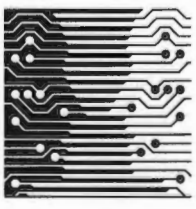
INFORMATION VISUALISATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Wayne Paverd
October 1995

Supervised by
H.A. Goosen



Abstract

Information visualisation uses interactive three-dimensional (3D) graphics to create an immersive environment for the exploration of large amounts of data. Unlike scientific visualisation, where the underlying physical process usually takes place in 3D space, information visualisation deals with purely abstract data. Because abstract data often lacks an intuitive visual representation, selecting an appropriate representation of the data becomes a challenge.

As a result, the *creation* of information visualisation involves as much exploration and investigation as the eventual exploration of that data itself. Unless the user of the data is also the creator of the visualisations, the turnaround time can therefore become prohibitive.

In our experience, existing visualisation applications often lack the flexibility required to easily create information visualisations. These solutions do not provide sufficiently flexible and powerful means of both visually representing the data, and specifying user-interface interactions with the underlying database.

This thesis describes a library of classes that allows the user to easily implement visualisation primitives, with their accompanying interactions. These classes are not individual visualisations but can be combined to form more complex visualisations.

Classes for creating various primitive visual representations have been created. In addition to this, a number of auxillary classes have been created that provide the user with the ability to swap between visualisations, scale whole scenes, and use automatic level of detail control.

The classes all have built-in interaction methods which allow the user to easily incorporate the forms of interaction that we found the most useful, for example the ability to select a data item and thereby obtain more information about it, or the ability to allow the user to change the position of certain data items.

To demonstrate the effectiveness of the classes we implemented and evaluated a number of example systems. We found that the result of using the classes was a decrease in development time as well as enabling people with little, or no visualisation experience to create information visualisations.

Acknowledgments

I would like to thank my supervisor, Henk Goosen, for introducing me to the field of visualization and writing most of iIsh,

The Foundation for Research and Development's Manufacturing Core Program for their financial support,

Prof. A.P. Fairall for providing feedback and the data for the universe visualisation,

Dave Watson for his algorithm for generating convex hulls, used in the VoPoints class,

My friends and collegues for their help and the diversions which made life more interesting.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Organisations and individuals increasingly have access to large amounts of on-line data. Three technological trends are contributing to this situation. First, the increase in the speed to cost ratio of computers makes it easier to generate large amounts of data more quickly. Second, the increase in mass storage capacity enables people to easily store large data sets. Finally, increased connectivity and network bandwidth allows people to have access to more stores of information.

These data sets may contain potentially useful information if they can be analysed and understood efficiently. Until recently, analysis of these data sets relied on traditional approaches to data analysis. These traditional approaches include statistical reduction of the data to more manageable proportions, the use of algorithms to search and analyse the data to determine where the anomalies exist, and the use of graphical representations of the data.

However, each of these methods have limitations. The statistical reduction of the data results in some information being lost to decrease the size of the data set. The use of search algorithms is only applicable where a person searches for recognised anomalies. When data is being analysed for the first time, or the recognition of these anomalies can not easily be defined in mathematical terms, this method cannot be easily applied.

It has long been recognised that data can often be more readily understood by studying a visual representation of the data. William Playfair is recognised as one of the pioneers in using graphical representation [34]. Subsequent research has been done to improve the effectiveness of visual data analysis [34].

Displaying data in a visual forms often aids comprehension of complex and/or large data sets. There are a number of psychological and physiological reasons for this. First, pictures and images play an important role in our every day communication [28]. Second, the human visual system has a very high bandwidth, and a single image is able to convey a large amount of information [20]. Third, the human visual system is adept at being able to track and recognise patterns and shapes [22, 16].

These factors make it possible for humans to easily recognise patterns, anomalies and trends, provided the data is presented effectively. The problem then becomes one of determining how best

to display the data in such a way as to display as much information as possible.

Traditional visual representations were either static, or involved cumbersome manual methods of manipulating the data as outlined by Bertin [8]. With the advent of the computer and the ability to quickly and easily display not only two-dimensional but also three dimensional images, the scientific community was quick to recognise the way computers could be employed to generate visual images of complex data sets. The visual images no longer were limited to being static, but could include interaction. This technique of data analysis was initially used by the scientific community and became known as scientific visualisation [26].

We start by explaining what visualisation is, and how it differs from presentation graphics. Next we distinguish between scientific visualisation and information visualisation. We then examine the problems with information visualisation, existing solutions to theses problems, and then briefly describe our approach to creating information visualisations.

1.1 What is Visualisation?

The term visualisation refers to a number of different methods of displaying large data sets in such a way as to allow the user to 1) discover trends, patterns, structure and/or anomalies [6, 19] and 2) verify the importance of these discoveries. It would not be easy to make these discoveries if one only had access to the raw data.

Visualisation provides a graphical representation of data sets that can be animated to allow the user to explore the data sets. The animation can take the form of manipulating and rotating the whole data sets, predefined animations of processes or computational steering [6]. Computational steering allows the user to change input parameters and directly see how they affect the visualisation.

1.2 Visualisation vs. Presentation Graphics

The difference between visualisation and presentation graphics is in terms of the functionality of each, where presentation graphics is used as a form of communication, visualisation is used for exploration.

The purpose of presentation graphics is to present information to an audience in such a way as to emphasise or clarify some aspect of the data. It is therefore assumed that the user of the presentation graphics understands the data that is being discussed.

In contrast to this, the purpose of visualisation is to further understanding of the data by allowing the user to discover anomalies and patterns in the data and then provide insight into why these anomalies and patterns occur.

To do this, any visualisation should exhibit the following properties:

Exploratory: The user should be able to explore the data [19]. This refers to an underlying functional difference. Where presentation graphics allows the user to emphasize interesting characteristics of the data, visualisation permits the user to discover these interesting characteristics and determine the factors that caused them.

Interactive: Visualisation should allow the user to manipulate the data [26]. This manipulation may take different forms, it can involve rotating the visualisation, selecting and obtaining more information about a data point or moving a data point and then being able to obtain feedback on the impact this change will have on the rest of the data set. The important factor here is one of speed, the user must feel that they are interacting with the data. Only by interacting with the data directly will the chances that the researcher will gain some insight be increased [26].

A visualisation should also support the stages of observation as outlined by Jack Estes (UCSB). He defines observation as consisting of three recursive phases, 1) *Detection*, 2) *Identification* and 3) *Measurement and analysis* [24].

Any good visualisation system must support these phases of observation. *Detection* and *Identification* involve being able to identify patterns, trends or anomalies the are present in the data. *Measurement and analysis* involve more than just recognising patterns, it at this phase that quantitative information becomes necessary.

There is an inherent conflict between type of information that needs to be displayed for the *Detection-Identification* and the *Measurement and analysis* stages of the observation process in that the latter requires fine detail, or text, which becomes noise when viewed from a distance. This noise then hinders the *Detection-Identification* stages.

Our proposed solution uses the fact that when the viewer is sufficiently far away from the visualisation, the fine detail becomes noise and should therefore be removed. This was achieved by creating an automatic level of detail control tool that would allow the user to control the amount of detail displayed depending on the distance between the viewer and the visualisation.

1.3 Information vs Scientific Visualisation

The field of scientific visualisation traditionally deals with visualizing the results of simulations or experimentation. Computational fluid dynamics and finite element analysis are two applications in which much of this research has taken place, and for which there exist systems for doing visualisation [28]. Existing tools like AVS, Khoros, IRIS Explorer, and Data Explorer focus on scientific visualisation as opposed to information visualisation, although they can certainly also be used for information visualisation.

Information visualisation involves the visualisation of abstract data. Abstract data differs from other forms of data by not having any recognised underlying spatial model, or conversely, we cannot represent the data graphically in such a way as to make the underlying process easily identifiable.

If, for example, one wants to visualise stock market data for a certain period, it is not obvious how to display this information in such a way as to give the viewer as much insight as possible into the data. The problem is that there is no clearly defined underlying physical process.

Scientific visualisation, on the other hand, has a clearly defined underlying process that has been modeled and is then visualised. For example, for the data from a simulation of blood flow through an artificial heart, the underlying process that has been modeled is the path blood cells take when flowing through the heart, and the visualisation would show the path that each of the cells took when flowing through the heart. There is therefore a clear underlying spatial model of how the process should be visualised.

This highlights the difficulty of information visualisation: the process of creating a visualisation is as much of an exploratory process as the actual exploration and interrogation of the data. This problem has been recognised by other researchers and one approach, which has been adopted by the creators of Glyphmaker [28], is to provide the user with the ability to create their own visualisations based on a number of basic components which can be combined to create a visualisation.

Different methods of visualising data can be used on the same data set with varying levels of success; different visualisation techniques can also be used to highlight different characteristics of the data. Therefore the *method* of visualisation plays a key role in the understanding and interpreting of any data set.

1.4 The Challenge of Creating Visualisations

One of the factors that prevents more people using visualisations is that it requires an intimate knowledge of graphics and interaction techniques. This often means that there is a distinction between the *user* of the visualisation and the *creator* of the visualisation.

The process of creating a new visualisation can be divided into the following steps:

Conceptual Design: Given the data, possible techniques of turning that data into visual representations are investigated and the techniques that appear to offer the best possible solutions are then carried forward to the next stage.

Implementation: The visualisation and the accompanying interactions are implemented.

Evaluation: The user uses the visualisation to investigate the data. At this point the strengths and weaknesses of the visualisation technique become apparent and modifications may be necessary. The modifications may require changes in either the implementation or in the underlying concept design. This iterative process continues until the user determines that the visualisation is satisfactory.

As can be seen from the above steps, the creation process is an iterative process [7], and when the user is not also the creator of the visualisation the process can be very time consuming.

Therefore, the ideal situation is one where the owner of the data set also creates and uses the visualisation. It is therefore necessary to find some way to empower the user.

1.5 Existing Visualisation Solutions

Systems that try to combine the user and creator, that have come to the foreground in recent years, are ones that either use visual programming techniques to allow the user to create visualisations, or high level libraries that encapsulate many of the complex 3D drawing primitives.

The visual programming systems, the most prominent are AVS, IRIS Explorer, Data Explorer and Khoros are all based upon the data flow model, which defines the way the data flows through the system. These systems are each discussed in more detail in the following chapters.

Inventor, on the other hand, is a C++ library that encapsulates many of the basic graphics primitives in a suite of classes. Inventor was designed with the aim of simplifying the creation and manipulation of 3D scenes and as such does not cater specifically for visualisation.

The problem we found with the existing visualisation solutions was that they did not offer a satisfactory combination of flexibility, performance and simplicity.

In the following chapter these systems are examined in more detail and evaluated with respect to their effectiveness in creating fully interactive visualisations of abstract data.

1.6 A Modular Approach

It was the aim of this investigation to simplify the process of creating visualisations by offering the user a high level set of tools. The objective was to to sacrifice as little power and flexibility of the underlying system as possible while still maintaining a relatively simple technique for creating the visualisations.

To this end a suite of classes has been developed that can be used as building blocks for creating information visualisations. The goal in creating the classes was as follows:

Simplify the creation: Simplify the process of creation and interaction of both simple and complex visualisations.

Maintain flexibility: The user must be provided with a set of tools that provide enough flexibility to create a variety of visualisations.

Sacrifice as little performance as possible: The additional ease of creating visualisations must not have a negative effect on the rendering speed.

Effectively display fine detail: The display of fine detail when one wishes to see an overall view often amounts to no more than noise that decreases the effectiveness of the visualisation.

Our approach was to design a number of classes that were based upon Inventor and encapsulate the Inventor code for creating and interacting with a 3D scene.

The classes provide a standard interface that allows the user to easily substitute one class for another, thus making the modification of visualisations relatively quick. They also provide methods

for determining exactly which data point was selected when the user interacts directly with the visualisation.

We found that the classes we created could be easily combined in number of ways to create visualisations of varying complexity.

To simplify the creation process further we created iIsh which is based on the scripting language Tcl. The Tcl interpreter was extended to include commands that would allow the user to create and manipulate the classes and other 3D scenes. The user could then write scripts which iIsh would interpret and then use to create visualisations. This eliminated the need for the user to know C++.

The display of text, or any other fine detail, only at a point where it becomes relevant prevents it becoming noise when one views the whole visualisation. We identified a need for a tool that would only display the text once it became relevant. The class that was created for this purpose was the SoDetailLevel class. This class is discussed in more detail in Section 4.15.

1.7 Thesis Overview

In the following chapter, we comment on related research that has taken place in the field of visualisation.

In Chapter 3 we discuss our approach to simplifying the process of creating visualisation.

Chapter 4 explains in detail each of the different classes that we have created as well as example implementations where they were used. It also describes the automatic level of detail node, SoDetailLevel, that was created. Automatic level of detail control allows certain aspects of the visualisation to only be displayed once they become relevant, through this method it is possible to allow the user to define which parts of the visualisation are relatively less important and should not be displayed when they are far from the viewer.

In evaluating the classes a number of systems were implemented and in Chapter 5 we evaluate each of these visualisations in terms of their effectiveness and the ease with which they were created.

Finally, we have documented the conclusions that can be drawn from our experiençes using each of the classes. We look at each class with respect to its strengths and weaknesses.

Chapter 2

Related Work

Previous research has been done in the field of information visualisation and in this chapter we look at a number of more well known systems that have been developed based upon this research.

Not all of these systems were developed specifically for information visualisation but were orginally developed for scientific visualisation and were then extended or modified to allow them to be used for information visualisation.

The visualisation systems we have evaluated can be classified into two main groups on the basis of how the user programmes the system; namely *visual programming systems* and *scene description systems.*

Visual Programming Systems

The dataflow model allows the user to create a visual representation of how the data flows from the input through a series of transformations to produce some form of output. The systems described produce their output in the form of a visualisation, but this does not always have to be the case.

The model lends itself to being implemented as a visual programming language where the user defines a *program* by creating a network of modules. The connections between the modules represent the channels for the data flow, and the nodes the transformations that are performed on the data.

The systems that fall into this category are: *IRIS Explorer*, *Data Explorer*, *Khoros* and *Application Visualisation System.*

Scene Description Systems

GlyphMaker use a graphical interface to describe a scene, it allows the user to link certain properties of models in the scene to data fields in the input stream. For instance the position of the model can be linked to a certain data field, then as the data is read, the model is animated. This allows the user to easily create animation sequences as the system reads the data set.

Inventor and *Obliq-3D* both use a scene graph to describe a scene. The main difference between the two is the manner in which properties and attributes are represented within the scene graph. Scene graphs are discussed in more detail in Section 2.6.1.

Inventor uses separate nodes in the scene graph to define properties and attributes which then applies to a well defined subset of the all the nodes, while Obliq-3D stores the properties and attributes for each node within each node. Functionally the two methods are equivalent in so far as the same results can be achieved with both.

Another graphics library that was examined was *BRender*, a PC based rendering library that has characteristics in common with Obliq-3D in the way it describes the scene graph. The scene graph consists out of a number of actors, each actor stores all the attributes and properties that determine how that actor is rendered.

2.1 IRIS Explorer

IRIS Explorer uses a visual programming language to implement the dataflow model, a visual program being defined by creating a network of modules. A module is, in essence, a procedure or function that accepts some data, processes it, and then produces some form of output. The network is then responsible for taking the output of one module and providing it as the input to the next module, or modules.

Modules tend to be fairly computationally complex and therefore it is possible to create visualisation with relatively simple networks. If, however, networks do become too complex then a group of modules can be grouped into a single module [32]. This does not create a new module, but merely hides the connections between the group of modules and presents the user with a single interface for all the modules in the group.

The modules can be roughly divided into four groups [35, 33]:

input: These modules read the data from a file, a pipe or another application which is producing the data.

filters: These modules perform some form of filtering or modification of the data.

transform: These modules take the data and transform it into some geometric representation.

output: The output module is most often the render module.

The Explorer visualisation system consists out of 3 main sections:

Map Editor The Map Editor is used to modify and create maps or networks.

DataScribe The DataScribe is a conversion utility that allows data to be converted between Explorer's format and a number of other formats.

Builder The Module Builder allows the user to create custom modules.

A network is created by selecting a number of modules from a collection of modules and placing them on the work area using a simple drag and drop interface. The user then defines the data paths by either creating or destroying connections between the modules.

Each module can have a number of controls which allow the user to modify the parameters that control the module. These controls form the user interface of the visual program; through manipulating these controls the user can interact with the visualisation.

2.1.1 Comparison to our approach

The method IRIS Explorer uses is suitable for many forms of visualisation. However, it does not provide any easy facilities for directly interacting with the visualisation, all interactions take place through a number of modules. The results of these interactions then fire off modules which cause the scene to be rendered.

This form of interaction allows the user to modify and fine tune the inputs to a visualisation or to see how a number of factors influence the visualisation. However, once presented with a visualisation that exhibits interesting phenomena, patterns or trends it is difficult to select a single data item and obtain more detail information about that data item. No facilities for doing direct manipulation are provided and using external controls to try to pinpoint which data item is responsible for these anomalies is far less intuitive than directly querying the visualisation.

2.2 Application Visualisation System

AVS is a visualisation system that is widely available on many platforms from work stations to supercomputers. It was created to provide a relatively simple method of allowing non-experts to make use of complex 3D graphics [2]. AVS is based on the dataflow model with some extensions that provide the ability to make stand alone applications.

2.2.1 Components of AVS

AVS consists out of five interactive parts, namely the Geometry Viewer, the Image Viewer, the Graph Viewer, the Data Viewer, and the Network Editor [6].

The geometry viewer is used for rendering views of 3D geometric primitives; these primitive include points, lines, polygons, spheres, etc. [5]

The primitives are used in conjunction with lights, cameras and other property information to create 3D scenes which can be rendered into one or more windows.

The image viewer is designed for use in displaying 2D images for image processing applications; it provides ability to display and arrange images, as well as a limited set of image processing functions.

The graph viewer is a used for viewing 2D graphs as opposed to 2D images, while the data viewer is used to create visualisation tutorials. Finally, the network editor is equivalent to IRIS Explorer's map editor. It is used to modify and create networks of nodes that define the dataflow model.

2.2.2 Graphical User Interface

AVS uses X11 as the basis for all its widgets. The widgets that AVS uses include buttons, sliders, dials, etc. This is similar to IRIS Explorer in the type of interaction widgets that are provided. These widgets can then be used to scale, rotate, or otherwise modify objects that appear in the viewers.

Each viewer has an interaction window that displays the option associated with that viewer; therefore, to change an option the user can use a simple point and click approach. The viewers also use interactive picking, i.e. selection of objects in the viewport. This eliminates the need for moving the mouse pointer out of the viewport, selecting an object by using some widget or list, and then moving back to the viewport. These features make the use of the viewers very easy [6].

AVS allows the user to change the layout of the graphical interface of the visual program. It also makes possible the grouping of a number of interface widgets, usually the most important ones, on a page and the concealment of others, thereby simplifying the overall user interface. A similar feature is also present in IRIS Explorer where a number of modules can be grouped into a single module.

2.2.3 Command Language

AVS includes a command language, which can control all aspects of AVS. It also includes a journal and scripting facility that allows the user interaction to be recorded and played back at any time.

In addition to this modules and user applications can send commands to the command language interpreter for the following tasks: build and modify visual programs, execute these visual programs and modify module parameter values, change rendering properties, modify user interface layouts, and other operations normally performed directly by user interaction within AVS [6].

The command language does have some limitations, two of which are highlighted by Bethel [9]. In developing a virtual reality interaction module it was impossible to query the spatial and orientation information about the currently selected camera. Another problem was that no information could be obtained about the parent of a given object, which made difficult the construction a hierarchy of transformation that was more than two levels deep.

2.2.4 Entering data into the network

AVS has five basic data types, they are:

Fields: These are equivalent to IRIS Explorer's lattice type. They can be used to represent multidimensional data, which have regular topologies. In other words a 1D field represents a line, while a 3D field represents a cubic volume.

Unstructured Cell Data: This is for volume data which is not structured enough to be used with a field, finite element data is a prime example.

Geometry: This is similar to the geometry type of IRIS Explorer; it is used to describe 3D geometric objects.

Molecular Data: As its name describes it is geared specifically towards the needs of the chemistry community.

User Defined: This is used to describe data that does not fit into any of the above types, it is comparable to IRIS Explorer's *unknown*.

2.2.5 Comparison to our approach

As AVS is based upon the dataflow model there are naturally many similarities to other dataflow systems such as IRIS Explorer and Khoros. However, AVS does provide a greater ability to interact directly with the data. The Command Language and the ability to interact with the data comes close to giving AVS the functionality that a fully interactive visualisation system should provide.

AVS also focuses on creating a presentation tool where modification of the data input streams can be used to investigate the data, but our focus is in slightly different direction. We believe that the user should be presented with the data and then allowed to start exploring the data through investigating anomalies that should be self evident from the visualisation technique that has been employed.

The methods of rendering provided do not directly address the problems of information visualisation but focus more on problem domains in scientific visualisation. This results in there being a lack of tools that can be easily employed to view multidimensional data (data with a dimension greater than three). It is these types of views that we can create using either one, or a number, of our classes.

IRIS Explorer and other visualisation packages do not all have scripting languages. Although using our classes also does not provide any form of scripting, when they are used in conjunction with iIsh, then an interpreted language, in the form of Tcl, is provided.

2.3 Data Explorer

Data Explorer is, like IRIS Explorer and AVS, an application builder. It is also based on a dataflow model where applications can be written by connecting modules together by means of a network [6].

Unlike AVS and IRIS Explorer, modules do not have a widget interface, by default, but instead use type-ins to change the values of parameters. Widgets can be placed on a separate control panel and connected to the parameters to create a custom user interface.

It is also possible to run an application without loading the visual program editor; this results in only the control panel and the viewport being displayed. This provides a method of creating custom application where the end user need never interact with the visual program editor.

Camera movement within the viewport can be done by use of the mouse, however, scaling, rotating and translation require the use of the corresponding modules. As far as we can determine, Data Explorer does not provide comprehensive facilities for picking and manipulating data directly; all interaction takes place through the modules and the control panel.

