The Application of Semantic Depth of Field to Visualisation Problems

Ben Logan

May 2003
BSc (Hon's) Computing Science Dissertation

School of Computing Science
University of Newcastle upon Tyne

Supervisors: Dr. A. Wipat, Dr. J.L. Lloyd

Abstract

There is a major problem associated with the vast improvements in computing power and the expansion of technologies such as the World-Wide-Web. The problem is the inability to perceive the resulting large amount of data without the assistance of some visualisation technique. Data visualisation is an exciting and developing area of Computer Science that attempts to address this problem. In particular, Semantic Depth of Field is a recently proposed concept that applies the traditional focus techniques used in photography to modern day Computer Graphics. The aim of this project was to develop an application that demonstrated the potential of Semantic Depth of Field as well as any other techniques that the extensive research may have revealed. An application was designed and implemented in Java using the Java 3D API to address these aims. The implementation was based on a modular design pattern to allow for the addition of visualisation techniques that are not addressed by this project. The application, named Java Visualisation System (or JVS), provides the user with views of a data model using graph structures. The major conclusions of this project are that Semantic Depth of Field is indeed a powerful visualisation technique, but that its implementation using current technology requires an open-minded and innovative approach.

Declaration

I declare that this dissertation represents my own work except where otherwise stated:

Acknowledgments

Many thanks are owed to Neil, for providing support and advice when it was needed. I would like to thank Robert Kosara for his acknowledgement of this work and for sharing his thoughts. Following successful integration with another project it also seems sensible to thank the person behind that project, Jon Bainbridge. The author would also like to thank his father for financial support throughout the duration of his degree and his girlfriend for understanding…

Contents:

Chapter One

Introduction

1 Introduction

The world is a complex and multi-dimensional place; the paper and the screen are static and flat. Communication between the readers of an image and the makers of an image must take place on a two-dimensional surface. The essential task of information visualisation is escaping this flatland. All of the interesting worlds (physical, biological and human) that require a better understanding are multivariate in nature, not flatlands.[1] The primary purpose of visualisation is to simulate the 'real world' thereby making vast and complex data easier for humans to understand. Given recent technological advances in a number of areas, including bioinformatics, there is an increasing amount of this vast and complex data. That is the motivation for data visualisation (US 'visualization') and the reason it has only recently been recognised as an important field of study.

There is some confusion surrounding the differences between imaging, computer graphics, and visualisation. It is crucial to distinguish between these areas so that the targets that are set are realistic: [2]

- **Imaging, or image processing;**
  The study of 2D pictures, or images. This includes techniques to transform (e.g. rotate, scale, and shear), extract information from, analyze, and enhance images.

- **Computer graphics;**
  The process of creating images using a computer. This includes 2D paint-and-draw techniques as well as more sophisticated 3D drawing (or rendering) techniques.

- **Visualisation;**
  The process of exploring, transforming, and viewing data as images to gain understanding and insight into the data.

This project is concerned with large scale implementations of visualisation solutions that aim to dramatically improve the user's understanding of a given set of data. The field of information visualisation is relatively young, and is developing fast, however the approaches are numerous. Visualisation can range from simple computer graphics, such as a curve illustrating numerical relationships to elaborate shapes illustrating far more complex data sets, such as, metrological phenomena.[30] Visualisation is a vast subject and its applications can vary dramatically, in both scale and usefulness.

A common feature of visualisation applications is the ability to direct the user's attention to certain objects. This can help alert the user to a problem or, perhaps, show the matching objects in response to a query.[26] The visualisation application that has been created during this project strove to successfully implement such a feature. The aim of this project was to evaluate a number of data visualisation aids and to implement an application that employed a selected visualisation technique to aid in data visualisation. A technique known as Semantic Depth of Field (SDoF)[27] was the primary concern of this project and was used to blur irrelevant objects and focus on the relevant ones. SDoF is a technique proposed recently by Robert Kosara[26] at the Vienna University of Technology. The method is based on the concept of blurring the non-relevant parts of the displayand retaining them in the view as context. Blurring objects based on their relevance is the key idea behind Semantic Depth of Field.[26] SDoF is based on the Depth of Field (DoF) effect, which has its origins in photography. The idea behind Depth of Field (DoF) is that objects are depicted sharply or blurred depending on their distance from the lens. Semantic Depth of Field extends this effect to decide whether to display an object sharply or blurred, not based on geometry, but on the objects current relevance.

Originally it was envisaged that the implementation would focus around a biological data set, for the purposes of experimenting and testing. Although the visualisation of the relationship between entities in complex biological systems is a very real problem area, a more useful visualisation system would be flexible in its approach to handling data input. One of the strongest aspects of the final program is its ability to demonstrate its usefulness on a wide-range of data sets. As a starting point, it was decided to apply Semantic Depth of Field to visualisation problems in graph drawing systems. Graph structures are a very common tool often employed to represent the relationships between any entities that this system may wish to model. To help in the understanding and requirements analysis for the project, the Sun Graph-Layout applet [37] (Figure 1.1) was selected as the basis for further study. This dynamic graph system provides run-time layout manipulation to ensure that the user is viewing the data in its most presentable form. A Swing implementation of this application was made available by Mr Oliver Shaw. Oliver has fixed a few of the problems with the original Abstract Window Toolkit (AWT) version, and has made some improvements of his own.

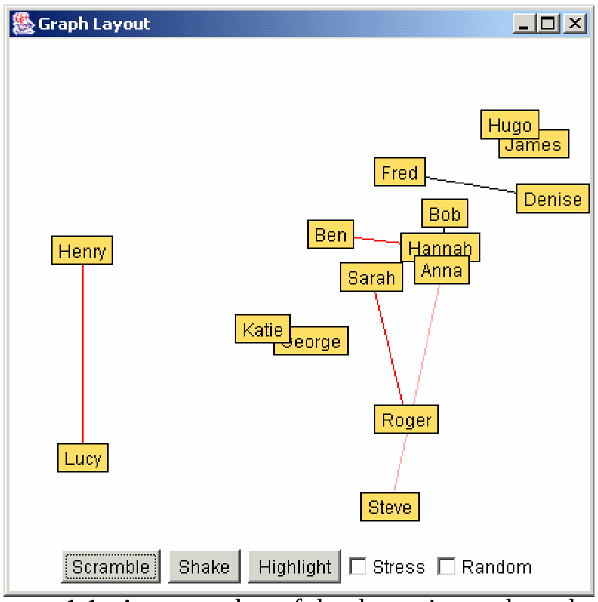


Figure 1.1: A screen shot of the dynamic graph applet.[37]

The Graph-Layout applet produces a visual representation of some ‘nodes’ and ‘edges’ and shows the relationships (edges) between some entities (nodes) within a system. For larger data sets things quickly become complex and difficult to understand. The aim of this project was to implement a program that allowed the user to highlight a portion of this graph (or certain nodes/edges) in order to help them understand the given data. To explain; imagine a large number of people (represented by nodes), if the user attempted to analyse that data and there were hundreds of people floating around it would be impossible to get an idea of any of the relationships or characteristics of the data. The ideal application would have a button to, for example, highlight all males or highlight all couples. That functionality could instantly improve the user's understanding of the data. The result of the project was a generic set of Java classes that can be used to implement a graph layout system for user applications, including Semantic Depth of Field. The results are discussed in more detail in Chapter 7.

1.1 Project Scope

There is no shortage of motivation for this project. Although examples of three-dimensional computer graphics are numerous, information visualisation work in three dimensions is still novel.[3] The field of information visualisation especially in the context of 3D computer graphics is young and developing fast. Information visualisation has only recently been recognised as a field of study in its own right and work in the area has grown significantly over the last few years.[31] Since this project was started in September, 2002, there have been noticeable developments in the area of information visualisation. Indeed there has been recent recognition that information visualisation combined with data-mining techniques could result in an even more useful end product.

Researchers in information visualisation believe in the importance of giving users an overview and insight into data while data mining researchers believe that statistical algorithms and machine learning can be relied on to find interesting patterns. It is recognised that a combined approach could lead to novel discovery tools.[21,32] Although that is outside the scope of this project; one of the primary objectives was to develop a piece of software that was flexible enough to be easily integrated with such a system. Due to the modularity of design and flexibility of the data model this integration was achieved and the results are presented in Chapter 7.

This project follows on from an MSc project of a similar title that was completed just prior to the commencement of work on this project. That dissertation made the observation that the Java 3D API did not provide an in-built blur effect, and because of that, the work centred around creating a blur effect from the features that were available.[22] Although it is true that Java 3D has no facility to implement blur directly, the Application Programming Interface (API) does provide a fog function which, it was hoped, would be capable of achieving the desired effect. However, there was a potential problem with the use of fog; it works differently to a traditional implementation of blur, blending colours instead of reducing the definition of an object.[22] Rather than be concerned with the specifics of blur, this project regards the aims of Semantic Depth of Field as the principle factor in determining whether the application is achieving the desired effect. The technique uses the concept of relevancy; fogging can display the relevancy of one object in relation to another by using depth cueing, i.e. setting the non-relevant object further back than the relevant object. It is important to highlight these differing opinions so as to make sure that this project is not perceived as overlapping with any previous work.

This title provided an excellent opportunity to pursue original research. The work carried out during the course of this project has genuinely not been undertaken before. This is an interesting and exciting application of technology and, more importantly, it has the potential to be very useful to scientists across a range of fields.

1.2 Project Specification

The aim of this project was to evaluate the use of Semantic Depth of Field (SDoF) as a method for aiding data visualisation. The expected outcome of this project was a working implementation of some data visualisation aids. Remembering the key principles of visualisation; it does not matter how good a technique is in terms of its efficiency, its ingenuity in design, or the pleasing pictures it produces – it must convey information to the user efficiently and effortlessly.[17] Although the specific aim was to successfully implement Semantic Depth of Field, the overall end-product should be a tool that will improve the user's understanding of a given data set.

"The field of computer-based information visualisation is about creating tools that exploit the human visual system to help people explore or explain data. Interacting with a carefully designed visual representation of data can help us form mental models that let us perform specific tasks more effectively." [20]

The above quote sums the topic up beautifully. This is a data visualisation project. The original project requirements have been slightly refined and are defined as follows:

1. Create a graph visualisation system that incorporates facilities for highlighting portions of the graph using Semantic Depth of Field.

2. The program will implement the Java 3D fog effect on a group of nodes and there will be an intelligent discussion of the results in relation to the aims and objectives of Semantic Depth of Field (SDoF).

3. The application will attempt to implement an Emissive Lighting effect on a group of nodes, with the aim of further improving visualisation.

4. The application will be flexible in design so that it could be easily adapted to suit visualisation problems in other fields with different input data.

5. There will be implementation of a dynamic graph algorithm, with the aim of creating nodes that appear to move in and out of the fog.

6. The application will combine the above visualisations with the Sun Graph-Layout applet [37] to provide some new views on the underlying data model.

7. The application will be downloadable from a web page that provides support for its use and examples of its output.

1.3 Project Management

A "software development process model" is a description of the work practices, tools, and techniques used to develop software. The development of the software during this project followed a model in order to ensure systematic working and the completion of objectives. The waterfall method is a well-known model that consists of the following stages:

- Requirements
- Design
- Implementation
- Testing
- Maintenance & Review

Each stage feeds into the next. The purpose of the requirements stage (requirements analysis) is to understand and clarify the requirements and often involves resolving the conflicting views of different users. The next stage is concerned with the production of a document, the specification, which defines as accurately as possible the problem to be solved. The requirements analysis and specification are often regarded as the most difficult tasks in software development.[4]

Having defined what the system is to achieve, the next step is to design a solution and implement the design on a computer. It is at this, the implementation stage, that the programming language becomes directly involved.[4]

The aim of testing is to show that the implemented solution does what the user expects and satisfies the original specification. One of the problems with testing is that it can only show the presence of errors, it can never prove their absence.

The final stages of software development covers two distinct activities:

- The correction of errors that were missed at an earlier stage but have been detected after the program has been in active service.
- Modification of the program to take account of additions or changes in the users' requirements.

The last stage is outside the scope of this project, though it is important when designing the program to bear it in mind as maintenance, as well as changes or updates, may need to take place in the future. There are numerous advantages of using the waterfall method:

- Every stage produces a concrete deliverable.

-Requirements documents, code, etc.
- Provides simple measure of development status.
  -What phase are we in?
- All of the most widely used methods are underpinned by the waterfall method.
- All the research is done before the coding begins, this will inevitably lead to better quality program design.

Here are the relevant disadvantages of using the waterfall method:

- Difficult to accurately and completely specify requirements prior to any implementation.
- Phase of development does not necessarily reflect progress toward completion. Entering testing could mean 80% or 20% completion.
- One phase must be completed before proceeding onto the next.

It was decided to adopt the waterfall model of software development. The reason is that most final year projects fall into a well-defined category:

- Problem (given to the student)
- Analysis: refine and detail the problem
- Design: produce a specification
  - Architecture (subsystem overview)
  - Data, interfaces, processes
- Code
- Test
- Final Report & demonstration

A project with these distinguishable features can be easily matched to the Waterfall model of software development.

"A wide variety of 'flexible' models of software development are available as alternatives to the traditional waterfall model. We find that, while some characteristics of a more flexible process have a negative impact on performance when considered alone, these potential 'trade-offs' disappear when considered in combination with other attributes of a more flexible process." [16]

Given that argument, the development of the program will, in actual fact, be more flexible than the traditional waterfall approach. The Gantt chart that follows highlights the key stages in this project and gives an idea as to when they each took place.

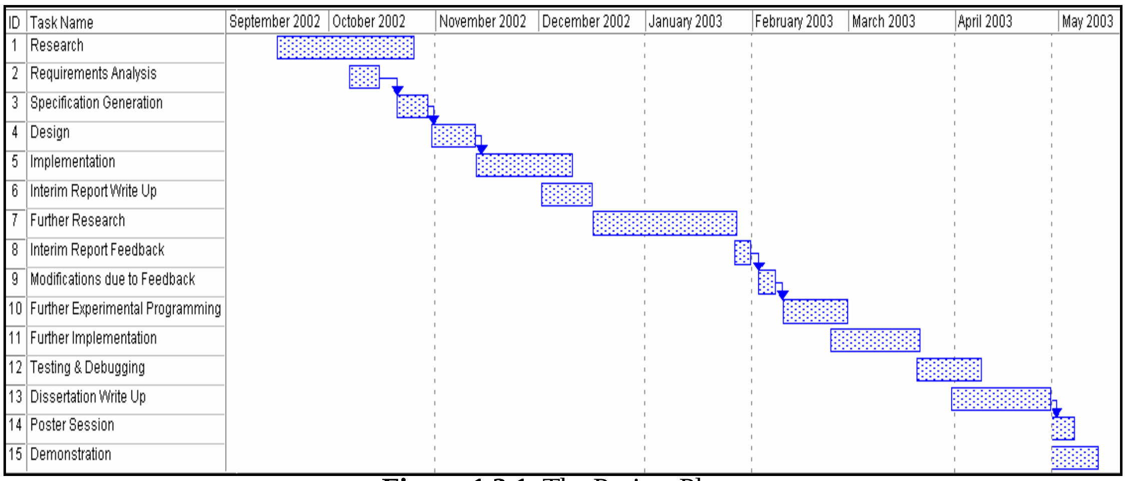


Figure 1.3.1: The Project Plan.

1.4 Dissertation Outline

Chapter 2 of this dissertation provides all the necessary background information for this project. It starts with a discussion of the relevant theories and concepts and moves on to cover the tools and technologies used in creating a solution to the given problem.

Chapter 3 discusses the design issues relating to the application itself. It provides an overview of the application and then comments on the design issues relevant to each of the main aspects of the software.

Chapter 4 takes an in-depth look at the implementation of the software and includes a code-description for each of the classes in the system.

Chapter 5 includes a discussion of the testing strategies.

Chapter 6 contains the user manual for the software. It includes help sheets for the installation, operation and integration of the software developed.

Chapter 7 presents the results of the program in the context of the aims of Semantic Depth of Field, and data visualisation as a whole. The results of the integration with a data-mining application are also discussed in this chapter. Integration is not mentioned in the design or implementation because those stages focussed on the development of a package of classes that would be flexible enough to allow for integration, but the integration was not physically built into the system; it was made possible by flexible, modular design of the software.

Chapter 8 analyses to what extent the original objectives, as set out in this chapter, have been met. It then makes some conclusions regarding the work carried out, as well as making a comment on future areas for development within the application.

Chapter Two

Background

2 Background

2.1 Theory & Concepts

2.1.1 Visualisation

To gain an appreciation for the purpose, requirements and specification of the software developed during this project, some of the basic concepts of data visualisation are reviewed in this section.

The goal of visualisation research is to transform data into a perceptually efficient visual format. It is necessary to say something about the types of data that can exist to be visualised. Bertin (1977) suggested that there are two fundamental forms of data: data values and data structures. More recently it has been accepted that a better way to express this idea is to divide data into entities and relationships.[8] Entities are objects that user wishes to visualise; relationships (or relations) define the structures and patterns that relate entities to one another. In the context of this work the concept of entities and relationships relates closely to graph theory and the representation of this data. Entities can be considered to be the nodes of the graph, it is these that are to be visualised. Relations can be thought of as the edges of the graph. The only difference that needs to be highlighted is that not all nodes are being visualised; technically there are nodes in the graph that are not entities, in the visualisation sense of the word. The closeness of this mapping is the core reason that the final program can be so flexibly applied to other sources of data. There is a more thorough discussion on graphs later in this chapter, for now just consider the data at an abstracted level. There are clearly distinguishable stages to visualisation: [8]

- The collection and storage of data itself.
- The pre-processing designed to transform the data into something the user can understand.
- The display hardware and the graphics algorithms that produce an image on the screen.
- The human perceptual and cognitive system (the perceiver).

Visualisation has the potential to yield some major benefits. The list that follows is impressive, but it is crucial to realise that these are potential benefits, and that a visualisation technique that is successful in all of these areas would be very challenging to implement. Visualisation can:

- Help a user comprehend huge amounts of data.
- Allow the user to notice properties of the data that were not anticipated.
- Help facilitate the understanding of both large-scale and small-scale features of the data.
- Enable problems with the data itself to become immediately apparent. It is common for a visualisation to reveal not only things about data itself, but about the way it is collected.

- Help to facilitate hypothesis information.

The reason visualisation can be so effective is that it uses one of the channels to the human brain that has a high bandwidth: the eyes.[26] As with most things though, this channel can be used more or less efficiently. A successful visualisation allows for a large amount of understandable data to be transferred to the brain, and for that data to be perceived in a very short period of time. That can be quite a challenge.

To summarise, the field of information visualisation is only a few years old, but it is growing rapidly and so is recognition that visualisation will play an important role in many future applications. In the future processor speed will no doubt continue to grow according to Moore's Law, but it is expected that the amount of data to process will increase even faster.[20] There is little doubt that this explosion of data poses problems, especially in the area of human perception and the understanding of this data. The data collection is not an end itself, but a means to the end of helping humans deal with the world.[20] Visualisation lets humans claw their way through these mountains of data, making decisions based on a more sound understanding. Visualisation is necessary because although the human visual system is very flexible, it is tuned to receiving data presented in certain ways, but not in others.[8] The understanding of this mechanism is the key to the science of visualisation; it will result in the production of high-quality displays.

