scaling and multiple planes, the system was able to improve the visibility between links and enhance the understanding of closeness of the vertices. Since the main focus of the ZASH system was to

support browsing and visualizing relations, the system did not provide any elaborate user interaction other than textual input using a traditional keyboard.

The aforementioned systems are all ground breaking and interesting systems, but the development of VR systems has come a long way since they were first developed. In out case we are working in a more immersive and capable environment than was the case in the Starlight, Semnet and ZASH systems. Therefore we build upon different parts of these systems when we design our virtual environment for graph visualization. We try in particular to mimic Starlight's ability to freely navigate the information environment and query each individual data element, by allowing the user to freely navigate our environment and retrieve information associated with any selected vertex or edge in the graph. This work also builds upon ZACH's development of viewing methods like fisheye and multiple planes. By implementing different views we are able to address the problem of information overload, a problem that occurs in both immersive environments and graph visualization.

Chapter 4: Design

This chapter discusses the design decisions that were made prior to implementing the application. First we discuss the input devices chosen for interaction with the virtual environment, and then we cover the overall design chosen for the application. We also reveal the structure a VR Juggler application and discuss how this influences our design. Then we cover the decisions that were made regarding the graph and graph formats in our application. Finally, we examine the views that were drawn from our research and deemed suitable for inclusion in our application.

4.1 User Interaction

In a virtual environment designed for visualization of graphs, both navigation and interaction is important. The user should be able to navigate in the virtual environment while interacting freely with the vertices, edges and other graph information. To enable the user to perform these actions, we use two different input devices concurrently: a wand type device for navigation and a palm computer for further interaction with the application. In our virtual environment information overload is a big problem. If the user is immersed in a huge graph with many edges and many vertices it is important that there is an effective method for interaction with the environment. It is also important that the method does not occlude the view and provides detailed information about the graph.

By moving some of the interaction from the virtual environment to the palm computer we allow the user to interact with the environment through a more user-friendly and familiar windows based interface. The user is still able to interact with the vertices and

edges in the environment, but textual based interaction is performed through interacting with the palm computer.

There are several advantages with using a palm computer as an additional input device. Some of the information stored with each vertex in the graph might be of textual nature with comprehensive narrative; this kind of information is more suited for displaying on a palm computer. It also allows the user of the virtual environment to use the palm computer to take notes and store information extracted from the virtual environment. In our test case, metabolic pathways, there exist two-dimensional applications that visualize the same data. It is our belief that using the palm computer as a bridge between our virtual environment and any other visualization package would be a great asset. This approach has not been implemented but looks promising.

4.2 VR Juggler Call Sequence

Since the application is built upon the VR Juggler framework, the application needs to adhere to the VR Juggler call sequence. The call sequence depicted in figure 4.1 is typical for a VR Juggler OpenGL application. The following section provides an overview of some of the important functions from the VR Juggler calls sequence that the main application class needs to override, these functions are called continuously by the kernel while the application is running.

The `preFrame()` function is invoked by VR Juggler's kernel before each frame is rendered. When this function is invoked the information from the input devices are the freshest. This is because the kernel queries the input manager after the previous post frame, invoking the `updateAllData()` function. It is during this function that the application

needs to update the navigation, perform collision detection and update animation based upon this new information.

The `draw()` function renders a frame of the virtual environment. The frame rendered is determined by the state entered during the `preFrame()`.

The `postFrame` is performed after a frame has been rendered. This function is used to determine timing issues. We can calculate how long it took for a frame to be rendered and update the timing of any animations.

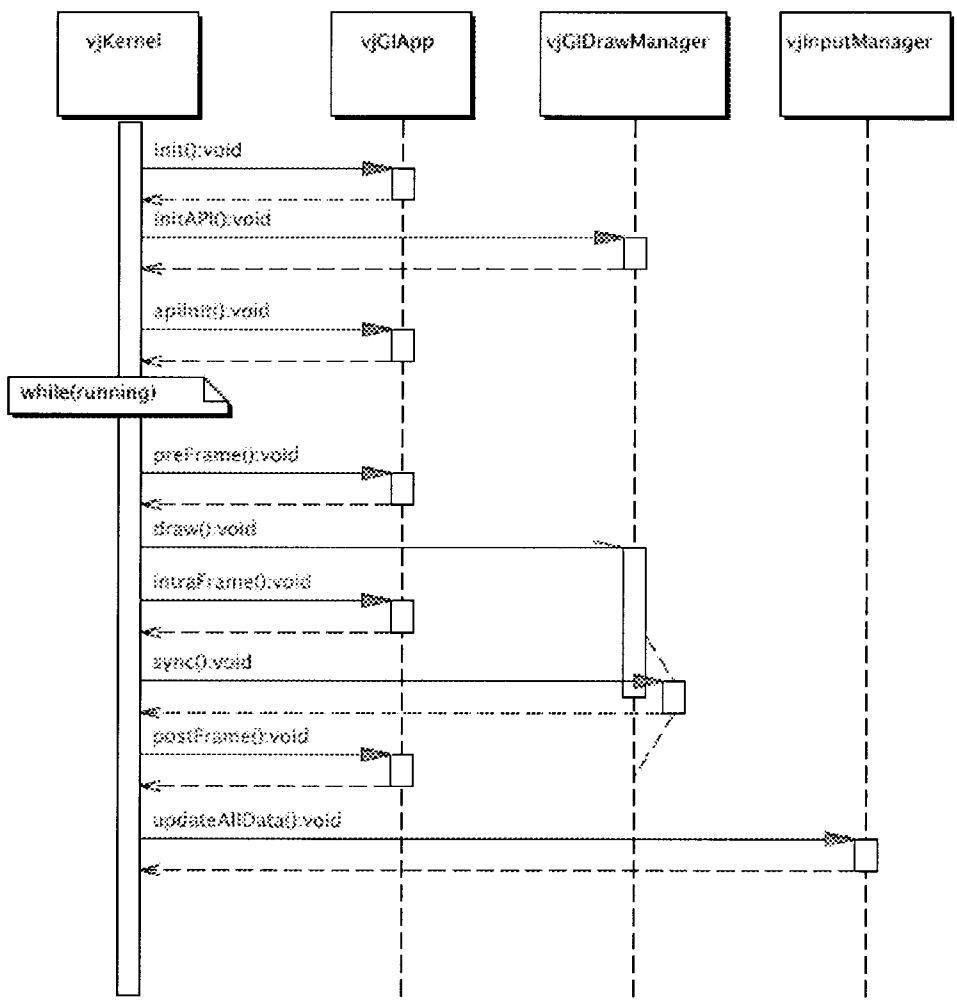

Figure 4.1 VR Juggler call sequence

4.3 Design Overview

This section explains the design of the system in more detail. Figure 4.2 gives an overview of the graph visualization environment's design. The design consists of two main categories of classes, view and application.

The application category of class consists of classes that deal with the user interaction and underlying system interaction. Form the information obtained from the input devices the classes must handle the user navigation and communicate between the application and the GUI via Tweek. To archive integration with the underlying system, the main application class must inherit from VR Juggler's `vjGlApp` class. The main application class also needs to provide an interface for Tweek, an interface that the Tweek objects can use to access and modify variables in the main application class. In order to facilitate navigation, the navigation class needs to be able to access tracker and input device data from the main application class. The navigation must then update this information according to the selected navigation scheme and send the updated data back to then main application class.