2.3.1 Script Language

Data Explorer contains an object oriented interpreted script language. The interpreter can either be accessed using a shell and entering the commands, one by one, or through a script file that contains the lines to be interpreted.

The script language allows the user to produce the same results as the visual language; in fact, when a visual program is saved it is saved as a script.

2.3.2 Data Structures

In contrast to AVS and IRIS Explorer, Data Explorer only uses one type of data representation, namely a *field*. The field is composed of a number of *components*. The more important components are:

- positions, which contain spatial information.
- connections, which contains information about the relationships of the nodes.
- data, which contains the data for each node. This data can be either simple numerical data or, in contrast to other packages, complex data.
- color, which determines the color associated with the data.

This form of data contains all the information needed by the renderer to create visualisation, unlike AVS and IRIS Explorer which require additional information to create the geometric representation.

2.3.3 Comparison to our approach

Our approach does not restrict the user to creating a visualisation based upon any one data structure, but instead provides a number of ways that data can be represented and then allows the user to place the data in a way that gives the most insight to them.

Data Explorer does not provide the user with the ability to interact and query the data in the visualisation viewport, as is easily possible by the use of our classes.

2.4 Khoros

Khoros is a visualisation system that is partially based upon the data flow model and partially on an event driven system. It makes use of the the visual programming language Cantata.

2.4.1 Cantata

Cantata is a visual programming language that consist out of *glyphs* or *operators* that are placed in the *workspace*. These glyphs are then connected to form a network that defines the visual program.

Unlike visual programming languages used in other visualisation systems, the network in Cantata does not only indicate the data flow, but is also used to control the execution. A control connection can be established between two glyphs to transfer synchronisation information. This can be used where the order of execution is not dictated by the data flow, as in the case where parallel paths exist[37].

Cantata includes all the programming constructs of a textual programming languages. This allows users to create loops using counters or conditions, conditional branches and switches, as well as merging and splitting data flows.

Workspaces can be encapsulated to form an application that can be run without displaying the visual programming glyphs[37, 36]. This is comparable to creating a "stand alone" application in AVS and IRIS Explorer.

These encapsulated workspaces can also be used as procedural glyphs. The workspace contains definitions for inputs and outputs and these then become the inputs and outputs for the glyph. In this way it is far easier for the non-expert user to create custom glyphs than it is in AVS, Data Explorer or IRIS Explorer.

2.4.2 Visualisation Tools

While Khoros has been used extensively for visualisation and the teaching of image processing [27] it does have the tools to create 3D visualisations, namely the geometry toolbox. However, Khoros has been more focused on visualisation with respect to image processing and digital signal processing, and as a result these tools are still in the development phase [3].

Khoros provides the facilities to create simple 3D geometric shapes, these include isosurfaces and spheres. Tools for manipulating the color maps and creating slices through the data are also provided. The user has control over camera parameters, a number of different shaders, and object transformations.

2.4.3 Comparison to our approach

Khoros is more focussed on 2D visualisation with 3D, and higher dimensions, only being added recently. The system does not support easy-to-use 3D building blocks that can be used to create complex visualisations.

The interaction in the 3D visualisation is limited to controlling the inputs to the visualisation and not directly interacting with the 3D graphics. No reference could be found that indicated any ability to interact with the visualisation within the viewport.

2.5 GlyphMaker

GlyphMaker [28] is a system for visualising multivariate data sets. Instead of trying to find a form of visualisation that is applicable to all data sets the system use glyphs and allows the user to bind properties of these glyphs to fields in the data set.

Glyphs can be viewed as 3D icons that have a number of properties that define their appearance; these properties include size, shape, orientation, color, position, etc. GlyphMaker then allows the user to bind these properties to a field in the data set, and then view the data in 3D space.

GlyphMaker was designed to be used by non-experts with little or no programming experience to create their own custom visualisations of data. It was built on top of IRIS Explorer, using the underlying dataflow model as well as specially written modules.

The main objectives of GlyphMaker were to create a system of visualisation that was both descriptive and flexible, but still accessible to the non-expert, those with little or no programming knowledge. A point and click approach was adopted that allows the user interested in the data to create and explore the data.

GlyphMaker was designed as an exploratory tool. The focus was on the user who does not fully understand the data and who does not always know which visualisation technique is going to increase their understanding.

The objectives in creating GlyphMaker were:

Detailed control of visualisation With being able to bind individual data elements to a single glyph the user has very fine control over the visualisation.

Highly responsive controls The responsiveness of the controls does not only refer to the time lag between adjusting any control and seeing the result in the rendered visualisation, but also includes the time taken to find and adjust the control. This approach leads to the controls being prioritised, those with a higher priority being conveniently placed for the user.

Interactivity in visualisation Waiting a number of seconds for the system to respond to any changes made, detracts from the interactivity of the system. For this reason the Editor is written using the SGI graphics library.

It was also recognised that visualising large data sets negatively impacts on performance, therefore a facility was provided to reduce glyphs to points or not to render them at all. This is similar to our approach of using an automatic level system to automatically remove, or reduce in complexity, any distant shapes.

Focus and conditions Statistical graphics research has afforded a number of guidelines as to how to approach the visualisation of multivariate data sets. Many of these guidelines focus on reducing the data set to a more pertinent subset.

GlyphMaker uses its conditional box to allow the user to eliminate large portions of the visualisation which may be nothing more than noise.

Correlative linking Correlative linking is a process whereby a number of views, each containing partial information, of a complex data set are linked. The linking can take place through animation or by highlighting linked features in simultaneous displays.

GlyphMaker provides an easy method to implement correlative linking; the user defines a conditional box and then binds this box to a glyph.

Efficient accommodation for data filtering and transformation While this is being investigated on an ongoing basis, investigations thus far have created an appreciation of the effectiveness of self describing data structures.

The GlyphMaker system was built on top of IRIS Explorer. The idea was to supplement, not replace, the functionality provided by the underlying dataflow model. Where possible as many as possible of IRIS Explorer's modules were used and any new modules were written using C, Motif and GL.

The GlyphMaker system consists of four main parts:

Read Module The problem with many modern day visualisations is that they need to use data from a variety of different sources. This data has to be converted into some common format that can then be used to create the applicable visualisations.

GlyphMaker uses a system of self describing files; each file has a header that gives the description of each variable. The description for every variable includes the name, primitive type, number of instances, maximum value and minimum value.

The Read Module accepts data files in straight ASCII form, ASCII header and binary data or straight binary data.

Glyph Editor The glyph editor is a simple 3D editor that allows the user to create and edit glyphs. The editor can be used to combine glyphs into compound glyphs.

To construct custom compound glyphs the user is provided with some geometric primitives; these include points, lines, spheres, cuboids, cylinders, cones and arrows.

Glyph Binder The binder presents the user with a list of active elements as well as a list of all variables. The user then uses a simple point and click to associate variables with active elements.

In addition a text widget is associated with every active element and variable. This widget displays the maximum or minimum values for the variable or active element; initially these contain default values but they can be modified by the user.

Conditional Box The conditional box allows the user to create a condition that isolates a certain spatial region for closer examination.

2.5.1 Comparison to our approach

These objectives closely parallel our own with regard to iIsh [10].

Although GlyphMaker was built using Iris Explorer, it could equally well have been built on AVS or Khoros. The underlying model is a dataflow model that lends itself to the visualisation of time dependent data.

GlyphMaker and our set of classes are similar in the respect that we try to combine in one person the owner of the data, the creator of the visualisation and the user of the visualisation.

2.6 Inventor

Inventor is a object oriented 3D toolkit; it consists of a library of objects and methods that can be used to create interactive 3D applications. The Inventor library consists of a number of objects that can be used, modified and extended to meet the user's needs. The library consists out of 3 main types of objects[4], namely:

Database primitives The objects are the basic geometric shapes and properties that are used to create scene. These include geometry nodes, material nodes and property nodes.

Manipulators The manipulators are used to manipulate objects in the scene; these include handle boxes and track balls.

Components Components are editors that can be used to interactively change one or more properties. Examples of components are the material editor, directional light editor and any one of the number of viewers that are provided.

2.6.1 The Scene Graph

Inventor stores details of the scene to be rendered in a scene graph. A scene graph is a directed acyclic graph of nodes, where each node can represent a geometry, property or grouping of nodes. [31] Hierarchical scene graph are created by adding nodes to grouping nodes as children. An example of a scene graph is given in Figure 1. This scene graph renders two cubes each with its own material and transform nodes.

When a scene graph is rendered the nodes are traversed from top to bottom and from left to right.

When the render action begins it has an initial state which is simply all the current properties and matrices that are used to render any geometric shape. When a property node, like an SoMaterial or SoTransform node, is encountered then the current state is modified. Therefore when a geometric node is traversed the current state is applied to it to render it.

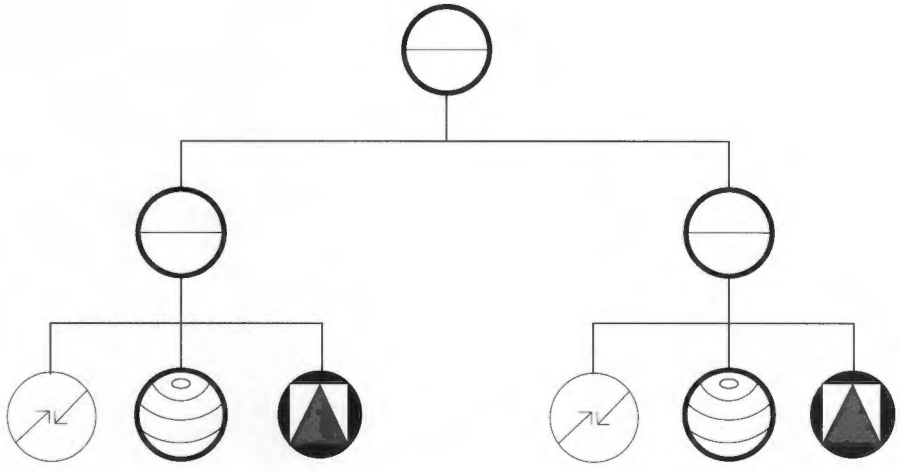


Figure 1: An Example Scene Graph

All nodes have the prefix *So* which stands for *Scene Object.*

There are 3 main groups of property nodes:

Transform The transform group includes all those property nodes that perform transformations, such as SoTransform, SoScale, SoRotation, SoTranslation, etc.

Appearance The appearance nodes are those nodes that modify the appearance of subsequent objects. These include SoMaterial, SoBaseColor, SoMaterialBinding, SoDrawStyle, SoLightModel, etc.

Metric When meshes and surfaces are defined, the point information is not stored within the shape node, but instead in a metric node. Metric nodes include SoCoordinate3, SoCoordinate4, SoProfileCoordinate2, SoNormal, etc.

The SoSeparator node is a group node to which children nodes can be added; in addition to this it also preserves the state before it traverses all its children nodes. Once all its children have been traversed it restores the state. This allows one to create a sub-graph which is guaranteed not to modify the current state and can therefore be inserted anywhere without fear of it causing unpredictable results.

2.6.2 Picking

A node can appear multiple times in the same scenegraph; this node can then represent different objects in the virtual 3D space. This means that a single instantiation of the SoCube can be used multiple times in a scene graph to display a group of cubes. This is also useful for displaying multiple copies of a more complex shape which is not represented by a single node but by a complete graph.

The advantage of using a single instantiation of a class multiple times, as opposed to multiple instantiations, is that every node needs to do a number of calculations which may be repeated if

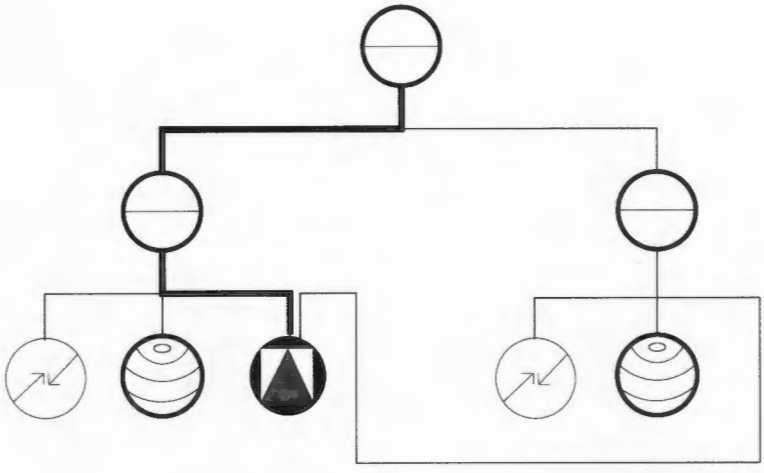


Figure 2: A Path to a Shared Node

there are a number of instantiations of the same class; an example of these calculations is calculating the normals for its faces. If there is only a single instantiation of the class then these calculations need only be performed once, otherwise they need to be performed for every instantiation. This results in a decrease in performance. In Figure 2, if a cube is selected, then just getting a pointer to the SoCube class does not uniquely identify which cube has been selected. The only way the cube can be uniquely identified is by the path from the root of the scene graph to the SoCube class.

Figure 2 shows how the path identifies which cube was selected, although there is actually only one cube in the scene graph.

Paths are used when an object needs to be uniquely identified so that properties of the node can be identified or modified.

The SoSelection node is also a group node. Like all group nodes it traverses its children from left to right. When a node is selected, the path that is returned stems from the SoSelection above it to the node that was selected.

2.6.3 Comparison to our approach

The class library we created was implemented on top of Inventor for the following reasons:

Maintaining Flexiblity

Inventor is a C++ class library for general purpose 3D interactive graphics and as such was very flexible in what it allowed us to do. This flexibility was coupled with an increased level of complexity that we aimed to eliminate.

By using a graphics library it was possible to develop the classes in such a way that the underlying Inventor libraries were not totally hidden from the user, but instead supplemented. This was

achieved by allowing the classes to be preprocessors that accept the information necessary to create the visualisation, and then generate a scene graph that Inventor can then use.

This approach was adopted as we did not want to appeal exclusively to either the non-expert user or the expert user, instead we wanted to create a system whereby we simplified the use of Inventor for the non-expert in such a way as not to frustrate the more advanced users.

By making use of the classes, the non-expert can quickly get a visualisation on the screen and start experimenting with it. The more advanced user may tend to move away from using the classes exclusively and instead start using native Inventor to supplement the classes.

Simplify Interaction

We wanted to create a system whereby the user can interact directly with the data and not have to use a number of external gadgets to interact with the data.

Inventor provides facilities for taking the 2D input of a traditional pointing device like a mouse and converting back to 3D for selecting and manipulating objects within the 3D scene. This is done by converting the 2D point into a selection ray that is cast through the 3D scene and then determining which points it intersects with. In addition to this Inventor provides a number of manipulators that can be attached to scenes to manipulate them.

Manipulators are typically nodes that have geometric and interaction parts; the geometric part appears in the scene and allows the user to interact with the scene, while the interaction part defines how the manipulator modifies the scene.

Our main interest was not in the manipulators themselves, but the way in which it was possible to directly query the rendered scenes. We can not only determine which geometric object was selected but also determine which was the nearest vertex to selection ray. Thereby we can use this to do fine resolution picking.

Inventor provided the freedom and flexibility to be able to create the visualisation we required. However, the creation of these visualisations was relatively complex, the most complex aspect being the interaction. Our classes aimed to simplify the creation of the scene graphs as well as the creation of the interaction.

2.7 Obliq-3D

Obliq-3D is a 3D animation system that allows for animations to be created and modified easily and quickly. This is achieved by using an interpreted language, Obliq, in conjunction with an animation library called Anim3D [25].

The problem in creating animations of abstract information is that it requires not only animation skill but artistic and communicative skills as well. As the process of creating a visualisation is an iterative one, the benefits of being able to use a system with a fast turnaround time can be seen in the increased speed with which visualisations can be created.

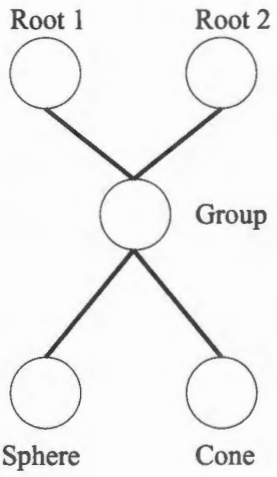


Figure 3: A DAG with two root nodes.

Obliq-3D is based upon 3 basic concepts: graphical objects, properties and callbacks. The first two are discussed in more deatil but not the latter as callbacks are treated in the same ways as most modern visualisations systems.

2.7.1 Graphical Objects

Graphical objects are geometric shapes (cubes, spheres, cones, ellipsoids, etc), lights, cameras, groups, which group together a number of other graphical objects, and roots for displaying graphical objects.

A scene is described by creating a directed acyclic graph (DAG) with each node in the graph being a graphical object. The scene is created by making one graphical object the child of another. The root graphical object then traverses its children and renders the scene.

This method of using directed acyclic graphcs as the basic underlying data structures is common in many graphics libraries, one of the advantages is that a node can be the child of more than one node. This allows nodes and whole scene graphs to be reused.

If, for instance, a scene graph was created that contained a sphere and a cone and was attached to two root graphical objects. The same scene is rendered into two different windows. By moving the camera of one root object two independent views of the same scene are possible. An example of what the DAG for the scene is given in Figure 3.

The render process in Obliq-3D is based upon a *damage-repair* model. Whenever the scene graph is modified, it is said to be damaged as the scene represented in the window now no longer corresponds to the scene graph. An animation thread detects the damage and repairs it by redrawing the scene.

2.7.2 Properties

Every graphical object has associated with it a number of properties, these properties include color, location, size, etc. A property consists of a name and a value. Properties affect not only

the graphical object they are attached to, but also all descendant objects, unless that property is explicitly overridden by one of the descendants.

Properties can also be attached to objects if they are not applicable, for instance a *SphereGO_Center* property, which determines the center of a sphere, can be attached to a cone. In this case the property is simply ignored as it is not applicable to the current object. However, it is applicable to any descendants of the cone; therefore if a sphere is a descendant of the cone this property is then applied to it.

Properties are also time variant; in other words, they can change with time. A property can be defined to have an initial value, a final value and a time frame during which the transition is to take place. As properties include such values as position and size, this makes animations very easy to define.

Not all property values change with time. To deal with this problem Obliq-3D defines 4 types of properties, namely constant, asynchronous, synchronous and dependent. Constant property values are time invariant while asynchronous property values change irrespective of other property values. Synchronous property values change when signaled, and dependent property values change depending on another property value.

2.7.3 Lights and Cameras

Lights and cameras are handled like all other graphical objects. They can be added to a scene graph and inherent properties from their parents. In this way cameras and lights can be manipulated in much the same way as normal graphical objects.

2.7.4 Comparison to our approach

Obliq-3D system is very closely related to Inventor, insofar that scene graphs are represented as directed acyclic graphs of nodes/objects, and that all geometric primitives are treated in the same manner. However, there are also significant differences in the way properties are handled

While Obliq-3D attaches properties directly to graphical objects, Inventor places the properties in the scene graph. A current state is then maintained and a property node modifies this state. When a geometric node is encountered the node is rendered using the current state.

The creation of animations is also different in that all properties in Inventor are constant. Therefore property values have to be explicitly modified. This can be done through a timer sensor that invokes a callback function at regular time intervals, thereby mimicking the Obliq-3D animation thread. However, this is not true of Inventor 2.0 where animation and time dependent properties are now possible.

Another crucial difference is that Inventor does not have any form of interpreted language. This, however, has been remedied with the development of ilsh [10].

Obliq-3D has its origins in the field of algorithm animation. Another system that deals with algorithm and program animation is PAVANE [12]. PAVANE works on the principle of a limited

number of primitives whose position and properties are updated by modifying values in a tuple space.

As the system we use is based upon Inventor there are many similarities in the way visualisations are created. However, the fundamental difference lies in the way that the user interacts with the visualisation. In both Obliq-3D and PAVANE, the interaction is through the creation of the animation; this animation is then viewed and information is derived from the viewing, or alternatively it is used for instructional purposes as outlined in other works by the same author [11].

Our system is based upon the principle that a visualisation is created and the user then interacts directly with the visualisation. The interaction does not end with the creation of the visualisation.

2.8 Closing comments

The visualisation systems that we have examined in this chapter try to create applications that allow the user to define a visualisation. AVS, Khoros, IRIS Explorer and Data Explorer allow the user to define data flow model that produces the visualisation as output. Obliq-3D and Glyph-Maker, on the other hand, allow the user to define a scene and then tie properties of objects in the scene to data fields in the input stream.

While these techniques have been shown to be applicable for a large number of visualisation applications, they do have some limitations. One of these limitations is that it is difficult to allow the user to directly interact with data set. This means that tools external to the visualisation have to be used when the user wants to interact with the data set.

Another limitation is that many of these systems offer a set number of ways in which the data can be displayed, and it is not easy to create new methods of displaying the data.

While Inventor does not have these limitations, it is more difficult to create visualisation. To create a visualisation the user needs to have broad knowledge of both Inventor and C++.

Our approach was not to try and create an application, but instead create a number of tools that could be used to create an application. These tools took the form of a suite of classes that automatically generate an Inventor scene graph. The classes can be used either through C++ or from within iIsh.

This approach therefore requires the user to have some knowledge of either C++ and Inventor, or iIsh. The user needs to know very little about Inventor to create a visualisation, however, the expert user can still use all the power and flexibility that Inventor provides. Using the classes from within iIsh only requires the user to have a basic knowledge of how scene graphs are structured.

In the following chapter we discuss our approach in more detail as well as motivate why we selected this method.

Chapter 3

Class Description

While high level, off-the-shelf visualisation packages are easy to use they are by their very nature specialised. The most important limitations that they impose are the following:

- In most cases they only allow the user to display the data and not interact, or query the data [35].
- They constrain the user to view the data in a number of predetermined ways or, sometimes, a limited combination of these.