Focussing on the way in which visualisation was to be used in this project; it is accepted that there are multiple ways to visualise a given data set.[7] Geometric objects can be used to represent individual data entities, they can be coloured, animated or transformed (change of orientation). Although this project is aimed primarily at implementing one particular technique for visualisation (Semantic Depth of Field), it was recognised at an early stage that alternatives may appear during the course of the project. The next section discusses the research done in the area of Semantic Depth of Field and 'focus & context' visualisation.

2.1.2 Semantic Depth of Field

Pointing the user to the parts of a visual display that are currently the most relevant is an important task in information visualisation. The same problem exists in photography, where a number of solutions have been known for a long time. One of these methods is Depth of Field (DoF), which depicts objects in or out of focus depending on their distance from the camera (Figure 2.1.2.1). There have been surprisingly few attempts to use Depth of Field or blur in visualisation. There are approaches that use blur in a limited way, but few of them are based on a thorough model or founded out of in-depth research in perceptual psychology.[27]

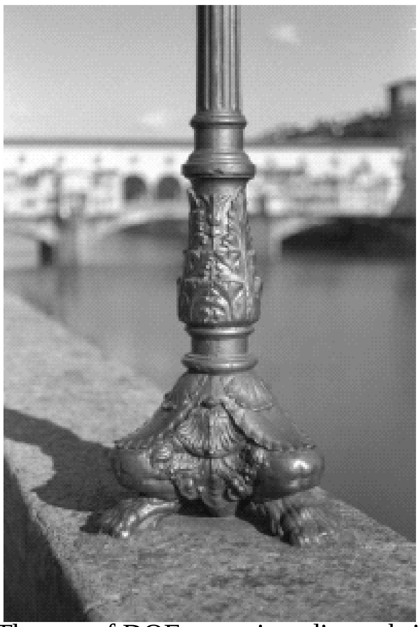


Figure 2.1.2.1: The use of DOF to retain a distant bridge as context.[29]

A recent paper proposes the generalisation of this idea to a concept known as Semantic Depth of Field (SDoF)[27], where the sharpness of objects is controlled by their current relevance, rather than their distance from the camera lens. This enables the user to quickly switch between different sets of criteria without having to get used to a different visualisation layout or geometric distortion.

The blurring of objects by Depth of Field, in photography, depends solely on their spatial depth within the scene. In contrast with this approach, Semantic Depth of Field provides blurring based on the semantics of the scene. The general approach is to use blur to de-emphasise context objects.[27] After Semantic Depth of Field has been 'turned on' the user can still see the features of relevant objects without having to adapt to a different kind of display. This is a key reason for the success of SDoF and it seems crucial that this characteristic is maintained in any implementation of the technique.

The details regarding the specific functioning of blur belong in perceptual psychology. Basically blur works by using a visual feature that is inherent in the human eye and therefore blur is perceptually effective. Research has shown that Semantic Depth of Field is a pre-attentive process and not significantly slower than colour, as was previously thought.[26] That is the justification for this project; it is known that the use of blur is perceptually effective and therefore practical investigation into possible solutions is warranted. Given that blur has the advantage of working independently of colour, it can be useful for black and white images as well as for providing visualisation to colour-blind users.[26]

The concept of Semantic Depth of Field relies on a process of finding sharp targets among blurred distracters (contextual objects). This process is performed pre-attentively and this is an important property of SDoF; the perception takes place in a very short time (usually within ten milliseconds).[17]

"[Pre-attentive processes] are performed in a highly parallel way, no serial search is performed. The viewer immediately perceives the one object which is different from all others, there is no need for examining all objects one after the other." [27]

The reason that the human brain is so fast to notice these differences in depth is that human perception divides our field our view into two categories; those objects that are in the foreground and those that are in the background (*or preferred and non-preferred stimuli* [5]). This division is semantic, it does not depend on the physical positions of objects – closer objects could be considered background objects while more distant ones could become foreground objects.[26] Semantic Depth of Field can assist the human eye in establishing this division: a blurred object will become part of the semantic background while a target object will remain in the foreground.[26]

The argument is that by blurring irrelevant aspects of a data set and leaving everything else un-touched the programmer can trick the brain into categorising (relevant & non-relevant) the data thereby improving data visualisation. A chess board is commonly used to demonstrate the effective use Semantic Depth of Field:

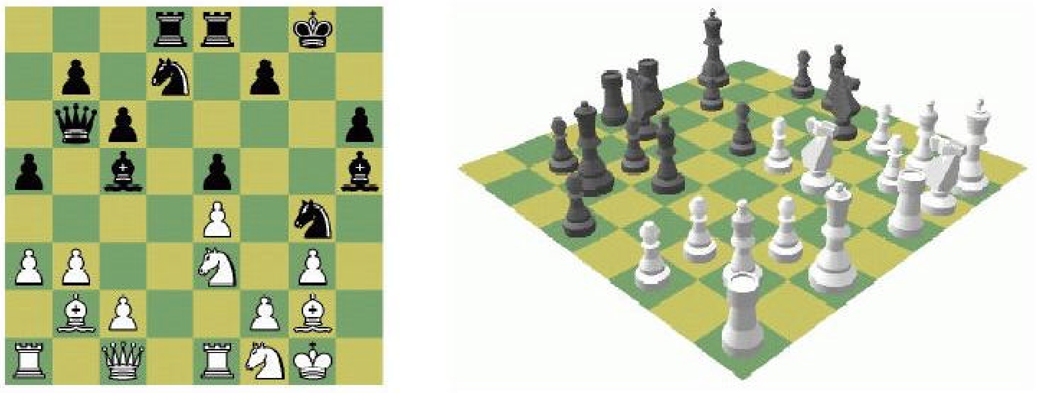


Figure 2.1.2.2: The board in 2D and 3D, with no use of visualisation aids.[27]

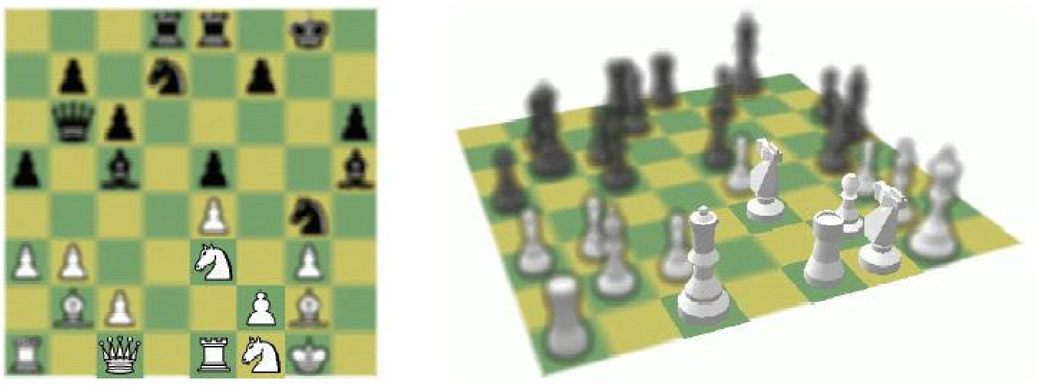


Figure 2.1.2.3: The same board, this time with the application of SDoF.[27]

The second figure shows how effectively Semantic Depth of Field can be used to highlight relevant parts of the display. In this case the relevant parts of the display are those chess pieces covering the white knight, they appear sharply and the other non-relevant parts of the display are blurred.

At the time of writing, a project to look into this technique (SDoF) was still gathering pace. It was founded in autumn of 2000 and is a joint research activity between;

1.) The research on information visualisation at the Institute of Software Technology at Vienna University of Technology, Austria.
2.) The basic research on visualisation at VRVis, Vienna, Austria.

There were some key findings made by this report, they are summarised as follows: [17]

- Semantic Depth of Field can be used to quickly and effectively guide the user's attention.
- Semantic Depth of Field makes it possible to discriminate between a small number (about two to four) of object groups.
- Interaction is very important; people do not like looking at blurred objects (if they do so, the application is badly designed).
- Blurred text should remain readable.
- Things that do not need to be blurred should not be blurred.

These points will be given much consideration when it comes to the design and implementation of a solution. One other issue, highlighted in that report, is that it is perceptually difficult to identify objects of the same level of blur.[17] This is a problem with using blur, however, with only two blur levels the user would be aware that there were only two possibilities and that all blurred objects belonged to one group. The user will never need to distinguish between those objects that are blurred because there would be no difference between them. As mentioned earlier, work in this field is still gathering pace, but the results so far are very promising. The study into Semantic Depth of Field has revealed a lot of interesting information, but it is acknowledged that it is only a first step.[17]

2.1.3 Graph Theory & Visualisation

The most general mechanism for representing relations between data is the graph. Graphs are demanding to construct but, as a result, they can represent more detail information. The versatility of graphs allows them to represent many of the most difficult theoretical problems that exist in Computer Science.[13] A graph can be described as an abstract mathematical structure defined from two sets; a set of nodes and a set of edges that form connections between nodes (see Figure 2.1.3.1). More formally:

A graph G = (V, E) is composed of a finite (and non-empty) set V of nodes and a set E of unordered pairs of nodes.

$$V = \{n1, n2, \ldots, n_m\}$$
$$\&$$
$$E = \{e1, e2, \ldots, e_p\}$$

Where each edge $e_k = \{n_i, n_j\}$ for some nodes n_i, n_j that are members of the set V.

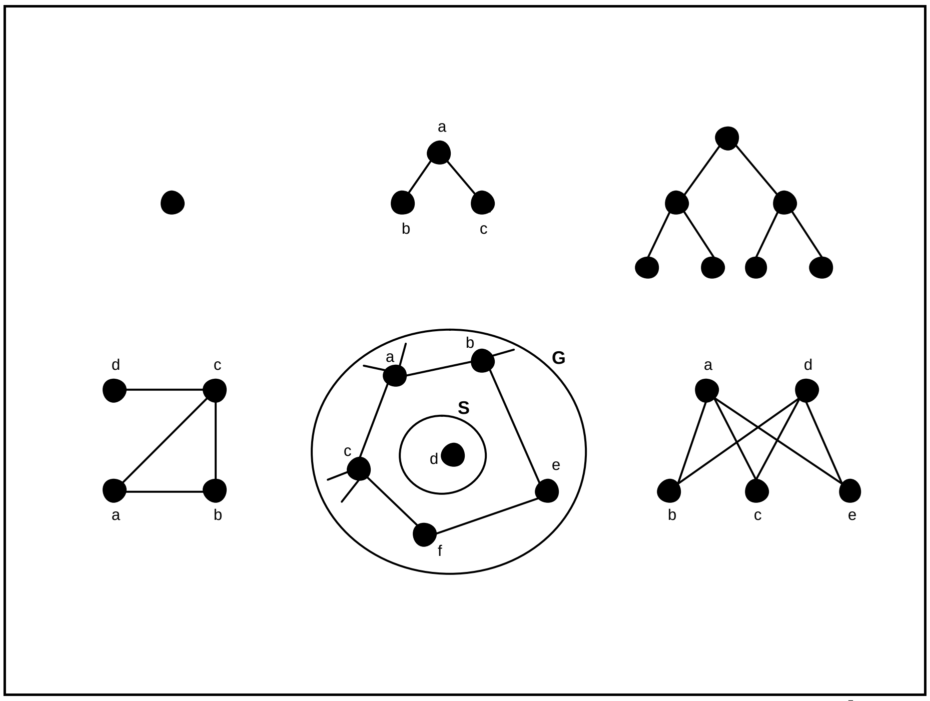


Figure 2.1.3.1; Each node *a* is adjacent to node *b*, but never to *d*. Graph *G* has two components, one of which is *S*. Only the top left graph is complete.[13]

'Data Visualisation' can be defined as an approach in which a computer-generated visual representation is used to improve our understanding of some data. In its simplest form, that could mean turning numbers into pictures. This project focused primarily on these 'visual representations' and not the data itself. The aim was to create a system that can be adjusted with relative ease to suit various forms of data and scenarios. However, it is important to recognise that to successfully implement a visualisation technique would require an environment in which to test any possible solutions.

Across a range of fields such as software engineering, telecommunications, and financial analysis graphs are commonly used to model information.[23] Graphs are a useful way of presenting data and graph drawing is an area of much research and discussion. The discipline of graph drawing is concerned with methods for drawing abstract versions of node/edge diagrams. At the heart of the discipline are a set of graph aesthetics (rules for graph layout) that, it is assumed, will produce graphs that can be clearly understood.[28]

There was the potential for involving complex graph display algorithms. This is a field of study in its own right and was not allocated much time in this project for this reason. There was an analysis of the workings of the Sun Graph-Layout applet implementation[37], for the purposes of understanding its operation. The software did go on to implement a modification of that algorithm under slightly different circumstances. The graph layout algorithms and data structures were not of the up-most importance; though it is obvious that there is scope for another project to tackle these issues. The graph drawing algorithm at work in the Sun Graph-Layout applet [37] is based on the Force-Directed approach. These algorithms are intelligent methods for creating straight-line drawings of undirected graphs. They are popular due to the fact that their basic versions are relatively easy to understand. A force-directed algorithm simulates a system of forces defined on an input graph, and outputs a locally minimum energy configuration.[10]

Given that the application behind this dissertation was defined as a graph visualisation system it is important to distinguish between the two key words in that title; 'graph' and 'visualisation'. Even more than in 2D, the display of graphs in 3D is a two-fold task:[24]

1. **Layout phase;**
   The drawing of the graph aesthetically.
2. **Presentation phase;**
   The application of viewing strategies, techniques and tools to present a meaningful view on the graph to the observer.

This project is more concerned with the 'presentation phase' and was using a graph system merely to demonstrate the actual application of theories such as Semantic Depth of Field. One issue that does need consideration when dealing with graphs is their representation (modelling) of data. The system currently models nodes and edges as Java objects (with coordinate parameters) and the data model stores arrays of these objects. There was exploratory programming with matrix representations but this is outside the scope of this visualisation project and matrices do not appear to have the same importance as they once did in graph science. Matrices are rarely used to represent graphs anymore, largely due to the proliferation of zeros that occurs in matrix representations of all but the densest graphs.[11]

2.2 Tools & Technologies

2.2.1 Java

The Java platform is based on the power of networks and the idea that the same software should run on many different kinds of computers. Since its initial commercial release in 1995, Java technology has grown in popularity and usage because of its portability. The Java platform allows you to run the same Java application on lots of different kinds of computers. Any Java application can easily be delivered over the Internet, or any network, without operating system or hardware platform compatibility issues. Java is an object orientated programming language and the object-orientated facilities of Java are essentially those of C++, with extensions from Objective C for more dynamic method resolution.

With most programming languages, you either compile or interpret a program so that you can run it on your computer. The Java programming language is unusual in that a program is both compiled and interpreted. With the compiler, first you translate a program into an intermediate language called Java bytecodes - the platform-independent codes interpreted by the interpreter on the Java platform. The interpreter parses and runs each Java bytecode instruction on the computer. Compilation happens just once; interpretation occurs each time the program is executed.

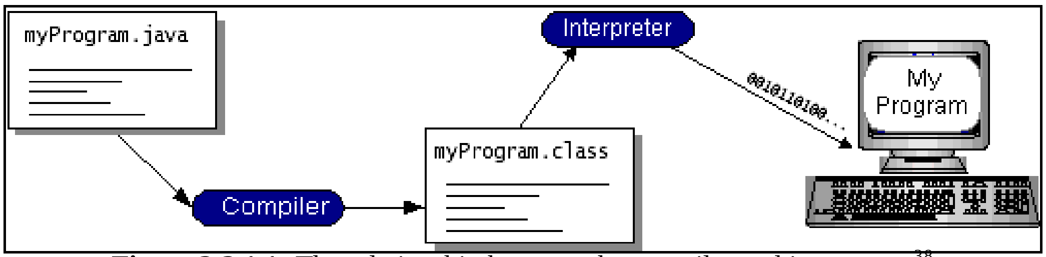
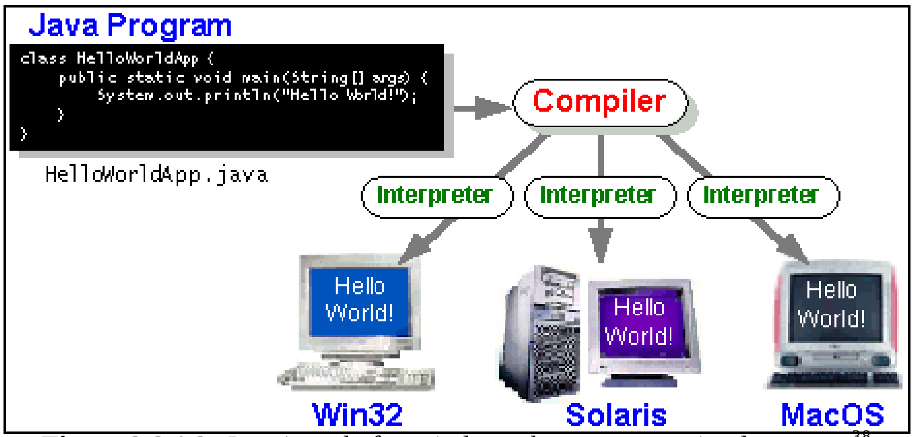


Figure 2.2.1.1: The relationship between the compiler and interpreter.[38]