The view category of classes consists of classes that are designed for providing the visuals for the application. The view classes consist of a graph class that is responsible for parsing and storing information about graphs, a shared data class that holds information that needs to be shared between the different views and a hierarchy of classes that implements the different views. A particular view needs to implements its own versions of the important functions from the VR Juggler call sequence described in the previous section. The view should also be able to access the graph and system information through the shared data class, and from this information being able to render the graph in the virtual environment according to the view's design.

This logical separation between application and view is desirable because it creates a clear separation between the graphics and system interaction parts of the application.

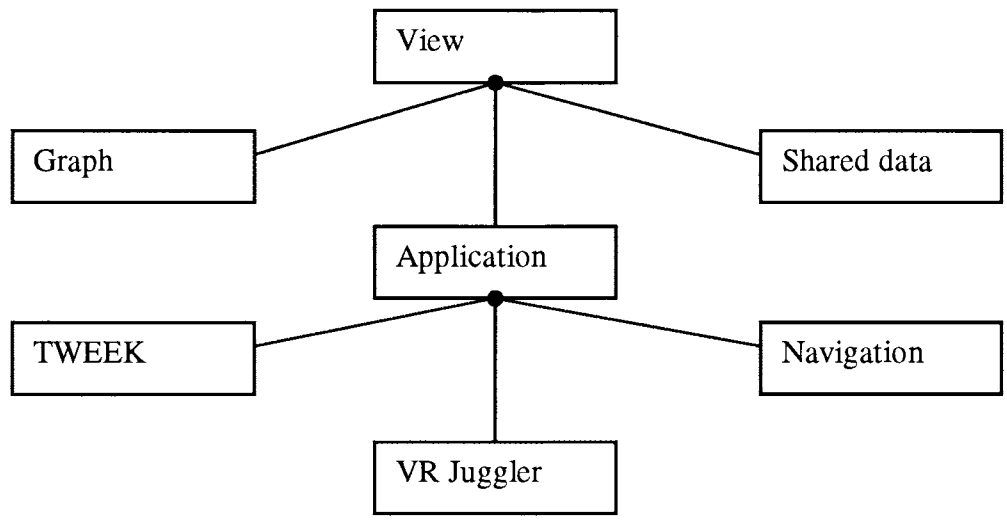

Figure 4.2 Top level class diagram of our system.

4.4 Graph Formats

Currently there are no standards for a file format to store and manipulate graphs. Therefore, due to this lack of any common standard for graph file formats, we decide to create a structure for parsing that allows for easy addition of new parsers for different file formats as they become available. Initially the system implemented parsers for XML [W301], [Apache01] and GML [Himsolt97]. This allowed us to load different graphs with data from a variety of different fields. For this work we have obtained graphical data from biology, data that describes a metabolic pathway. This data consists of 403 vertices and 540

edges, so it is a rather large dataset, but still small compared to other possible datasets, it is our hope is that our methods will scale well.

4.5 Views

In accordance with the scope of work, one the main focuses was to create a simple framework for adding different views of graphs to the virtual environment. Most of the current methods and algorithms for graph drawing and graph visualization [Tamassia97], [Herman00] have been developed for two-dimensional applications in mind. These two-dimensional methods do not easily translate into to our three-dimensional virtual environment. We also went to the literature on virtual environments and searched for methods, not specifically developed for graph visualization, but that could be useful for our purposes after some modifications. We implemented the Graphical Fisheye View [Fuarns86], [Noik93], Magic Lens [Napari00] and a layered approach.

4.5.1 Fisheye View

The graphical fisheye [Furnas86] view of graphs is built on the notion of fisheye lenses. Fisheye lenses are lenses that have a very wide angle, thereby showing nearby entities in greater detail while showing remote entities in successively less detail. This intuition can also be used in graph exploration as vertices close to the center of the user's viewpoint could be represented in greater detail. This is particularly useful in virtual environments since we can always calculate the user's position and direction accurately and then we can draw the vertices in the graph with an appropriate level of detail according the calculated user's

position. The works of [Sheelagh95], [Farchild88] and [Sarkar92] are examples of systems that have successfully implemented such a fisheye view of data.

Converting Furnars's implementation of the fisheye view from its native form into a form useful for virtual environments was straightforward. Fisheye views require the a-priori importance of a vertex and the distance from the vertex to the user's current position to be known. Both these parameters can be varied and calculated independently in order to create different and interesting views. In particular, we could let the a-priori importance of a vertex be the weight of the vertex, or a combination of weight and the size of the sub-graph at the vertex. The distance from the vertex to the user's position could be calculated using either the Euclidian or polar distance, each yielding different views of the particular graph.

The fisheye approach to a viewing method shows promise. We have implemented variations of the traditional fisheye view. One implementation has a uniformly evolving lens where the vertices grow and shrink in real-time as the user's position changes in the virtual environment while the closest vertex always is in focus and has additional information attached to it. Another implementation is a level evolving lens, where we define a set of levels and define the distance limits for each level and provide a distinct draw function for each level. This approach allows us to view the vertices closest to the user with the same level of detail while the vertices father way has successively less detail. We also allow for the user to freeze the view at any given time. This fixes the graph and allows the user to navigate in the virtual environment while the graph maintains the state it was in at the time the user invoked the freeze command.

One of the benefits of our implementation of this fisheye view is that it can also be used in other virtual environments. By displaying objects close to the user in greater detail

than those farther can be used to significantly increase the frame rate in many graphics heavy applications. Our implementation could be useful in the example of a room with several objects; the objects closest to the user are the ones he is the most interested in and these could be drawn with greater detail while the details of the objects farther away could be blurred and reduced.

4.5.2 Lens View

The lens metaphor is built on the intuition of a lens that the user might hold in front of an object in the virtual environment. Normally when holding a lens in front of an object would magnify the object assuming we have a magnifying lens. But since we have control over the rendering process, instead of magnifying the object we can draw the object differently when viewed through the lens. According to [Napari00], the lens metaphor has been applied successfully on both two-dimensional and three-dimensional environments before. In [Bier93], the authors observe that lenses can serve many roles in enhancing the user's interaction and understanding of software applications. Lenses can show local details in the context of larger scale information and limit clutter to a small region in the environment. In our application we envision the lens as a tool to further investigate the information associated with an object in the virtual environment, in particular additional information about a vertex or an edge in the graph.

One use of the lens is to increase the level of details of the objects covered by the view through the lens. For example, the edges could be displayed with its weight, name or any other information associated with the edge. The vertices could be displayed with name, number of incident vertices, weight, description, etc. In particular the lens allows us to reduce

the information displayed about the graph in the environment, the user is able to select a region of the graph by putting the lens in front of it and then this region will be displayed through the lens with more details and information.

Our system implements the lens view using OpenGL's built in clipping planes. Clipping planes allow us to draw two versions of the scene, one version of the regular scene and one version of the scene as seen through the lens.

As with the fisheye view, the magic lens view works well and could prove useful for other types of virtual environments. One example is a transparent lens; with such a lens the user could be able to see through objects in the scene that are within boundaries of the lens.

4.5.3 Layered View

We also implemented a layered view based loosely upon the work done in the Starlight [Risch96] and Zach [Orimo99] systems. The two aforementioned systems relate elements from the data set by projecting links between elements onto different planes, where each plane represents a specific parameter from the elements in the data set.

This work will use this approach to reduce the amount of information displayed in the virtual environment, and hence reduce user confusion and improve understanding of the data. This approach uses a weight function to identify vertices that are more 'important' than the others. The weighting function for a vertex might be calculated from the number of edges incident to that vertex, weight of that vertex etc. This enables the user to navigate a smaller subset of the original data and observe the more 'important' vertices. The user is then able to select a vertex and explore the incident vertices. By selecting one of the incident vertices the user has the option of adding this vertex to the current view of vertices.