A graphics library on, the other hand, provides the flexibility and power to create almost any visualisation with its accompanying interactions and interrogation techniques. However, graphics libraries have the following disadvantages:

- There is a significant time delay between when the visualisation is conceived and when it can be analysed and evaluated.
- Modification and corrections are also time consuming, and as designing visualisations is often an interactive process of design - implement - evaluate, this causes the whole process to be slowed down.
- The use of graphics libraries require the creator to have knowledge of how to program as graphics libraries mainly deal with primitives and not with higher levels of abstractions.

In designing our classes the objective was to create a system whereby a non-expert user can easily able create a visualisation, but at the same time the system still allows the more advanced user to take full advantage of the power of the underlying system. Our goals in creating such a system were as follows:

- Simplify the creation
  - Provide easy to use methods of interaction.

  - Simplify the creation of visualisations, thereby allowing the the user to be the creator of the visualisation.
  - Design the classes so that they can be easily combined to form more complex visualisations.

- Maintain the Flexibility
  - The increased simplification must not restrict the user with respects to the types visualisation that can be created.

- Sacrifice as little performance as possible.
  - Our system must have as little impact as possible on the underlying rendering system as possible.

- Effectively display fine detail
  - The display of fine detail when one wishes to see an overall view often amounts to no more than noise that decreases the effectiveness of the visualisation. Our system must allow the user to eliminate the data once it is no longer relevant, or its significance as fallen below some given threshold.

In the following section we shall consider each of the points above and explain the significance and relevance of each.

3.1 Simplification

3.1.1 Interaction

As previously stated, in Section 1.2, visualisation must provide the ability for the user to interact and query the data, not just produce a graphic output. It is for this reason that interaction is considered to be vital [29, 23].

In our suite of classes, each class is given a form of query interaction. This interaction takes the form of callback functions and the user is required to to define a callback function that is called each time any object, or part of an object, is selected or deselected.

The callback function takes as parameters, firstly, a name, that uniquely identifies the class that called it, and secondly, a number of parameters that allow the user to uniquely identify which data item was selected.

This form of interaction frees the user from determining which element of the visualisation was selected and then determining which data item is the most likely one to be select. Instead the user now has the freedom to purely define how the system should react to the selection of that data item.

By using iIsh this process is simplified further. The user defines a script that is called every time the user selects any data item. This script is passed enough information to be able to uniquely identify which data item was selected.

3.1.2 Combining the user and creator

Scientific visualisation has an underlying 3D spatial model, where no such model exists for information visualisation. This makes the creation of a visualisation more complex, as the user often has to go through an iterative process of creating and then evaluating the visualisations.

In many cases the creation of a visualisation is a complex process and can involve specialised knowledge. For this reason the owner of the data, who is often also to be the user of the visualisation, is not necessarily the creator of the visualisation.

In an iterative process this has two distinct draw backs:

1. The user has to communicate his ideas to the creator, and then the creator has to interpret what the user wishes and implement it. Often there is a problem in this communication stemming from the different backgrounds of the two individuals.
2. The process is time-consuming. The creation process is an iterative one, and therefore when the creator and the user are two different people the turnaround time from conceiving an idea to evaluating it is extended.

In creating the classes using Inventor 1.0 we identified two main areas where we felt there existed the greatest complexity and decided to simplify these areas.

The areas were simplified as follows:

1. Converting the data into a geometric representation.
2. Identifying which data element the user selected.

The turnaround time can be significantly decreased by the use of iIsh as it is an interpreted language, and therefore does not need to be compiled and linked for changes to the visualisation can be evaluated.

Creating a visualisation

The 3D representation of the data is defined by a scenegraph. The classes create this scenegraph by using the data given to them by the user and then return a pointer to the root of the scenegraph. The user is then responsible for adding the scenegraph to another node, or passing it to a viewer.

There are methods in each class that allow the user to modify values within the class, so that the data can be changed even after the scenegraph has been created.

Defining the interaction

To make use of the interaction the user defines two callback functions, one that is called when a data element is selected and one when the element is deselected.

By having two separate callbacks it is possible to highlight data elements that have been selected, and when a new element is selected then the previous data element can be unhighlighted.

The callback functions take a variable number of parameters, depending on the class. The parameters are:

- A name that is given to each class when it is created, this allows the user to uniquely identify which class the data element belongs to.
- Either one or two coordinates which allow the user to identify which data element was selected.

Through this technique the user only has to define actions, or events, that must take place when a data element is selected, and is not necessarily concerned on how to determine which data element was selected.

3.1.3 Combining classes

In creating the classes we realised that it is impossible, and counter-productive, to try to create a range of classes that cover every single type of visualisation. We therefore decided to find components that can be combined to create visualisation. It was these components that were then encapsulated into classes.

The classes can be divided in two main groups:

Functional The functional classes are those classes that provide additional functionality, without creating a visible object within the rendered scene.

The functional classes are VoSwap and VoScale.

Visible These classes consist of classes that are either *constrained* or *unconstrained*.

Constrained Constrained classes are the classes that are constrained to a uniform rectangular grid. Each data element is placed on a vertex in this grid, the data element's value determines the height at which the data element is placed.

The constrained classes have the data elements uniformally spaced 1 unit apart on the xy plane.

The classes that are constrained are the VoMesh, VoSquareMesh, VoRibbon, VoLines and the VoCube.

Unconstrained Unconstrained classes are classes where the 3D position of the data elements has to be defined.

The unconstrained classes are the VoVectorField and the VoPoints classes.

The constrained classes can easily be combined because the x and y positions of the data elements are predetermined. Therefore if two classes, like the VoMesh and VoCube are given the same data set, and the scenegraphs that the classes returned are placed in a scenegraph, without any transformation nodes between them, the center of the cubes from the VoCube class is the same as the vertices of the mesh created by the VoMesh class.

In Section 3.5 we demonstrate how complex visualisations can be created by combining a number of classes.

3.2 Maintain the Flexibility

The flexibility of the system was maintained by creating classes that could be combined to create more complex visualisation. The process of combining the classes to form complex visualisations is discussed in Section 3.5.

3.3 Performance

When considering the performance of the classes we looked at two aspects of the performance, namely how responsive was the system and, secondly, how much visual information was conveyed.

3.3.1 Rendering Performance

The rendering performance directly influences the responsiveness of the system. If the system is not sufficiently responsive the user does not feel able to interact with the data.

When deciding on how we were going to render our visualisations we had two alternatives: to create our own library or use a custom library.

Inventor 1.0 vs. Custom Library

The task of writing a custom library was beyond the scope of this thesis and in addition to this it was felt that a graphics library written by specialists, which is continually being improved and upgraded is more efficient than a custom written library.

We therefore decided to use Inventor 1.0 because it provided us with a sufficiently well structured and comprehensive set of classes that we could then expand upon.

The difference in rendering speeds between using Inventor and straight GL did not warrant the additional complexity of using GL.

Performance Penalty

The classes interact with Inventor in two areas:

Rendering

The classes do not directly impact on the rendering speed as they act only to create the scenegraph, and therefore the overhead of the classes is only incurred in the creation of the classes.

In the creation of the scenegraph we have attempted to create optimal scenes that provide the best visual effect with the best rendering speeds. An example of one of the problems encountered was the fact that creating a mesh and giving each vertex an emissive color created a scene that was more expensive to render than if each vertex were assigned a diffuse color. These are the sort of performance issues that the user should not be forced to know about in order to create a visualisation.

In the case of ilsh, although the script is interpreted and is in this case slower than the compiled C++ code. The performance penalty is only during the creation of the visualisation, all rendering is done by compiled C++ code.

Interaction

When the user selects an object that lies within the scenegraph a general purpose callback function is called. This callback function then determines which object within the scene was picked and which user-defined callback function must be called.

The selection process must not only determine which class was selected but which data item within that class has been selected. This process of determining the closest data item to the picking ray is done using native Inventor code.

The additional overhead that the classes create is the process of taking a selected data item and then mapping it to a callback function. This involves a single look-up in a table and a function call which does not constitute a large overhead.

When using the ilsh the callback function invokes a script, this is the only time that a script is interpreted while the visualisation is being manipulated. However, as the time taken to do the picking is in the order of milliseconds already, the additional time taken to interpret the script is not very noticeable. This is of course dependent on the script that is being interpreted.

3.4 Effectively Display Fine Detail

In measuring the quality of visualisation we identified three types of information that were being displayed:

Graphical Representation of the Data This is the 3D geometric object that represents either a single data item or a whole data set.

Relative Indicators These are indicators that give the users information with regard to scale, orientation and position. They can be in the form of a grid, a normalised data set, a minimum data set, a maximum data set or any other method that is most applicable to the current data set.

Quantitative Information This is information is more accurate and often involves the display of alpha-numeric text or fine detail.

Fine detail or text is necessary to provide the user with as much information as possible, however, this fine detail can become noise and obscure interesting phenomenon when the whole visualisation is viewed.

To overcome this problem we created an automatic level of detail tool that can automatically eliminate certain parts of the scene when they become relatively less important. This is class is discussed in Section 4.15.

3.5 Creating Complex Visualisations

The classes that have been created have not been designed to be stand alone visualisations, but instead were designed in such a way as to make it easy to integrate them into more complex visualisations.

3.5.1 Underlying Concepts

Before multiple-class visualisations can be created it is necessary to give some information with respect to positioning and scale of the classes and the scenegraphs they generate.

Scale: In the case of the visualisation objects that are constrained to a rectilinear grid, the spacing of the data points is always one unit apart. This means that all classes that are constrained to a rectilinear grid can be overlayed and that corresponding data points are correctly positioned.

Orientation: All classes are designed in such a way so that all references to height are along the z-axis. This means that in the case of any of the rectilinear classes the columns and rows refer to the x and y-axes, respectively and the height refers to the z-axis.

As many of the classes conform to the above restrictions, it is easy to overlay, and thereby combine classes.

3.5.2 An Example of Combining Classes

An example of combining classes is given to demonstrate how the classes can be combined.

A natural and logical starting point is the VoGrid class. We combine this with a VoRibbon class.

We start by determining the size of the data set that we are working with in terms of the number of rows and columns that we need. In this case it is x columns by y rows.

We then use this as the number of rows and columns for all the classes that are constrained to a uniform grid, namely the VoGrid, VoRibbon, VoCube and VoLines classes.

We now have to tell the VoGrid the size we want and also the number of divisions we want as well as the labels for the x,y and z axes. A VoGrid is then generated similar to the one in Figure 4.

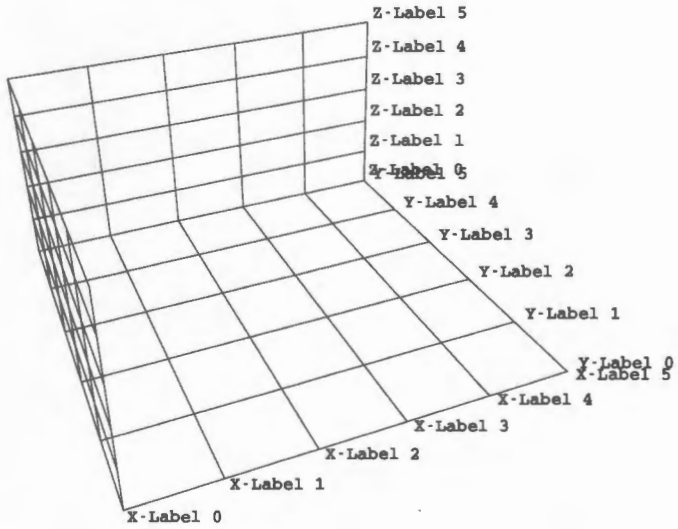


Figure 4: First step in combing the various classes

Once we have created the VoGrid we can now create the VoRibbon. The VoRibbon is also restricted to a rectilinear grid.

Now that the VoRibbon and VoGrid have been created, we create the VoCube and add that to the scene graph. The positioning allows the cubes from the VoCubes to be placed directly over the corresponding data points in the VoRibbon. The result of combining these classes can then be seen in Figure 5

We also use the VoLines to connect the cubes with the data points in the ribbon. The resulting visualisation can be seen in Figure 6.

From the above example we can see that by combining the classes we can create more complex visualisations that convey more information than is otherwise possible.

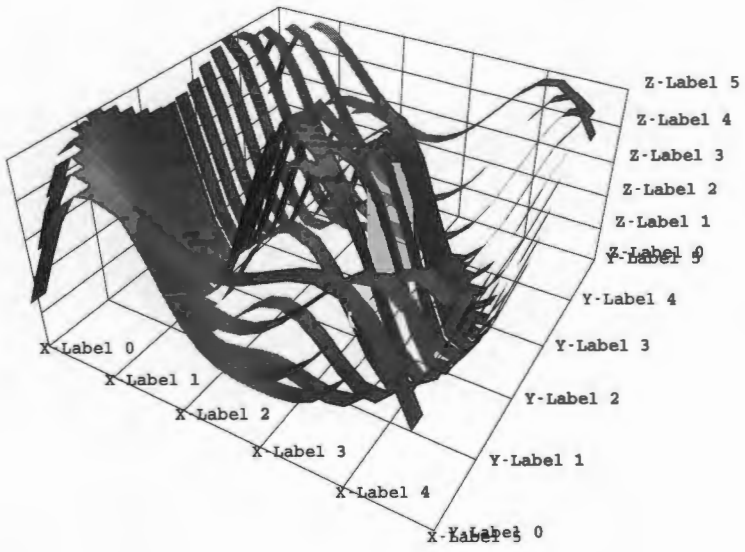


Figure 5: Combining the VoRibbon with VoGrid.

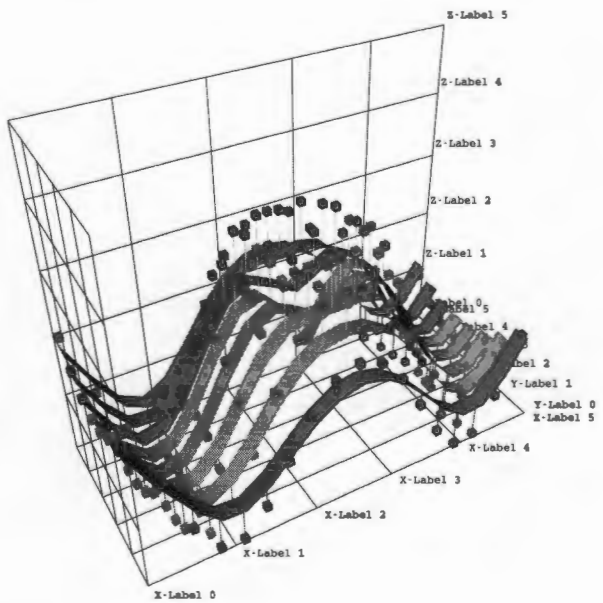


Figure 6: Combining the VoGrid with VoRibbon, VoLines and VoCubes.

Chapter 4

Implementation Issues

In Chapter 3 we discussed some of the classes that we created. In this chapter we look at the rest of the classes that were implemented and discuss each one in more detail.

4.1 Visualisation Classes

While Inventor allows the creation of visualisation at a higher level of abstraction, it still does not allow direct translation of visualisation primitives into Inventor code. In order to overcome this deficiency we have created a set of classes that can take the data and translate it into a scene graph.

The classes also allow easy interaction with the data through callback functions. Every time an item is selected the callback function is called with parameters that uniquely identify the data item that was selected. This allows the data item to be identified and more information about it to be provided to the user.

The primitives that were found to be often reused have been identified and encapsulated in classes. The classes created to date are:

The *VoBase* is an abstract base class for all the other classes. The *VoMesh*, *VoSquareMesh*, *VoRibbon*, *VoCube*, *VoPoints*, *VoLines*, *VoVectorField* and *VoSpiral* all generate data dependent graphical objects. The *VoGrid* generates a grid which can act as a reference point for size, position and orientation. The *VoSwap* allows the user to swap between pre-generated scene graphs and the *VoScale* allows a whole scene graph to be scaled to fit into a predetermined cubic region.

The *Vo* prefix stands for *Visualisation Object.*

4.2 VoBase

This is the base class from which all the visualisation objects are derived. It contains the following functions:

GetSoGraph() This virtual function creates the scene graph. Each class has to define its own function that creates the scene graph.

SetSelectionNode() This function sets the selection node. The selection node is needed so that the callback functions can be associated with a selection. Every time part of the subgraph generated by the class is selected then the appropriate callback functions is called.

SetClassName() This functions stores the class name in a node in the graph so that when there are multiple instantiations of the same class then these instantiations can be uniquely identified.

4.3 VoMesh

The VoMesh is a uniform rectangular mesh that employs the user-defined heights and colors for each vertex. The mesh has its rows and columns evenly spaced in a uniform grid. This allows the user to easily generate a landscape type of view.

The view created allows vertices with unusually high, or low, values to be easily identified.

The advantages of using the mesh is that for $n \times m$ data points there are $(n-1) \times (m-1)$ polygons. This means of visualisation has a very low impact on the responsiveness of the visualisation when the scene is rerendered.

The data points are joined both vertically and laterally by lines; this can imply a non-existent relationship between the data points. For instance, if one uses this method to visualize the stock market, each point can be used to represent the data for a single share on a specific day. One may expect there to be a relationship between the price of a single share from day to day; however, there may be no relationship between the prices of the two adjacent shares. Thus the line that joins the two adjacent data points can be misleading by implying a relationship.

Another factor that must be taken into account is that the color is interpolated between the vertices. This can cause misleading observations if the real data does not follow this trend.

This form of visualisation is best suited to situations where relationships between both adjacent columns and rows exist. For instance a regular point sampling of a characteristic over a certain area is ideally suited to this form of visualisation.

Looking for minerals across a certain area of land is an example of this type of data. Core samples can be taken at regular intervals and then the data can be visualized. The depth at which the mineral is found can be assigned to the height of the vertex while a color can indicate the quality of the mineral. This then provides an easy to read picture of the distribution of the minerals beneath the surface.

4.4 VoSquareMesh

The VoSquareMesh is a modification of the VoMesh. The difference lies in the fact that instead of each data point being represented by a vertex it is now represented as a square horizontal face. This allows the user to generate what can be seen to be 3D bar graphs.

The advantage of this is that the data points can be distinctly identified because the square is uniformly colored. The color interpolation only takes place between the squares. It is also easier for the user to select a square when the data is to be queried.

There is also no implied relationship between the data points, as the transition from one data point to the next is represent by a vertical face.

The disadvantage is that there is an increase in the number of polygons. For a $n \times m$ set of data points there are now $(2 \times n - 1) \times (2 \times m - 1)$ polygons. This obviously has a direct impact on the time taken to render the visualisation, and thus on the responsiveness of the system.

This form of visualisation is better suited to the stock market visualisation. No relationship is implied between different shares, or from one day to the next. It is also clear the the data being visualized is from a discrete data set, not a continuous one as implied by the VoMesh.

4.5 VoRibbon

The VoRibbon allows the user to define a group of ribbons by giving their height, color and width at each vertex.

When the user selects a ribbon a user-defined callback function is called with the name specified in the constructor and the ribbon and vertex number of the vertex that was selected.

The user is also able to move the selected ribbon around by selecting a ribbon and then using the mouse and the middle mouse button to specify where the ribbon should be placed. The new position of the ribbon is such that the ribbon is still parallel and in line with its previous position.

Jacque Bertin [8] describes a matrix based permutation device called the "domino" device perfected by *Laboratoire de Graphique de l'Ecoles des Hautes Etudes en Sciences Sociales.* He describes 3 types of devices, the first two being manual manipulation devices and the third a cathode screen based device. In the VoRibbon class aspects of the "domino" device have been incorporated insofar as one is able to rearrange the ribbons in order to group ribbons with similar characteristics together.

The VoRibbon combines 2 forms of data; each ribbon represents a discrete data item while vertices in the ribbon are best used to describe a continuous data set. The reason for this is that the joining of the vertices implies a relationships and the color of each vertex is interpolated between it and its adjacent vertices.

Again the VoRibbon can be used to visualize the stock market, each share being a separate share and each vertex within a share being the data for a day's trading. This then implies that there is

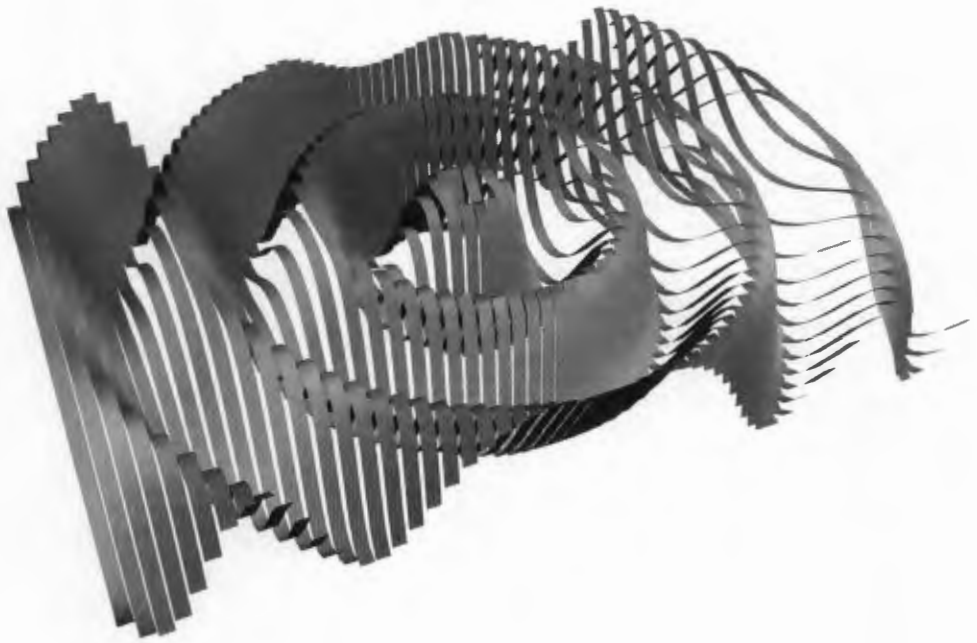


Figure 7: Example of a VoRibbon.

a relationship between the price of a share from one day to the next; this assumption may or may not be valid depending on which school of thought one favours in the field of technical analysis.

This class was also used in the analysis of rough-cut capacity planning. The maximum production capacity of each machine was represented by means of a ribbon, and then the VoCubes class was used to represent the allocated production for each machine on a particular day. This allowed the user to determine which machines were being over-utilised and which were being under-utilised.