Java byte-codes can be thought of as the machine code instructions for the Java Virtual Machine (JVM). Every Java interpreter, whether it is a development tool or a Web browser that can run applets, is an implementation of the JVM. Java bytecodes help make portability possible. A programmer can compile a program into bytecodes on any platform that has a Java compiler. The bytecodes can then be run on any implementation of the Java Virtual Machine (JVM). That means that as long as a computer has a JVM, the same program written in the Java programming language can run on a Windows machine, a UNIX workstation, or on a Mac.



```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Figure 2.2.1.2: Java is a platform-independent programming language.[38]

A platform is the hardware or software environment in which a program runs. Most platforms can be described as a combination of the operating system and hardware. The Java platform differs from most other platforms in that it's a software-only platform that runs on top of other hardware-based platforms. The Java platform has two components:

- The Java Virtual Machine (JVM)
- The Java Application Programming Interface (Java API)

The Java Virtual Machine is the base for the Java platform and is ported onto various hardware-based platforms. The Java Application Programming Interface (API) is a large collection of ready-made software components that provide many useful capabilities, such as graphical user interface (GUI) widgets. The Java API is grouped into libraries of related classes and interfaces; these libraries are known as packages. The following figure depicts a program that is running on the Java platform.

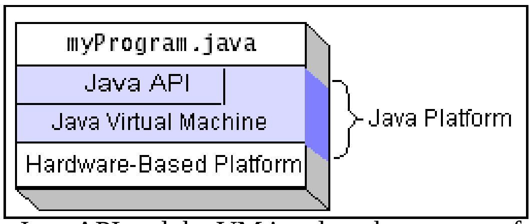


Figure 2.2.1.3: The Java API and the VM insulate the program from the hardware.[38]

Native code is code that after you compile it, the compiled code runs on a specific hardware platform. As a platform-independent environment, the Java platform can be a bit slower than native code.

The most common types of programs written in the Java programming language are applets and applications. An applet is a program that adheres to certain conventions that allow it to run within a Java-enabled web browser. The aim of this project was to develop a Java application, although if there had been time the changes necessary to allow the program to run as an applet would have been implemented. Every full implementation of the Java platform provides the programmer with the following features:

- ***Essentials***: Objects, strings, threads, numbers, input and output, data structures, system properties, date and time, and so on.
- ***Applets***: The set of conventions used by applets.
- ***Networking***: URLs, TCP (Transmission Control Protocol), UDP (User Datagram Protocol) sockets, and IP (Internet Protocol) addresses.
- ***Internationalisation***: Help for writing programs that can be localized for users worldwide. Programs can automatically adapt to specific locales and be displayed in the appropriate language.
- ***Security***: Both low level and high level, including electronic signatures, public and private key management, access control, and certificates.
- ***Software components***: Known as JavaBeans, can plug into existing component architectures.
- ***Object serialisation***: Allows lightweight persistence and communication via Remote Method Invocation (RMI).
- ***Java Database Connectivity (JDBC)***: Provides uniform access to a wide range of relational databases.

The Java platform also has APIs for 2D and 3D graphics, accessibility, servers, collaboration, telephony, speech, animation, and more. The Java 3D Application Programming Interface (API) is of particular interest to this project and is the focus of the next section.

2.2.2 Java 3D

Java 3D is a high level, scene graph based Application Programming Interface (API). Java 3D uses either DirectX or the OpenGL low level API to take advantage of 3D hardware acceleration. This maintains the platform-independent aspect of the Java language and provides the high level of performance achieved by the lower level APIs.

Java 3D provides an interface for writing programs to display and interact with three-dimensional graphics. Java 3D is a standard extension to the Java 2 Software Development Kit (SDK). The API provides a collection of high-level constructs for creating and manipulating 3D geometry and structures for rendering that geometry. Java 3D provides the functions for creation of imagery, visualisations, animations, and interactive 3D graphics. The Java 3D API is a hierarchy of Java classes which serve as the interface to a sophisticated three-dimensional graphics rendering and sound rendering system. The programmer works with high-level constructs for creating and manipulating 3D geometric objects. These geometric objects reside in a virtual universe, which is then rendered. The API is designed with the flexibility to create precise virtual universes of a wide variety of sizes, from astronomical to subatomic.

A Java 3D program creates instances of Java 3D objects and places them into a scene graph data structure. The scene graph is an arrangement of 3D objects in a tree structure that completely specifies the content of a virtual universe, and how it is to be rendered. Much like standard Java applications, Java 3D programs can be written to run as stand alone applications, or as applets in browsers which have been extended to support Java 3D.

The Java 3D Application Programming Interface (API) defines over 100 classes presented in the *javax.media.j3d* package. These classes are commonly referred to as the Java 3D core classes. In addition to the Java 3D core package, other packages are used in writing Java 3D programs. One such package is the *com.sun.j3d.utils* package that is commonly referred to as the Java 3D utility classes. The core class package includes only the lowest-level classes necessary in Java 3D programming. The utility classes, on the other hand, are convenient and powerful additions to the core. The utility classes fall into four major categories:

- Content loaders
- Scene-graph construction aids
- Geometry classes
- Convenience utilities

Using utility classes significantly reduces the number of lines of code in a Java 3D program. One of the aims of this project is to utilise these utility classes when implementing the program. In addition to the Java 3D core and utility class packages, every Java 3D program uses classes from the *java.awt* package and *javax.vecmath* package. The *java.awt* package defines the Abstract Windowing Toolkit (AWT). These classes create a window to display the rendering. The *javax.vecmath* package defines vector math classes for points, vectors, matrices, and other mathematical objects.

The next section will look at a central feature of Java 3D known as environment nodes. These environment nodes proved to be the key to how best implement the visualisation solutions. A good strategy for designing a visualisation is to transform the data so that it appears like a common environment – a kind of data landscape. The hope then is that skills obtained in interpreting the real environment can be transferred to understanding the given data.[8] Environment nodes affect the environment in an area of the virtual universe. They are not directly visible, but instead they change the space in which the shape nodes are viewed. Light and fog are two such environment nodes and these were used to help implement the visualisation improvements that are at the core of this project.

2.2.3 Java 3D - Fog

Fog nodes, as their name suggests, simulate atmospheric effects like fog or smoke. They can be used to increase the realism of a scene by fading the appearance of objects that are farther from the viewer. Fog nodes are designed to simulate the way in which real-world objects appear to fade into the background when viewed from a distance.[6]

Fog is a surprisingly useful feature that can not only increase the realism of your scenes, but can also improve overall performance by allowing you to reduce the amount of detail required for objects positioned far away from the viewer. In the real world, distant objects seem to fade somewhat because the light reflected from them has to travel farther through the earth's atmosphere to reach your eyes. Depending on atmospheric conditions, this fading effect can be quite extreme. The use of fog is sometimes referred to as 'depth cueing' because it gives the brain a visual cue as to the depth of an object in the scene; it is in this context that fog has the potential to assist in the implementation of Semantic Depth of Field.[6] Fog is implemented in Java 3D by blending the fog colour with the colour of the objects in the scene based on distance from the viewer. The farther away an object is from the viewer, the more it gets blended with the fog colour. Java 3D supports the following two basic types of fog:

- Linear Fog has a constant density, so that an object that is twice as far away appears to be twice as "fogged".
- Exponential Fog has a density that approximates the way fog appears in the real world.

Given that linear fog is useful for 'depth cueing' and exponential fog is more suited to simulating atmospheric effects the implementation of fog to simulate the blurring of objects would best use the linear fog function available in Java 3D.[6] Linear fog has a pair of distance values that define the "fog bank". Anything closer than the front distance is not fogged at all, anything beyond the back distance is completely obscured by the fog (it is drawn in the fog colour), and anything in between the two distances is fogged depending on the distance.

```
java.lang.Object
  |
  +--javax.media.j3d.SceneGraphObject
        |
        +--javax.media.j3d.Node
              |
              +--javax.media.j3d.Leaf
                    |
                    +--javax.media.j3d.Fog
                          |
                          +--javax.media.j3d.ExponentialFog
                          +--javax.media.j3d.LinearFog

public abstract class Fog extends Leaf
```

Fig 2.2.3.1: The position of the Fog classes in relation to the rest of the Java 3D library.

The Fog leaf node defines a set of fog parameters common to all types of fog. These parameters include the fog colour and a region of influence in which this Fog node is active. A Fog node also contains a list of Group nodes that specifies the hierarchical scope of this Fog. If the scope list is empty, then the Fog node has universe scope: all nodes within the region of influence are affected by this Fog node. If the scope list is non-empty then only those Leaf nodes under the Group nodes in the scope list are affected by this Fog node (subject to the influencing bounds).

If the regions of influence of multiple Fog nodes overlap, the Java 3D system will choose a single set of fog parameters for those objects that lie in the intersection. This is done in an implementation-dependent manner, but in general, the Fog node that is "closest" to the object is chosen.

The *ExponentialFog* leaf node extends the *Fog* leaf node by adding a fog density that is used as the exponent of the fog equation. The density is defined in the local coordinate system of the node, but the actual fog equation will ideally take place in eye coordinates. The fog blending factor, f, is computed as follows:

$f = e^{-(\text{density} * z)}$

Where *z* is the distance from the viewpoint.

In addition to specifying the fog density, exponential fog lets you specify the fog colour, which is represented by R, G, and B colour values, where a colour of (0,0,0) represents black.

The *LinearFog* leaf node defines fog distance parameters for linear fog. *LinearFog* extends the *Fog* node by adding a pair of distance values, in Z, at which the fog should start obscuring the scene and should maximally obscure the scene. The front and back fog distances are defined in the local coordinate system of the node, but the actual fog equation will ideally take place in eye coordinates.

The linear fog blending factor, f, is computed as follows:

$f = \text{backDistance} - z / \text{backDistance} - \text{frontDistance}$

Where *z* is the distance from the viewpoint.
frontDistance is the distance at which fog starts obscuring objects.
backDistance is the distance at which fog totally obscures objects.

2.2.4 Java 3D - Light

These environment nodes are responsible for lighting the scene. The various types of light available are:

- ***Directional light;*** These provide a simple, fast light type that simulates light from a distant source, such as the sun.
- ***Point light;*** Position the light at a point in space like the light from a bare light bulb.
- ***Spot light;*** These are like the point light which can be focussed in a particular direction.
- ***Ambient light;*** This simulates the diffused light that fills a room, lighting areas that are not directly illuminated.

After a lot of prototyping and exploratory programming (following initial research) it was decided that any of the above methods would be far to complicated to successfully implement, and to be dynamic would have required moving light sources. What was needed was a way of lighting a given shape, and that was not really provided by the above functions. This frustration lead to further research and experimentation and the discovery of a more likely and novel approach.

An interesting idea that this project has produced is the idea of "glow in the dark" nodes. This could be made possible through the use of emissive colouring. Emissive colour in Java 3D is not actually a light node. Emissive colour is the colour of the object independent of any light sources. It produces glowing colours that appear to emanate from within the shape itself. Whereas a diffuse colour might be used to colour a light bulb that has not been lit up, an emissive colour can be used to render a light that is turned on; light appears to be emitted from within the bulb, making it glow. The result is that a shape has a constant colour. This is similar to ambient colouring; the difference is that the emissive colouring is independent of any lights, while the ambient colouring is a combination of the ambient colour and the ambient light colour.

Although emissive colours seem to create shapes that emit lights, they do not actually illuminate objects around them (objects with emissive colours do not act as light sources). This would help to ensure that there was no interference between the numerous nodes in the graph visualisation system.

```
java.lang.Object
  |
  +--javax.media.j3d.SceneGraphObject
        |
        +--javax.media.j3d.NodeComponent
              |
              +--javax.media.j3d.Material