The application also facilitates real-time adjustment to the weight function, so that the user can experiment with the weight function in order to get a manageable sized subset of the original data.

We have implemented two different layered approaches, cone-layer and normal layer. In the normal layer approach we display the vertices incident to the current vertex selected by the user in the xy-plane, the vertices are projected in the xy-plane relative to the position it has to the selected vertex. VR Juggler uses a right-handed coordinate system, so when we refer to coordinates; positive x axis points right, positive y points up and negative z points forward in the virtual environment. This way the user can get a feel of the structure and relationships between the selected vertex and its incident edges.

In the cone layered approach we project the vertices incident to the selected vertex spread out like in a cone under the selected vertex. In this approach we loose the structure and relative position of the incident vertices to the selected vertex, but we get a spatial condensed representation of the vertices incident to the selected vertex. This approach is more useful for a vertex with several incident vertices and can display these incident vertices in a useful and convenient manner for the user.

4.6 World in Miniature

The World in Miniature (WIM) [Stoakley95] is a navigation metaphor that has been used with success as a tool for navigation in large virtual environments. The WIM is a scaled down or an appropriately rendered miniature version of the original environment. While the user is navigating in the original environment the user sees his position in the original environment through the relative position in the WIM. This helps the user to understand

where in the environment he is located and can help him better understand the overall structure of the data. The WIM can be regarded as a map of the virtual environment, thus aiding the user in navigation and way-finding. The WIM can be implemented in several ways; one can use it as a simple map, one can allow the user to manipulate things in the WIM and let these changes occur in the real environment, or one can render a different view of the virtual environment in the WIM. In our case, we will strictly implement the WIM as a navigation and overview tool, and not allow any manipulation of the environment in the WIM.

The traditional variation of the WIM is used for representing a top-down view of the virtual environment with the user's position displayed with an icon in the WIM. WIM's were developed for applications where the user is mainly allowed to move in the xz-plane with not much movement among the y-axis. In our case the user should be able to move freely among all of the x, y and z axis and a pure top-down view of the environment would not easily reflect the users movement among the y-axis. In order to remedy this we implemented a three-dimensional WIM. The three-dimensional WIM is a scaled down model of the current virtual environment. This allows us to display the user's position in the WIM accurately in three-dimensional space.

Chapter 5: Implementation

We now present the implementation of our graph visualization environment based on the background and design discussed in the three previous chapters. The discussion is based upon the top level class diagram of our system shown in figure 4.2. We present a more detailed description of the two major components of the application, the view component and the application component.

5.1 View

The view classes are the part of the application that handles the implementation of the different views described in the previous chapter. The application provides an abstract class `graphView` for all the subsequent view classes to inherit from. This class includes all the functions each subclass class is expected to implement. A view essentially specifies a particular method to display the information stored in the graph in the virtual environment. Therefore it is necessary for each sub class of `graphView` to implement each own version of the methods expected by the OpenGL Draw Manager (figure 4.1). So by adding `preDraw()`, `draw()` and `postDraw()` as pure virtual functions in the `graphView` class the subsequent subclasses are forced to implement their own versions of these important functions.

Each particular subclass is also allowed to implement their own distinct `drawWIM()` function. By doing this, each different view can provide a unique way to display the graph in the WIM, a way that is useful for that particular view method. This is useful since, for example, a layered view might only have a subset of the whole graph displayed in the virtual

environment at a particular time and therefore it might also be useful to only display this subset of the graph in the WIM also.

Figure 5.1 gives an overview of the views that this work has implemented and the relationships between these views. There are three main groups of views implemented, `gvFisheye`, `gvLayered` and `gvLens`.

The `gvFisheye` views are bases upon the fisheye view metaphor discussed earlier. In this view the weight associated with each vertex is added to the Euclidian distance from each vertex to the user, the resulting weight is used by the application to classify the vertex. The Euclidian distance from the vertex to the user is calculated by calling the `distance()` function from the `oglPQPObject` class with the PQP matrix for the wand stored in the `gvSharedData` class. Then `gvFisheye` class classifies the vertices in layers according to this calculated weight, where each layer has its own level of detail applied for each vertex classified into that layer. The `gvFisheyeSmooth` class scales each vertex according to the vertex's weight so that the closest vertices are bigger and display more detail while the vertices further away are show smaller with successively less detail.

The `gvLayered` views are based on the layered approached discussed earlier. The application uses the `weightCalc()` function to determine which vertices are to be included in the current layer of the view. The current layer holds the vertices that are being displayed in the virtual environment at any time. The class uses the map container from the standard template library (STL) to create a map of the vertices. In particular, the class map the pair <string vertex_name, vertex *>, a string with the name of the vertex and a pointer, to the vertex. The reason for using the map structure is to facilitate fast and easy look up of

vertices based upon the vertex name. The class maintains two such mappings, one for the current layer and one for the other vertices not in the current layer.

When the user selects one vertex in the virtual environment, the vertex selection is determined when the distance from the wand to the vertex is within a threshold from the value returned form `oglPQPObject`'s `distance()` function. Then the class determines which layer the selected vertex is contained in, if the selected vertex is in the current layer the incident vertices to the selected vertex are then displayed. The gvLayered class draws the incident vertices to the selected vertex in the xz-plane, the incident vertices are drawn in their respective positions only rotated to appear in the xz-plane. The user is then able to select any one of the incident vertices in the xz-plane, and by selecting one of them the user adds the selected vertex to the current layer. The vertex most recently promoted to the current layer has its vertex and incident edge colors changed from the normal vertex and edge colors. This additional visual cue helps aiding the user by emphasizing the last action, thus distinguish the vertex selected in the last action from the other vertices.

The `gvLayeredCone` view class takes a little different approach to displaying the incident vertices. While the previous gvLayered class drew the incident vertices in their relative positions only rotated into the xz-plane, this class displays the incident vertices in a circle in the xz-planed directly below the selected vertex.

The `gvLens` class implements the Magic Lens approach discussed earlier. This work implements this method by using OpenGL's clipping planes. OpenGL supports the use of at least six clipping planes, although some implementations might supply more, this work utilizes five clipping planes. The five clipping planes are the left, right up, down and front clipping planes. Currently no far clipping plane is specified, this implies that the lens in

theory has an unlimited visual range. The clipping plane's positions are calculated from the position of the wand and the lens that is drawn in front of the wand. To archive the lens effect the `draw()` function in the `gvLens` class, first draws one version of the scene with the clipping planes enables then disables the clipping planes and draws another version of the scene without the clipping planes.

This work implements two variations of the lens. The `gvLensFixed` class specify the clipping planes by planes that extend from the boundaries of the lens towards infinity, like a square shaped tunnel. The `gvLensAngle` class also uses clipping planes but extend towards infinity from the boundaries of the lens at an angle, extending the clipping volume towards infinity in a square shaped cone.