4.6 VoCubes

The VoCubes class allows the user to define a rectangular region of cubes; the height and color of each cube can be specified. In addition the dimensions of the cubes can be changed. Each cube can have a different height, depth and width.

The VoCubes can be used to display discrete data items as no relationship is implied between the data items.

Visualisation Example

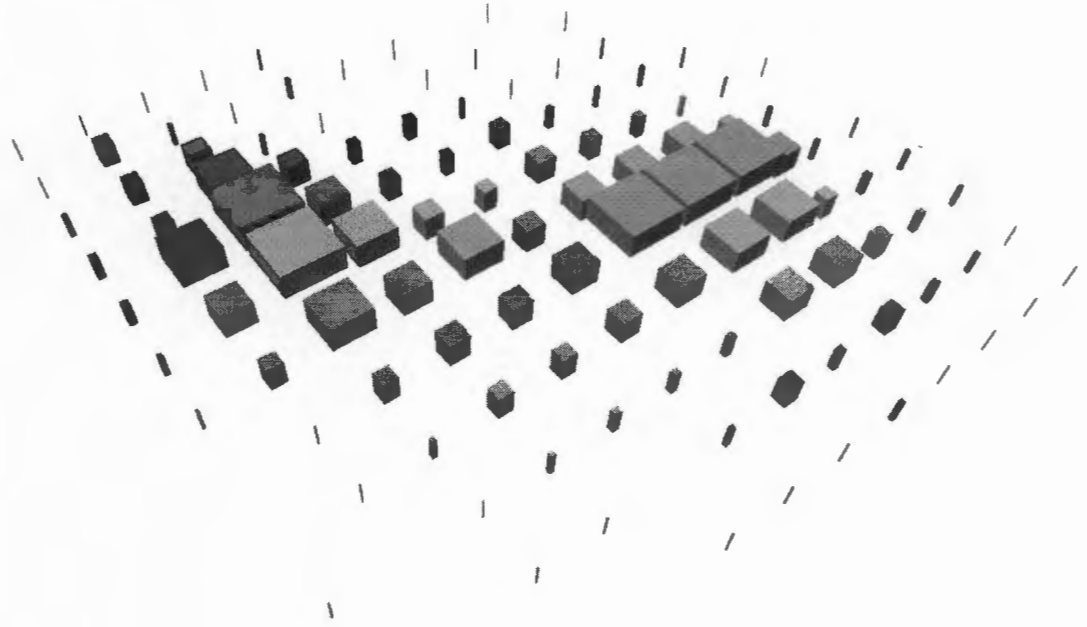


Figure 8: Example of a VoCube.

The VoCube class can be used to do scatter plots very effectively; an example of this is given in figure 8. In this example the distribution of males having certain heights and weights is visualised. The size of the cubes determines the number of males that fall into each group, corresponding to a given height and weight. This example is adapted from an example in the book by J Bertin [8, page 106]

The color and height values of the cubes have not been used but can be utilized to display additional information about each group of males. For instance the average age of the males in each group can be plotted on the vertical axis. This can then be used to view a height versus age, and a weight versus age distribution within the context of the above case study.

4.7 VoPoints

This class allows for the definition of various points in 3D space. All the points can have the same color or have an individual color assigned to them. The points can also be enclosed in a convex polyhedron.

This class can be used very effectively for cluster analysis where the data is in a 3D format, or the data set is very large and each data item has a singular characteristic. In other words, each data item is defined solely in terms of its position and one other characteristics.

Visualisation Example

Figure 9: Extra-Galactic Super-Structures

This was used effectively by Prof A.P. Fairall from the University of Cape Town's Astronomy Department in the search for extra-galactic super-structures and sub-structures within these super-structures. He was able to find a cellular sub-structure within the super-structures. [15, 14].

4.8 VoLines

This defines a class which allows you to join points to form line segments parallel to the x, y, or z axes. Line graphs can be drawn parallel to the x or y axes, or two corresponding points in the primary and secondary data sets can be joined.

Like the VoCube, VoRibbon, VoMesh and VoSquareMesh classes this class is aligned to a rectilinear grid.

The user is asked to give the class two sets of heights, the primary data set and the secondary data set. Lines can be drawn between points in adjacent rows and/or adjacent columns. In addition to this lines can be drawn between corresponding points in the two data sets.

4.9 VoVectorField

The VoVectorField class is used to visualize any data set that can be represented by groups of vectors. The wind patterns in or around a building, current flows, population migrations, etc. are all examples of data that can converted into vectors.

The class can define any number of vectors; these vectors are then placed in 3D space and given a direction, color and magnitude.

The interactive component of this class allows each vector to be queried individually; the callback function is then passed the index of the vector that was selected.

In contrast to other vector visualisation methodologies [30, 13] we did not take the approach of depicting the vectors as being a stream or a flow, but instead used the methodology that allows each vector to be depicted individually. The reason for this is that in the abstract data sets that we encountered the data is generally discrete, where as in other fields, like fluid dynamics the data is discrete but is attempting to model a continuous process.

This approach allows one to use the vector field in cases where one does not wish to show any movement, but just the forces that are applied at that point. An example of this is the upward, or downward, force exerted on a share in the stock market due to inflation, business confidence, or a number of other factors.

4.10 VoSpiral

The VoSpiral class is used when one has a matrix of information that represents all the relationships between the data elements. An example is shown in Table 1, which represents the relationships that exist between five variables, A to E. In this case the relationship is symmetrical, i.e. the relationship between C and A is the same as that between A and C. This need not always be the case.

	A	B	C	D	E
A	1	4	8	9	3
B	4	1	2	2	9
C	8	2	1	3	9
D	9	2	3	1	0
E	3	9	9	0	1

Table 1: Matrix Example for VoSpiral.

This data can be given to the VoSpiral class, each variable being represented by means of a cube; these cubes are then placed in a spiral. The relationships between the variables are represented by cones that point from one cube to the other.

While it is possible to have relationships from both A to B and B to A, this does cause the two cones to intersect and can decrease the effectiveness of the visualisation. This class is particularly

useful when there can be a relationship only one way or the other; in other words only the upper or lower triangular half of the matrix contains values.

Visualisation Example

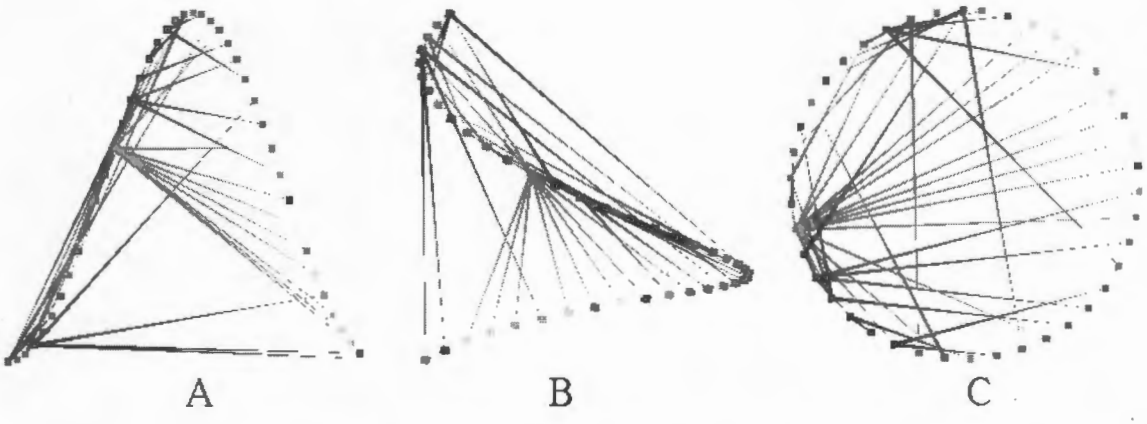


Figure 10: The VoSpiral class being used in Chiron.

In Chiron [18, 17] the VoSpiral is employed to give a profile of parallel program execution. Each cube is a function and the cone represents a function call from one to the other. In Figure 10 an example program execution is given, with views along the x, y and z-axes represented by A, B and C respectively.

4.11 VoSwap

This class allows the user to switch between various scene graphs, it can be likened to a *case* statement. The user places various scene graphs within the class and then with one function call can switch between the different scene graphs.

This class was designed to make short interactive animations scenes possible. The user sets up a number of *frames* beforehand and then allows the computer to pregenerate each scene. The frames are not 2D static images but 3D *snapshots* which can be rotated and manipulated.

Through an interactive control, such as a slider bar or thumbwheel, the user can interactively select which frame to look at. This allows the changes between frames to be studied.

The frames usually represent different time instances. An example is the investigation of how heat travels through a solid surface. The frames showing the isosurfaces for a specific temperature at fixed time intervals can be generated and then displayed in succession to investigate how the heat propagates through the solid object.

Similarly the frames can be used to investigate the isosurfaces of different temperatures at different time instances. The user can then interactively look at the different temperatures at those time instances.

This approach can then be taken one step further and both the above examples can be combined to allow the user to interactively select any isosurface for any time instance then either change the temperature or time instance interactively.

4.12 VoGrid

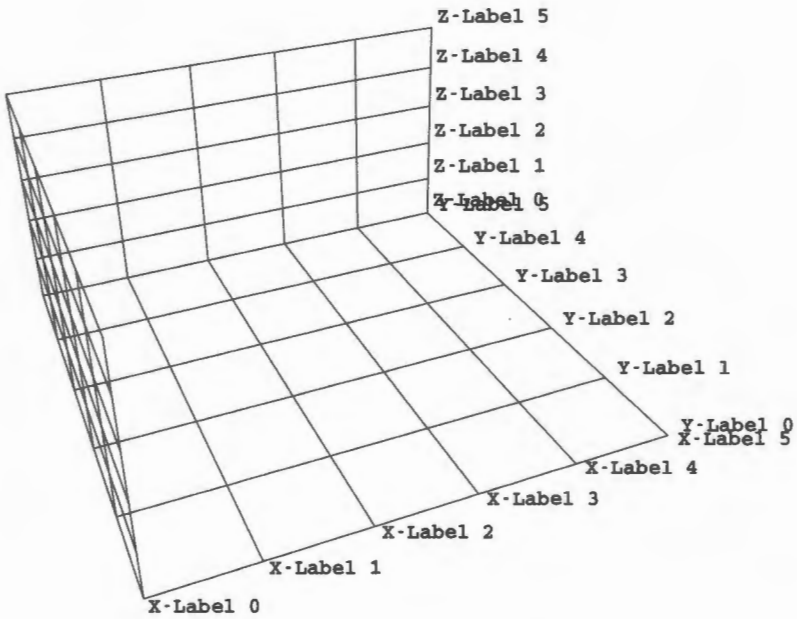


Figure 11: 3D Grid with labels on the X,Y and Z axes.

This class allows the user to create a grid by specifying the width, depth and height of the grid. Either the number of subdivisions in the grid or the distance between the lines may be specified. The grid may be labeled with two sets of labels which are placed on any two adjoining sides. An example grid with no data is given in Figure 11.

Vertical grids have also been created that can be placed on two adjoining sides of the grid. This allows the z-axis to be labeled as well.

In 3D virtual environments it is easy for one to switch up and down as well as left and right. This can cause confusion and disorientation when one is dealing with a single visualisation. The grid is designed to be able to give the viewer a reference point for orientation in 3D space. It also provides information as to the scale and magnitude of the data.

4.13 VoScale

This class allows whole subgraphs to be scaled to fit into a box of a certain volume. This is often necessary when a variety of complex objects must be displayed simultaneously in the same visualisation and one wishes to ensure that the objects are correctly positioned and scaled.

VoScale allows graphs to be displayed in a unit sized cubic area. This means that creating complex scenes featuring a number of graphs can be done easily by always ensuring that the graph remains a fixed size and adjacent graphs do not intersect.

One pitfall that must be mentioned is that if different graphs are all scaled to fit into a predetermined cubic volume then any information about relationships between graphs can well be lost.

4.14 Mathematical Comparisons

Table 2 compares the different classes with respect to the number of data points and the number of graphics primitives.

Class Name	Data items per data point	Data Points	Number of graphics primitives
VoMesh	2	$n \times m$	$(n-1) \times (m-1)$ polygons
VoSquareMesh	2	$n \times m$	$(2 \times n-1) \times (2 \times m-1)$ polygons
VoRibbon	3	$n \times m$	$(n-1) \times m$ polygons
VoCubes	5	$n \times m$	$6 \times (n \times m)$ polygons
VoPoints	2	n	n points
VoLines	2	n	$\max 2 \times (n-1) \times (m-1) + n \times m$ lines
VoSpiral	2	n	$\max 38 \times n$ polygons
VoSwap	N/A	N/A	N/A
VoGrid	N/A	N/A	N/A
VoScale	N/A	N/A	N/A

Table 2: Relative expense of the Vo Classes.

4.15 Level of Detail Control

In our experimentation a need to automatically eliminate unwanted detail was discovered. Situations arose where parts of the scene no longer added information to the overall visualisation, and in fact in many cases obscured vital information. An example of this is the use of the VoGrid, when the camera is sufficiently far away from the grid then the lines from the grid tend to obscure the visualisation and not provide any useful information.

The level of detail suppression is a new node that was created that can be placed into the scenegraph to allow different subgraphs to be displayed, depending on the distance between the node and the

camera. It differs from the other classes in that it is placed in the scene graph and is traversed every time the scene is rendered and can not be viewed as only doing preprocessing work.

This effectively allows the user to create a number of subgraphs that can then be used to depict the same visualisations with varying levels of detail.

It must be pointed out that node was developed for Inventor 1.0. Open Inventor, the most recent version of Inventor, now includes a level of detail node as a standard node.

4.15.1 Eliminating Noise

Not all visualisations consist of three-dimensional objects; lines and points are also regularly used. Both lines and points are not true three-dimensional objects, a line is one-dimensional and a point has zero dimensions; therefore, when a perspective projection is applied to these primitives their size is not scaled correctly. For instance a point is always one pixel, irrespective of the distance from the camera to the point; likewise a line always has a width of one pixel irrespective of distance. True three-dimensional objects on the other hand decrease in size with an increase of distance from the camera to the object.

The different ways the objects are rendered produce a difference in relative size depending on the distance between the visualisation and the camera. This can cause lines, points and other objects to become nothing more than noise when the viewer is sufficiently far away from the visualisation.

In order to eliminate this we have investigated ways to allow the user to remove these primitives that can obscure the visualisation.

Including Text

It is often informative to include text within the visualisation to provide more information. The two types of text available to us under Inventor 1.0 were 3D text and 2D text.

3D Text

Three-dimensional text can be created that can be placed in the scene as a 3D object. However, there are certain advantages and disadvantages to creating and using such text.

The advantages are:

Perspective The text consists of 3D polygons and is therefore rendered correctly. More distant text appears smaller than text that is closer to the camera.

The disadvantages are:

Performance To create a three-dimensional piece of text requires a number of polygons; for instance the letter 'T' consists of at least 24 triangles, while more complex letters require even more triangles. This has a negative effect on the performance of the system and can slow the rendering speed down significantly.

Orientation Because it is a 3D object; the text is not always oriented facing the user. This means that often the user is unable to read the text, i.e. either it is edge-on or the text is reversed.

This problem can be overcome by always ensuring that the normal for the text is pointing towards the camera. However, this has a performance impact every time the visualisation is manipulated.

2D Text

Two dimensional text uses a bitmapped font with a fixed size that is always facing the user. The position of the text is determined by the 3D scene; this position is then used to plot the 2D text on the viewport.

The advantages are:

Performance The 2D text has a very low performance impact as it only require a single matrix multiply to convert a 3D point within the scene to a 2D point within the viewport. This 2D point is then used to place the text.

Orientation The text is always oriented in such a way as to make it readable for the user. This does not require any additional calculations but is a side-effect of the way it is placed in the viewport.

The disadvantages are:

Perspective While the 2D text does not vary with distance, it is still placed in the Z-buffer, and therefore can be obscured by other objects. However, a number of 2D text labels in a scene can tend to clutter the scene and become noise instead of information, once the user moves away from the scene and it becomes smaller.

The SoDetailLevel (see 4.15.3) node can be used to eliminate the disadvantages of 2D text. This can be done by placing a number of instantiations of a 2D text class as the children of the SoDetailLevel node, each of the instantiations uses a different size of text, going from largest to smallest, with the last child having no text at all.

This results in the text closer to the user appearing larger than text further away, and text that is sufficiently far away is not displayed at all. This mimics the effects of perspective projection by breaking the distance from the camera to the object into discrete regions and associating a certain sized text with each region.

An example of different sized text that has been assigned to a number of cones can be seen in Figure 12.

4.15.2 Performance Issues

The level of detail suppression can also be used to increase the rendering speed of the overall visualisation.

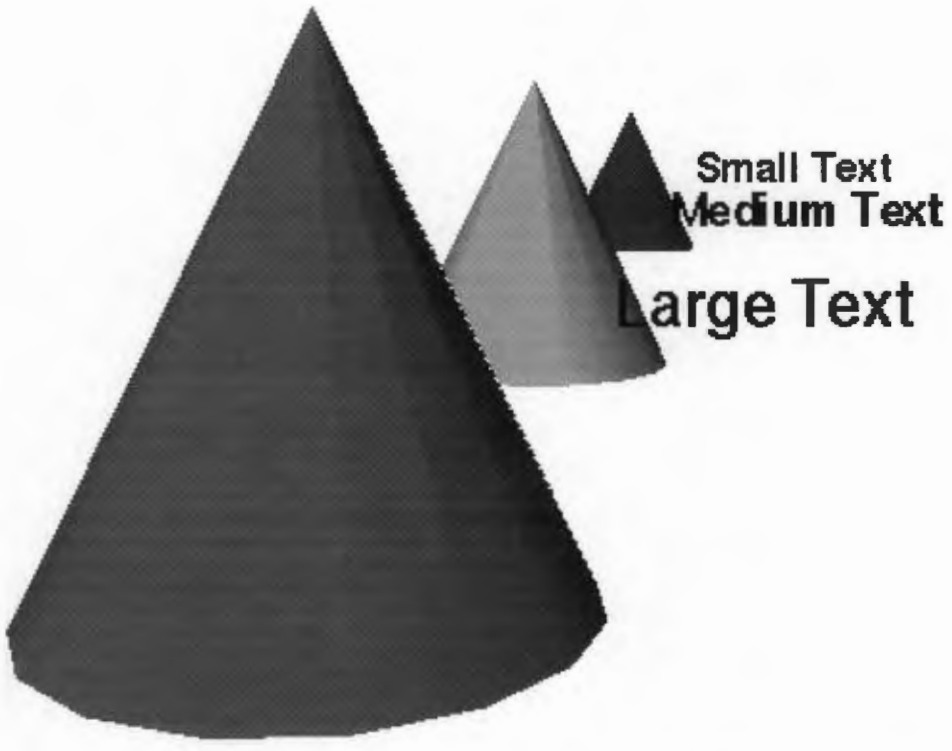


Figure 12: 2D Text that is automatically scaled using the SoDetailLevel node.

When large data sets are being visualised, or a number of visualisations are being displayed simultaneously then the SoDetailLevel node can be used to create simplified versions that is displayed when the visualisation is further away. These simplified versions should consist of fewer polygons and therefore can be rendered faster.

4.15.3 SoDetailLevel Node

Our solution to this was to create a node that can be placed in the scene graph which automatically alters the the scene graph depending on the distance of the camera to that part of scene graph. This allows different sub-scenegraphs to be rendered depending on the distance to the camera. For instance, a sphere can be used to represent a data point; with hundreds of data points this translates into thousands of polygons. This can be simplified by rendering spheres that are far away as cubes instead of spheres. That means that only data that is close to the camera, and thus the user, is rendered in full detail, and with an increase in distance, there is a decrease in detail. This closely mimics how we perceive the natural world.

This form of detail control is also far more intuitive than menus or toggle switches because it is natural for a person to approach something that they find interesting. If one sees an interesting anomaly it is a natural reaction to approach the anomaly until one is at a distance that allows one to see the most detail without losing information pertaining to its context; in other words one approaches until one reaches a comfortable viewing distance. It is this behaviour that is encourage within the virtual world.

4.15.4 An Example of Level Of Detail Suppression

In order to display the maximum amount of information simultaneously and still preserve some order it is often necessary to reduce some information and only display it once it becomes relevant. In order to achieve this we decided to use a system whereby the user automatically determines the level of detail by moving closer to the visualisation. In this way information that will otherwise clutter and overwhelm the viewer is only displayed once it is apparent that the viewer is interested in that part of the visualisation.

In displaying a number of 3D graphs simultaneously using the VoRibbon class the grid and labeling of the grids began to obscure the actual graphs. By placing the grids and labeling under a SoLevelOfDetail node it is possible to specify the distance at which these feature become visible.

Figures 13, 14, 15 and 16 exemplify a visualisation that employs the level of detail node. Each figure is a view of the same scene graph, differing only in the distance between the viewer and the visualisation.

It can be seen that as one approaches the visualisation the amount of detail steadily increases until eventually one is able to see all the relevant information.

Figure 13: Level of Detail control, far away.

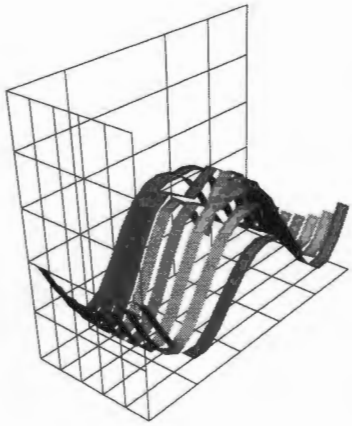


Figure 14: Level of Detail control, moving closer to the visualisation.