public class Material extends NodeComponent
```
Fig 2.2.4.1: The position of the Material class in relation to the rest of the Java 3D library.

The Material object defines the appearance of an object under illumination. If the Material object in an Appearance object is null, lighting is disabled for all nodes that use that Appearance object. The properties that can be set for a Material object are:

- ***Ambient colour;*** The ambient red-green-blue (RGB) colour reflected off the surface of the material. The range of values is 0.0 to 1.0. The default ambient colour is (0.2, 0.2, 0.2).
- ***Diffuse colour;*** The RGB colour of the material when illuminated. The range of values is 0.0 to 1.0. The default diffuse colour is (1.0, 1.0, 1.0).
- ***Specular colour;*** The RGB specular colour of the material (highlights). The range of values is 0.0 to 1.0. The default specular colour is (1.0, 1.0, 1.0).
- ***Emissive colour ;*** The RGB colour of the light the material emits, if any. The range of values is 0.0 to 1.0. The default emissive colour is (0.0, 0.0, 0.0).

- ***Shininess***; The material's shininess, in the range [1.0, 128.0] with 1.0 being not shiny and 128.0 being very shiny. Values outside this range are clamped. The default value for the material's shininess is 64.

2.2.5 Java Foundation Classes & Swing

The Java Foundation Classes (JFC) consist of a group of features to help programmers build graphical user interfaces (GUIs). The JFC was first announced at the 1997 JavaOne developer conference and is define d as containing the following features:

- ***The Swing Components;***
  Include everything from buttons to split panes to tables.
- ***Pluggable Look and Feel Support;***
  Gives any program that uses Swing components a choice of looks and feels. For example, the same program can use either the Java look and feel or the Windows look and feel.
- ***Accessibility API;***
  Enables assistive technologies such as screen readers and Braille displays to get information from the user interface.
- ***Java 2D API;***
  Enables developers to easily incorporate high-quality 2D graphics, text, and images in applications and in applets.
- ***Drag and Drop Support;***
  Provides the ability to drag and drop between a Java application and a native application.

The first three JFC features were implemented without any native code, relying only on the API defined in JDK 1.1. As a result, they could and did become available as an extension to JDK 1.1. This extension was released as JFC 1.1, which is sometimes called "the Swing release." The API in JFC 1.1 is often called "the Swing API."

Note: "Swing" was the codename of the project that developed the new components. Although it is an unofficial name, it is frequently used to refer to the new components and related API. It is immortalised in the package names for the Swing API, which begin with *`javax.swing`*.

2.2.6 Software Engineering Tools

RealJ 3.5 (formerly FreeJava) uses the tools available in the Sun Java Software Development Kit (SDK), but offers a more convenient Windows-based front end to the compiler, applet runner and runtime module supplied. The current version comes with new features previously only found in expensive applications from major software companies. A new suite of tools gives menu access to some of the more useful SDK tools that were previously only accessible by using the command line.

RealJ is an integrated development environment (IDE) for writing and building programs in the Java language. The tools provided help the programmer develop their source code file, and allow them to compile their source code using the separate tools of the SDK. The key features of RealJ are:

- Class and function browser lets you navigate easily through large projects.
- Integrated build window shows compiler output and output of Java programs while they run.
- Double-click on compiler errors in the build window to view and edit the source code.
- Syntax colouring in Java source files and HTML files.

RealJ is written in C++ specifically for 32-bit windows machines, which makes it much faster than IDEs built in Java. As a small, fast, powerful and efficient Windows front-end to the command line tools of the SDK, RealJ is ideal for small to mid-size applet and application development projects.

RealJ is used along side the current version of the Java programming language, Java 2 SDK - Standard Edition, Version 1.4.1. The version of the Java 3D API that was used to develop the program was 1.3.1 Beta for Win32/OpenGL.

2.3 Architecture & Design Patterns

2.3.1 Model-View-Controller

The Model-View-Controller (MVC) design pattern was developed using the Smalltalk programming language for the creation of user interfaces. The MVC pattern is surprisingly simple, yet incredibly useful.

"The goal of the MVC design pattern is to separate the application object (model) from the way it is represented to the user (view) from the way in which the user controls it (controller)." [35]

The MVC design pattern essentially forces the programmer to think of the application in terms of three modules:

- **Model:** The core of the application. This maintains the state and data that the application represents. When changes occur in the model, it updates all of the views that are connected to it.
- **Controller:** The Graphical User Interface (GUI) presented to the user to manipulate the application.
- **View:** The GUI which displays information about the model to the user. This is where the visualisation techniques will be implemented.

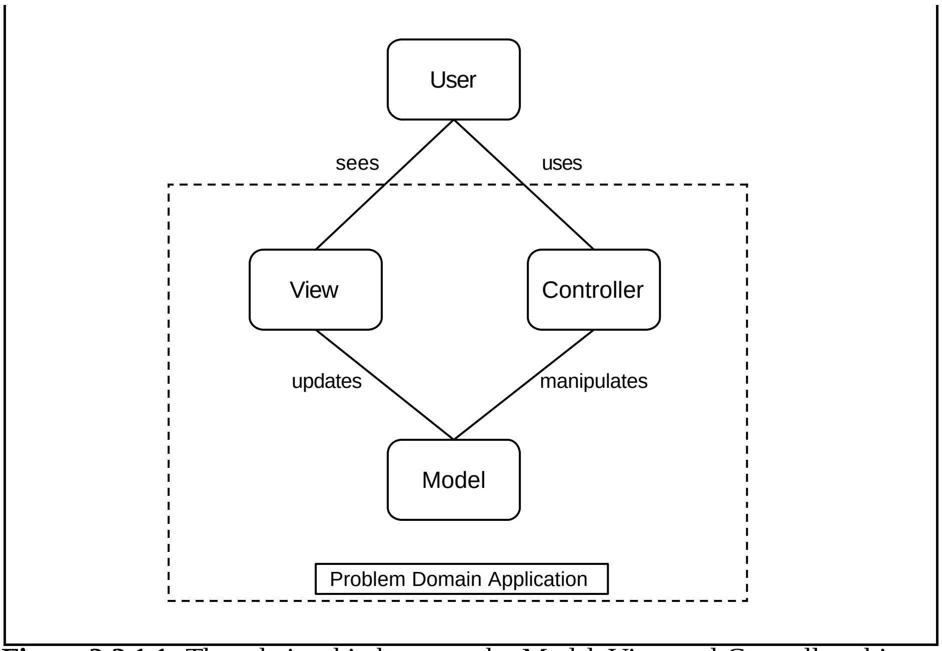


Figure 2.3.1.1: The relationship between the Model, View and Controller objects.

The programmer gains many advantages from breaking down the program and using this Model-View-Controller architecture:

- **Clarity of design:**
  The public methods in the model stand as an API for all the commands available to manipulate its data and state. By glancing at the model's public method list, it should be easy to understand how to control the model's behaviour. When designing the application, this trait makes the entire program easier to implement and maintain.[36]

- **Efficient modularity of design:**
  This allows any of the components to be swapped in and out as the user or programmer desires, even the model. Changes to one aspect of the program are not coupled to other aspects, eliminating many awkward debugging situations. Also, development of the various components can progress in parallel, once the interface between the components is clearly defined.[36]

- **Multiple views:**
  The application can display the state of the model in a variety of ways, and create/design them in a scalable, modular way. Two (or more) views can be using the same data; they just use the information differently.[36]

- **Ease of growth:**
  Controllers and views can grow as the model grows; and older versions of the views and controllers can still be used as long as a common interface is maintained. For instance, if an application needs two types of users, regular and administrator, they could use the same model, but just have different controller and view implementations. This is related to the similarity with the client/server architecture - where the new views and servers are analogous to the clients.[36]

- **Powerful user interfaces:**
  By using the model's API, the user interface can combine the method calls when presenting commands to the user. Macros can be seen as a series of "standard" commands sent to the model, all triggered by a single user action. This allows the program to present the user with a cleaner, more efficient interface.[36]

2.3.2 Model-View-Presenter

A more recent development has been the introduction of a new design pattern based on the Model-View-Controller, the Model-View-Presenter (MVP).[33]

Mike Potel
VP & CTO, Taligent, Inc
January 2000

"Model-View-Presenter or MVP is a next generation programming model for the C++ and Java programming languages. MVP is based on a generalisation of the classic MVC programming model of Smalltalk and provides a powerful yet easy to understand design methodology for a broad range of application and component development tasks. The framework-based implementation of these concepts adds great value to developer programs that employ MVP."

The overall approach was to decompose the basic MVC concept into its constituent parts and then further refine them, with the aim of assisting programmers in developing more complex applications. The first step was to formalise the separation between the Model and the View-Controller, which is now referred to as a Presentation. That separation represents the breaking down of a programming problem into two fundamental concepts: Data Management and User Interface.[18]

From the data management point of view, the programmer is concerned with more than just the underlying data representation within a model but also what data structures, access methods, distributability, etc. are employed.
From the user interface point of view, the programmer is concerned with more than just drawing an object on the screen, but also what semantic operations are enabled, what user actions are recognised, and what feedback is given.

Potel suggests that there are several benefits to generalising the model concept. The first is that it enables a clean separation of the Model and the Presentation. This would mean that over time the programmer could change the underlying data structures in the model, for example, and still reuse the same presentation code. Multiple presentations could be created without re-implementing the underlying model.

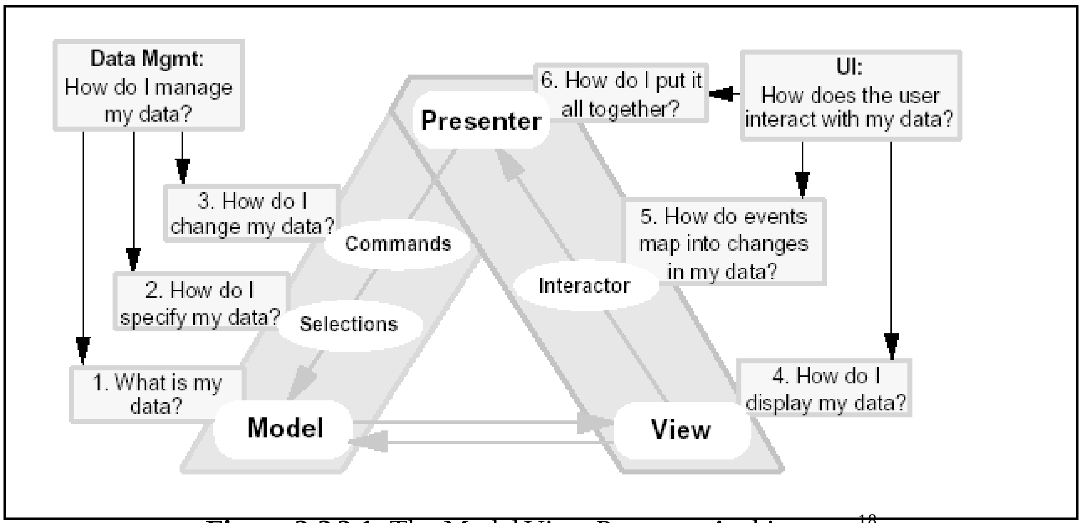

Data Mgmt:
How do I manage my data?
6. How do I put it all together?
UI:
How does the user interact with my data?
Presenter
3. How do I change my data?
Commands
5. How do events map into changes in my data?
Interactor
2. How do I specify my data?
Selections
4. How do I display my data?
1. What is my data?
Model
View

Figure 2.3.2.1: The Model-View-Presenter Architecture.[18]

There are numerous benefits of the approach that has been taken to establish a stronger design pattern. Abstracting out the model permits greater flexibility of use among multiple users. Having different program models encapsulate the same remote data allows multiple users to share the same data.[18] This confirms what was already known, the MVC is a good architecture, but it is out-dated. For the purposes of this project the design will follow the Model-View-Controller architecture but will be flexible in doing so, to allow for modifications along the lines of tha t which Potel has suggested.

Chapter Three

Design

3 Design

The approach taken, based on the extensive research carried out, was to design a program, that would be implemented in an Object-Oriented language (most likely Java), to test Semantic Depth of Field. Given the need to have information to visualise in the first place it was decided to implement a graph visualisation system. A graph visualisation system basically contains a series of 'nodes' connected by links known as 'edges'. This provides a meaningful and useful environment in which to test any visualisation improvements. The concept of a relevancy parameter is associated with each node, that way the results of the program can be evaluated against the aims of Semantic Depth of Field.

Given that one of the key aims set out earlier in this dissertation was for a 'bolt-on' system; the classes were all to be written with the flexibility that would enable them to be deployed in such a fashion. The application was designed to take two forms; a stand-alone executable program and a series of bolt-on classes for use along-side an existing system.

The design included the provision of three possible views, but considered the requirement that additional views may need to be added to the system in the future. The first view is the Sun Graph-Layout applet [37], there are no visualisation techniques available in this view; it is included to show the motivation for the project. The two views developed during this project are the fog view and the emissive light view. The fog view simulates the effects of Semantic Depth of Field using the linear fog function in the Java 3D API.

3.1 Design: Overall Application

The application was created using a design pattern known as Model-View-Controller (MVC) architecture. The MVC architecture is elegant and simple, but adopts a quite different approach than traditional application programs.[21] The MVC architecture was used to provide the application with the flexibility to be easily developed in the future. The use of such a design pattern also allows the program to be used in conjunction with other existing programs and data models, which was an essential requirement of the software described in Chapter 1.

The initial design stated that the program would include a model and a controller and then any number of view objects. The number of view objects created would depend on how many useful methods of improving data visualisation were implemented. For testing purposes (and standalone use of the program) there was a need for an executable file; this is the class *Application*. The interactions between this executable application class, the data model, the controller, and the various views are shown below.

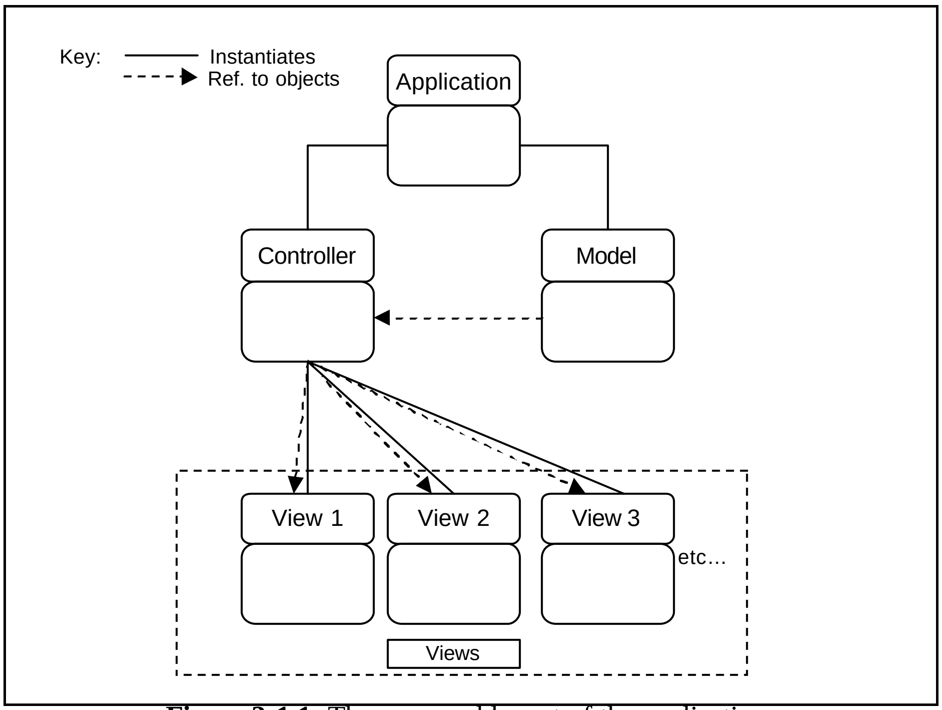


Figure 3.1.1: The proposed layout of the application.

This figure shows the proposed layout of the application, it differs slightly from the traditional MVC design pattern, but all the benefits remain. Basically the application class instantiates a controller class and a model. The controller provides the user with a mechanism for selecting which view to display (or which visualisation aid to apply to the model). The executable class instantiates a data model and a controller. The controller is passed a reference to the data model and in turn is able to instantiate any number of different views. Each view is passed a reference to the data model.

The controller provides the Graphical User Interface (GUI) that the user can interact with. This GUI appears only if the program is going to be used as a stand alone entity; otherwise the methods can be called from other external classes. The GUI basically gives the user a variety of different ways to view the data model. Depending on the method of visualisation chosen by the user, the controller will create the appropriate view. An important reason for the use of views for this program is that they allow both the standard Java API and the Java3D API can be used side by side.

There are some differences between the above design and the conventional MVC design pattern mentioned earlier. This is due to the assumption that the data will not need to be modified by the user while the program is running. This would mean that the controller could interact directly with the views without passing via the model.

The program has implemented methods of making 'live' modifications to the data model (e.g. add a node to the graph) and so there needed to be a way to keep the various views up-to-date. When a data manipulator method is called in the model, the model will need to automatically broadcast a "notify" message to all associated views. This notifies the views that something has changed and will cause them to re-draw themselves, first obtaining the new data from the model with an “update” method. The idea is that the controller would call the manipulation methods in the model and then the model would tell its associated views to re-draw themselves. If the user was interacting via the views directly (not the controller) then the views could manipulate the model directly and then tell themselves to re-draw. These changes could be easily made at a later date due to the flexibility of the modular design pattern.

3.2 The Data Model

The data model is the backbone of this system, this would house all the data that the rest of the system would manipulate and provide views on. Given that the program was implementing a graph system to test the visualisations then the program would need to model ‘nodes’ and ‘edges’. The design for the model is very simple; it needs to be easy to understand as it is going to be used repeatedly throughout the system.

There were two objects designed to represent a node and an edge. Although the object would later be given additional parameters each node would essentially be required to hold its coordinates, a unique identifier and a name. Each edge would simply hold the node identifiers that it was connecting (from/to) as well as a unique identifier and an optional weight. Having implemented these data structures the next step was to design a model object that could hold arrays of any number of these ‘nodes’ and ‘edges’.

The model is accessible to all other areas of the program and changes can be made it to it at run-time. These changes were to be made possible through the inclusion of some additional methods into the model class, these were required to;

- Add a node
- Delete a node
- Add an edge
- Delete an edge

Given that the number of nodes and edges had to be flexible there were two more important methods needed;

- Grow an array
- Shrink an array

That was the initial design for the model, as the project progressed it was recognised that there was a need for some additional methods on the model and one or two additional node parameters.

3.3 The Graph System

There must be a thorough understanding of data types and representations before making a decision in choosing or designing a visualisation.[33] The approach taken needed to be based on this understanding and so it was realised that it would be beneficial to apply the techniques to an existing application. The Sun Graph-Layout applet [37] implementation of a dynamic graph system was used as a basis for making these visual improvements.

The software would need to implement a 'graph' class that would import the node and edge data from the model (passed to it). There would then be some functionality much like that in the Sun implementation [37] that would layout the given data and perhaps perform some threaded activity to dynamically update (animate) the graph. These methods would simply change the coordinates of the nodes (and therefore edges) and then call an 'update' method that would make the changes in the display.

Regarding the actual drawing of the graph this class would also be responsible for setting up a Java3D scene-graph and then adding the results of two additional classes to its branch-group. These additional classes would use the Java3D API as well as the data in the model to draw a series of nodes and edges (DrawNode.class & DrawEdge.class). Given that the actual drawing takes place in these two classes then the 'update' method mentioned earlier would in fact call 'update' methods in these two classes instead. Also within each of these classes would be methods to change the colour of the nodes/edges and show/hide the node/edge labels. The graph class (Graph3D.class) would incorporate two additional methods that may or may not be available to any subsequent visualisations; snap-image, zooming and panning.

There are two aspects of the graph system that add to the strength of the application in terms of its use as a visualisation tool; the fact that connections exist between nodes and that those connections have a well perceived direction. Connectivity is known to be a powerful grouping principle that is stronger than proximity (a), colour (b), size (c), and shape (d).[8]

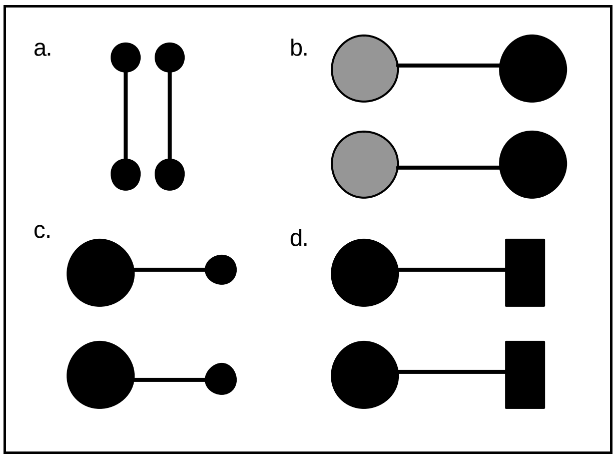


Figure 3.3.1: Connectivity as a powerful grouping principle.[8]

The direction of an edge is also an important characteristic of the data that needs to be made clear to the end-user. Vector direction can be unambiguously given by implementing a colour change along the edge relative to the background colour. [8]

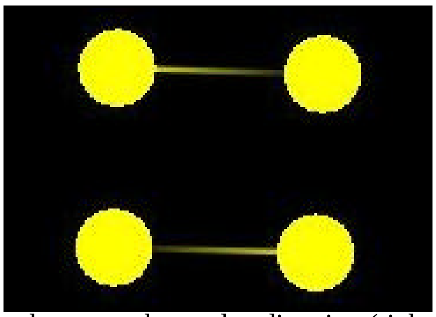


Figure 3.3.2: Colour change to show edge direction (right-left, then left-right).

3.4 The Views

It was decided at a relatively early stage that each visualisation (or view) on the data produced by the program should be based on the same class to save massive duplication of code, this is the class that is discussed in the section just prior to this (Graph3D.class). The idea is that there will be some abstract methods to visualise (de-visualise) some nodes and they will make subsequent calls on different areas of the program depending on the visualisation technique that has been selected. The technique chosen will be signalled by a code number that is constant throughout the program. Currently, signal-1 represents the Semantic Depth of Field view and signal-2 represents the Emissive Lighting view. So, the graph class will implement the following methods that will allow for any number of visualisation techniques to be incorporated in the future;

- Visualise (or De-visualise) a selected set of nodes.
- Visualise (or De-visualise) all nodes.

3.5 View 1: Semantic Depth of Field

Semantic Depth of Field was to be simulated by using the linear fog function in the Java 3D API. This works by setting up a 'fog-bank' that determines where an item in the scene needs to be positioned in order to appear fogged. Obviously the further into the fog bank the denser the fog. The immediate implication of using this method was that the graph would have to be represented in 2D, not 3D. The reason for this is that the 3rd dimension (Z axis) would have to be used for 'depth-cueing'. The basic idea was to layout all the nodes on a particular plane (e.g. Z=10) and then when a particular node is deemed irrelevant move it back into the fog bank (e.g. Z=9). Moving a node into the fog will have the effect of blending the colour of the fog with the colour of the node, if the colour of the fog is set to be the same as the colour of the background then the node will appear blurred;

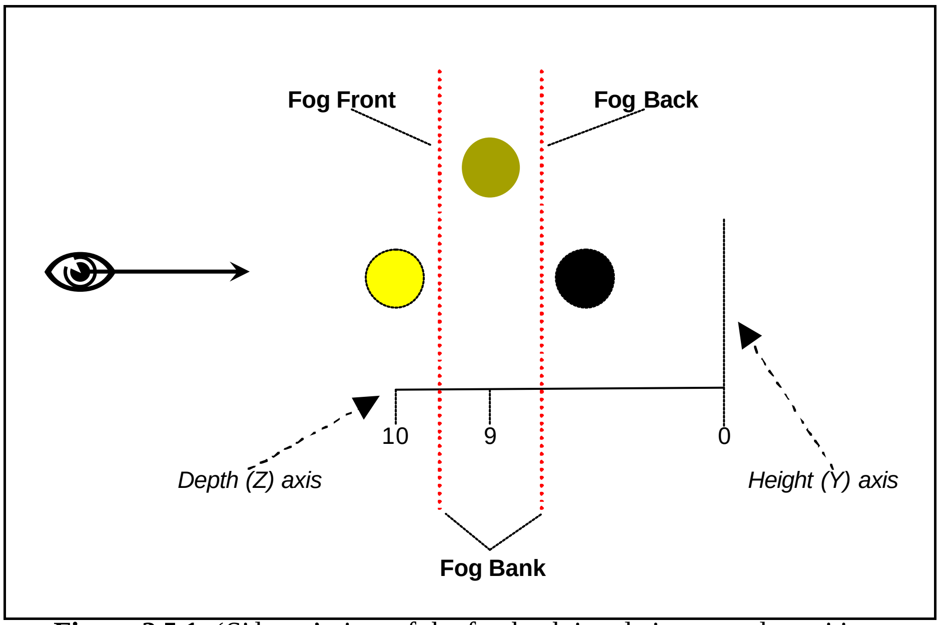


Figure 3.5.1: 'Side-on' view of the fog-bank in relation to node position.

'Focus and Context' techniques are, in general, methods used to enhance certain scene elements of interest while still using the rest as (non-distracting) context. Many of the solutions are distortion oriented in their approach. The parts of interest (relevance) are enlarged while other parts are scaled down to fit into the remaining space.[3] This is not ideal; there should not be any changes to the interesting parts of the scene as that will affect the time in which the user can perceive the visualisation. When introducing Semantic Depth of Field into a graph display system it is important that the information that is to be deemed relevant is still easily, and quickly, readable. Semantic Depth of Field does not suffer from this 'perception delay' due to the fact that it distorts irrelevant objects, not the relevant ones.

The node inside the fog bank will appear fogged (or blurred) where as the appearance of the node that is being visualised will remain un-changed. This is one of the crucial requirements of Semantic Depth of Field. So, in order to set up this fog there needs to be the addition of a method to the graph class, and it needs to have the fog permanently active when visualising with signal-1, the idea is to simply move nodes in and out of it.

Another important issue with Semantic Depth of Field is that the irrelevant items need to appear to be the same size, or at least size can not change at any point; this would disturb the perception and make analysis very difficult. No one can ask the question; to what extend is Semantic Depth of Field effective or was the success related to the fact that the size of the nodes was changing? Of course the actual size of the nodes would not change anyway, but when they move back they will appear further away (and therefore smaller), this meant that the getting the balance right was going to be crucial;

1. Not moving them back so far that they appear smaller,

2. Moving them far enough into a fog bank so as they appear significantly blurred.

This is made yet more interesting by the fact that the fog-bank can not begin until a point beyond the back of the relevant nodes. Now there needs be a little discussion about the mechanisms for visualisation and determining which nodes are to be deemed relevant.

The user wishes to know what part of the visualisation is more relevant as compared to others. Given a certain 2D visualisation of some data, for example, a map of a city, the scene can be extended into 3D, by assigning z-values to each scene object according to their relevance (as decided on by the user). Semantic Depth of Field extends this idea to assume that the depth values assigned to scene objects correspond to DOF in such a way relevant objects lie in the vicinity of the plane of focus, and irrelevant objects are behind it.[3]

The original idea was that nodes would be assigned additional values that determine their position on a third axis. However it seemed to make more sense simply to visualise a series of nodes based on their unique identifiers. Given that this project is implementing relevancy values with two possibilities (relevant and non-relevant) then the node identifier can either be passed to the visualise method, or not.

Fog is not technically the same as blur but this project has subsequently judged the effects to be very similar. This is the principle argument behind this work; that fog can be used as a replacement to a strict implementation of blur. This could have some interesting implications for data visualisation; although simulating depth of focus is recognised as an excellent way of highlighting information by blurring everything except that which is critical, the technique of blur is computationally expensive and has therefore been limited in its applications.[8] It is known then that implementing blur directly can be difficult and tediously slow. It would be possible to implement an actual blur feature, instead of simulating or mimicking blur, but to achieve the effect of blur requires that any point of an image be drawn in a dispersed, circular pattern. That effect can be achieved by creating multiple translucent copies of the object and arranging them in a circular motion.[22] The problem with this is that it still requires a lot of objects (inefficient, slow and complicated) and does still not produce particularly amazing results.

“Unfortunately, simulating depth of focus using a flat-screen display is a major technical problem. It has two parts: simulating optical blur and simulating the optical distance of the virtual object.” [8]

The difficulty in simulating optical blur has been discussed, the problem with regard to optical distance, is not actually relevant with SDoF. The issue is that the human brain decides what it wants to see in focus and what it wants to use as context. With Semantic Depth of Field that decision is made for the user, or at least they actively tell the program what they wish to see as being relevant.

A key part of Semantic Depth of Field is interaction, quite simply blurring objects is useless if users can not change the focus or see what happens after they have changed some of the parameters. There are some typical interactions for SDoF applications:

- ***Selecting the relevance function**:* If the objects carry different parameters the user must be able to see the relevancies relating to each parameter. For example, if a node represented a person, the user may wish to highlight all males or all people less than 20 years of age (assuming the two parameters were sex and age).

- ***Changing the threshold:*** As soon as the display shows SDoF, the user should be able to change the blur function threshold. In the case of this project it may well be that this means changing the depth cueing dynamically so as to increase or decrease the level of fog affecting a given set of nodes.

- ***Using auto-focus:*** As soon as a user has seen the relevant information, they may want to go back to the sharp display. This is done with the auto-focus feature, which brings all objects back into sharp focus again – after a certain timeout or after being triggered by users.

In early versions of the software developed in this project the visualisation of nodes was based on relevancy functions. The classes developed are designed to be flexible though, and it was an important requirement that the completed system would be compatible with a wide variety of data. For that reason the visualisation is based on a list of node identifiers (as mentioned earlier) and the decision on what nodes are in that list is left up to the user. The idea being that if the system was integrated with another application then that application could, with relative ease, make adjustments to their code to determine which item to visualise.

There was no way of changing the fog threshold once it had been set up due to the complicated argument discussed earlier (fog density versus node size).

The auto-focus concept is not so difficult to incorporate, there was already the inclusion of methods to visualise all of the nodes, but there was the addition of a 'reset' function that would allow for the graph to be totally re-drawn in case of any ambiguities.

3.6 View 2: Emissive Lighting

Another method that this project revealed is the possibility of using emissive lighting to improve visualisation; the relevant parts of the display could 'glow in the dark'. The lighting aspect is of secondary importance to Semantic Depth of Field in this project and it must be made clear that the research for it was not as thorough.

Perhaps the key aspect of this approach was that it allowed for a 3D representation of the graph. So, whereas in signal-1 mode (Fog) the Z coordinate of a node was restricted to being one of two values (depending on relevancy) it could now be governed by the layout algorithms. Of course there is a disadvantage of this; the size of the nodes will now appear varied and this will significantly affect the analysis of the results of emissive lighting. This could be covered later by the inclusion of some testing of this approach in two dimensions.

The basic approach was to develop a simple function that would darken the colour of the light that a node was emitting. The easiest way to do this was to extract the RGB (red, green, blue) components from the colour and then reduce them by some constant value before using the new RGB values to construct a new (darker) colour.

Chapter Four

Implementation

4 Implementation

This chapter will begin with a high-level explanation of the implementation of the software. There will also be a thorough analysis of the implementation of the core components of the system as determined by the design (Chapter 3), which will include segments of the code itself. The code in this section is partial segments, simplified for the purposes of making a clear explanation of the implementation. For a full listing of all the source code see the Appendices (A & B).

4.1 Overall Application

The software was implemented in Java, making much use of the Java 3D API. There was an implementation of two distinct pieces of software; stand-alone executable and the 'bolt-on' system. At the core of this visualisation system lies an implementation of a graph system similar to that used in the Sun Graph-Layout applet [37], only extended into three-dimensions. The system implements two visualisation techniques; Semantic Depth of Field and Emissive Lighting. These two techniques are presented in their own 'views' on the model, along side the Sun Applet, so the controller instantiates three possible views.

The stand-alone application has been constructed to allow for the analysis of data (using Semantic Depth of Field) that is taken in from a file (.mdl). A file loader class is used to convert a file to a standard data model which is used throughout both systems. When using the bolt-on classes it is necessary to manually set up the model and use the add/remove methods that it provides to bring about changes in the model.

The classes written form the package *com.logan.JVS*, some of which are used for the 'bolt-on' system, all of which are used for 'stand-alone' application. The implementation consists of 25 classes in total, of which 21 are required when using the program in its 'bolt-on' form. There are 9 classes which make use of the Java 3D API.

4.2 The Data Model

The first, rather straight-forward, stage was to implement two classes to represent an individual node and edge:

```
class Node {
  float x;
  float y;
  float z;
  float zFog;         // Z axis positon when using fog

  // Additional data required for graph layout function
  double dx;
  double dy;
  double dz;

  boolean fixed;      // Motion allowed?
  String label;       // Node name
  int ident;          // Node unique identifier
  int group;          // For grouping (area-code)
}
```

Fig 4.2.1: The class *Node*.

```
class Edge {
int ident;  // Unique identifier

int fromID; // From Node ID
int toID;   // To Node ID

float len;  // Weight
}
```

Fig 4.2.2: The class *Edge*.

The next stage was to implement a mechanism for storage of these nodes and edges in a data model that could then be passed around the system. This was achieved with two simple arrays:

```
               public Node[] nodeArray;  // Array of nodes
               public Edge[] edgeArray;  // Array of edges
```

Fig 4.2.3: The creation of arrays to hold instances of the classes *Node* and *Edge*.

The initial design for the data model (Chapter 3) stated that, fundamentally, the model needed to implement the following functions:

- Add / Delete a node
- Add / Delete an edge
- Grow / Shrink an array

The system was implemented in such a way that edges are added one by one, and if a node within that edge does not exist (node exists function) then it too is added to the model. Each time an addition to the model is made, the counters are incremented and the relevant array is grown by one. The core of the *addEdge* method is as follows:

```
if(nodeExists(temp1) == false) {          // From node does not exist
      addNode(from);
}
if(nodeExists(temp2) == false) {          // To node does not exist
      addNode(to);
}
edgeArray = (Edge[])arrayGrow(edgeArray);// Increase size of array by 1
numEdges++;                              // Increment number of edges
edgeArray[numEdges-1] = new Edge();      // Create new edge
```

Fig 4.2.4: The key functionality within the method to add edges.

To prevent the system being limited by an initial assignment dictating how many nodes and edges the system would hold, two methods were implemented to allow the growing and shrinking of either array during the execution of the program. The methods were implemented using reflection to write generic array code.[12] The class *Array* in the *java.lang.reflect* package allows the programmer to create arrays dynamically. The grow method is very similar to the shrink method and both are designed to cope with arrays of any type:

```
private Object arrayGrow(Object a) {
      Class c1 = a.getClass();
      if(!c1.isArray()) {
            return null;
      }
      Class componentType = c1.getComponentType();
      int length = Array.getLength(a);
      int newLength = length + 1;
      Object newArray = Array.newInstance(componentType, newLength);
      System.arraycopy(a, 0, newArray, 0, length);
      return newArray;
}
```

Fig 4.2.5: A method to grow the size of an array by one item.

4.3 The Graph System

The software implemented a 'graph' class, the class *Graph3D* imports the node and edge data from the model (passed to it). The first step that class takes is to collect the node and edge data from the model:

```
nodes = model.returnNodes();
edges = model.returnEdges();
numNodes = model.nodeArraySize();
numEdges = model.edgeArraySize();
```

Fig 4.3.1: The use of the model to retrieve node and edge data.

The next stage was to implement the actual Java 3D elements that would be necessary for creating a graphical representation of the data. The main steps are creating the 'canvas', then the 'scene-graph' and then adding this to the 'universe':

```
canvas = new Canvas3D(config);
u = new SimpleUniverse(canvas);
scene = createSceneGraph();
u.addBranchGraph(scene);
```

Fig 4.3.2: The four main steps involved in incorporating Java 3D.

The layout code is quite complex and lengthy and can be seen, in full, in the Appendices. Basically it centres on the use of a thread and some code which makes dynamic alterations to the coordinates of a node depending on its current coordinates and the overall characteristics of the data set:

```
public void run() {
  Thread me = Thread.currentThread();
  while(relaxer == me) {
    relax();
    if(Math.random() < 0.03) {
      Node n = nodes[(int)(Math.random() * numNodes)];
      if(!n.fixed) {
        n.x += 200*Math.random() - 50; // Were all 100 - 50
        n.y += 200*Math.random() - 50;
        n.z += 200*Math.random() - 50;
      }
    }
  try {
    Thread.sleep(5);
  }
  catch(InterruptedException e) {
    break;
}}}
```

Fig 4.3.3: The initial stages in implementing the dynamic aspect.

In terms of the actual drawing of the nodes and edges, this is done in two separate classes, class *DrawNode* and class *DrawEdge*. These are both Java 3D intensive classes and the results of their execution are simply added to the 'scene' created in the class *Graph3D*:

```
public BranchGroup createSceneGraph() {
      objRoot = new BranchGroup();

      JVSnode = new DrawNode(this, signal);
      JVSedge = new DrawEdge(this, signal);

      objRoot.addChild(JVSnode);
      objRoot.addChild(JVSedge);

      return objRoot;
}
```

Fig 4.3.4: The calls resulting in the drawing of nodes and edges.

There is quite frequent use of a method to make changes in the display following alterations to coordinates; this method simply results in calls to 'update' methods in the draw classes that know how to re-draw a node or edge. The drawing of nodes in the class *DrawNode* is achieved using Java 3D *Sphere* objects. The drawing of edges in the class *DrawEdge* is achieved using Java 3D *LineArray* objects. Each edge is drawn with a width that is determined by its weight, obviously if all the weights are the same (or defaulted) then the edges will be created with the same width. At no stage in the implementation are there any changes to the size of a node, this is not permitted by Semantic Depth of Field.

4.4 View 1: Semantic Depth of Field

The class *Graph3D* contains the following method to add the fog to the scene; the approach requires that the fog is initialised from the onset and then the location of the nodes on the depth axis (Z axis) is manipulated depending on their perceived relevance:

```
private void setupFogs() {
      .........

      // Set up the Linear fog - distances from the viewer
      float fogFront = (viewDistance - coordLimit) + 0.1f;
      float fogBack  = (viewDistance - coordLimit) + 1.2f;

      LinearFog fogLinear = new LinearFog(black, fogFront, fogBack);
      fogLinear.setInfluencingBounds(infiniteBounds);

      .........
}
```

Fig 4.4.1: The basic steps involved in setting up linear fog.

The class *Graph3D* also contains a visualise method that is passed an array of node identifiers from the graphical user interface. This method then makes the necessary calls in the class *DrawNode* to move the nodes accordingly:

```
public void visualise(int[] relevantNodes) {
      .........

      JVSnode.applyFog(allNodes,false);     // False -> Fogs
      JVSnode.applyFog(relevantNodes,true); // True  -> Un-Fogs
      JVSedge.update();
      // spheres are updated automatically through DrawNode!

      .........
}
```

Fig 4.4.2: The method to trigger movement of a group of nodes in or out of the fog.

The fogging method in the class *DrawNode* does not actually have anything to do with fog. The method simply changes the node coordinates, resulting in some nodes (irrelevant/context) moving into the fog that has already been created. The method takes as input a Boolean (true/false) that tells the method to either move the nodes forward (out of the fog), or backward (into the fog):

```
public void applyFog(int[] toMove, boolean front) {
      float distance;       // Distance to position node at
      if(front == true) {   // Moving selected nodes to the front
            distance = 10.0f;
      }
      else {                // Moving selected nodes to the back (fog)
            distance = 9.0f;
      }
      .........
      for(int y = 0; y < moveLen; y++){ // Each node to be moved
        for(int nodeNum = 0; nodeNum < nodeLen; nodeNum++) {
          int ID = nodes[nodeNum].ident;
          if (ID == toMove[y]) { // Match - move
            nodes[nodeNum].zFog = distance; // Change the depth
            break;
          }
        }
      }
      .........
}
```

Fig 4.4.3: The actual method to alter the depth coordinates of a group of nodes.

4.5 View 2: Emissive Lighting

The class *Graph3D* also contains a visualise method that is passed an array of node identifiers from the graphical user interface. It was demonstrated in the previous section how this method brings about fogging and in this section the same method makes the necessary calls in the class *DrawNode* to light the nodes accordingly:

```
public void visualise(int[] relevantNodes) {
      .........
      JVSnode.applyLight(allNodes,true);        // True  -> Dims
      JVSnode.applyLight(relevantNodes,false);  // False -> Brightens
      .........
}
```

Fig 4.5.1: The method to trigger the darkening of a group of nodes.

The lighting method in the class *DrawNode* changes the emissive colouring of nodes, resulting in some nodes (irrelevant/context) appearing darker than the visualised nodes, which ‘glow in the dark’. The method takes as input a boolean (true/false) that tells the method to either brighten the nodes (identifiers passed to it also), or dim them:

```
public void applyLight(int[] toColour, boolean darken) {
      .........
      if(darken == true) {
        darkenColour();
        for(int y = 0; y < colLen; y++){// Each node to colour
          for(int nodeNum = 0; nodeNum < nodeLen; nodeNum++) {
            int ID = nodes[nodeNum].ident;
            if (ID == toColour[y]) { // Match - change colour
              ColoringAttributes colrAttr = new
                ColoringAttributes(darkColour,
                ColoringAttributes.SHADE_GOURAUD);
              appear[nodeNum].setColoringAttributes(colrAttr);
              break;
        }}}
      }
      .........
}
```

Fig 4.5.2: The actual method that sets the new colour of a group of nodes.

If the Boolean passed is false and the nodes are to be brightened, then the code is very similar to the above, only the colour is set to the original colour before darkening. The darkening method, called in the above segment of code, simply subtracts values from each of the red, green and blue components of the current node colour:

```
public void darkenColour() {
      Color temp = curColour.get();
      int red   = temp.getRed();
      int green = temp.getGreen();
      int blue  = temp.getBlue();