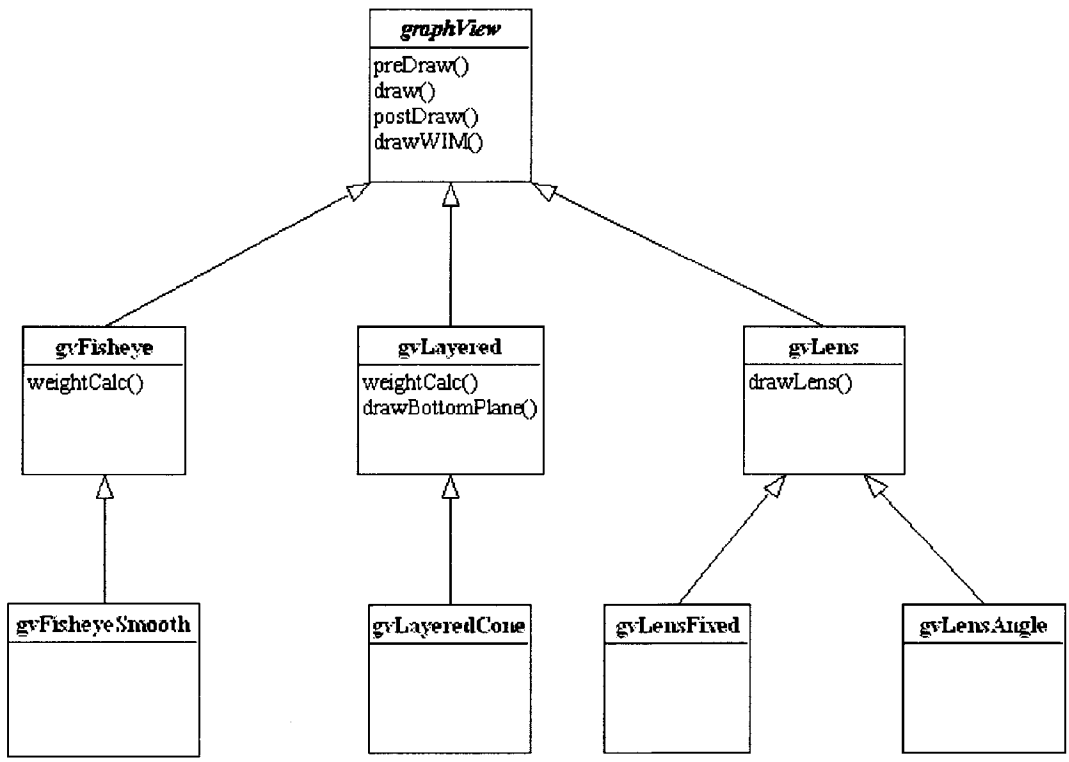

Figure 5.1 `graphView` class diagram.

5.2.1 Draw Manager

The `gvDrawManager` class in figure 5.2 is a utility class that manages and stores the available views and graphs. This class acts as an intermediary between the system and visual parts of our application or between the `graphVizApp` and `graphView` classes. The graphVizApp class maintains an instance of this class and calls its `preDraw()`, `draw()` and `postDraw()` functions when appropriate, while the `gvDrawManager` class in turn calls the `graphView` class's corresponding functions. This class also provides the `graphVizApp` class methods for changing the views and graphs.

This class maintains a vector of all the available graphs and all the available views. It also maintains a vector of all the WIMs associated with each available graph. When the user changes the current graph through the GUI the draw manager creates views for this graph and notifies the shared data that the graph has changed and sends the shared data a pointer to this newly selected graph.

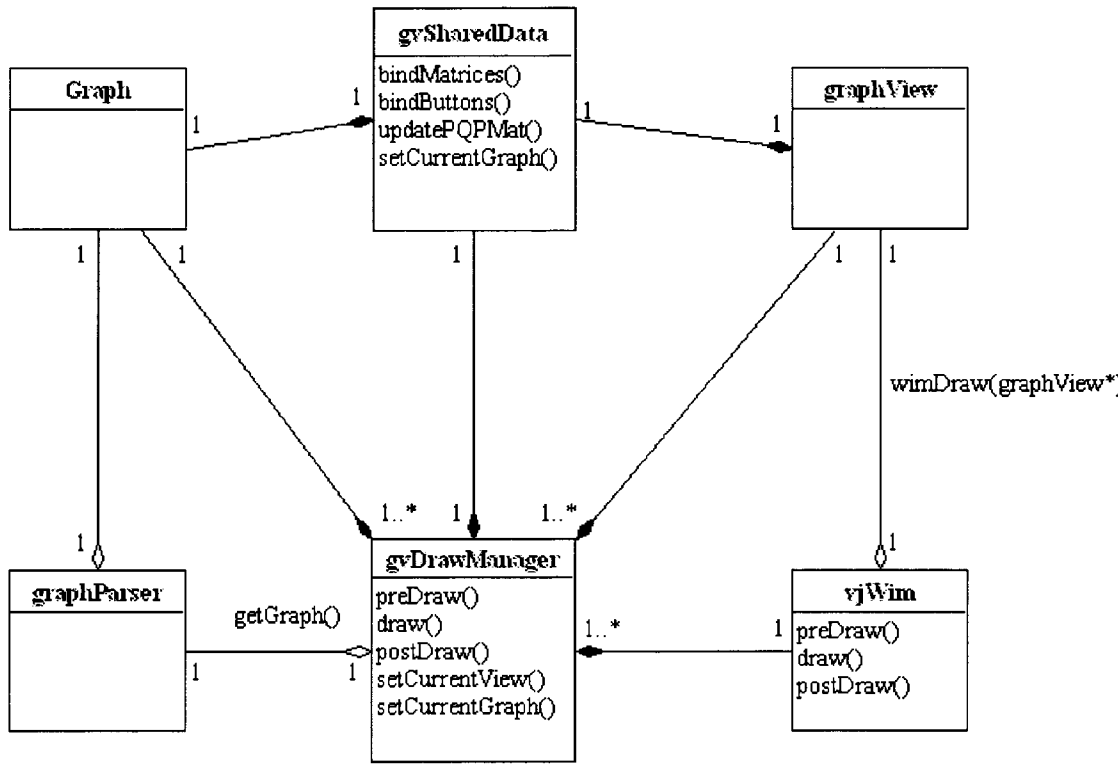

Figure 5.2 `gvDrawManager` class diagram.

5.2.2 Shared Data

The class `vjSharedData` is a utility class to the `vjGraphView` class and `gvDrawManager` class. It stores some current information about the system, information that has to be accessible to all the different view classes. This class is responsible for storing information about the head position, wand position and it also holds the current navigation matrix and the state of the buttons on the wand. The class uses this information to maintain PQP models of the head and wand, which are calculated using the current navigation matrix along with the respective matrices. The shared data also stores information about the vertices and edges in the graph, information like the closest vertex, selected vertex, marked vertices

etc. By having a class like this separated from the views, the user is allowed to change views while the shared data maintains information about the user's position, actions and preferences. This class also maintains a pointer to the graph currently being shown in the virtual environment, this pointer allows the views to access and display the correct graph. The pointer to the current graph is maintained and set by the `gvDrawManager` class through the `setCurrentGraph()` function.

5.2.3 Graph

The graph class contains the data structures needed for parsing and representing graphs. The class `graphParser` is the abstract class the specific parsers must inherit from. This application has currently implemented two parsers `GMLParser` and `XMLParser`.

The graphs are parsed and then stored in the `graph` class depicted in figure 5.3. This class maintains a list of the edges and vertices present in the current graph. The application does not need a more elaborate data structure for storing the graph since traversing the entire vertex and edge list is only necessary every time the graph is drawn in the virtual environment. The only functionality that could be a target for optimization is the `graph` class's `findVertex()` function. This function traverses list of vertices in the graph class in order to find a specified vertex. By changing the way vertices are stored from a list to another data structure like sorted list, the complexity of this function could be reduced. The same is true for the `findEdge()` function. But since neither of these functions are likely to be called a significantly number of times the simpler data structure for storing the vertices and edges were selected.