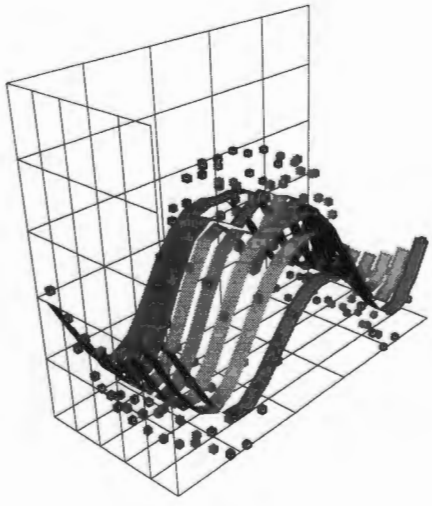


Figure 15: Level of Detail control, close to the visualisation.

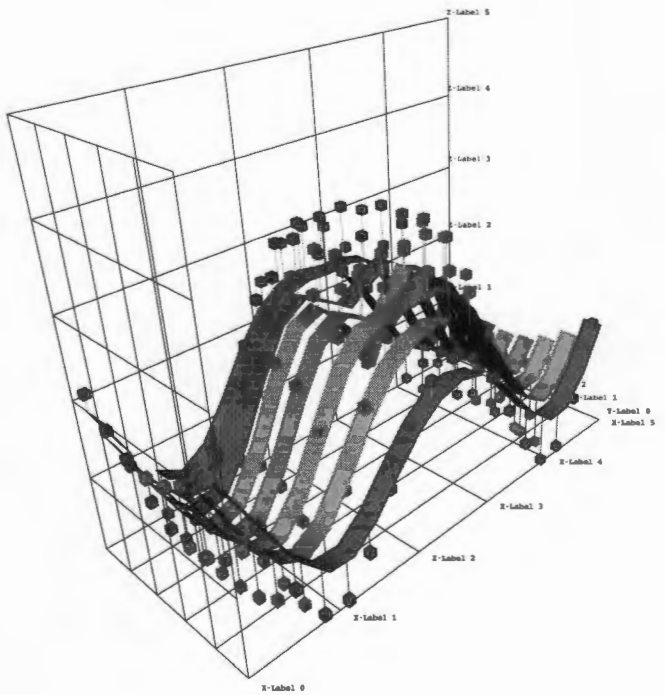


Figure 16: Level of Detail control, close to the visualisation.

Chapter 5

Case Studies

In this chapter we discuss our experiences in using the classes while developing a number of applications. We show how the use of our classes simplifies the creation of the systems.

The first system we created was used by the Astronomy department at the University of Cape Town for investigating extra-galactic super-structures. These super- structures are formed when a number of galaxies form clusters of various shapes and sizes. The researchers are looking at the relationships between these structures as well as their internal structure. This visualisation was our first experience in creating a scientific visualisation.

The second system we implemented used data from the Johannesburg Stock Exchange. The data used was the daily trading information for all the shares, this allowed us to investigate the usefulness of the classes when used to visualise abstract data.

The final system was used for visualising capacity planning data in a manufacturing environment. The data used was obtained from a commercial package and then visualised using the classes from within iIsh.

5.1 C++ vs iIsh

The systems that we have implemented have been created using either C++ or iIsh.

iIsh is an interactive Inventor shell that uses the Tcl interpreter as its kernel. The Tcl interpreter was extended to included a number of commands that allowed the user to make use of Inventor primitives. It was also extended to allow the user to create the instantiations of the Vo classes.

Some of the systems were implemented using iIsh as it provided the ability to easily create visualisation by non-C++ programmers. However, as Tcl is interpreted and because of the internal way numbers are stored, computationally intensive programs are very inefficient. For this reason we decided to implement some of the systems in C++ only.

5.2 Search for extra-galactic super-structures

Study of extra-galactic structures involves the search for what are known as super-galactic structures. These structures are recognized by regions in the universe where there is higher density of galaxies as opposed to the rest of the universe. Previous techniques for discovering these structures involved taking the data and generating 2D plots that represented wedges, or slices, out of the data. This meant that the 3D aspect of the data was lost.

By employing visualisation techniques the data can be interactively explored. Although present techniques still limited the data to being displayed in 2D, the animation and interactivity gave the data a 3D feel.

5.2.1 The Data used

The positions and velocities of over 14000 galaxies, visible from the southern hemisphere, have been collected over the past 30 years. This data gives the position of galaxies in red-shift space, as opposed to 3D space.

Red shifted space coordinates are in the form of declination, azimuth and z-factor. The z-factor determines the red shift which gives the galaxy's velocity relative to Earth.

Although there is a correlation between red shift space and 3D space, by convention all investigation are done in red shift space due to problems in converting from one to the other.

Different super-structures have been highlighted by defining the volume that they occupy and then coloring all galaxies within the volume a different color.

5.2.2 How the classes were used

The VoPoints class was used to display the data. This class was chosen because of the natural association between the concept of a single galaxy and a simple point in 3D space.

Each of the super-structures was associated with its own instantiation of the VoPoints class. This allows for them to be easily given a unique color. It also allowed each one to be enclosed within a convex polyhedron. All galaxies that did not fall within a super-structure were allocated to another instantiation of a VoPoints class.

By placing the different structures in a separate instantiations of a class the points were separated into groups that could be manipulated as single objects. This allowed the structures to be hidden or displayed, or the color of the structures to be changed, this allowed the structures to be manipulated and made the systems more flexible.

5.2.3 Results

This system was used by Prof. Fairall of the University of Cape Town for investigation of large scale structures in the galaxies visible from the southern hemisphere. Using this system he was

able to determine that there was a cellular substructure [15] within the super structure that had never been seen before. This substructure only became visible when the data could be interactively manipulated and observed.

The system also has lead Prof. Fairall to believe that two previously unrelated structures, one visible from the southern hemisphere and the other visible from the northern hemisphere *could* be in fact two parts of one larger super-structure. However, more investigation is needed before this can be verified.

The visualisation were also used to create all-sky projections. All-sky projections are sets of 6 slides that are projected onto a planetarium dome to create hemispherical view of the night sky. This was, to the best of our knowledge, the first time an all-sky projection of the galactic night sky had ever been created. Copies of these slides have been used in some of the major planetaria around the world, amongst others the Munich and London Planetaria.

Another set of all-sky projections were created with each of the galaxies color coded according to distance from the Earth, more distant galaxies were coded blue while the closer galaxies were coded red. When this was viewed through chromatic glasses they produce a 3D image, with the blue galaxies appearing to be more distant than the red ones. This made it possible to use an all-sky projection to view the galactic super-structures with their accompanying cellular sub-structure.

5.2.4 Implementation

C++ Implementation

The implementation using C++ and Inventor involved opening the file, creating the data structures and then passing this information to Inventor. We then had to create a scenegraph that would describe the scene, pass this scenegraph to a viewer and call the Inventor event handler.

The data is read from a file by a function, *ReadData*. This function read in a number of attributes that define each galaxies position and these then get converted to an (x, y, z) tupple that defines its position in 3D space. These tupples are stored in a the class *GalaxyList*.

The creation of the scenegraph is relatively simple, it involves making a SoPointSet and SoCoordinate3 nodes the children of a SoSeparator (see 2.6).

```
SoSeparator *
CreateScenegraph() {
   SoSeparator *sep = new SoSeparator;
   SoPointSet *pointset = new SoPointSet;
   SoCoordinate3 *coordnode = new SoCoordinate3;
   SoMaterial *mat = new SoMaterial;
```

We create a a SoSeparator, which is going to be the root of the scene graph, and then an SoPointSet, an SoCoordinate3 and an SoMaterial node. The SoCoordinate3 store all the individual locations

of the galaxies and the SoPointSet uses these coordinates as individual point that must be plotted. The SoMaterial node define the appearance of all the points.

```
sep -> addChild( mat );
sep -> addChild( coordnode );
sep -> addChild( pointset );
```

This section of code creates the scenegraph, each of the three nodes are added to the separator node *sep* as a child. The order in which they are added is important. The *pointset* must be last as it needs the data contained in the other two nodes to render correctly. The order in which *mat* and *coordnode* are added is not important.

```
coordnode -> point = GalaxyList;
```

The tupples that were read are now stored Within the SoCoordinate3 node *coordnode*.

```
mat -> emissiveColor.setValue(255,255,255);
```

The color of the points is set by defining the RGB value of the color, in this case it is set to pure white. The *emissiveColor* is set as the points do not consist out of faces, and thus can not reflect light but must emit light if they are to be seen.

```
pointset -> startIndex.setValue(0);
pointset -> numPoints.setValue(GalaxyList -> point.getNum());
```

We now tell the SoPointSet *pointset* how many points it must use, in this case all the points that are stored in *GalaxyList*.

```
   return sep;
}
```

We then return a pointer to the root node of the scenegraph.

Now that we have read in the data and created the scenegraph, we need to create a viewer and pass the scenegraph to the viewer.

```
void
OpenViewer() {
   Widget appWindow = SoXt::init("Universe");
   if ( appWindow == NULL ) exit( 1 );

   ReadData("SIZEPLOT.ALL");
```

First we need to create an X-Windows application. We create a new application window and check if it was created correctly, if not we exit. Next we read the data from *SIZEPLOT.ALL*.

```
SoSeparator *Mainroot;
Mainroot = CreateScenegraph();
```

We generate the scene graph by calling the function *CreateScenegraph* which returns a pointer to the root of a scenegraph.

```
TopNode = new SoSelection;
TopNode -> addChild(Mainroot);
```

The newly created scenegraph is the added as a child to *TopNode*. *TopNode* is of type SoSelection, this node is used if one wishes to be able to do picking on the scene graph.

```
viewer = new SoSceneViewer(TopNode);
viewer -> setHeadlight(FALSE);
viewer -> setDrawStyle(SoXtViewer::VIEW_LINE);
viewer -> setSize(SbVec2s(736+56,578+54));
(void) viewer->build( appWindow );
viewer->show();
```

We now create a viewer, a viewer consists of render area, buttons and controls as well different interaction techniques. The interaction techniques define how one manipulates the scenegraph.

We then turn the headlight off, the headlight is a light connected to the camera and always points in the same direction as which the camera is pointing. We also change the draw style so that no shading is done and then set the size of the viewer by defining its width and height in pixels.

We then tell the viewer to build and show itself.

```
SoXt::show( appWindow );
SoXt::mainLoop();
}
```

Finally we tell the application to show itself and then go into the main event loop which handles all the X-Windows and Inventor events.

C++ Implementation using VoPoints

When using the VoPoints class the class handles the creation of the scenegraph, therefore the reading in of the data is the same as the above example.

It is only the creation of the scenegraph that changes. The VoPoints class does this all automatically, therefore the *CreateScenegraph* function falls away and is replaced by the *GetScenegraph* method of the VoPoints class.

```
void
OpenViewer() {
   VoPoints *Points = new VoPoints("Universe Points");
   if (Points == NULL)  exit(1);
```

We create an instantiation of the class and pass it a name that is used later if the callback functions are used. This allows one callback functions to uniquely identify which class called it and then take appropriate action if it is necessary.

```
   Widget appWindow = SoXt::init("Universe");
   if ( appWindow == NULL ) exit( 1 );

   ReadData("SIZEPLOT.ALL");

   Points -> SetPoints(GalaxyList);
```

As described above we create the application window and read in the data. We then pass the list of points to the VoPoints class.

```
   TopNode = new SoSelection;
   Mainroot = Points -> GetSoGraph();
```

Where previously we had to write the function to generate the scene graph, the scene graph generation is now done by the VoPoints class.

```
   TopNode -> addChild(Mainroot);

   viewer = new SoSceneViewer(TopNode);
   viewer -> setHeadlight(FALSE);
   viewer -> setDrawStyle(SoXtViewer::VIEW_LINE);
   viewer -> setSize(SbVec2s(736+56,578+54));
   (void) viewer->build( appWindow );
   viewer->show();

   SoXt::show( appWindow );
   SoXt::mainLoop();
}
```

The rest of the implementation is the same as above.

iIsh Implementation

The following implementation reads the file *SIZEPLOT.ALL* and then uses the class VoPoints to create a point cloud. The body of the function converts the points from polar coordinates to 3D coordinates.

```
set datafile [open "SIZEPLOT.ALL" r]
```

We first open the data file *SIZEPLOT.ALL* for reading.

```
set coords ""
set count 0
set pi 3.141592

# Convert the coordinates from polar coordinates to 3D coordinates.
while {[gets $datafile line] >= 0} {
  set alpha [lindex $line 0]
  set delta [lindex $line 1]
  set zee [lindex $line 2]
  set temp [expr $zee * cos($delta*$pi/180) * cos(15*$alpha*$pi/180)]
  lappend temp [expr $zee * cos($delta*$pi/180) * sin(15*$alpha*$pi/180)]
  lappend temp [expr $zee * sin($delta*$pi/180)]
  lappend coords $temp
  incr count
}
```

The above lines replace the ReadData function, this reads in the data and then converts it to the (x, y, z) tupple. Each tupple is stored as a list and the whole data set is stored as a list of lists in the variable *coords*.

```
ivCreate VoPoints galaxy
ivVoPointsSetPoints galaxy $coords
```

We now create the VoPoints class and give it the name *galaxy*. We then pass it the variable *coords* which is a list of all the points.

```
ivCreate SoSeparator root
ivSetSceneGraph root
ivAddChildToGroup root galaxy
```

We now create the node *root* which we set to be the root node of the scenegraph. We then add *galaxy* to it as a child of *root*. While this may now appear that we have added the VoPoints class

into the scene graph, what in fact happens is that iIsh detects the *galaxy* is not a normal node, and then gets is to generate a scene graph which is then added to *root*.

The last implementation, using iIsh, is shorter and easier to understand. In this example this is the best solution as the amount of time spent processing the data is negligible in comparison to the time spent viewing the data. In addition to this the data set is static; the processing of all the data and generation of the scenegraph is all done beforehand.

5.3 Stock market visualisation

The data generated by the stock markets around the world runs into the megabytes. The analysis of this data relies heavily on statistical models where the data is reduced to more manageable proportions[21].

By its very nature of reducing the data, detail is lost. What we have hoped to achieve is to still have the all the data including the detail but presented in such a way that not only the overall trends can be seen but also on closer inspection, the more detailed information.

5.3.1 Data

The data was obtained from a local feed to one of our university computers. It has information going back 5 years for the Johannesburg Stock Exchange.

We extracted the data for the mining sector for the period from 1 June 1990 to 31 December 1990. We then visualized the data in a number of different ways, all of them making use of the classes we had created.

The following are different views that were created:

- Mesh View
- Square Mesh View
- Ribbon View
- Cube View

Mesh View

The mesh view used a VoMesh with the different shares along the y axis and time progressing along the x axis. The height, or z axis, was used to represent some aspect of the data, either the price normalised relative to a certain date or the percentage change in price from one day to the next.

Each vertex of the mesh was colored according to the volume traded on that day. To colour the vertices we used hue, saturation and value to define the color, by associating the volume traded to the hue we obtained the largest color variation.

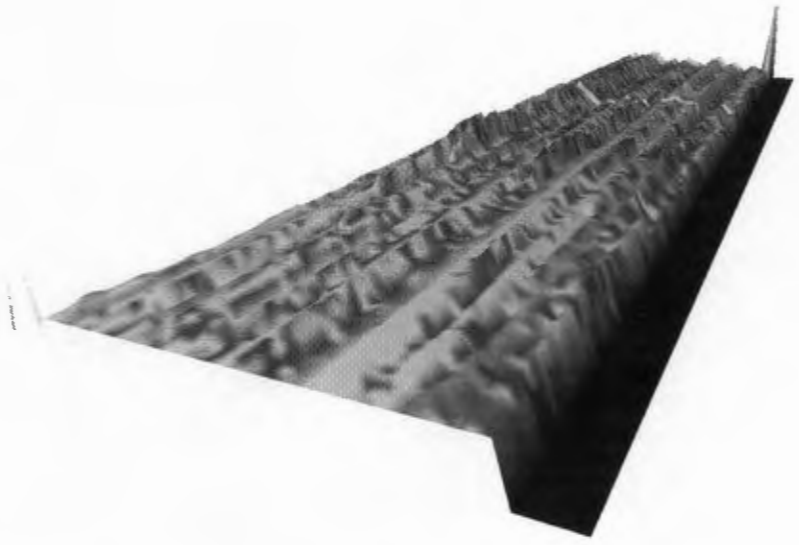


Figure 17: The VoMesh class used in the Stock Market visualisation.

Square Mesh View

The data layout of the square mesh was identical to that of the Mesh View. The difference being that unlike the Mesh View where each data point is represented by a vertex, in the Square Mesh View each data point is represented by a square horizontal face.

This has the advantage that, unlike the mesh view, there is no interpolation of colors or height between the data points. The interpolation can be misleading in the effect that it implies a relationship where none need exist.

Ribbon View

The ribbon view used the VoRibbons class, with each ribbon representing a share and the vertices within each ribbon representing the data for a single day.

As with the Mesh View and the Square Mesh View the height of each vertex represents the data for a single day.

Cube View

The cube view used the VoCube and VoLines classes, Figure 20. Each cube represented the data for a particular day, with the color of the data representing the volume traded for that particular day. The height of the cubes represented either the normalised price or the percentage change in price from one day to the next.

The dimensions of the cube can be used to represent three other pieces of data per day. Unfortunately the data feed did not have sufficient data to usefully utilize these additional dimensions in

Figure 18: The VoSquareMesh class used in the Stock Market visualisation.

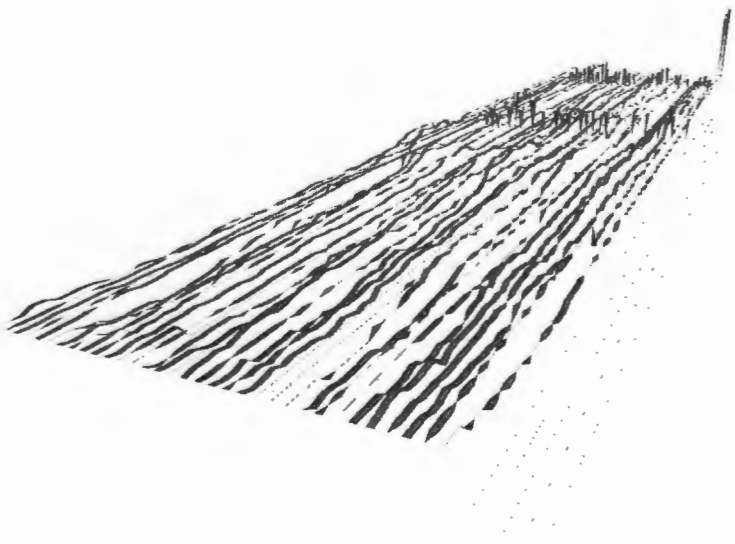


Figure 19: The VoRibbon class used in the Stock Market visualisation.

the data space. In order to illustrate the way this can be employed we used random data.

The VoLines were used to connect the all the cubes that refer to the same share together so that all the information for a particular share can be clearly grouped together. The lines also show very clearly whether there was an increase from one day to the next. This can become obscured when both the height and vertical position of a cube are changed simultaneously.

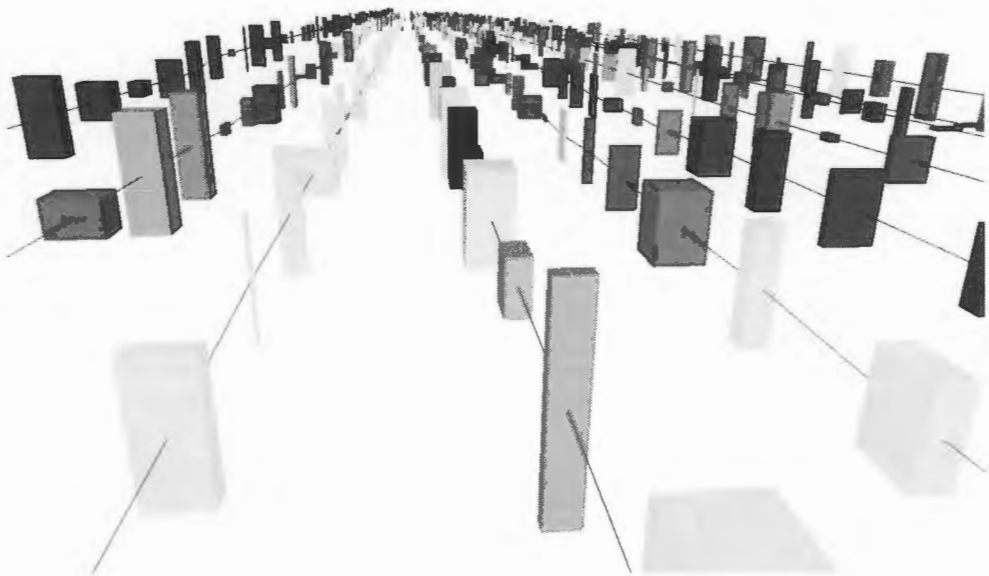


Figure 20: The VoCube class used in the Stock Market visualisation.

5.3.2 Results

The system was a prototype and used to investigate the ideas and principles of visualisation. As such it was not presented to stock market analysts for a proper evaluation.

We, however, did look at the mining sector from the Johannesburg Stock Exchange and discovered that there were shares that clearly showed a positive price correlation, while others showed a possible negative correlation. These can be seen in the Figure 21.

We also implemented a "water level" facility that placed a blue plane at a height of 1.0 in the Mesh and Square Mesh Views. The price of each share could be normalised with respect to its price on a given day. This effectively showed whether the price of the share had risen or fallen. All shares that were above the water level appeared as "islands" while share that were below the water level were obscured by the plane. This allowed us to very easily see which share prices had risen and which had fallen.

This sort of visualisation can allow us to easily see patterns in the share prices. However, unless it is associated with a sufficiently rich database that includes detail company information, macro- and micro-economic indicators, news items, and any other information that has an impact on share prices; it is difficult to turn these patterns into theories that can then be tested.