      red = red - 110;
      if(red < 0) {
            red = 0;
      }
      green = green - 110;
      if(green < 0) {
            green = 0;
      }
      blue = blue - 110;
      if(blue < 0) {
            blue = 0;
      }
      temp = new Color(red, green, blue);
      darkColour = new Color3f(temp);
}
```

Fig 4.5.3: The method to create the new, darker, colour.

4.6 Code Description – Core Classes

This section provides a summary of each individual class that is used in the 'bolt-on' system; the more important classes are highlighted in **bold**. All of the classes in this section are also required (as well as additional classes) for use of the stand-alone application. For more detail see the documented source code listing in Appendix A.

4.6.1 **Class - Model**

This class represents the Model from the Model-View-Controller paradigm. It forms the Object that is passed around the program in order to access the details of the graph elements (nodes and edges). It provides methods to manipulate what it holds (e.g. add a node) and also methods to report on its status (e.g. the number of nodes). It can hold arrays (of nodes and edges) of any size through the inclusion of array 'grow' and 'shrink' methods.

4.6.2 Class - Node

This class and the one that follows are very closely related to the class *Model*. Node objects are created (and stored in an array) in the class *Model* and represent the individual entities in the graph visualisation system. They do not actually represent the visual representation of the nodes in Java 3D; the nodes are drawn in a separate class (*DrawNode* – 4.6.6).

4.6.3 Class - Edge

This class and the one previous are very closely related to the class *Model*. Edge objects are created (and stored in an array) in the class *Model* and represent any connections that exist between entities in the graph visualisation system. They do not actually represent the visual representation of the edges in Java 3D; the edges are drawn in a separate class (*DrawEdge* – 4.6.7).

4.6.4 **Class - Graph3D**

This class represents the View from the Model-View-Controller paradigm. This class is responsible for creating the 'canvas' upon which the nodes and edges will be drawn. It is also responsible for the layout of the nodes and the dynamic aspect of the graph. It makes calls to two additional classes in order to bring about the drawing of the nodes and edges; see section 4.6.6 and 4.6.7. The class also provides a substantial amount of additional functionality, such as that which enables the user to take snap-shots of the screen, and that which enables zooming and panning of the view.

4.6.5 Class - Graph3DPanel

This represents the Controller from the Model-View-Controller paradigm, there is a second controller that is necessary for when the program is ran in stand-alone mode. This control panel is optional for users of the bolt-on system; if it is not required then many of the classes in the package would not be needed either.

4.6.6 **Class - DrawNode**

This class is responsible for drawing a series of nodes based on information in the model. This is the transition between the representation of a node as an object (class *Node*) and the drawing of that node using spheres in Java 3D. This class also contains the code that allows each node to display a label showing its name. The emissive lighting technique is implemented solely in this class and the code that moves the nodes in and out of the fog is also here (the fog itself is actually set up in the view; 4.6.4). The 'update' function allows the graph view to bring about changes in the display resulting from the work of the layout algorithm.

4.6.7 **Class - DrawEdge**

This class is responsible for drawing a series of edges based on information in the model. This is the transition between the representation of an edge as an object (class *Edge*) and the drawing of that edge using lines in Java 3D. This class also contains the code that allows each edge to display a label showing its name.

4.6.8 Class - DrawGrid

This class is responsible for drawing the lines that mark the coordinate system. This is not a particularly important feature but can be useful when panning in 3D.

4.6.9 Class - Constants

This class consists of numerous definitions and objects that are used throughout the program. All the various colours used by the application are probably the most important component of this class. It also stores URLs and file/folder names as well as anything else that is used repeatedly in the application.

4.6.10 Class - HtmlViewer

This is a useful class that provides an external window to view help files, web sites and a textual representation of the model currently in memory. The idea is simple; pass it a URL and it returns an elegant little window displaying the results. The textual representation of the model does not currently work in 'bolt-on' mode due to the fact that it is actually based on the file currently in use, not the model itself.

4.6.11 Class - KeyPrint

This class deals with key presses from the user, the system is not set up to respond to any of these, though it is straight-forward to change that.

4.6.12 Class - IntersectInfo

This class was non-existent until the latter stages of the project. It is included to show the perceived importance of future development in this area of the program.

4.6.13 Class - About

This class creates a small panel detailing a selection of both program and local environment details.

4.6.14 Class - BackTool

This class contains the code that allows the user to change the background displayed in the view (class *Graph3D*). It is included in the package as 'proof of concept' more than anything else. This class is the sole user of three other classes in the system; class *IntChooser*, *IntEvent* and *IntListener*. These classes are not yet implemented efficiently and are not discussed here as it seems unnecessary.

4.6.15 Class - OffScreen

This class contains the functionality for taking screen-shots of the scene. This function has proved invaluable in evaluating results as the project progressed and has helped keep the relevant staff informed of any developments.

4.7 Code Description – Additional Classes

This section provides a more detailed look at the implementation details of the additional classes used in the stand-alone version of the system, the more important classes are highlighted in **bold**. For more detail see the documented source code listing in Appendix B.

4.7.1 **Class - FileLoader**

This class liaises with the class *Model* to allow loading of data from a file. This functionality was crucial for testing and development of the program prior to the integration with a database system. It was designed to follow the same procedure as that which is recommended to integrators using the 'bolt-on' system. This means that it reads each line of a file and calls *addEdge* in the class *Model* to fill the data model with the information in the file.

4.7.2 **Class - Controller**

This is the main Controller (MVC) object. This is the user interface that the end-user is presented with when running the program in its stand-alone form. It does not serve as a replacement to the optional control-panel (class *Graph3DPanel*) which provides control functionality more closely related to manipulating the view itself. This controller is more concerned with the aspects of model manipulation than view manipulation. It is assumed that for any manipulation of the model in 'bolt-on' mode the designer of that system will develop their own classes tailored to their specific needs.

4.7.3 Class - Application

This is the class that contains the *main* method that will be called to begin execution of the stand-alone program. Class *Application* forms the executable object for JVS. This class is the only one outside the package, instead it imports the package to utilise the classes it provides. It calls the constructors of class *LoadScreen* (to display a small splash screen) and then class *Controller* (to load the GUI).

4.7.4 Class - ModelFilter

This is a small class that provides some file filtering functionality to ensure that when loading models the user can only view files of the correct format (.mdl).

4.7.5 Class - LoadScreen

This is another small class that creates and then displays a brief 'splash screen' that entertains the user while the remainder of the program is loaded into memory.

4.7.6 Class - SunGraph

This is an implementation based on the Sun Graph-Layout applet [37] that can be used to view an additional representation of the model in memory. Given that part of the motivation for this project was the lack of visualisation in what was otherwise a useful view this seemed like an appropriate inclusion.

4.7.7 Class - SunGraphPanel

This class forms part of the Sun Graph-Layout implementation mentioned above. Much of the dynamic layout code in the class *Graph3D* is based on the graph layout algorithm implemented here.

Chapter Five

Testing

5 Testing

The essence of software testing is to determine a set of test cases for the item to be tested. A test case should consist of:

- Inputs: pre-conditions and the actual inputs.
- Outputs: post-conditions and the actual outputs.

The act of testing entails establishing the necessary pre-conditions, providing the test case inputs, observing the outputs, and then comparing these with the expected outputs to determine whether the test passed.[14] Two fundamental approaches are used to identify test cases:

- Functional (or Black Box) Testing
- Structural (or White Box) Testing

The key areas of the program have been tested and one of the benefits of integration was that there was a physical user-base that could report additional errors or ambiguities. One of the most important problem areas, highlighted by the integration with the data-mining application, was the requirement of the previous implementation that edge weight data was inputted. Indeed in many applications this weight data may not be relevant, or available. This lead to modifications allowing the passing of 'pure' edge data, with no additional information required. The result is additional flexibility of the program to handle a more varied form of data input. Perhaps a less important problem, but a further discovery via integration leads on from the difficulties just mentioned: the displaying of unnecessary edge labels. If the system was to set up default edge weights when none were passed to it then labels detailing those weights would almost certainly not be required. The system was modified to allow for the node and edge labels to be turned on or off, independently of one another.

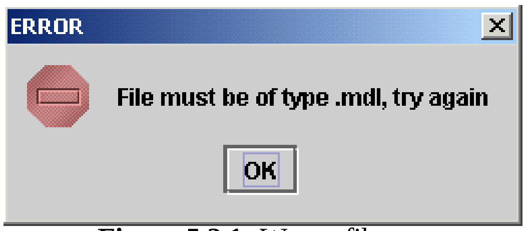
5.1 Functional Testing

The functional approach to software testing considers only the information in the specification document. [14] The key aims and objectives of this project as governed by the specification are presented in Chapter 1. Given the nature of this project as primarily an implementation of visualisation techniques, it was difficult to carry out functional testing. The completion of objectives is discussed in section 8.2, Chapter 8.

5.2 Structural Testing

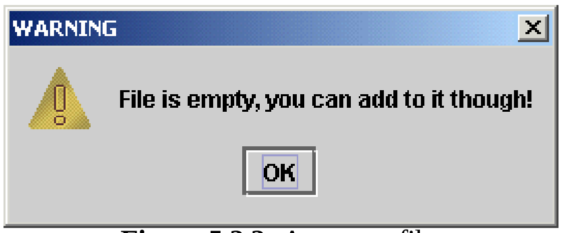
The essential difference between this approach and the functional approach is that the implementation is known and used to identify test cases.[14] The essential aspect of the implementation that needed testing was the data model. The testing of the model would actually focus around two of the classes in the system; class *Model* and *FileLoader*. This is a shortened summary of the tests, followed by the resulting action of the program;

1. Wrong file type (e.g. '.jpg') – requires changing view option in the loader dialog.
2. Empty file of the correct extension (.mdl).
3. File full of random characters.
4. Grouping nodes using the 'area-code' concept (e.g. "bob/2").
5. Connecting a node with itself.
6. Creating duplicate edges (Note: opposite direction is not an identical edge).


ERROR

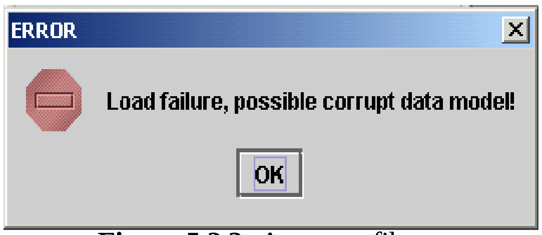
File must be of type .mdl, try again

Figure 5.2.1; Wrong file type.


WARNING

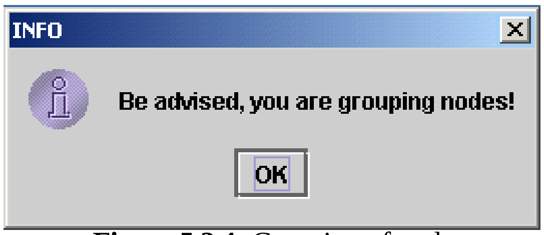
File is empty, you can add to it though!

Figure 5.2.2; An empty file.


ERROR

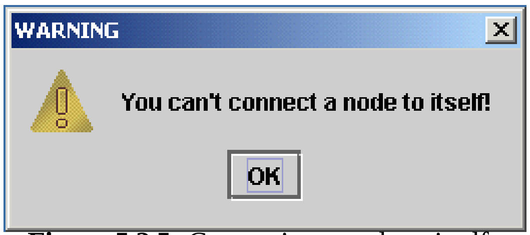
Load failure, possible corrupt data model!

Figure 5.2.3; A corrupt file.


INFO

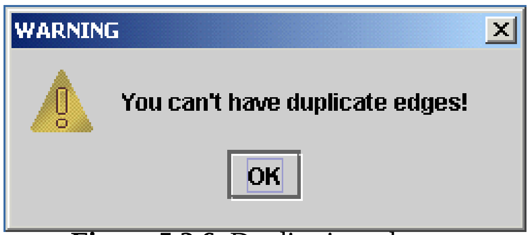
Be advised, you are grouping nodes!

Figure 5.2.4; Grouping of nodes.


WARNING

You can't connect a node to itself!

Figure 5.2.5; Connecting a node to itself.

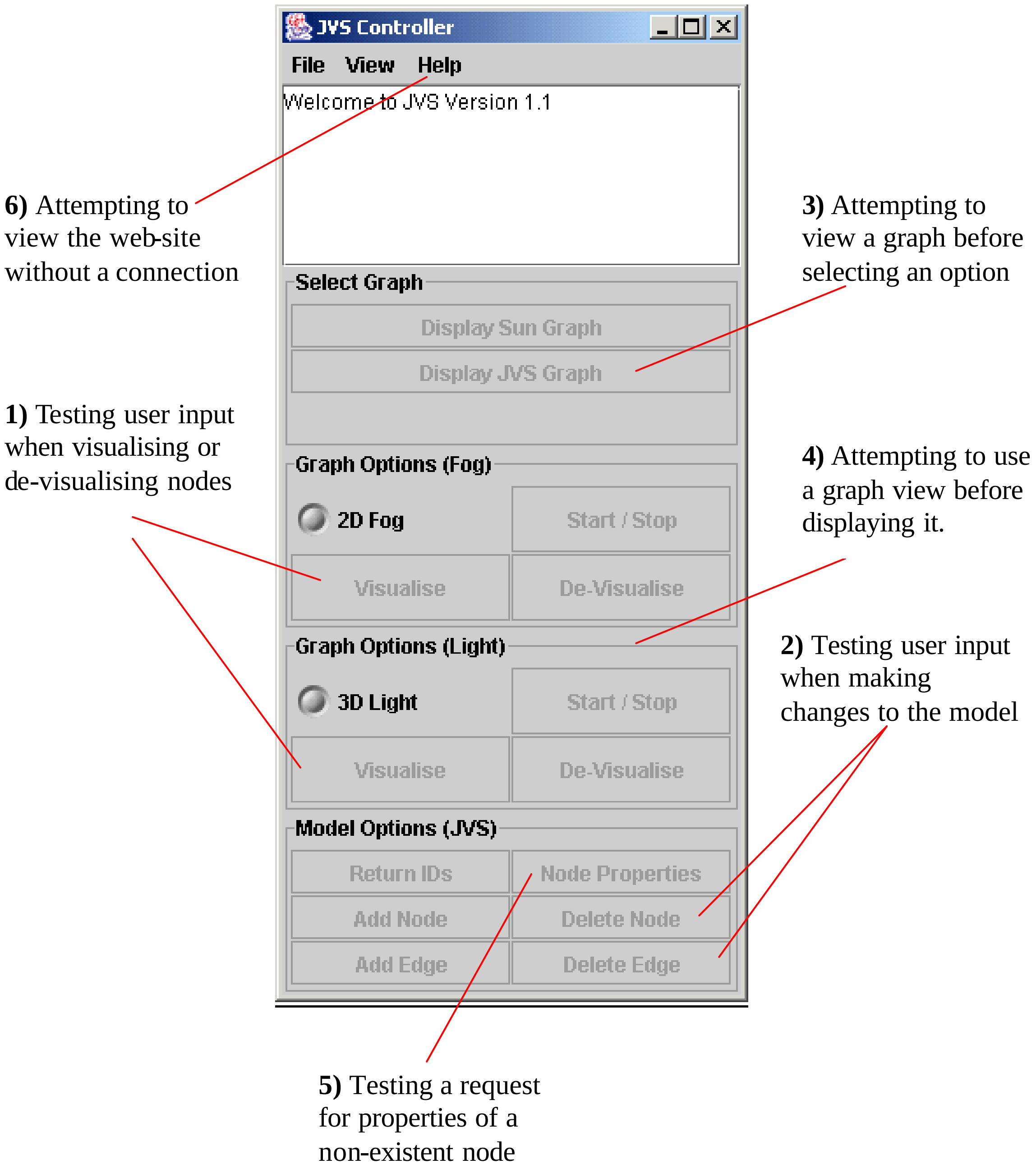

WARNING

You can't have duplicate edges!

Figure 5.2.6; Duplicating edges.

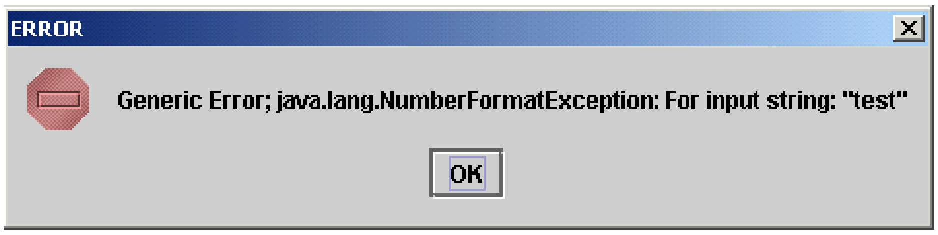
5.3 User Interface Testing

The main characteristic of any graphical user interface (GUI) application is that it is event driven. The user interface for this application aimed to create a 'guided' event sequence.[14] In this case this means that the user is guided through the system by the enabling and disabling of buttons relevant to their current position in the system. For example, if a user has yet to load a data model into the system then the user interface will physically prevent them from loading a graph view. There were still various aspects of the controller (stand-alone application) that needed thorough testing, illustrated by the diagram:


JVS Controller
File View Help
Welcome to JVS Version 1.1
Select Graph
Display Sun Graph
Display JVS Graph
Graph Options (Fog)
2D Fog
Start / Stop
Visualise
De-Visualise
Graph Options (Light)
3D Light
Start / Stop
Visualise
De-Visualise
Model Options (JVS)
Return IDs
Node Properties
Add Node
Delete Node
Add Edge
Delete Edge
6) Attempting to view the web-site without a connection
3) Attempting to view a graph before selecting an option
1) Testing user input when visualising or de-visualising nodes
4) Attempting to use a graph view before displaying it.
2) Testing user input when making changes to the model
5) Testing a request for properties of a non-existent node

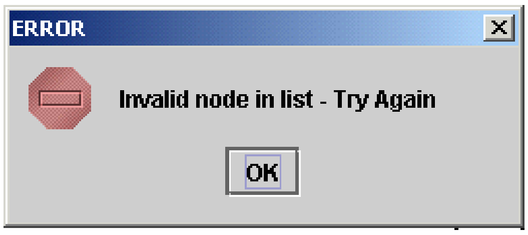
Test 1:

The testing of user-input when visualising or de-visualising nodes. The user is expected to enter data in the format correct format (node IDs separated by blank spaces). Cancelling of user input is handled and if the user enters anything other than a number or series of numbers they will be presented with this message;


ERROR

Generic Error; java.lang.NumberFormatException: For input string: "test"

If the user attempts to visualise any nodes that do not exist then they will see the next message. A list of node identifiers for the current view is available by clicking 'Return IDs'.

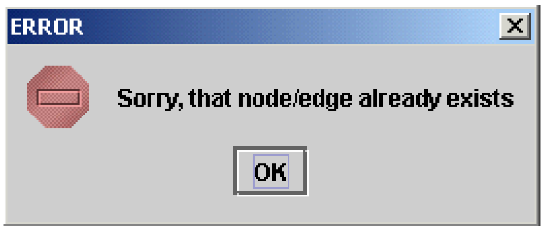

ERROR

Invalid node in list - Try Again

Test 2.

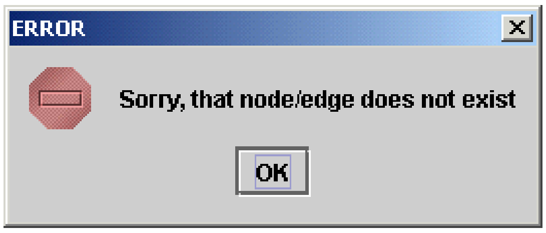
The testing of user-input when making changes to the model. There are four possible problems here;

- Adding a node/edge that already exists


ERROR

Sorry, that node/edge already exists

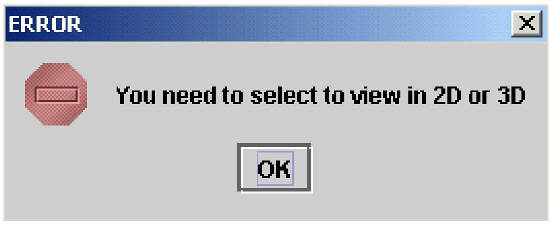
- Deleting a node/edge that does not exist


ERROR

Sorry, that node/edge does not exist

Test 3.

Attempting to view a graph before selection (i.e. clicking on 'Display JVS Graph') will result in the following warning message:

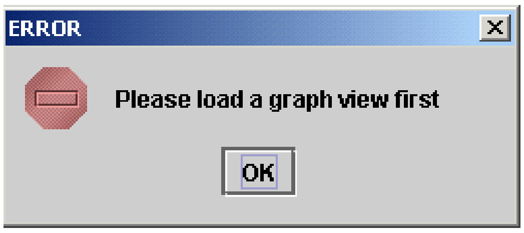

ERROR

You need to select to view in 2D or 3D

OK

Test 4.

Attempting to manipulate a graph before it exists will result in the following warning message:


ERROR

Please load a graph view first

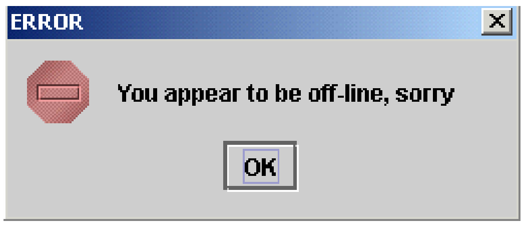
OK

Test 5.

A request for the properties of a node that does not exist will result in the same warning message as when attempting to delete a node/edge when it does not exist.

Test 6.

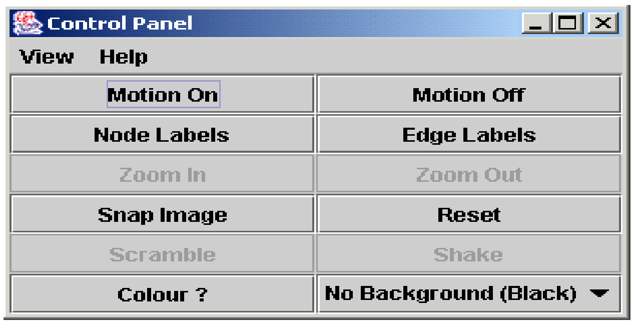
If the us er attempt to view the JVS web site and the application can not find a network connection to resolve the URL then the following message will result;


ERROR

You appear to be off-line, sorry

OK

On the optional control panel (shown below) there is no scope for any input errors. The buttons representing functionality that at a given time should not be available are simply turned off and unavailable to the user. The only other issue is the cancelling or resetting of the colour dialog, which are both handled correctly.



5.4 Error Handling

The Java programming language uses a form of error-trapping known as ‘exception handling’.[12] The reliability in the software presented here is, in a large part, due to the use of exception handling. If an operation can not be completed because of an error, the program attempts to either:

- Return to a safe state and enable the user to execute other commands;

or

- Allow the user to save their work and terminate the program gracefully.

If a method is unable to complete its task in the normal way it ‘throws’ an object that encapsulates the error information. The method exits immediately and does not return any value. This is a very useful feature and was used where possible to deal with errors in the program. Of course it is still sensible to design software to be as reliable as possible, but exception handling can provide back-up error recovery. What follows is a code segment and the results of action following error detection:

```
if(event.getSource() == properties) {
  try {
      String find = JOptionPane.showInputDialog("Enter node ID");
      String props = model.properties(Integer.parseInt(find));
      JOptionPane.showMessageDialog(Controller.this, props,
           "Node Details", JOptionPane.INFORMATION_MESSAGE);
      }
  catch (Exception e) {
      genericErrorMessage(e);
  }
}
```

This is the method for dealing with requests from the user to provide information on a given node. Placing the code that takes user-input inside a ‘try’ clause implies that if any exceptional circumstances arise the method will terminate and the ‘catch’ clause that follows will deal with the error. In most areas of the program there is more advanced error detection and handling whereby values are analysed and more specific actions can be taken, but this is demonstrates the basic approach.

Chapter Six

User Manual

6 User Manual

This chapter is designed to assist the end-user with the installation, operation and maintenance of the software described in this dissertation. The complete documentation is available in Appendix C. The program developed has been named Java Visualisation System (JVS). The software was designed to take two forms; as an add-on to an existing application and as a stand-alone application. The deployment will differ slightly depending on the version that has been chosen. The manual is broken up into 6 distinct Help-Sheets:

1. Helps the user with the installation of the program.
2. Helps the user with integration aspects relating to the Class Model.
3. Helps the user with integration aspects relating to the Class Graph3D.
4. Helps the user with operation of the program.
5. Helps the user with the various views that the program produces.
6. Helps the user with the operation of the optional view control-panel.

6.1 Help Sheet – 1; Installation

Java Visualisation System
Version 1.1
Installation
Help Sheet - 1

This document is designed to assist you with installation of the Java Visualisation System (JVS). It will tell you where to download the program and how to install it on your machine. Once you have completed installation there are additional documents designed to help you with the running of the program.

This program was not written for use with any particular operating system. The current version has been tested on all three major operating systems (Win32, Unix/Linux and Mac). It was designed to exploit the platform-independence that developing in Java allows.

Follow these simple steps for a pain-free installation:

- ✓ Check that you have the latest Java Software Development Kit (SDK); this is available from the Sun Java web site.
  http://java.sun.com/j2se/downloads.html

- ✓ Check that you have the latest Java 3D SDK. The program was developed using the OpenGL version of the Java 3D software; there are a number of limitations in the DirectX release.
  http://java.sun.com/products/java-media/3D/download.html

- ✓ Download the JVS software, either from the CD or the web site.
  http://homepages.cs.ncl.ac.uk/ben.logan/home.formal/project.htm

- ✓ Install the program by simply copying the downloaded directory to the current working directory of your project or, if running JVS as a stand-alone application, anywhere you like!

This is a listing of exactly what the JVS folder contains, if you only plan to use the 'bolt-on' classes then much of this is not required:

Directory / File	**Description**
Application.class	'java Application' if the class-path has been set correctly
JVS.jar	executable archive of the program
JVS.jpx	project file for use with Borland JBuilder
readme.txt	a little extra information
com	the first directory in the package hierarchy
images	images used during execution of the program
models	test models available
screenshots	screenshots will be stored here
Non-essential Directories;	
doc	documentation for the program classes
help	help files
html	nicely formatted html files of the source code
src	the actual Java source code
uml	UML diagrams of each of the classes

Stand-Alone

If you are running JVS as a stand-alone application then the set-up is now complete, to run the program simply enter either of the following;