The class `vertex` stores information associated with each vertex in the graph. This class is a subclass of the `oglPQPObject` class. This inheritance creates a PQP object for each vertex. The Proximity query package (PQP) [PQP01] is a library for different types of proximity queries performed on geometric objects. This work utilizes the library's ability to calculate the distance between two geometric objects, to efficiently calculate the distance between the user and each vertex. According to [Lin98] PQP is the most efficient package. This allows us to determine the distance from the user and conclude if the vertex is selected or not. In the case of fisheye views the layout of the graph can be determined by querying each vertex for the distance from the user.

The class edge stores the information about the edges in the graph. As the case with the vertex class `edge` is also a subclass of `oglPQPObject`, this allows us to query the edges about their distance to the user. The class edge also maintains a pointer to each of the two vertices associated with that edge.

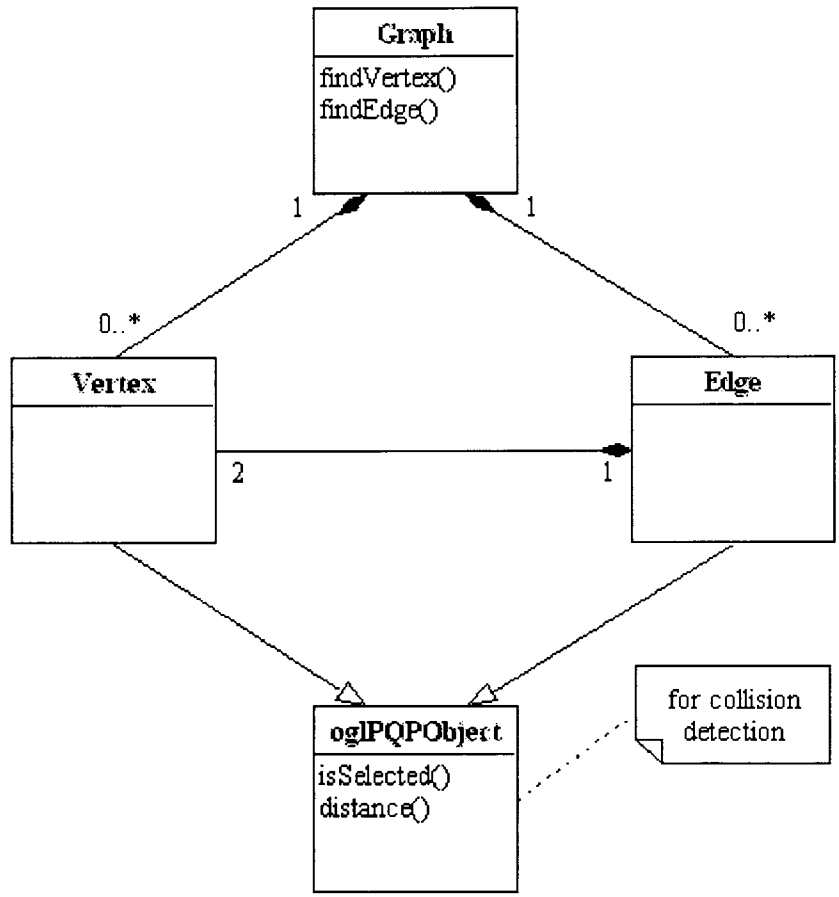

Figure 5.3 `graph` class diagram.

5.3 Application

The application family of classes deals with the interaction and system parts of the system. This section will discuss the main parts of the application category of classes displayed in figure 4.2. First we will discuss the main class in this class category; the `graphVizApp` class. Then we will describe the `vjNavigator` class and then we will explain how Tweek is incorporated into the `graphVizApp` class to facilitate communication with the Java GUI.

5.3.1 The Main Application Class

The `graphVizApp` class is inherited from the VR Juggler `GlApp` class, see figure 5.4. This inheritance is vital and ensures that the `graphVizApp` interacts correctly with the VR Juggler kernel. Among the responsibilities of this class it to query the VR Juggler kernel for up to date information from the input devices, in this case information about the wand's position and button state and the users head position. After collecting this information the class passes the information, stored in matrices for the positions and state variables for the buttons, to the other classes that needs this information. This passing of information is done in the `preFrame()` function, because this is right before the kernel has gathered this information from the input manager (figure 4.1).

The `graphVizClass` also uses the `vjAnimator` class, this class implements animation from one point to another point in the virtual environment. When the user selects a vertex from the vertex list in the GUI the `graphVizApp` creates an instance of the `vjAnimator` class. This animator class now replaces the navigation class and updates the navigation matrix. So when we start an animation we suspend the user input, this way when an animation is taking place the user cannot interact with the environment.

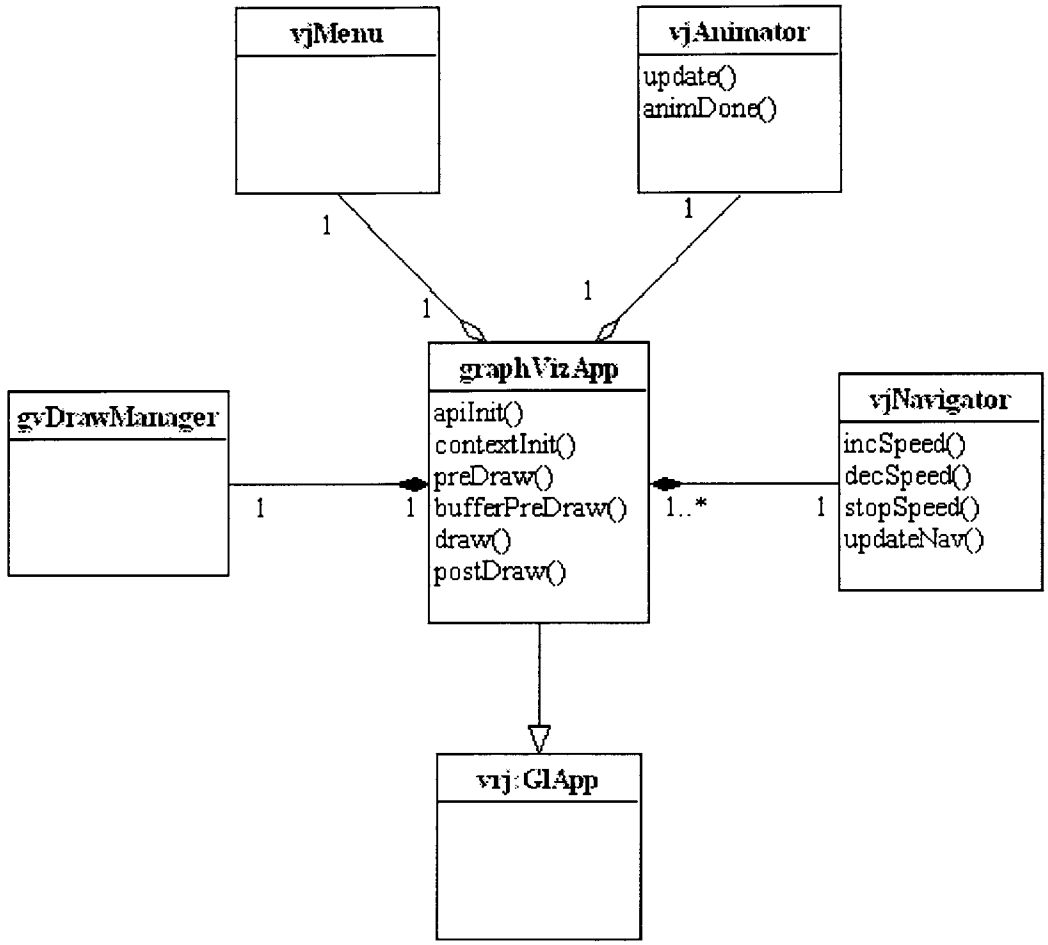

Figure 5.4 `graphVizApp` class diagram

5.3.2 Tweek

As explained earlier Tweek is built upon CORBA, which allows us to communicate between our virtual environment and our Java GUI. In order for the communicating to work, we have created an interface to our application and an interface to the Java GUI using the Interface Definition Language (IDL). The IDL generates stubs that are included in the application at compile time, hence provides the GUI to access and manipulate different