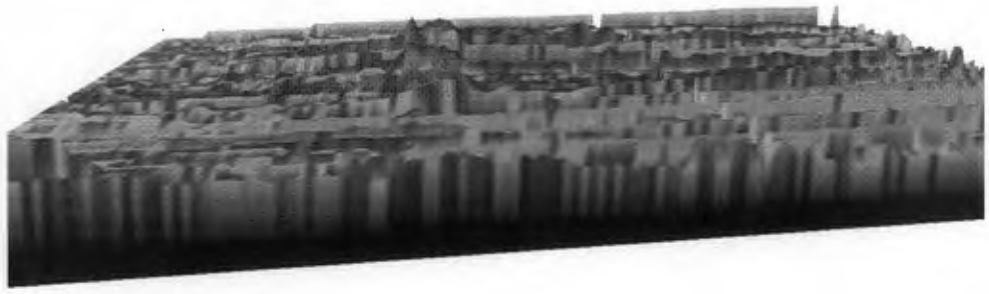


Figure 21: Correlation between a number of shares.

5.3.3 Implementation

Due to the amount of data processing that is performed on the data and the volume of the data it was not feasible to implement this system in iIsh. Instead all implementation was only done in C++.

As iIsh is a scripting language that is interpreted, and the manner in which Tcl handles mathematical calculations, this resulted in the data processing being inefficient.

5.4 Production capacity planning in a manufacturing environment

Working in conjunction with MCBA, a local company that provides capacity planning software, we developed a front end to their system that allowed the data generated to be displayed in a graphic format.

Their software is based on the MRP-II system and produces a large amount of data that details the allocation of resources to achieve certain production levels. In the case of the rufcut capacity planning, which we investigated, the resources are given in terms of man-hours and machine-hours.

The factory consists out of a number of work stations. Each work station has a certain capacity in terms of the number of machine- and man-hours that it can deliver every day. This capacity need not remain constant but does tend to fluctuate as people go on leave, resign, or get relocated and machines are serviced or replaced.

The constraints on the system are therefore the number of machine- and man-hours available and the production required to fulfill the current orders. The software provided by MCBA attempts to

allocate the resources in such a way as to be able to produce the required output. However, it is not always possible to obtain a solution and this results in certain work stations being required to produce beyond their capacity.

Our system was designed to identify which work stations were over capacity and which orders were creating the problems. These orders could then be rescheduled or station that were under capacity could be employed to make up the deficit, if this were possible.

5.4.1 Data

The data generated by their system gives the capacity in machine hours for every machine as well as the required capacity to complete all the allotted jobs. The required and available man hours are also given.

A communication program was written to extract the information directly from their database and send it to the machine that was being used for doing the visualisation.

5.4.2 Results

The visualisation created to view this data used ribbons, line and cubes to create a view that allowed the user to easily see which machines were being over utilised and which were being under utilised.

Figure 22 shows a view of the visualisation. Along the one axis we have the different work centers and along the other the time, divided into days. The height represents the number of either man or machine hours.

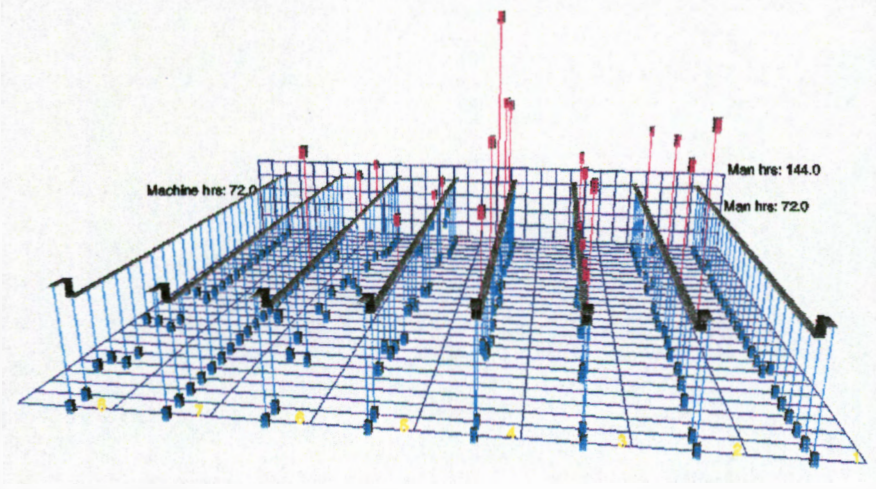


Figure 22: Rufcut capacity planning.

The ribbons represent the capacity of each work center while the cube shows the number of hours allocated to that work center for a particular day. The vertical line serves as a method of connecting the cube to the ribbon and thereby relating the two.

The coloring used allows the over-scheduled work centers to be clearly identified; if the time scheduled for a work center is greater than the capacity then the cube for that work center is colored magenta, otherwise it is colored cyan. This coloring allows the places where the allocated capacity is greater than the actual capacity to be easily identified.

From this visualisation it can be seen that a number of work centers have be allocated more capacity than they can handle. It is not enough to just be able to identify which centers have been over-scheduled, one also wants to know why this occurred.

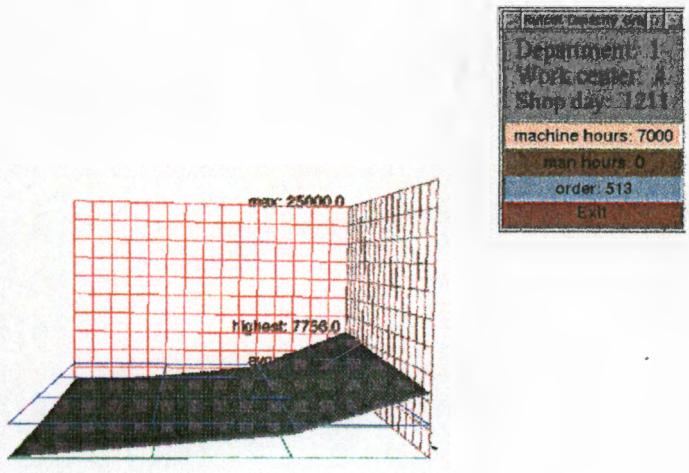


Figure 23: Detail of orders that make up the capacity for one work center.

By selecting any cube, the user is presented with a new visualisation that shows all the orders that contribute to the allocated time for that work center. The orders are displayed in the form of a grid with the largest being at one corner of the grid and the smallest at the opposite corner.

The user can select one of the vertices of the grid and will then be given information on which order it is and how many hours have been allocated to completing it. The user can then determine which orders need to be rescheduled in order to make the capacity plan feasible.

It can be confusing and distracting to have all information on the screen simultaneously, therefore when the user is examining the visualisation it is possible to hide all the information for work centers in which the allocated capacity is less than the actual capacity. This then makes it easy to see which work centers are potential problems, Figure 24 shows which work centers are potential problems.

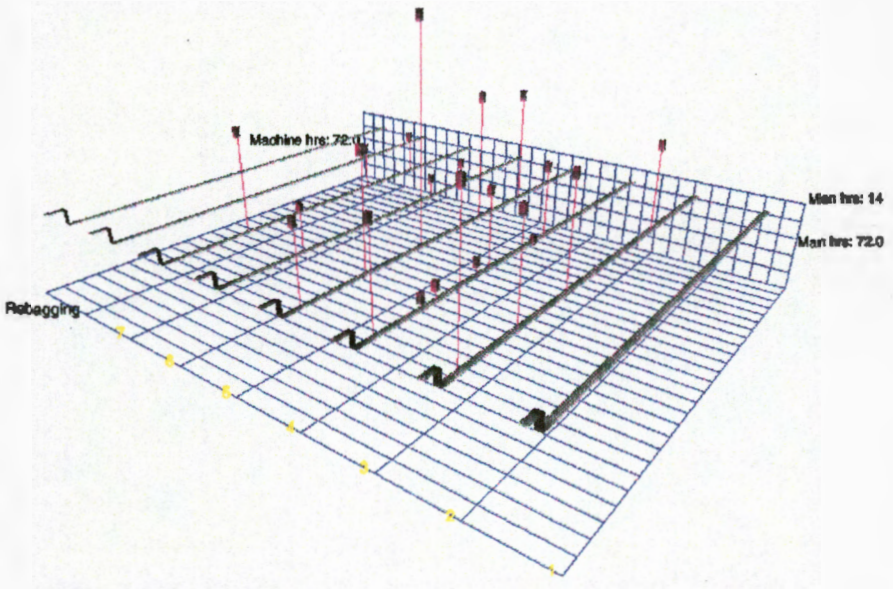


Figure 24: Rufcut capacity planning only showing the order that exceed capacity.

5.4.3 Implementation

This system was written in 3 months using ilsh by two students who had not done any graphics programming and were not familiar with Tcl. While the students did have some programming experience, it demonstrates the ease with which an interactive client-server visualisation, that also supports queries, can be created by relatively non-expert users.

Use of the classes

In the overall view of all the job centers, as shown in Figure 22, 4 main elements, namely a VoRibbon class, a VoCube class, a VoLines class and a VoGrid class have been used.

The VoRibbon class is used to create the ribbons that show the capacity of each work center. The VoCubes class creates the cubes that depict the planned capacity for each work center. The lines joining the two are created by the VoLines class by setting the primary and secondary data sets to the actual and allocated capacity of each work center.

A callback function was associated with the cube class that created the visualisation that gave more detail on a work center when a cube was selected.

The detailed work center visualisation, as seen in Figure 23 was made using the VoLines, VoMesh and VoGrid classes. The VoGrid created the surrounding grid, while the horizontal wire mesh that shows the average for all the orders was created using a VoLines class. The mesh was created using the VoMesh class and a callback routine was then associated with the VoMesh class that was called every time a vertex in the mesh was selected. The information for the vertex was then displayed in the window.

The whole system was written under iIsh. From the above description it can be seen that a relatively complex and informative visualisation was created by simply combining a number of simple building blocks.

Chapter 6

Conclusions

The factor limiting the extensive use of information visualisation is the inability of users to create visualisations and not the inability of information visualisation to assist the user. More often than not the creation of visualisations that are flexible requires an advanced knowledge of programming and graphics, skills that are not available to the average user of visualisations.

By encapsulating certain building blocks into C++ classes we have simplified the process of creating visualisations. However, knowledge is still required of C++ and the fields and classes that are used by Inventor. However, the user no longer requires a knowledge of the underlying 3D graphics that is need to create a visualisation. This still does not make the classes accessible to the average user but does at least eliminate the need for specialised 3D graphics knowledge.

The design of these building blocks allows for more complex visualisations to be created by combining them in creative ways. This method was found to be better than defining a set number of predetermined visualisations, as it gave the user more freedom as well as increasing the number of possible visualisations. This also fulfilled one of the requirements of our system, that it be flexible.

The classes where created in such a way as to strike a balance between the complex, but very flexible, options offered by Inventor and a simple, but more restricted, system that is easier to use and that has a fairly flat learning curve.

iIsh [10, 1] further simplifies the creation process by not requiring the user to have any knowledge of C++. It is an interactive system whereby visualisation can now be created using a much simpler scripting language. iIsh still uses many of the constructs and syntax of formal programming languages, but visual programming interfaces are being developed which require the user to have less and less knowledge of programming.

This frees the user from the need to know how to program, and allows them to concentrate all their efforts in making and using visualisations.

This suite of classes is not presented as a definitive list, but is being updated and improved. As we gain more experience in the field of information visualisation we become more aware of extensions that can be made.

6.1 Simplifying the creation of visualisations

In the case studies we looked at a number of system and demonstrated the ways in which the classes simplify the creation of visualisations. The visualisation of the extra-galactic super-structures showed that using iIsh greatly simplified the process of creating a visualisation as opposed to using straight Inventor and C++.

In the stock market visualisation we were able to create a number of views of the data quickly. More complex visualisations, like the Cube View, were easily created by combining two classes.

The ability to query the visualisation to find out more details about a particular data item, which in this case was the trading information of a share on a particular day, was also easily achieved. The process involved creating a callback function that would get detailed information from the database given a share and a date.

The visualisation of the capacity planning data demonstrated that by combining a number of classes, complex visualisation could be created by non-expert users in a short period of time. The system included both the graphical display of the data as well the ability to query the underlying data and obtain more information. In this system the more detailed information was in the form of another visualisation, which could also be queried to give alpha-numeric information.

Our experience in developing these systems was that when one is able to concentrate on the problem of visualising the data and not on how to implement the visualisation, then the process of creating visualisations became easier.

6.2 Classes

In developing different visualisations we discovered that certain classes lend themselves more readily to visualising data with certain properties. In the following section we discuss each of the classes we have created and what type of data can be best visualised with each class.

6.2.1 VoMesh

The VoMesh class implies a relationship between each data point and the four data points adjacent to it by joining the points with a line and also interpolating the colors between the points. In addition to this, because the colors and height are interpolated, the data should be continuous as opposed to discrete.

6.2.2 VoSquareMesh

The VoSquareMesh is suited for discrete data sets because neither the colors nor the heights are interpolated.

6.2.3 VoRibbon

If the data set consists of a discrete group of continuous data sets then the VoRibbon should be used. Each ribbon is a discrete group of continuous data that can be approximated by a number of discrete points.

An example of where this is most applicable is in the visualisation of the stock market. Each share is associated with a ribbon and then each ribbon contains the data for a certain time period. While in this case the data for each day is discrete, insofar as only the closing price is known, it is assumed to be a reflection of the overall trading on that day, in other words the continuous price fluctuations are approximated by one sample per day.

6.2.4 VoCube

This class is suitable for data sets where there is no relationship between the elements in the data set, and each element in the data set has more than 2 distinguishing characteristics. The 2D equivalent of the VoCube class would be a scatter plot.

6.2.5 VoPoints

The VoPoints can be used when the data set has the following characteristics:

- The data set consists of a number of positions; these positions can be in either 2D or 3D.
- Each data element only has one other characteristic that can be easily associated with a color.
- The data set is large.

This class then provides a method of visualising this type of data set. However, because each data point is represented by a single point the 3D perspective is lost once the scene becomes stationary. Therefore this form of visualisation is really only applicable in cases where the visualisation has to be fully interactive.

6.2.6 VoLines

The VoLines is more often used not as a visualisation class on its own but as a means of showing associations between data elements within one class or between data elements in different classes.

This can be clearly seen by the way the VoLines class was used to join the cube in the Cube View in the stock market visualisation. In the rough cut capacity planning example we can see the way the lines were used to join the relevant data points in the VoRibbon class with those in the VoCube class.

6.2.7 VoVectorField

The VoVectorField can be used when the data set exhibits the following properties:

- Each data element in the data set has a direction and one more characteristic that can be associated with a color.
- Large data set.

6.2.8 VoSpiral

If the given data set consists of a number of elements and their relationship to one another then this class the best one to use. However, if the relationship is bidirectional then the cones depicting the relationship will intersect and the visualisation can become confusing and therefore be less effective.

6.2.9 SoDetailLevel

The level of detail class allowed us to display text information, and other fine detail information. By using this class it was possible to have the information be automatically display only once it became relevant. This allowed us to create visualisations that contained fine detail, however this fine detail did not become noise when the whole visualisation was viewed and it was no longer relevant.

6.3 Future Work

These classes are dynamic and changing and new ideas and concepts are continually being developed. In conclusion some future changes that are envisioned are outlined.

6.3.1 New Classes

One of our investigations revealed the need for classes that allow the user to visualise data feeds, in other words, non-static data sets. Such a class will have to incorporate some form of time dependent animation to display these data sets. Inventor 2.0 already includes nodes that are time dependent and can be used to create animations, these will be employed to create this new class.

6.3.2 Class Extensions

The one obvious extension to the classes is to extend the interaction to include the ability to not only query but also interact with the data points. This will allow the user to select a data

point and manipulate it in a predetermined manner; this feedback can then be used to modify the visualisation appropriately.

This form of interaction will be the foundation for implementing graphical what-if analysis. By being able to select a data point the user can modify it interactively and discover the affects of such modifications on the rest of the data set.

The SoDetailLevel class can also be extended to include a number of different ways to determine when different levels of detail should be used. At the moment this is entirely dependent on the distance between the object and the camera. Another suggested method can take into account the size of the image that the rendered scene graph will create and then, based upon this, determine the level of detail to use.

Appendix A

Visualisation Objects

Conceptually what we wished to create with our visualisation objects was a class of objects that could encapsulate the Inventor implementation and present the user with a common interface into all the classes.

The interface consists out of some basic functions and can be broadly classified as follows:

- Setup Functions
- Generating the Scene Graph
- Modifying the Scene Graph

Setup Functions

These functions are the functions that are need to provide the Visualisation Object with the initial information it needs. None of these functions need be called, with the exception of the constructor, as the Visualisation Object inserts default values where there is insufficient data.

Some of the objects provide facilities whereby if the user selects that object then a user defined call back function can be called. If the user wished to make use of this facilities the call back functions as well as the SoSelection node must be passed to the Visualisation Object.

The dimensions and number of data items also need to be passed to the object and this needs to be done before the scene is generated.

Generating the Scene Graph

The scene graph is generated in all cases by the 'GetSoGraph' function. This function returns a node of type **SoSeparator**. This can then be inserted into a scene graph and/or passed to a viewer.

Changing the Scene Graph

Once the scene graph has been rendered and the user is able to select points in the scene graph it often becomes necessary to change characteristics of that scene graph. It would be not only tedious, but also a waste of time, if we had to regenerate the scene graph every time. In order to by pass this functions have been provided with each class that allows the user to modify aspects of the scene graph.

These functions vary from scene graph to scene graph but they essentially all involve changing the position, dimensions and color of a specific data item.

A.1 Class Descriptions

As Inventor uses the prefix 'So', standing for Scene Object, for all its classes we have decided to use the prefix 'Vo', standing for Visualisation Objects.

To date the following different Visualisation Objects have been implemented:

- VoBase
- VoMesh
- VoSquareMesh
- VoRibbon
- VoCube
- VoPoints
- VoLines
- VoVectorField
- VoGrid
- VoSwap
- VoScale

A.1.1 VoBase

This is the base class from which all the Visualisation Objects inherit. It contains the virtual function GetSoGraph() that returns a pointer to an SoSeparator node that contains the scene graph for that Visualisation Object.

The SetSelectionNode function stores the selection node that is needed to attach the selection callback function. A scene graph can only contain one SoSelection node, this makes it necessary for the user to create a SoSelection node and inform the class.

The base class also contains two label nodes, one that gives the name of the class, this is stored in VoLabel. The other is the LabelNode which gives the user the opportunity to store a label for each instantiation of the subclass. This label is then passed to the call back functions in order that the user can identify which Visualisation Object was selected.

A.1.2 VoMesh

The VoMesh is a rectangular mesh that uses the user defined heights and colors for each vertex. This allows the user to easily generate a landscape type of view. When a vertex in the grid is selected then a user defined callback is called. The row, column and user defined label are passed to the callback function.

Attribute	Value
Rows	10
Columns	10

VoMesh(char *name)
~VoMesh()

The constructor takes as a parameter a string that is used to identify this instantiation of the class. When the user defined call back is called then this string is passed to it so that that it is possible for the user to uniquely identify which VoMesh was selected.

void SetSelectionCallback(void f(char *,int,int))
void SetDeselectionCallback(void f(char *,int,int))

This allows the user to define call back function that are called every time the VoMesh is selected. The call back functions are passed as parameters; the name defined when the constructor was called and the row and column of the vertex that was selected.

The Deselection call back function is called when the point is deselected. This is usefully if the point has been highlighted with a color, and we wish to restore the original color when a new point is selected.

void SetRows(int rows)
void SetColumns(int columns)

This allows the dimensions of the VoMesh to be set. Changing these after the scene graph has been generated has no effect. To have an effect the scene graph needs to be regenerated.

void SetHeights(SoMFFloat &data)

This sets the heights for all the vertices. The data is passed in as a SoMFFloat, which is an Inventor class that can store multiple floating point values. The values must place in the SoMFFloat in row order.

If the number of values provided is less than the product of the number of rows and the number of columns then the missing values are assumed to be zero. If, on the other hand, more values than are necessary are provided then any extra values are ignored.

void SetColors(SoMFColor &color)

Like the HeightData this sets the color of each vertex, the data is stored in an Inventor SoMFColor in row order.

Should there be to few color values then the missing values are assumed to be white, more values that necessary results in the extra values being ignored.

SoSeparator *GetSoGraph()

This function uses the data passed to it by the previous functions to generate a rectangular grid with each vertex having the height and color specified by the user. This function returns a pointer of type **SoSeparator** that can be inserted into a scene graph. It also links the call back functions to the Selection node that has been specified by calling the function **SetSelectionNode** in VoBase.

void SetVertexColor(int row, int column, SbColor color)

This sets the color of the vertex at the specified row and column to the color given. This should only be called after **GetSoGraph()** has been called and the scene graph has been generated. Calling this function before the scene graph has been generated results in the changes being ignored.

void SetVertexHeight(int row, int column, float height)

This function changes the height of a the vertex specified by the row and column to the height given. As with the **SetVertexColor()** calling this function before the scene graph has been generated results in the changes being ignored.

A.1.3 VoSquareMesh

The VoSquareMesh is an modification of the VoMesh. The difference lies in the fact that instead of each data point being represent by a vertex it is now represented as a square face.

This allows the user to generate what can be seen to be 3D bar graphs. As with the VoMesh when a face is selected and deselected then a user defined call back functions are called.

VoSquareMesh(char *name)
~VoSquareMesh()

The constructor takes as a parameter a string that is used to identify this instantiation of the class. When the user defined call back is called then this string is passed to it so that that it is possible for the user to uniquely identify which VoSquareMesh was selected.

void SetSelectionCallback(void f(char *,int,int))
void SetDeselectionCallback(void f(char *,int,int))

This allows the user to define call back functions that are called every time a VoSquareMesh is selected. The call back functions are passed as parameters; the name defined when the constructor was called and the row and column of the face that was selected.

void SetRows(int rows)
void SetColumns(int columns)

This allows the dimensions of the VoSquareMesh to be set. Changing these after the scene graph has been generated has no effect. To have an effect the scene graph needs to be regenerated.

void SetHeights(SoMFFloat &data)

This function specifies the heights for each of the faces that make up the VoSquareMesh. As with the VoMesh the heights are stored in an SoMFFloat in row order.

If the number of floats in the SoMFFloat are less than the product of the number of rows and the number of columns then the missing values are assumed to be zero. If extra values are specified then the extra values are ignored.

void SetColors(SoMFColor &color)

All the colors for each face is stored in an SoMFColor, the colors must be stored in row order.