```
java Application
```

or

```
java –jar JVS.jar
```

Note: For an understanding of how to create new models for use with JVS check the example files in the 'model' directory. There is information on grouping nodes in *Help Sheet – 2*.

Bolt-On

If you wish to incorporate JVS into your existing application then it is necessary to add some additional code to your program. The first step is to include the JVS package at the top of any of your classes that will require use of it;

```
import com.logan.JVS.*;
```

Please refer to *Help Sheet - 2* and *Help Sheet - 3* for additional help with how to incorporate the JVS classes into your existing application.

6.2 Help Sheet – 2; The Model

Java Visualisation System
Version 1.1
The Data Model
Help Sheet - 2

This document is designed to assist you in creating and initialising a model for use with the 'bolt-on' version of the Java Visualisation System (JVS). It will walk through how to fill the model with data from a database, and then how to perform operations to manipulate the model once it has been created.

Remember that the first step is to include the JVS package at the top of any of your classes that will require it;

```
import com.logan.JVS.*;
```

The class *Model* constructor requires no parameters to be passed to it. To create an instance of the class *Model* in your program, simply add this code;

```
Model model = new Model();
```

The next step, now that you have created a model, is to fill it with your data. The standard (expected) way of doing this is to add a series of 'edges' to the model;

```
model.addEdge(from, to, weight);
```

Where 'from' and 'to' are Java Strings representing the names of the nodes at each end of the edge. Choose sensible names for the nodes as you will have the option of displaying them as labels along side each node. The parameter 'weight' is a Java Float representing the weight of the edge. If you are not interested in modelling edge weights then you should just pass the method '0' and the program will assign a default value.

Alternatively you can add individual (un-related) nodes as follows;

```
model.addNode(name);
```

Where 'name' is a String representing the name of the node that you wish to create.

Grouping Nodes:-

If you wish to group nodes together, by demanding that they are located in a given region of the screen, you need to add an 'area code' to the name of the node.

Ungrouped node:	Richard
Grouped node:	Richard/2

There are four possible locations, the area codes are shown opposite;

1	2
3	4

The other two important methods available are for deleting nodes and edges from the model, this can be achieved as follows;

```
model.delEdge(ID);
```

Where 'ID' is a Java Integer representing the unique identifier of the edge that you wish to remove from the model.

```
model.delNode(ID);
```

Where 'ID' is a Java Integer representing the unique identifier of the node that you wish to remove from the model.

Now that you have created a Model, it is time to pass it to the JVS viewer to see the visual representation of your data. This step is discussed in detail separately in *Help Sheet - 3*.

6.3 Help Sheet – 3; The Graph

Java Visualisation System

Version 1.1

The Graph

Help Sheet - 3

This document is designed to assist you in creating and initialising a graph view for use with the 'bolt-on' version of the Java Visualisation System (JVS). It will walk through how to use the visualisation techniques and will explain how to communicate with the graph to achieve the desired effect.

Remember that the first step is to include the JVS package at the top of any of your classes that will require it;

```
import com.logan.JVS.*;
```

If you have followed all the steps in *Help Sheet – 2* then you are ready to create a graph view of the data model;

```
Graph3D graph = new Graph3D(model, signal, panel);
```

Where 'model' is the instance of the class *Model* that you have just created. The parameter 'signal' is a Java Integer representing the way in which you wish to view the graph;

1 = Fog visualisation technique (2D).
2 = Emissive lighting techniques (3D).

The third parameter 'panel' is a Java Boolean that tells the program whether to load up the control panel (*Help Sheet - 6*), set this to true if you want to see the panel. This provides access to all the basic functionality that is associated with the graph.

After making the above call you should see a new graph view window, displayed in a separate JFrame over the top of your existing application. Now we will see how to access the facilities for manipulating this graph and improving the visualisation.

Note:-

If you make any changes to the Model while still displaying the graph it will be necessary to inform the graph so that they can take affect. To do this call;

```
graph.newData(model);
```

Perhaps the two most important methods are those which allow you to visualise (and de-visualise) a set of nodes;

```
graph.visualise(toVis);
```

Where ‘toVis’ is an array of Java Integers representing the node identifiers of the nodes that you wish to view in the context of the others.

```
graph.removeVis(deVis);
```

Where ‘deVis’ is an array of Integers representing the node identifiers of the nodes that you wish to remove from your current field of interest.

There is also a method available to remove the visualisation from all the nodes in the graph, to save you from passing the method a list of all the node identifiers;

```
graph.removeAllVis();
```

The methods available for using the dynamic aspect of the graph visualisation system are now discussed. The dynamic aspect is responsible for animating the graph. The two most useful methods here are obviously those that turn this feature on and off;

```
graph.start();
graph.stop();
```

The two other methods of interest consist of one which allows you to rearrange the nodes and another that gives the nodes a slight ‘shake’ to bring them back to life;

```
graph.scramble();
graph.shake();
```

As the above calls trigger the generation of new views there is no need for any operational discussion when running the program in ‘bolt-on’ mode. Skip *Help Sheet – 4* and proceed to 5 and 6 for help with understanding the views and operating the optional control-panel.

6.4 Help Sheet – 4; Operation

Java Visualisation System
Version 1.1

Operation

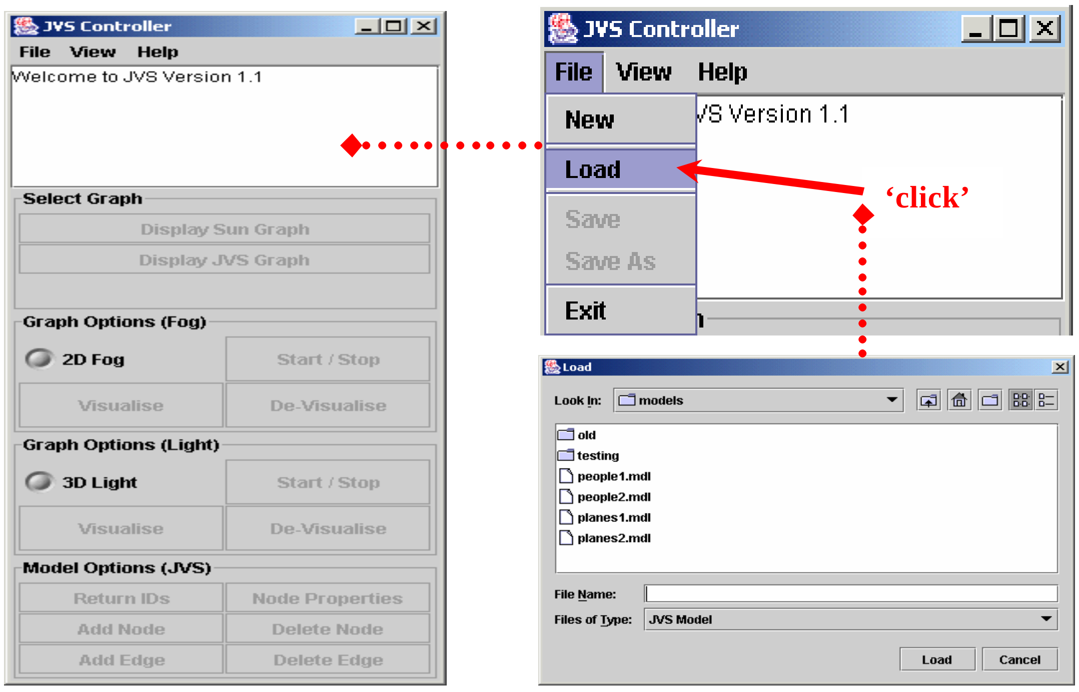
Help Sheet - 4

This document is designed to assist you in the operation of the stand-alone version of Java Visualisation System (JVS). It will walk through how to use the controller and explain how it interacts with the various other aspects of the program. For details on how to use the optional view control-panel read *Help Sheet – 6*.

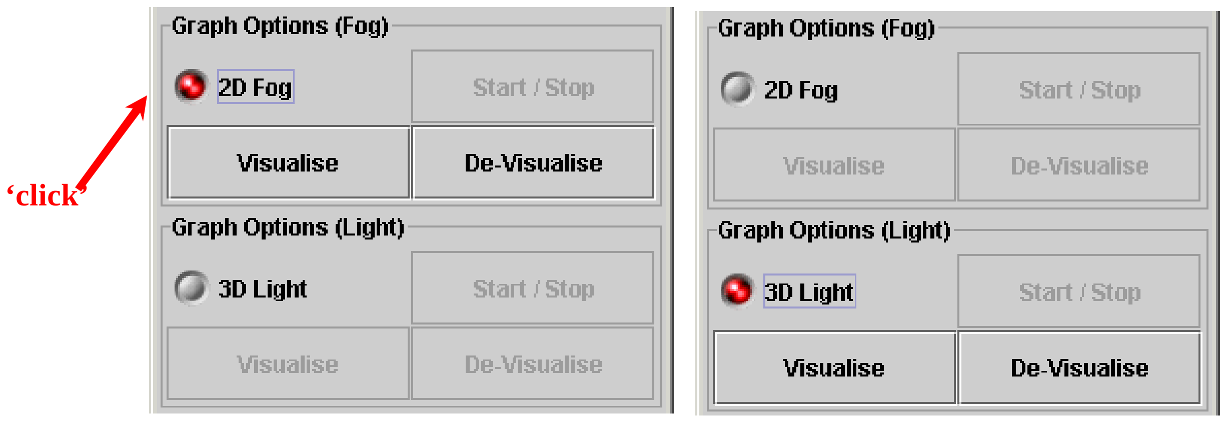
To run the application; enter either of the following at the command prompt;