variables in our virtual environment. The IDL interface provide the Java GUI access to the `graphVizApp` class for accessing information about the state of the navigation and other application level information. Through the `graphVizApp` class the Java GUI has indirect access to the `gvDrawManager` and `gvSharedData` classes, these classes provide information about the state of the current graph and the current view. Information flows both from the application to the GUI and from the GUI to the application, thereby allowing the user to change the state of the application as well as observing the current state of the application through the GUI.

5.3.3 Navigation

The navigation in the virtual environment is handled by the `vjNavigation` class. This class main functionality is to manipulate the current navigation matrix to reflect the wand's orientation and the state of the wand's buttons. For each call to `graphVizApp`'s `preFrame()` function the navigation class is responsible for updating the navigation matrix based upon information stored in the wand matrix and the previous navigation matrix. This information is retrieved from the `updateAllData()` call to the `vjInputManager` (figure 4.1). The abstract class `vjNavigation` provides the virtual class `updateNav()` that each sub class needs to implement in order to create their distinct flavor of navigation. This work has implemented a navigation scheme with no constraints, thus allowing the user to freely explore the environment. This method simply moves the user in the direction that the wand is pointing at a speed that is determined by pressing certain buttons on the wand. The `vjNavigation` class also provides the two functions `getSpeed()` and `getPos()`

these functions provide the GUI with information about the state of the current navigation that can be displayed in the GUI.

Chapter 6: Case Study

This section presents the case study that was conducted in order to evaluate our application. First we discuss the data set used for this case study and present the problems faced with while trying to visualize this data set. Then we discuss the different views and methods covered in the previous chapters and the experiences with applying these views and methods to the data set.

6.1 The Data Set

The data set used in the case study is taken from the field of biological chemistry, it represent a metabolic pathway occurring during the metabolism of a plant. A metabolic pathway closely resembles a graph, where vertices depict the different substances while the edges represent the relationships between these substances. Because of this similarity, work has been done to utilize graph drawing methods to visualize metabolic pathways [Karp94]. This work goes a step further and utilizes virtual environments for visualizing graphs. And in our case study, metabolic pathways stored as graphs.

In particular, the metabolic pathway data used in this work is taken from the Arabidopsis plant. It shows part of the metabolism for Acetyl CoA, a metabolite that is a critical part of the energy formation process in organisms. Figure 6.1 shows us the data set as displayed in our immersive virtual environment.

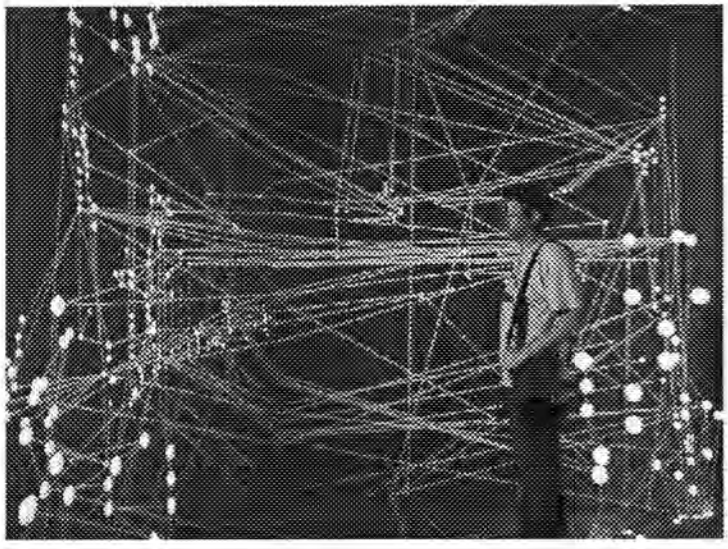

Figure 6.1 User immersed in the data set.

6.2 Problems

The data set in our case study is quite large; it has 403 vertices and 540 edges. Even though the data set is not enormous, the problems of information overload and spatial awareness must be addressed. Another problem with a data set of this size is the ability to interact easily with individual elements of the data set. When the data sets become bigger, the amount of information needed to be displayed also increases, this could cause the information space to become cluttered and interaction with individual elements form the data set becomes harder. Therefore a successful and useful virtual environment needs to address these issues and provide efficient methods for dealing with these problems.

6.3 Fisheye View

The implementation of the fisheye view works well in our virtual environment. This view method addresses the problem of information overload by only displaying the closest

vertices with a great level of detail. From figure 6.2 we observe that the closest vertices are displayed with their edges visible, the closest vertex is being displayed in another color and with its substance identification displayed. The vertices are that are further away are displayed in successively smaller sizes without any edges visible.

The fisheye view also addresses the problem with cluttering and interacting with individual elements of the data set. Since only the closest vertices are drawn with much detail, the user's view will not be cluttered by edges from vertices far away that might pass through the view. The user is also able to quickly determine which vertices are closest and query them for further information.

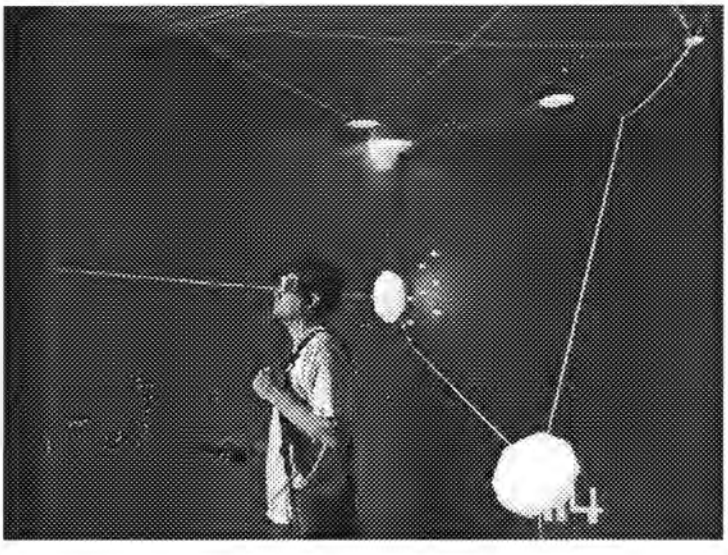

Figure 6.2 Fisheye view.

6.4 Layered View

The layered view is also a helpful view method. By initially only displaying the vertices with a calculated weight bigger than a specified threshold, we are able to reduce the

overall information displayed at one time. From figure 6.3 we have a case where the user has selected a vertex and the incident vertices are displayed in a cone shape directly below the selected vertex. This view addresses the problem of information overload by initially only displaying a subset of the actual graph. Since the user is able to interactively increase the number of vertices shown by selecting them, the user has the ability to work with a subset of the graph that is manageable. However since this view only shows a subset of the original graph the user can loose the overall structure of the graph. In the case study, initially only the vertices that represent substances that have many interactions (edges) are shown in this view. This gives us a good idea of the shape of the pathway, but it does not give us quantitative information about the specific substances. The user has to find a particular substance and query it by selecting it to determine all the substances that have interactions with the selected substance.