Should there not be enough colors then the missing colors are assumed to be white, similarly if there are to many colors specified then the extra colors are ignored.

SoSeparator *GetSoGraph()

This function uses the data passed to it by the previous functions to generate a rectangular grid with each face having the height and color specified by the user. This function returns a pointer of type **SoSeparator** that can be inserted into a scene graph. It also links the call back functions to the SoSelection node that has been specified by calling the function **SetSelectionNode** in VoBase.

void SetFaceColor(int row, int column, SbColor color)

This function allows the user to change the color of a single face. If this function is called before the scene graph is generated then the changes are ignored.

void SetFaceHeight(int row, int column, float height)

This function allows the user to change the height of a single face. As in the case of the SetFaceColor if it is called before the scene graph is generated then the changes are ignored.

A.1.4 VoRibbon

The VoRibbon allows the user to define a group of ribbons by giving their height, color and width at each vertex.

When the user selects a ribbon a user defined call back function is called with the name specified in the constructor and the ribbon and vertex number of the vertex that was selected.

The user is also able to move the selected ribbon around by selecting a ribbon and then using the mouse and the middle mouse button to specify where the ribbon should be placed. The new position of the ribbon is such that the ribbon is still parallel and in line with its previous position.

VoRibbon(char *name)
~VoRibbon()

The constructor takes as a parameter a string that is used to identify this instantiation of the class. When the user defined call back is called then this string is passed to it so that that it is possible for the user to uniquely identify which VoRibbon was selected.

void SetSelectionCallback(void f(char *,int,int))
void SetDeselectionCallback(void f(char *,int,int))

This allows the user to define call back functions that are called every time a VoRibbon is selected. The call back functions are passed the following values as parameters: the name defined when the constructor was called and the ribbon and the vertex that was selected.

void SetRibbons(int ribbons)
void SetRibbonVertices(int ribbonvertices)

This tells the class the number of ribbons and the number of vertices that make up each ribbon. The product of these two gives the number of data points needed.

void SetCamera(SoCamera *cam)

This specifies the camera that Inventor will use to render the scene graph. This is needed to calculate where the ribbon should be moved to, should no camera be given then it is impossible to move the ribbons.

void HeightData(SoMFFloat &data)

The data for the heights for each vertex of each ribbon is stored in an SoMFFloat. The data must be stored in ribbon order, that is all the data for the first ribbon and then all the data for the second ribbon, etc.

If insufficient data is provided then the missing data is assumed to be zero.

void SetColors(SoMFColor &color)

The colors, like the height data, must be given in ribbon order. The colors are stored in an SoMFColor.

Any missing colors are assumed to be white, and any extra colors are ignored.

void SetWidths(SoMFFloat &data)

As with the colors and the heights the data must be in ribbon order and is stored in an SoMFFloat.

If insufficient data is provided the width for the missing vertices is assumed to be one.

SoSeparator *GetSoGraph();

This function uses the data passed to it by the previous functions to generate a group of ribbons with each vertex in the different ribbons having the height, width and color specified by the user. This function returns a pointer of type **SoSeparator** that can be inserted into a scene graph. It also links the call back functions to the Selection node that has been specified by calling the function 'SetSelectionNode' in VoBase.

void SetVertexColor(int ribbon, int ribbonvertex, SbColor color)

This sets the color for a particular vertex on a certain ribbon to the color given. If this function is called before the scene graph is generated then the change is ignored.

void SetVertexHeight(int ribbon, int ribbonvertex, float height)

This sets the height for a particular vertex on a certain ribbon to the height given. If this function is called before the scene graph is generated then the change is ignored.

void SetVertexWidth(int ribbon, int ribbonvertex, float width)

This sets the width for a particular vertex on a certain ribbon to the width given. If this function is called before the scene graph is generated then the change is ignored.

A.1.5 VoCube

The VoCube allows the user to define a rectangular region of cubes, the height and color of each cube can be specified. In addition the dimensions of the cubes can be changed. Each cube can have a different height, depth and width.

VoCube(char *name)
~VoCube()

The constructor takes as a parameter a string that is used to identify this instantiation of the class. When the user defined call back is called then this string is passed to it so that that it is possible for the user to uniquely identify which VoCube was selected.

void SetSelectionCallback(void f(char *,int,int))
void SetDeselectionCallback(void f(char *,int,int))

This allows the user to define call back function that are called every time a VoCube is selected. The parameters passed to the call back functions are: the name defined when the constructor was called and the row and column of the cube that was selected.

void SetRows()
void SetColumns()

This sets the number of rows and columns in the rectangular grid of cubes. Changing this value after the scene graph has been generated has no effect.

void SetHeights(SoMFFloat &heights)

This sets the heights of each cube, the heights for the cubes must be in row order. If the number of heights is less than the product of the rows and columns then the missing values are assumed to be zero.

void SetColors(SoMFColor &colors)

Each cube can be assigned a different color, the SoMFColor variable *colors* contains these values. As with the height values the color values must be in row order. If there are too few colors, the missing colors are assumed to be white.

void SetDimensions(SoMFVec3f &dim)

Each cube can have a different size. By default all the cubes are of size 1x1x1, but this can be change by supplying different dimensions. As with the colors and the heights the dimensions must be in row order. Any missing dimensions are assumed to be 1x1x1.

SoSeparator *GetSoGraph()

This function generates a scene graph from the data previously passed to it by the user. The cubes are laid out in a rectangular shape that has the dimensions specified by **SetRows()** and **SetColumns()**. This function also links the callback functions into the SoSelection node and then returns a pointer to an **SoSeparator** node that can be inserted Into a scene graph.

void Set1Height(int r, int c, float value)

This sets the height of a single cube in row r and column c.

void Set1Color(int r, int c, SbColor color)

This sets the color of a cube in row r and column c.

void Set1Dimension(int r, int c, SbVec3f dim)

Set the height, width and depth of a cube in row r and column c.

A.1.6 VoPoints

This class allows for the definition of various points in 3D space. All the points can have the same color or have an individual color assigned to them. The points can also be enclosed in a convex polyhedron. The points are passed to the class as an SoMFVec3f, and the color is specified with an SbColor or an SoMFColor.

VoPoints(char *name)
~VoPoints()

The constructor takes as a parameter a string that is used to identify this instantiation of the class. When the user defined callback function is called then this string is passed to it so that that it is possible for the user to uniquely identify which VoPoints was selected.

void SetSelectionCallback(void f(char *,int))
void SetDeselectionCallback(void f(char *,int))

This allows the user to define callback function that are called every time a VoPoints is selected. The parameters passed to the call back functions are: the name defined when the constructor was called and the index of the point that was selected.

void SetPoints(SoMFVec3f points)

The points are stored in an SoMFVec3f which gives the position of the point in 3D space.

void SetColor(SbColor color)

Set the colors for all the points. Using this method means that all points are set to one color.

void SetColor(SoMFColor color)

Set the colors for all the points. In this case a unique color needs to be specified for each point. If there are more points than colors given then the missing colors are assumed to be white.

void SetPolyhedronColor(SbColor color)

All the points can be encapsulated in a convex polyhedron. This function defines the color of the polyhedron.

void SetPolyhedronTransparency(float trans)

The polyhedron can be set to be semi-transparent. The value of *trans* defines how transparent the polyhedron is. The value of *trans* can vary between 0 and 1, with 1 being completely transparent and 0 being opaque.

SoSeparator *GetSoGraph()

This function generates a scene graph from the data previously passed to the class by the user. The points can be any where in 3D space as defined by the values passed to the class in the *SetPoints()* function.

This function also links the callback functions into the SoSelection node and then returns a pointer to an SoSeparator node that can be inserted into a scene graph.

void SetPolyhedronOn()
void SetPolyhedronOff()

The convex polyhedron can be toggle on and off both before and after the scene graph has been generated. These functions switch the polyhedron on and off respectively.

void Set1Point(int num, SbVec3f &point)

This changes the position of a single point, it changes the position of point *num* to be at position *point*.

void Set1Color(int num, SbColor &color)

This changes the color of a single point, it changes the color of point *num* to be *color*.

A.1.7 VoLines

This defines a class which allows you to join points to form line segments parallel to the X, Y, or Z axes. This allows the user to draw line graphs parallel to the X or Y axes, or to join two corresponding points in the primary and secondary data sets.

Like the VoCube, VoMesh, VoSquareMesh, etc. this class is aligned to a rectilinear grid.

The user is asked to give the class two sets of heights, the Primary data set and the Secondary data set. Lines can be drawn between points in adjacent rows and/or adjacent columns. In addition to this lines can be drawn between corresponding points in the two data sets.

There are two possible ways the lines can be colored. They are :

1. Each point in each data set can be given a color.
2. All the points in a data set can be given one color. If vertical lines are specified in this case then the color of the line is determined by the data set whose corresponding point is the highest.

If any height data is missing from either the primary or secondary data sets then it is assumed to be zero.

VoLines(char *name)
~VoLines()

The constructor takes as a parameter a string that is used to identify this instantiation of the class. When the user defined call back is called then this string is passed to it so that that it is possible for the user to uniquely identify which VoLines was selected.

void SetSelectionCallback(void f(char *,int,int,DataSetType dataset, void *userdata), void *userdata)
void SetDeselectionCallback(void f(char *,int,int,DataSetType dataset, void *userdata),void *userdata)

This allows the user to define call back function that are called every time a VoLines is selected. The parameters passed to the call back functions are: the name defined when the constructor was called, the row and column of the point selected, which data set the point lies in, the primary or the secondary, and a void pointer that the user can use to transfer any data.

void SetLinesToDraw(LinesToDrawType Lines,
DataSetType dataset = PRIMARY)

There are three different ways the points can be joined together, namely:

1	Join points in adjacent columns.	VoLines::BETWEEN_COLUMNS
2	Join points in adjacent rows.	VoLines::BETWEEN_ROWS
3	Join corresponding points in the data sets.	VoLines::BETWEEN_DATA_SETS

This value can be set independently for both data sets, of course setting the value of one data set to VoLines::BETWEEN_DATA_SETS results in both sets being joined.

At all time the data set must also be specified, if it is omitted then it is assumed that we are working with the primary data set.

void SetHeights(SoMFFloat &heights, DataSetType dataset = PRIMARY)

The heights for the two data sets are set independently. If the data set is not specified then it is assumed that the heights refer to the primary data set. The heights are stored in row order in an SoMFFloat.

void SetColor(SoMFColor &color, DataSetType dataset = PRIMARY)

This sets the color for each point in the data set. If no data set is given the it is assumed that the colors refer to the primary data set.

void SetColor(SbColor color, DataSetType dataset = PRIMARY)

If all points in the data set have the same color then a global color for the data set can be given. As with the above functions, if the data set is omitted then it is assumed to be the primary data set.

void SetRows(int rows)
void SetColumns(int columns)

Set the number of rows and columns that make up both data sets. It is a constraint that both data sets must have the same dimensions.

SoSeparator *GetSoGraph()

This generates the scene graph using the data given. It also attaches the call back functions to the **SoSelection** node.

A pointer to an SoSeparator node is returned which can be inserted into a scene graph or passed to a viewer.

void SetVertexColor(int row, int column, SbColor color,
DataSetType dataset = PRIMARY)

This sets the color of a particular vertex in either of the data sets, if the data set is not specified then it is assumed to be the primary data set.

void SetVertexHeight(int row, int column, float height,
DataSetType dataset = PRIMARY)

This sets the height of a particular vertex, if data set is omitted then it assumed to be the primary data set. If the change in height causes the point to be moved to the opposite side of the other data set then the color is changed appropriately.

A.1.8 VoVectorField

This defines a number of vectors that can be placed anywhere in 3D space, each vector can be given a direction, size and color. The vectors can be selected and the number of the vector is returned.

VoVectorField(char *name)
VoVectorField()

The constructor takes as a parameter a string that is used to identify this instantiation of the class. When the user defined call back is called then this string is passed to it so that that it is possible for the user to uniquely identify which VoLines was selected.

void SetVectorStarts(SoMFVec3f &pos)
void SetVectors(SoMFVec4f &vec)
void SetVectorColors(SoMFColor &color)

This sets the start positions of the vectors, the color of the vector as well as the direction and magnitude of the vector.

The values for the start position are are passed in in a SoMFVec3f to give the x, y and z coordinates.

The actual vector is represented by an SoMFVec4f with the first 3 values giving the direction and the fourth value is the magnitude of the vector. The colors are passed in an SoMFColor.

When the values are entered the maximum number of entries in each field is recorded and this is used as the number of vectors that are to be displayed.

SoSeparator *GetSoGraph()

This function uses the previously entered information to create a vector field. If not enough colors were given then the missing colors are assumed to be white. Missing positions are assumed to be zero as are missing direction and magnitude values.

Each vector is the either represented by an arrow or a sphere. If a magnitude and direction is given then the arrow of the appropriate size pointing in the direction given is drawn. If only a magnitude is given then the a sphere is drawn.

void SetSelectionCallback(void f(char *,int))
void SetDeselectionCallback(void f(char *,int))

This allows the user to define call back function that are called every time a vector in a VoVectorField is selected. The parameters passed to the call back functions are: the name defined when the constructor was called and the number of the vector selected.

void SetVectorPosition(int num, SbVec3f &vec)

Change the position of a vector.

void SetVectorColor(int num, SbColor &color)

Change the color of a vector.

void SetVectorDirection(int num, SbVec4f &vector)

Change the direction and magnitude of a vector.

A.1.9 VoSwap

This class allows the user to easily switch between various scene graphs, it can be likened to a *case* statement. The user places various scene graphs within the class and then with one function calls can switch between the different scene graphs.

VoSwap()
~VoSwap()

Unlike the other classes this class does not take an identifying string in its constructor as this class can never be selected.

SoSeparator *GetSoGraph()

This returns a scene graph that includes all scene graphs passed to this class. This function can be called before any scene graphs have been inserted into the class.

void InsertNode(SoNode *node)

This inserts a scene graph into the list of possible scene graphs. The new scene graph can consist of a whole scene graph or it can be a single valid Inventor node.

The scene graphs are inserted into a queue so that the first scene graph inserted is scene graph number zero, and the second is scene graph number two, etc.

void ChangeGraph(int num)

This changes the scene graph that will be rendered next time the whole scene graph is rendered. The number of the graph rendered depend on the value of *num*. If *num* is zero then the first scene graph is rendered, if *num* is one then the second is rendered, etc.

A.1.10 VoGrid

This class allows the user to create a grid by specifying the width and depth of the grid. Either the number of subdivisions in the grid may be specified or the distance between the line may be specified.

Two vertical grids can also be added to indicate height. A vertical grid can be position at either the top or bottom of the horizontal grid. A vertical grid can also be placed at the left of right of the horizontal grid.

The grid may be labeled with three sets of labels which are placed on any two adjoining sides, and vertically along the z-axis at either of the four corners of the horizontal grid.

The grid is designed to be able to give the viewer a reference point and to orient themself in 3D space. It also provides information as to the what the subdivisions of the axes represent.

VoGrid()
~VoGrid()

The constructor requires no parameters and sets up the default values as follows:

Attribute	Value
Width	10
Depth	10
Height	10
BaseHeight	0
Rows	10
Columns	10
Levels	10
Color(RGB)	1,1,1
XStep	1
YStep	1
ZStep	1
XLabelPosition	Bottom
YLabelPosition	Left
ZLabelPosition	TopRight
XVerticalGrid	Top
YVerticalGrid	Left

SetX(float x)
SetY(float y)
SetZ(float z) SetHeight(float z)

SetX() specifies how wide the grid is while SetY specifies how deep it is. SetZ() set the height of the vertical grids. SetHeight tells the horizontal grid at what height the grid must be drawn along the z-axis.

void SetXLabelPositions(VoGridPosition pos)
void SetYLabelPosition(VoGridPosition pos)
void SetZLabelPosition(VoGridPosition pos)

The labels can be positioned along any two adjoining axes. This means the the labels for the X axis can be along the top or the bottom, while the Y axis labels can be along the left or the right.

The labels for the Z axis can be situated at one of the four corners of the bottom grid, i.e. bottom-left, bottom-right, top-right or top-left.

VoGridPosition is and enum type that can have the following values: **VoGrid::Top**, **VoGrid::Bottom**, **VoGrid::Left**, **VoGrid::Right**, **VoGrid::TopLeft**, **VoGrid::TopRight**, **VoGrid::BottomLeft** and **VoGrid::BottomRight**. If the XLabelPosition is set to **VoGrid::Left** or **VoGrid::LabelRight** then the default value for XLabelPosition is used. This is true for YLabelPosition as well. ZLabelPosition can take on only the last four of the positions listed above, if another value is given then the labels are not displayed.

void SetXVerticalGrid(VoGridPosition pos)
void SetYVerticalGrid(VoGridPosition pos)

The X vertical grid can be either on the *Left* or on the *Right*, while the Y vertical grid can be either on the *Top* or *Bottom*. If either of the vertical grids are not needed, they can be hidden. This can be done by setting the position to be *Hide* and then the vertical grid is not displayed.

void ToggleXLabels()
void ToggleYLabels()
void ToggleZLables()
void ToggleLabels()

It is sometimes necessary to toggle the labels on and off to make the viewing of the scene clearer. These functions allows you to toggle the display of the X, Y or Z labels separately or as a group.

void SetXLabels(SoMFString &labels)
void SetYLabels(SoMFString &labels)
void SetZLabels(SoMFString &labels)

This sets the labels for the X, Y and Z axes. If the number of strings in *labels* is more than the number of subdivisions then the extra labels are ignored and in the case that the number of labels are two few then the remaining labels are assumed to be blank.

void Set1YLabel(int num,SbString label)
void Set1XLabel(int num,SbString label)
void Set1ZLabel(int num,SbString label)

If the user wishes to change a single label then the number of the label and the string to be used can be specified and the label is changed.

void SetXStep(float step)
void SetYStep(float step)
void SetZStep(float step)

This sets space between lines along the X, Y and Z axes.

void SetXNumOfLines(int columns)
void SetYNumOfLines(int rows)
void SetZNumOfLines(int rows)

This sets the number of lines along each of the axes. Setting these values causes the SetXStep(), SetYStep() and SetZStep() values to be ignored.

void SetGridColor(SbColor color)

This sets the color that is used when the grid is drawn.

SoSeparator *GetSoGraph()

This returns an SoSeparator node that contains the scene graph for the grid. The values set in the previous functions is used to generate the grid. If any values have been omitted then the default values are used.

A.1.11 VoScale

The VoScale class allows scene graphs to be scaled to fit into a predefined cubic area. The is useful if a variety of visualisations are going to be combined and one wants to ensure that they do not overlap one another.

VoScale()
~VoScale()

The constructor does not take a name parameter like other classes as it can never be selected.

SoSeparator *GetSoGraph()

This generates a scene graph that contains an SoTransform node that scales the scenegraph so that the bounding box of the scene graph fits into the specified cubic volume.

void SetCube(SbVec3f &cube)

The sets the size of the scenegraph.

void SetSubGraph(SoNode *node)

This sets the scene graph that has to be scaled.

Appendix B

Tcl Interface

B.1 iIsh

iIsh is an Interactive Inventor Shell written by Prof Goosen. This allows the user to create and display Inventor scene graphs using Tcl scripts. Extra commands have been added to the Tcl command interpreter to allow inventor objects to be created and command for the manipulation of these objects.

The **Vo** classes have then been added to the interpreter so that the methods of the different classes can be used as Tcl commands. The interaction is handled by replacing the callback functions with Tcl procedures.

To use the classes from out of Tcl scripts you need to know how to call the methods within each class. The following section describes the Tcl commands that can be used to create and modify the Visualisation Objects.

B.2 VoMesh

Description

Commands

ivCreate VoMesh *name*

This creates an VoMesh and associates it with the name *name*.

ivVoMeshSetRows *name number*

This sets the number of rows in the mesh with the name *name*.

ivVoMeshSetColumns *name number*

This sets the number of columns in the mesh with the name *name*.

ivVoMeshSetSelectionProc *name proc_name*

This sets the procedure that is called whenever a vertex in the mesh is selected. The procedure must take three parameters. The first parameter is the name of the mesh that was selected, and then the row and column of the vertex within that mesh that was selected.

An example of such a procedure would be:

```
proc select {name r c} {
   ivVoMeshSetVertexColor $name $r $c 1.0 0 0
}
```

This changes the color of the selected vertex to red.

ivVoMeshSetDeselectionProc *name proc_name*

This sets the procedure that would be called when a point is deselected, in other words when another point is selected. The parameter list and structure of the procedure is the same as in the case of the selection procedure.

This is an example of a deselection procedure:

```
proc deselect {name r c} {
   ivVoMeshSetVertexColor $name $r $c 1.0 0.4 0.1
   ivVoMeshSetVertexHeight $name $r $c 40
}
```

This changes the color of the deselected vertex to brown and sets the height of that vertex to 40.

ivVoMeshSetHeights *name list*

Set the heights for the different vertices in the mesh, the vertices must be given in row order, i.e. all the heights for the first row and then the heights for the second row.

ivVoMeshSetColors *name list*

The specifies a list that gives the color for each vertex of the mesh. To define each color we use a list of 3 values, these values represent the red, green and blue values in the range 0 to 1.

ivVoMeshSetVertexHeight *name row column height*

This allows you to change the height of a single vertex after the mesh has been displayed. The vertex is specified by giving the *row* and *column* numbers.

ivVoMeshSetVertexColor *name row column red green blue*

This set the color of a single vertex. As in the case of setting a single vertex height this can be done after the mesh has been displayed. The color is changed to the color specified by the *red*, *green* and *blue* values, these values must be in the range 0 to 1.

B.3 VoSquareMesh

Description

Commands

ivCreate VoSquareMesh *name*

This creates an VoSquareMesh and associates it with the name *name*.

ivVoSquareMeshSetRows *name number*

This sets the number of rows in the mesh with the name *name*.

ivVoSquareMeshSetColumns *name number*

This sets the number of columns in the mesh with the name *name*.

ivVoSquareMeshSetSelectionProc *name proc_name*

This sets the procedure that is called whenever a face in the mesh is selected. The procedure must take three parameters. The first parameter is the name of the square mesh that was selected, and then the row and column of the face within that mesh that was selected.