The layered view also addresses the problem of cluttering and interaction with the individual elements in the data set. Since the user is able to determine the size of the initial subset of the graph to be displayed, the user has the freedom to create a manageable and clutter-free environment to facilitate ease of exploration.

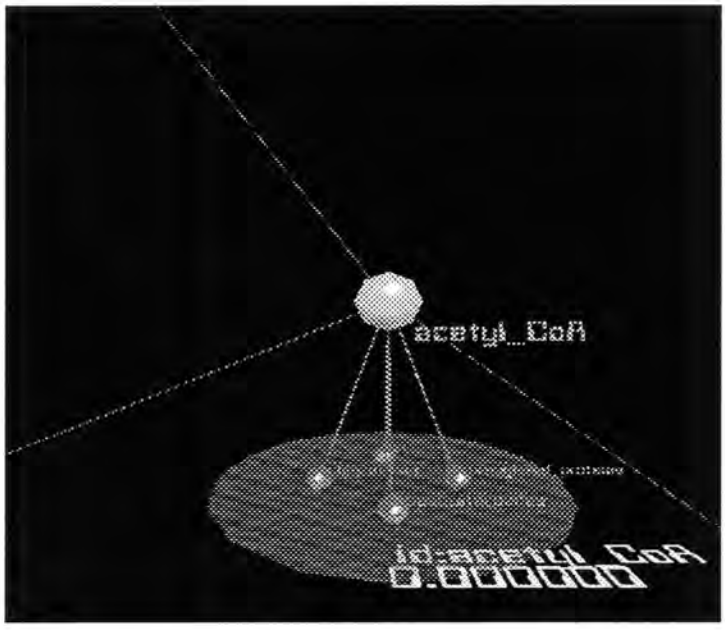

Figure 6.3 Layered view.

6.5 Lens View

The lens view is not as helpful as the layered and fisheye views. Because the lens is connected to the wand, it complicates interaction with the graph. However it is a useful tool for allowing the user to examine the overall structure of the data set. Figure 6.4 shows us the angled lens, the vertices and edges that fall within the area covered by the lens is displayed with more detail than the vertices and edges that are without the area.

The lens is useful when used together with the other views. By exploring the data set using the lens, the user is able to be positioned at a particular spot in the environment and examine the data set in any direction by using the wand. The user can then observe an interesting region and then switch to another view to explore that region in greater detail with the enhanced interaction options the other views provide.

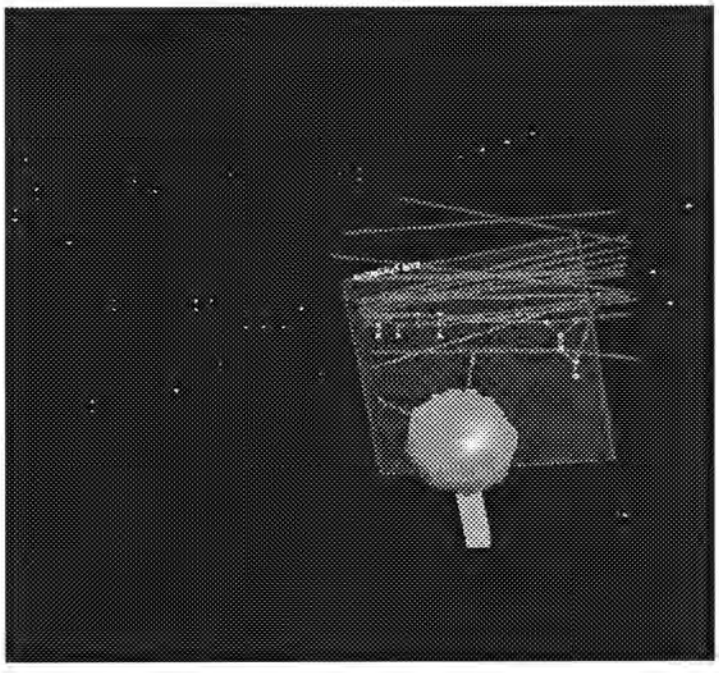

Figure 6.4 Lens view.

6.6 World in Miniature

The WIM is a useful addition to the other views in order to enhance the user spatial awareness. By consulting the WIM, the user is able to orientate in the virtual environment. Even though our data set is not very large, having the WIM display the user's position is very useful. By having the WIM attached to the wands position and rotation, it allows the user to rotate and reposition the wand in order to get the best possible view of the WIM. From the figure 6.5 we see the WIM displayed within the normal view of the graph, the WIM is also rotated and translated in order to get a better overview of the user's position. The user's position in the virtual environment is represented by the white crosshair drawn in the WIM.

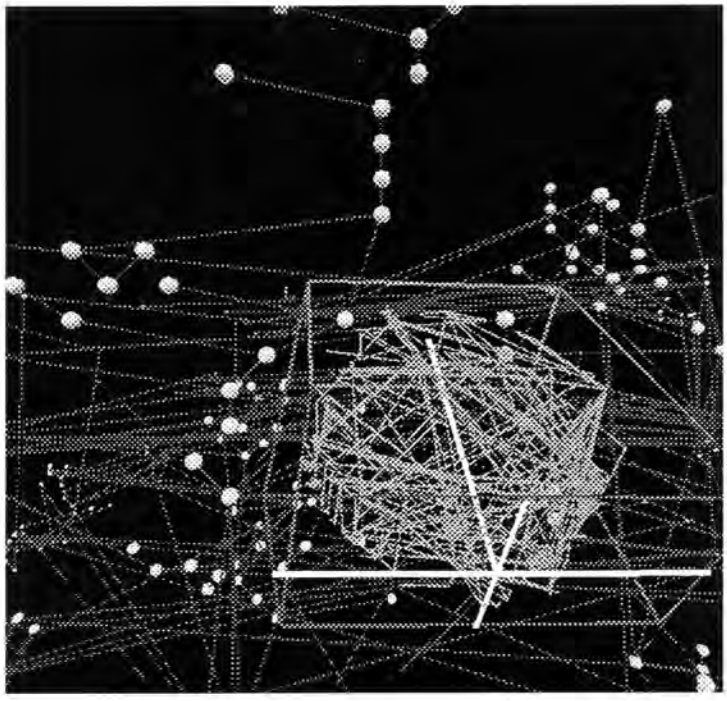

Figure 6.5 World in Miniature.

Chapter 7: Conclusions

In designing and implementation the virtual environment for visualization of graphs, we have developed a helpful environment for users to view and explore different graphs. The case study also strengthens the hypothesis that virtual environments can provide useful insights into the field of graph visualization.

By immersing the user in the actual graph, virtual environments have a clear advantage over traditional computer systems, since this represent a more intuitive interaction method than traditional computer systems. This immersion enables the user to get a better feel of the overall structure of the graph, hence understand the underlying data better.

The use of two different input devices increased the usefulness of our application. By displaying textual information about the vertices, edges and the graph on the tablet computer we reduced the information needed to be displayed in the actual virtual environment.

The views and methods implemented and used in the virtual environment are good tools for aiding visualization. The layered and fisheye views successfully addressed the problem of information overload and cluttered view space by reducing the amount of information displayed at once. The lens view was not as useful as the two other viewing methods, because of its interaction shortcomings. However the lens view is a useful addition, used together with the other views.