An example of such a procedure would be:

```
proc select {name r c} {
   ivVoSquareMeshSetFaceColor $name $r $c 1.0 0 0
}
```

This changes the color of the selected face to red.

ivVoSquareMeshSetDeselectionProc *name proc_name*

This sets the procedure that would be called when a face is deselected, in other words when another face is selected. The parameter list and structure of the procedure is the same as in the case of the selection procedure.

This is an example of a deselection procedure:

```
proc deselect {name r c} {
   ivVoSquareMeshSetFaceColor $name $r $c 1.0 0.4 0.1
   ivVoSquareMeshSetFaceHeight $name $r $c 40
}
```

This changes the color of the deselected face to brown and sets the height of that face to 40.

ivVoSquareMeshSetHeights *name list*

Set the heights for the different faces in the mesh, the faces must be given in row order, i.e. all the heights for the first row and then the heights for the second row.

ivVoSquareMeshSetColors *name list*

The specifies a list that gives the color for each face of the mesh. To define each color we use a list of 3 values, these values represent the red, green and blue values in the range 0 to 1.

ivVoSquareMeshSetFaceHeight *name row column height*

This allows you to change the height of a single face after the mesh has been displayed. The face is specified by giving the *row* and *column* numbers.

ivVoSquareMeshSetFaceColor *name row column red green blue*

This set the color of a single face. As in the case of setting a single face height this can be done after the mesh has been displayed. The color is changed to the color specified by the *red*, *green* and *blue* values, these values must be in the range 0 to 1.

B.4 VoRibbon

Description

Commands

ivCreate VoRibbon *name*

This creates an VoRibbon and associates it with the name *name*.

ivVoRibbonSetRibbons *name number*

This sets the number of rows in the ribbon set with the name *name*.

ivVoRibbonSetRibbonVertices *name number*

This sets the number of vertices in each ribbon in the ribbon set with the name *name*.

ivVoRibbonSetSelectionProc *name proc_name*

This sets the procedure that is called whenever a vertex in a ribbon is selected. The procedure must take three parameters. The first parameter is the name of the ribbon set that was selected, and then the ribbon and the vertex in ribbon that was selected.

An example of such a procedure would be:

```
proc select {name r c} {
   ivVoRibbonSetVertexColor $name $r $c 1.0 0 0
}
```

This changes the color of the selected vertex to red.

ivVoRibbonSetDeselectionProc *name proc_name*

This sets the procedure that would be called when a vertex is deselected, in other words when another vertex is selected. The parameter list and structure of the procedure is the same as in the case of the selection procedure.

This is an example of a deselection procedure:

```
proc deselect {name r c} {
   ivVoRibbonSetVertexColor $name $r $c 1.0 0.4 0.1
   ivVoRibbonSetVertexHeight $name $r $c 40
}
```

This changes the color of the deselected vertex to brown and sets the height of that vertex to 40.

ivVoRibbonSetHeights *name list*

Set the heights for the different vertices in the mesh, the faces must be given in row order, i.e. all the heights for the first row and then the heights for the second row.

ivVoRibbonSetWidths *name list*

Set the widths for the different vertices in the ribbons. The width are given as a list of floating point values. The values should be between 0 and 1 to avoid adjacent ribbons from overlapping. The values must be given in row order, i.e. all the width for the first row and then for the next row, etc.

ivVoRibbonSetColors *name list*

The specifies a list that gives the color for each vertex of the mesh. To define each color we use a list of 3 values, these values represent the red, green and blue values in the range 0 to 1.

ivVoRibbonSetVertexHeight *name row column height*

This allows you to change the height of a single vertex after the ribbon set has been displayed. The vertex is specified by giving the *ribbon* and *vertex* numbers.

This allows you to change the width of a single vertex after the ribbon set has been displayed. The vertex is specified by giving the *ribbon* and *vertex* numbers.

ivVoRibbonSetVertexColor *name row column red green blue*

This set the color of a single face. As in the case of setting a single face height this can be done after the mesh has been displayed. The color is changed to the color specified by the *red*, *green* and *blue* values, these values must be in the range 0 to 1.

B.5 VoPoints

Description

Commands

ivCreate VoPoints *name*

This creates an VoPoints and associates it with the name *name*.

ivVoPointsSetPoints *name list_of_points*

This sets the positions of the points. The points are stored as lists of 3 floating point numbers. These numbers are the position of the points in 3D space.

ivVoPointsSetColors *name list_of_colors*

This sets the colors for each point. The colors are defined by giving the red, green and blue values for each point. The values for the red, green and blue levels must be between 0 and 1.

ivVoPointsSet1Point *name number x y z*

This changes the position of a point. The *number* specifies the number of the point whose position must be changed to *(x,y,z)*.

ivVoPointsSet1Color *name number r g b*

This changes the color of a point. The *number* specifies the number of the point whose color must be changed and the *r, g and b* specify the red, green and blue values for the color.

ivVoRibbonSetSelectionProc *name proc_name*

This sets the procedure that is called whenever a point is selected. The procedure must take two parameters. The first parameter is the name of the point set that was selected and then second is the number of the point that was selected.

An example of such a procedure would be:

```
proc select {name num} {
   ivVoPointsSet1Color $name $num 1.0 0 0
}
```

This changes the color of the selected point to red.

ivVoPointsSetDeselectionProc *name proc_name*

This sets the procedure that would be called when a point is deselected, in other words when another point is selected. The parameter list and structure of the procedure is the same as in the case of the selection procedure.

This is an example of a deselection procedure:

```
proc deselect {name num} {
   ivVoPointsSetVertexColor $name $num 1.0 0.4 0.1
}
```

This changes the color of the deselected point to brown.

B.6 VoLines

Description

The

Commands

ivCreate VoLines *name*

This creates an VoLines and associates it with the name *name*.

ivVoLinesSetRows *name number*

This sets the number of rows in the line set with the name *name*.

ivVoLinesSetColumns *name number*

This sets the number of columns in the line set with the name *name*.

ivVoLinesSetHeights *name dataset list_of_heights*

This sets the heights in one of the two data sets. The command takes three parameters, the name associated with that line set, the dataset and a list of heights. The heights must be in row order, i.e. all the heights for the first row and the all the heights for the second row.

ivVoLinesSetColors *name dataset list_of_colors*

This sets the colors for the lines. The data set determines which data set the list of colors should be applied to. The *list_of_colors* consists of a list of lists. Each sub-list consist of 3 numbers which defines the red, green and blue values of each color.

ivVoLinesSetLinesToDraw *name dataset line_to_draw ..*

This specifies which lines must be drawn. You can draw lines between adjacent rows, adjacent columns or between the two data sets, or combinations of these. The name is the name associated with this VoLines and the data set is the data set for which you are going to tell it which lines to draw.

The *line_to_draw ..* parameter be one or more of the following:

NONE	Do not draw any l
BETWEEN_COLUMNS	Draw a line betwe
BETWEEN_ROWS	Draw a line betwe
BETWEEN_DATA_SETS	Draw a line betwe

ivVoLinesSetVertexColor *name dataset row column red green blue*

This sets the color of a vertex color of the dataset given by the parameter *dataset*. The *row* and *column* specify which vertex and the *red*, *green* and *blue* describe the color.

ivVoLinesSetVertexHeight *name dataset row column height*

This sets the vertex at (*row*,*column*) in the dataset *dataset* to the height given by *height*.

ivVoLinesSetSelectionProc *name proc_name* This sets the procedure that is called whenever a vertex is selected. The procedure must takes four parameters. The first parameter is the name of the VoLines that was selected, the second is the data set the vertex is in, and then the row and column of the vertex within that dataset that was selected.

An example of such a procedure would be:

```
proc select {name dataset r c} {
   ivVoLinesSetVertexColor $name $dataset $r $c 1.0 0 0
}
```

This changes the color of the selected vertex to red.

ivVoLinesSetDeselectionProc *name proc_name*

This sets the procedure that would be called when a vertex is deselected, in other words when another point is selected. The parameter list and structure of the procedure is the same as in the case of the selection procedure.

This is an example of a deselection procedure:

```
proc deselect {name dataset r c} {
   ivVoLinesSetVertexColor $name $dataset $r $c 1.0 0.4 0.1
   ivVoLinesSetVertexHeight $name $dataset $r $c 40
}
```

This changes the color of the deselected vertex to brown and sets the height of that vertex to 40.

B.7 VoCube

Description

VoCube creates a matrix of *nxm* cubes. Each cube can have a different height and color, in addition the dimensions of each cube can be changed.

Commands

ivCreate VoCube *name*

This creates a VoCube and associates it with the name *name*.

ivVoCubeSetRows *name number*

This sets the number of rows in the VoCube with the name *name*.

ivVoCubeSetColumns *name number*

This sets the number of columns in the VoCube with the name *name*.

ivVoCubeSetHeights *name list_of_heights*

This sets the heights of the cubes. The command takes two parameters, the name associated with that VoCube and a list of heights. The heights must be in row order, i.e. all the heights for the first row and the all the heights for the second row, etc.

ivVoCubeSetColors *name list_of_colors*

This sets the colors for the cubes. The *list_of_colors* consists of a list of lists. Each sub-list consist of 3 numbers which defines the red, green and blue values of each color. The sub-lists must be in the same order as the heights, i.e. in row order.

ivVoCubeSetDimensions *name list_of_dimensions*

This sets the colors for the cubes. The *list_of_dimension* consists of a list of lists. Each sub-list consist of 3 numbers which defines the height, depth and with of the cube. The sub-lists must be in the same order as the heights, i.e. in row order.

ivVoCubeSet1Color *name row column red green blue*

This sets the color of 1 cube. The cube is described by its *row* and *column* position and the color by its *red*, *green* and *blue* components.

ivVoCubeSet1Height *name row column height*

This sets the height of a cube. The cube whose height is to be set to *height* is at position (*row*,*column*).

ivVoCubeSet1Dimension *name row column height width depth*

This sets the dimensions of a cube. The cube whose dimensions are to be set is at position (*row*,*column*). The height of the cube (from the top to the bottom) is set to *height*, the width (from left to right) is set to *width* and the depth (from front to back) is set to *depth*.

ivVoCubeSetSelectionProc *name proc_name*

This sets the procedure that is called whenever a cube is selected. The procedure must take three parameters. The first parameter is the name of the VoCube that was selected, and then the row and column of the cube within that was selected.

An example of such a procedure would be:

```
proc select {name r c} {
   ivVoCubeSet1Color $name $r $c 1.0 0 0
}
```

This changes the color of the selected cube to red.

ivVoCubeSetDeselectionProc *name proc_name*

This sets the procedure that would be called when a cube is deselected. The parameter list and structure of the procedure is the same as in the case of the selection procedure.

This is an example of a deselection procedure:

```
proc deselect {name r c} {
   ivVoCubeSet1Color $name $r $c 1.0 0.4 0.1
   ivVoCubeSet1Height $name $r $c 20
   ivVoCubeSet1Dimension $name $r $c 0.5 0.5 0.5

}
```

This changes the color of the deselected cube to brown and sets the height of that vertex to 20 and the dimensions to (0.5, 0.5, 0.5).

B.8 VoVectorField

Description

The VoVectorField defines a number of vectors that can be placed anywhere in 3D space, each vector can be given a direction, size and color. The vectors can be selected and the number of the vector is returned. The vectors are drawn as arrows with a tail and a head they can therefore represent direction as well.

Commands

ivCreate VoVectorField *name*

This creates a VoVectorField and associates it with the name *name*.

ivVoVectorFieldSetVectorStarts *name list_of_start_positions*

This sets the start position of the vectors. The vectors are not laid out in a matrix and therefore the x, y and z positions must be given. The list_of_start_positions therefore be a list of lists, with each sub-list containing 3 floating point values.

ivVoVectorFieldSetVectors *name list_of_vectors*

The list_of_vectors comprises out of a number of sub-lists. Each of these sub-lists contains 4 floating point values. The first 3 values describe the direction of the vector while the fourth describes the magnitude of the vector.

ivVoVectorFieldSetVectorColors *name list_of_vector_colors*

This sets the colors for the vectors. The list_of_vector_colors is a list of lists. Each sub-list consists of 3 floating point values the represent the red, green and blue values.

ivVoVectorFieldSet1Color *name number red green blue*

This sets the color of a vector in the VoVectorField associated with the name *name*. The number of the vector whose color is to be changed is given by *number* and the color it is to be changed to is given by the red, green and blue components.

ivVoVectorFieldSet1Position *name number x y z*

This change the start position of the vector numbered *number* to (x,y,z).

ivVoVectorFieldSet1Direction *name number x y z mag*

This changes the direction of the vector numbered *number* to the vector from (0,0,0) to (x,y,z) with magnitude of *mag*.

ivVoVectorFieldSetSelectionProc *name proc_name*

This sets the procedure that is called whenever a vector is selected. The procedure must take two parameters. The first parameter is the name of the VoVectorField that was selected and then second is the number of the vector that was selected.

An example of such a procedure would be:

```
proc select {name num} {
   ivVoVectorFieldSet1Color $name $num 1.0 0 0
}
```

This changes the color of the selected vector to red.

ivVoVectorFieldSetDeselectionProc *name proc_name*

This sets the procedure that would be called when a vector is deselected. The parameter list and structure of the procedure is the same as in the case of the selection procedure.

This is an example of a deselection procedure:

```
proc deselect {name num} {
   ivVoVectorFieldSetVertexColor $name $num 1.0 0.4 0.1
}
```

This changes the color of the deselected vector to brown.

B.9 VoGrid

Description

This class allows the user to create a grid by specifying the width and depth of the grid. Either the number of subdivisions in the grid may be specified or the distance between the lines may be specified.

The grid may be labeled with two sets of labels which are placed on any two adjoining sides.

The grid is designed to be able to give the viewer a reference point and to orient themself in 3D space. It also provides information as to the what the subdivisions of the axes represent.

Commands

ivCreate VoGrid *name*

Create a new VoGrid and associate it with a the name *name*.

ivVoGridSetX *name value*

ivVoGridSetY *name value*

ivVoGridSetZ *name value*

Set the X, Y and Z size of the grid.

ivVoGridSetHeight *name value*

Set the height of the grid along the z-axis.

ivVoGridSetXLabelPosition *name position*

Set the position of the labels for the X axis. The *position* parameter can have the value of *Hide*, *Top* or *Bottom*.

ivVoGridSetYLabelPosition *name position*

Set the position of the labels for the Y axis. The *position* parameter can have the value of *Hide*, *Left* or *Right*.

ivVoGridSetZLabelPosition *name position* Set the position of the labels for the z-axis. The *position* parameter can have the value of *Hide*, *TopLeft*, *BottomLeft*, *BottomRight*, *TopRight*.

ivVoGridToggleXLabels *name*

ivVoGridToggleYLabels *name*

ivVoGridToggleZLabels *name*

This toggles the X or Y labels on or off.

ivVoGridToggleLabels *name*

This toggles the labels on/off depending on whether or not they are currently visible.

ivVoGridSetXLabels *name label_list*

ivVoGridSetYLabels *name label_list*

ivVoGridSetZLabels *name label_list*

This sets the labels for the x or y axis. The labels are taken to be the elements in the list *label_list.*

ivVoGridSet1XLabel *name label*

ivVoGridSet1YLabel *name label*

ivVoGridSet1ZLabel *name label*

This changes, or sets, 1 label on the x or y axis.

ivVoGridSetXStep *name value*

ivVoGridSetYStep *name value*

ivVoGridSetZStep *name value*

This sets the space between the lines along the x or y axes.

ivVoGridSetXNumOfLines *name value*

ivVoGridSetYNumOfLines *name value*

ivVoGridSetZNumOfLines *name value*

This sets the number of lines, thus the number of labels, that you want along x or y axes.

ivVoGridSetColor *name hue saturation value* Set the color of the grid.

Bibliography

[1] The iIsh User Manual. URL:http://www.cs.uct.ac.za/Local/Honours/Visualization/iish.html.

[2] AVS ... The Future of Visual Computing, 1994. URL: http://www.avs.com/products/avs.html.

[3] The Geometry Toolbox, 1995. URL: http://www.khoros.unm.edu/khoros/khoros2/toolboxes/geometry.html.

[4] Addison-Wesley Publishers. *Inventor Mentor.*

[5] Alan Barnum-Scrivener. How I Do An AVS Kickstart. URL: http://avs.ncsc.org/HTML/IAC/AVS95_www/avs95/netnews/barnum/barnum.pts1and2.html.

[6] Sander Belikn, Stefaan Poedts, Hans Goedbloed, Hans Spoelder, Ad Emmen, Jaap Hollenberg, and Rik Leenders. Comparison of Visualization Techniques and Packages. URL:http://www.sara.nl/Consumer.Report/Report.html.

[7] Lawrence D. Bergman, Jane S. Richardson, David C. Richardson, and Jr. Frederick P. Brooks. VIEW - An Exploratory Molecular Visualization System with User-Definable Interaction Sequences. In *SIGGRAPH '93, Anaheim, California*, 1993.

[8] Jacques Bertin. *Graphics and Graphics Information Processing.* Walter de Gruyter, Berlin, 1981.

[9] Wes Bethel. Modular Virtual Reality Visualization Tools. In *Proceeding of AVS '95*, 1995.

[10] Edwin H. Blake and Henk A. Goosen. The "no-paradigm" programming paradigm for information visualization. In Remco C. Veltkamp and Edwin H. Blake, editors, *Programming Paradigms for Graphics '95.* Springer, 1995. Revised version of paper in 5th Eurographics Workshop on Programming Paradigms for Comptuer Graphics.

[11] M.H. Brown and J. Hershberger. Color and Sound in Algorithm Animation. *Computer*, 25(12):p52–63, December 1992.

[12] Kenneth C. Cox and Gruica-Catalin Roman. Experiences with the PAVANE Program Visualisation Environment. Technical Report WUCS-92-40, Washington University in St Louis, October 1992.

[14] A.P Fairall and W.R. Paverd. Large-Scale Structure in the Southern Sky to 0.1c, In *35th Herstmonceux Conference on 'Wide-field Spectroscopy and the Distant Universe'*. World Scientific Publishing Company, (in press).

[15] A.P. Fairall, W.R. Paverd, and R.P. Ashley. Visualization of Nearby Large-Scale Structures. In *Unveiling Large-Scale Structures behind the Milky Way*. Astronomical Society of the Pacific Conference Series, 1994.

[16] Nahum D Gershon, Ricahrd Mark Friedhoff, Margaret S. Livingstone, Vilayanur S. Ramachadran, and Robert L. Savoy. From Perception to Visualisation. In *SIGGRAPH '92, Chicago*, 1992.

[17] H.A. Goosen, A.R. Karlin, and D.R.Cheriton. Chiron: A system for parallel program performance visualization. In *Proceedings of the Conference on Advanced Techniques in Animation, Rendering and Visualization*. Ankara, Turkey, July 1993.

[18] Peter Hinz. Visualising the Performance of Parallel Programs. Master's thesis, University of Cape Town, 1995.

[19] B.D. Johnson and K.W.J. Malafant. Visualisation Techniques for Geoscience Data and Modelling. In *Proceedings of the Third National Conference on the Management of Geoscience Information and Data*, pages pp 18.1–4, July 1995.

[20] Margaret S. Livingstone. Art, Illusion, and the Visual System. In *SIGGRAPH '92*, July 1992. panel discussion: From Perception to Visualization.

[21] Peter Lynch and John Rothchild. *One up on Wall Street*. Penguin Books, 1990.

[22] Dinesh P. Mehta and Sartaj Sahni. Models and Techniques for the Visualisation of Labeled Discrete Objects. Technical report, University of Florida.

[23] Barton P. Miller. What to Draw? When to Draw? An Essay on Parallel Program Visualization. Technical Report CS-TR-92-1103, University of Wisconsin-Madison, June 1992.

[24] Eugene N. Miya. comp.graphics.visualisation faq.

[25] Marc A. Najork and Marc H. Brown. Obliq-3D: A High-Level, Fast-Turnaround 3D Animation System. *IEEE Transactions on Visualization and Computer Graphics*, pages p175–193, June 1995.

[26] Gregory M. Nielson. Visualization in Scientific Computing. *IEEE Computer*, 22(8):pp 10–11, August 1989.

[27] Lotufo Rasure, Jordan. Teaching Image Processing with Khoros. In *1994 IEEE International Conference on Image Processing*, November 1994.

[28] William Ribarsky, Eric Ayers, John Eble, and Sougata Mukherjea. Glyphmaker: Creating Customized Visualisations of Complex Data. *IEEE Computer*, pages p57–64, July 1994.

[29] Manojit Sarkar and Steven P. Reiss. Generating Abstractions for Visualisation. Technical Report CS-92-35, Brown University, August 1992.

[30] W.J. Schroeder, C.R. Volpe, and W.E. Lorensen. Stream Polygon: A Technique for 3D Vector Field Visualisation. In *SIGGRAPH 93, Course Notes 2, Introduction to Scientific Visualisation Tools and Techniques*, pages 4–11 to 4–17, 1993.

[31] Silicon Graphics Computer Systems. *Iris Inventor Programming Guide, Volume 1: Using the Toolkit.*

[32] Silicon Graphics Computer Systems. *IRIS Explorer Users Guide*, 1992.

[33] Silicon Graphics Computer Systems. *Iris Explorer 2.1: Technical Report*, 1993.

[34] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, Connecticut, 1983.

[35] Jeremy Walton. Now You See It - Interactive Visualisation of Large Datasets. In *Apllications of Supercomputer in Engineering III.* Computational Mechanics Publications, 1993.

[36] M. Young, D. Argiro, and S. Kubica. An Object Oriented Visual Programming Language Toolkit. *Computer Graphics*, 29(2):pp 25–28, May 1995.

[37] M. Young, D. Argiro, and S. Kubica. Cantata: Visual Programming Environment for the Khoros System. *Computer Graphics*, 29(2):pp 22–24, May 1995.