The Worlds in Miniature method helped address the issue of spatial awareness and used in conjunction with the views it gives the user the ability to always know the position in the graph. The WIM also displays the overall structure of the graph, which helps the user navigate the graph and improves the spatial awareness.

Chapter 8: Future work

During the course of this work there were additional functionalities and other approaches that were discussed and considered. Ultimately many were dropped and deemed inappropriate within the scope of this work. This section lists and discusses some of the more interesting features that were discussed.

Sound could provide us with a useful additional cue. By adding auditory cues to our application we could possibly enhance the user's experience. According to [Stuart01] sound is a valuable cue and when used right could remove some of the dependency on visual cues. Thereby reduce the load on the visual senses, creating a more complete experience for the user. During the development of this work VR Juggler did not have any clear cut sound capabilities, which explains the omission of sound from our application.

It could also be useful to add functionality to facilitate the creation of a guided tour through any graph. By specifying a particular path through the graph, specify the vertices to visit and provide narrative associated with each vertex. It would allow teachers to tailor make a learning experience for students. In our case study on metabolic pathways, biology students could by guided down the metabolic map starting with the initial substances then follow the reactions as they occur until the final substance is synthesized.

It would also be useful to test our visualization methods on other data sets, not just metabolic pathways. Other data sets might have different properties and require different approaches to visualization.

Bibliography

[Apache01] Xerces C++ Parser. http://xml.apache.org/xerces-c. Verified June 2001.

[Ascension01] MotionStar Wireless. http://www.ascension-tech.com/products/mswireless. Verified December 2001.

[Bier93] E. A. Bier, M. C. Stone, K. Pier, W. Buxton and T. D. DeRose. "Toolglass and Magic Lenses: The See-Through Interface". Proceedings of Siggraph '93. Computer Graphics Annual Conference Series, ACM, pp 73-80. 1993.

[Bierbaum00] A. Bierbaum. "VR Juggler: A Virtual Platform for Virtual Reality Application Development." MS Thesis, Iowa State University. 2000.

[Bowman95] D. A. Bowman, L. F. Hodges. "User Interface Constraints for Immersive Virtual Environment Applications". Graphics, Visualization, and Usability Center Technical Report GIT-GVU-95-26, 1995.

[Bowman98] D. A. Bowman. L. F Hodges and J. Bolter. "The Virtual Venue: User-Computer Interaction in Information-Rich Virtual Environments". Graphics, Visualization, and Usability Center Technical Report GIT-GVU-96-22. 1998.

[Compaq02] Compaq iPAQ. http://athome.compaq.com/showroom/static/iPaq/3835.asp. Verified August 2002.

[CORBA02] CORBA. http://www.corba.org. Verified May 2002.

[Cruz93] C. Cruz-Neira. "Virtual Reality Overview". Siggraph 1993 Course Notes. 1993.

[Cruz95] C. Cruz-Neira. "Projection-based Virtual Reality: The CAVE and its Applications to Computational Science". Ph.D. Dissertation. University of Illinois at Chicago. 1995.

[di Battista94] G. di Battista, P. Eades, R. Tamassia, and I.G. Tollis, "Algorithms for drawing graphs: an annotated bibliography". Computational Geometry: Theory and Applications. pp 235-282. 1994.

[di Battista99] G. di Battista, P. Eades, R. Tamassia, and I.G. Tollis. Graph Drawing: Algorithms for the Visualization of Graphs. Prentice Hall. 1999.

[Fairchild88] K. M. Fairchild, S. E. Poltrock and G. W. Furnas. "SemNet: Three-dimensional graphic representations of large knowledge bases". Guidon, R. ed. Cognitive Science and its Applications for Human-Computer Interaction, Lawrence Erlbaum Associates, pp 201-233. 1988.

[Furnas86] G. W. Furnas. "Generalized Fisheye Views". ACM CHI'86, pp 16-23. 1986.

[Herman00] I.Herman, G.Melancon and M.S.Marshall. "Graph Visualization and Navigation in Information Visualization: a Survey". IEEE Transactions on Visualization and Computer Graphics. 2000.

[Himsolt97] M. Himsolt. GML: A portable Graph File Format. http://www.infosun.fmi.uni-passau.de/Graphlet/GML/gml-tr.html. Verified May 2002.

[Interlink01] ACM SIGCHI Curricula for Human-Computer Interaction. http://www.interlinkelec.com. Verified December 2001.

[Intermec02] Intermec Pen Computer, Model 6642. http://epsfiles.intermec.com/eps_files/eps_man/6642tr.pdf. Verified May 2002.

[Juggler02] VR Juggler – Open Source Virtual Reality Tools. http://www.vrjuggler.org. Verified December 2001.

[Jungnickel99] D. Jungnickel, Graphs, Networks and Algorithms, Springer Verlag, 1999.

[Karp94] P. Karp, S. Paley. "Automated drawing of metabolic pathways". Proceedings of 3rd International Conference on Bioinformatics and Genome Research. 1994.

[Lin98] M. Lin, S. Gottschalk. "Collision Detection between Geometric Models: A Survey". Proceedings of IMA Conference on Mathematics of Surfaces 1998.

[Lindeman99] Lindeman, R., Sibert, J., Hahn, J., "Towards Usable VR: An Empirical Study of User Interfaces for Immersive Virtual Environments," Proceedings of the SIGCHI '99, pp. 64-71. 1999.

[Napari00] H.Napari, T. Takala. "Magic Lights and 3D Magic Lenses- Projective Interaction Tools for Virtual Environments". Immersive Projection Technology 2000.

[Noik93] E. G. Noik. "Layout-independent Fisheye Views of Nested Graphs". Proceedings of the 1993 IEEE Symposium on Visual Languages, pp 336-341. 1993.

[Norman90] D. Norman. The Design of Everyday Things. Doubleday. New York. New York. 1990.

[OpenGL01] OpenGL. http://www.sgi.com/software/opengl/. Verified December 2001.

[Orimo99] E. Orimo, H. Koike. "ZASH. A Browsing System for Multi-Dimensional Data". IEEE Symposium on Visual Languages. 1999.

[PQP01] PQP – A Proximity Query Package. http://www.cs.unc.edu/~geom/SSV. Verified June 2001.

[Risch96] J. Risch, R. May, J. Thomas and S. Dowson. "Interactive Information Visualization for Exploratory Intelligence Data Analysis". VRAIS'96

[Sarkar92] M. Sarkar, M.H. Brown. "Graphical Fisheye Views of Graphs". ACM CHI'92, pp 83-91. 1992

[Sheelagh95] M. Sheelagh, T. Carpendale, D.J. Cowperthwait and F.D. Fracchia. "Distortion Viewing Techniques for 3-Dimensional Data". IEEE Symposium on Information Visualization, pp. 46--53. 1995.

[Stoakley95] R. Stoakley, M. Conway and R. Pausch. "Virtual reality on a WIM: interactive worlds in miniature". ACM CHI'95. 1995.

[Stuart01] R. Stuart. The Design of Virtual Environments. Barricade Books. New Jersey. 2001.

[Tamassia97] R. Tamassia. "Graph Drawing". Handbook of Discrete and Computational Geometry. Edited by J. E. Goldman and J. O'Rourke. CRC Press, pp. 815-832. 1997.

[W301] Extensible Markup Language (XML). http://www.w3.org/XML/. Verified June 2001.