```
java Application
      or
java –jar JVS.jar
```

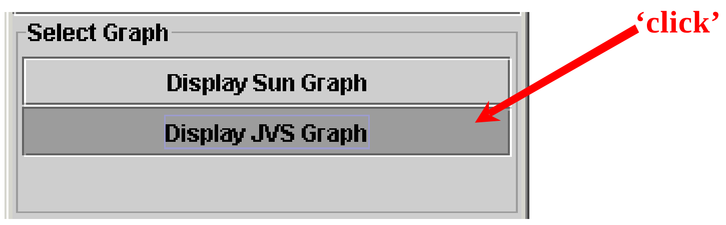
Once the application has loaded you will be presented with the controller, your first step should be to load a new model into memory;



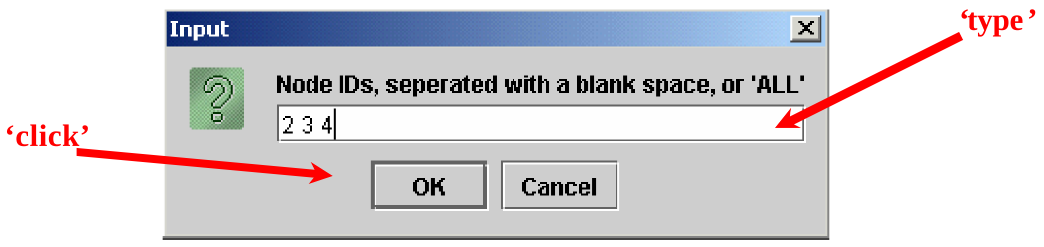
Once a model has been loaded you have the option to load a new view, for example;



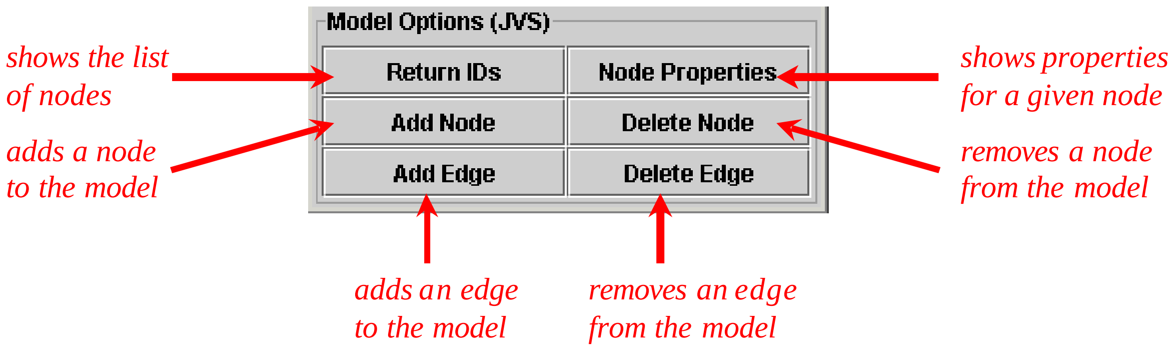
Once you have made a choice about which visualisation technique to use you should load up a new graph view of the data (click 'Display JVS Graph');



If you wish to visualise some nodes using either fog or emissive light click 'visualise', you will then be presented with the following user-input dialog;



If you wish to make changes to the data model during execution of the program you can do so using the model sub-panel located at the bottom of the controller;



6.5 Help Sheet – 5; The Views

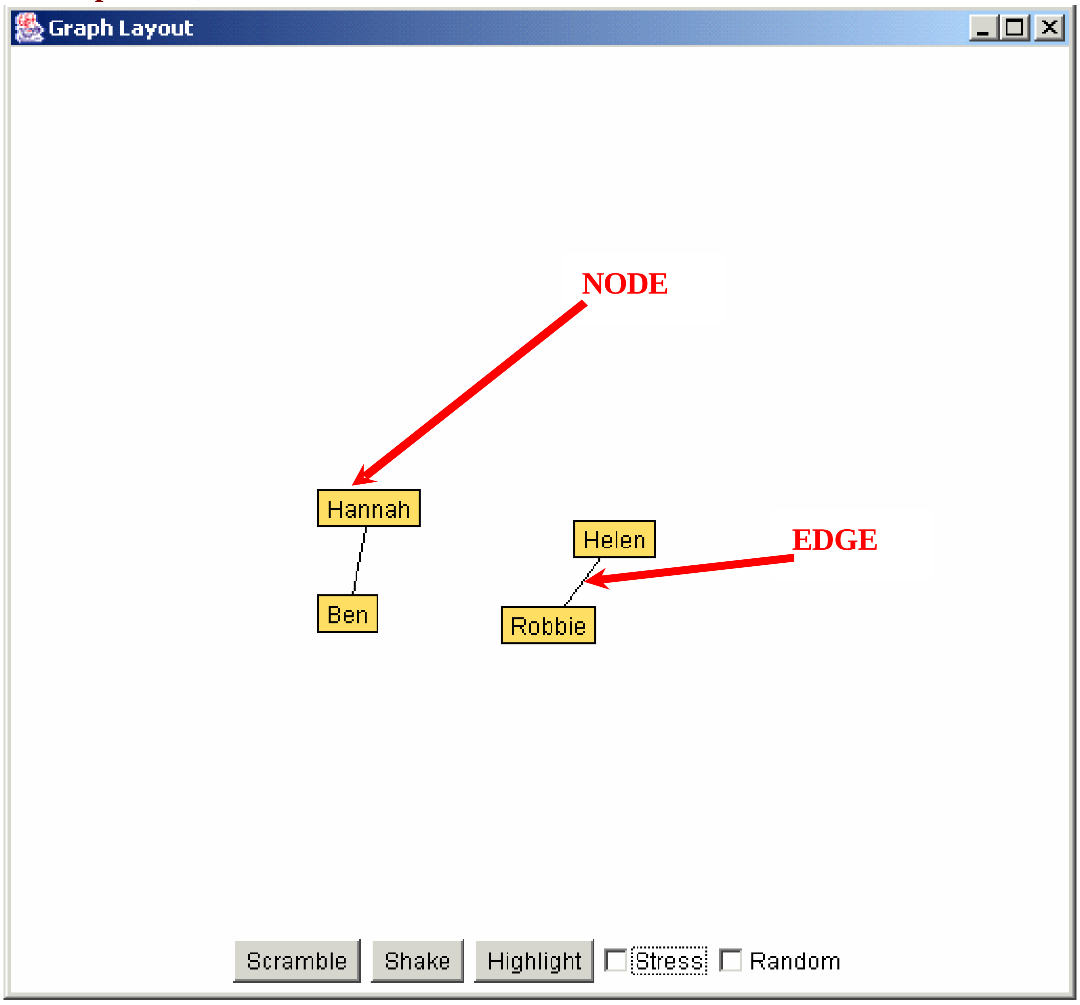
Java Visualisation System
Version 1.1

The Views

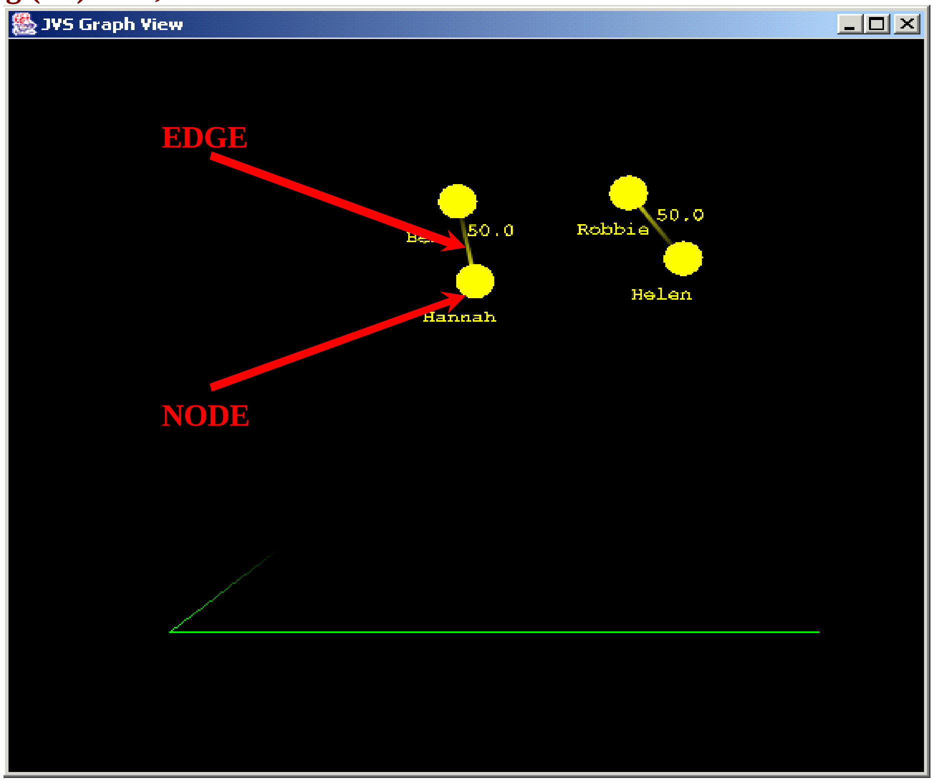
Help Sheet - 5

This document is designed to give you a brief overview of what you are seeing when you are presented with one of the three views available in the Java Visualisation System (JVS). The views are designed to be minimalist so as to improve the effects of any visualisation technique, so there is not much to say!

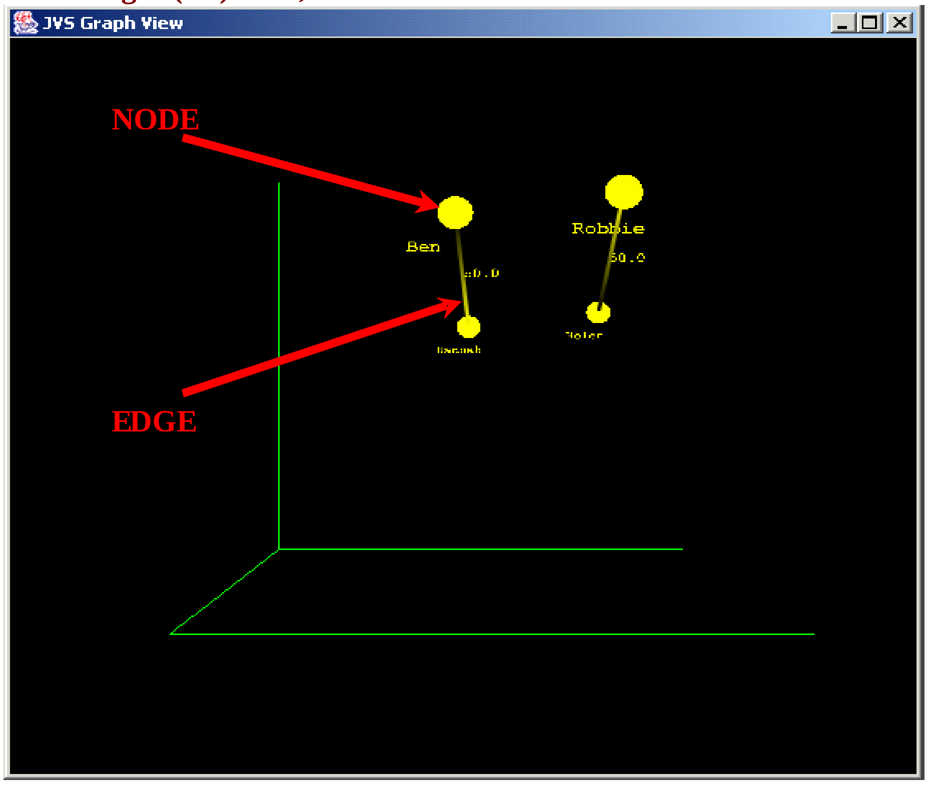
Sun Graph view;



JVS Fog (2D) view;



JVS Emissive Light (3D) view;

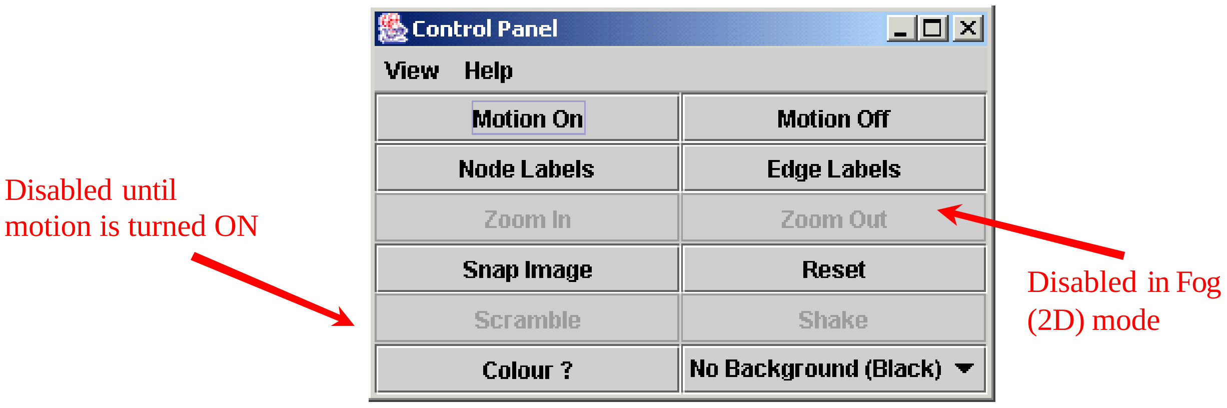


6.6 Help Sheet – 6; Control-Panel

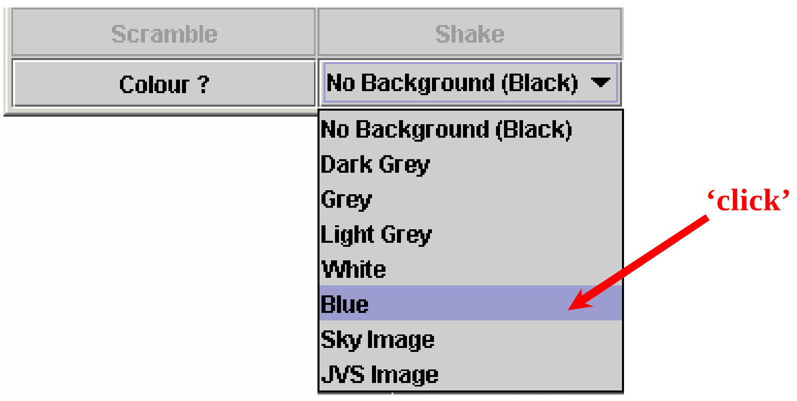
Java Visualisation System
Version 1.1
Control-Panel
Help Sheet - 6

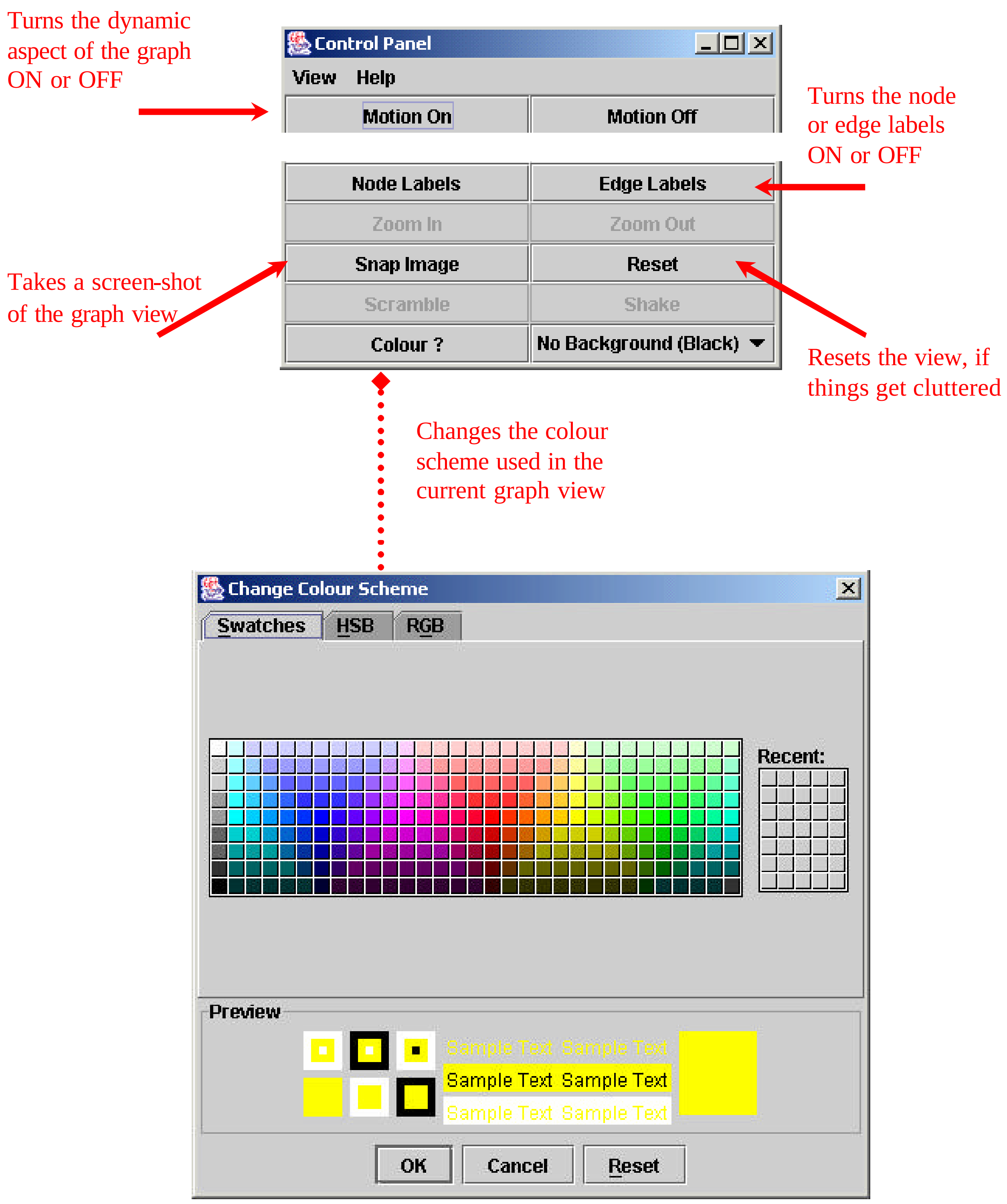
This document is designed to assist you with operation of the optional control-panel that is used to manipulate the graph view in the Java Visualisation System (JVS). It will walk through how to use the panel and explain how it interacts with the various other aspects of the program.

If you are running the stand-alone version of JVS or, if you are in 'bolt-on' mode and have chosen to view the control panel, then this is what you will be presented with;



At present changing the background is not particularly helpful but the functionality is included as it is an area of future development for the program;




Turns the dynamic aspect of the graph ON or OFF
Control Panel
View
Help
Motion On
Motion Off
Turns the node or edge labels ON or OFF
Node Labels
Edge Labels
Zoom In
Zoom Out
Takes a screen-shot of the graph view
Snap Image
Reset
Scramble
Shake
Colour ?
No Background (Black)
Resets the view, if things get cluttered
Changes the colour scheme used in the current graph view
Change Colour Scheme
Swatches
HSB
RGB
Recent:
Preview
Sample Text Sample Text
Sample Text Sample Text
Sample Text Sample Text
OK
Cancel
Reset

Chapter Seven

Results

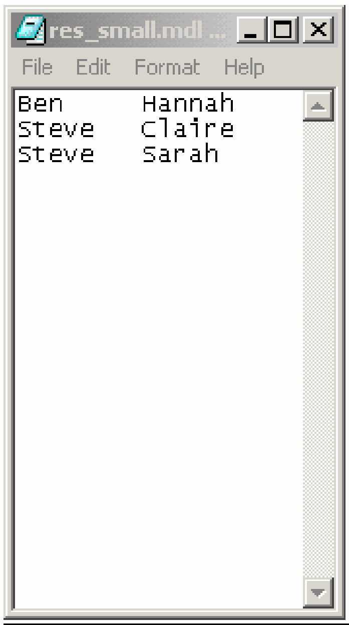
7 Results

This section will attempt to evaluate the completed application based on the specification as it is described in Chapter One. Due to the graphical nature of this project there will be numerous screen-shots in this section. This will hopefully demonstrate the success that was achieved in implementing the visualisation improvements, particularly Semantic Depth of Field. There will be a demonstration that interactively visualises many graphs of different size and complexity to support the claims made in Chapter Eight.

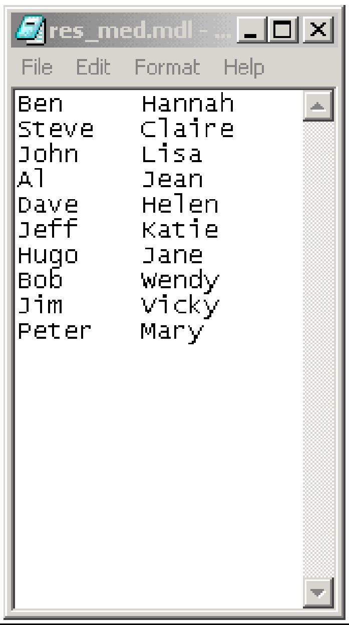
The tests were carried out using people to represent the nodes in the graph and the edges represent the relationships between those people. As the whole principle of Semantic Depth of Field (SDoF) is based on relevancy of nodes the gender of the individuals was used to demonstrate the effect. Gender is also the basis for visualisation in the Emissive Lighting section. For the purposes of testing; the relevant nodes are considered to be the females.

The relationships will, in general, be straightforward. At the end of the SDoF section there will be coverage of a more complex graph (Fig 7.1.7 & 7.1.8). There will be three graph sizes;

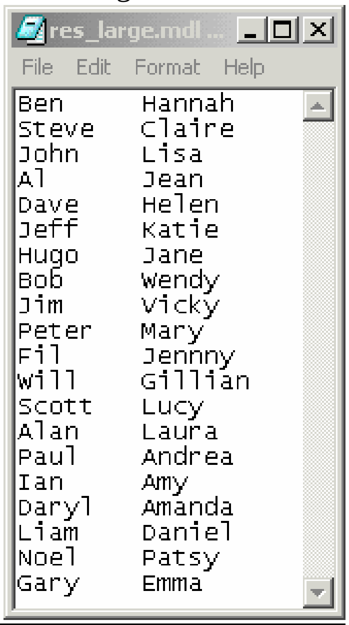
- Small - 5 nodes
- Medium - 20 nodes
- Large - 40 nodes



Ben	Hannah
Steve	Claire
Steve	Sarah



Ben	Hannah
Steve	Claire
John	Lisa
Al	Jean
Dave	Helen
Jeff	Katie
Hugo	Jane
Bob	Wendy
Jim	Vicky
Peter	Mary



Ben	Hannah
Steve	Claire
John	Lisa
Al	Jean
Dave	Helen
Jeff	Katie
Hugo	Jane
Bob	Wendy
Jim	Vicky
Peter	Mary
Fil	Jennny
Will	Gillian
Scott	Lucy
Alan	Laura
Paul	Andrea
Ian	Amy
Daryl	Amanda
Liam	Daniel
Noel	Patsy
Gary	Emma

7.1 Semantic Depth of Field

The following few pages will show the results of applying the implementation of Semantic Depth of Field to the test data. Each page will show a 'before' and 'after' shot, designed to simulate the visualisation as experienced by the user. The pictures show clearly that there is no distinguishable difference between the nodes before visualisation and the relevant nodes after visualisation; this is a crucial requirement of SDoF.

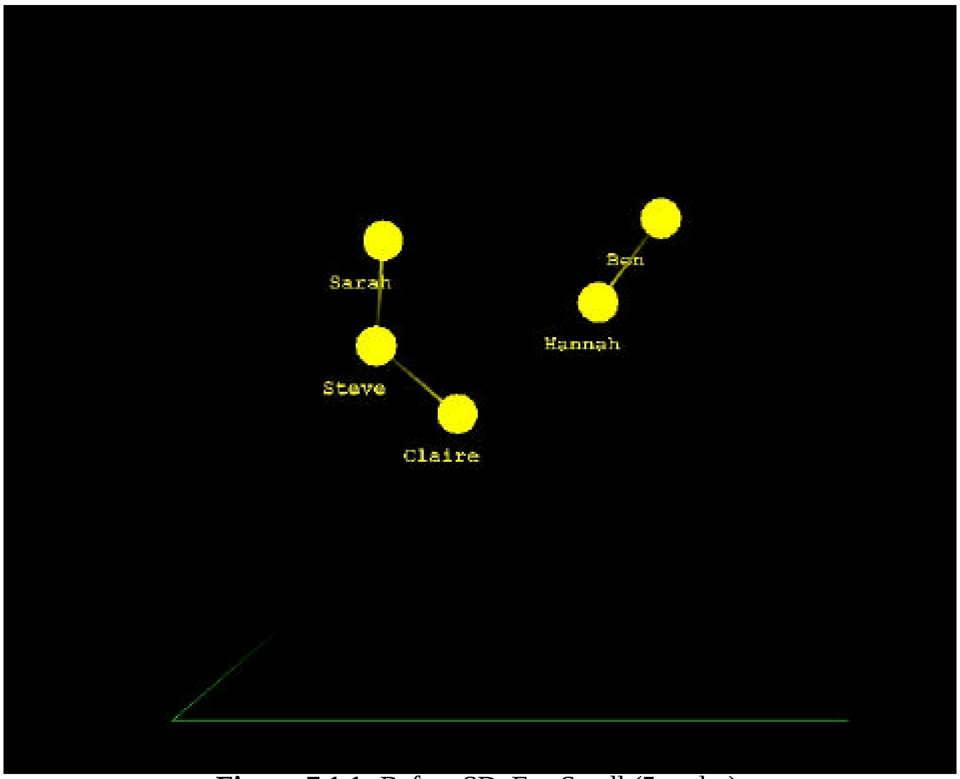


Figure 7.1.1: Before SDoF – Small (5 nodes).

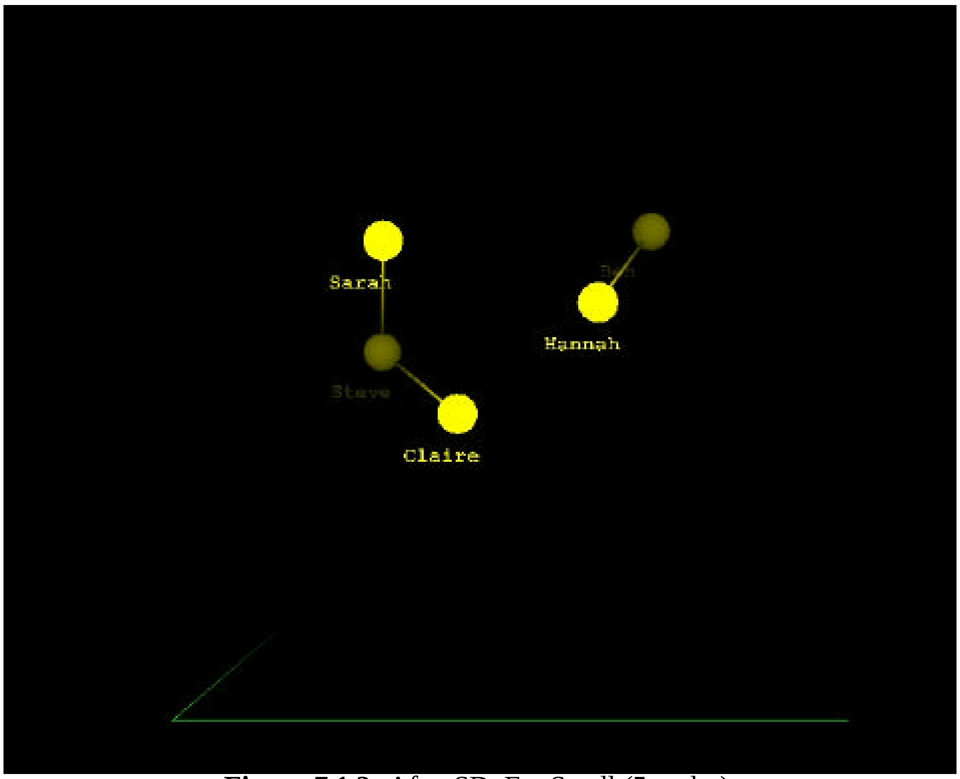


Figure 7.1.2: After SDoF – Small (5 nodes).

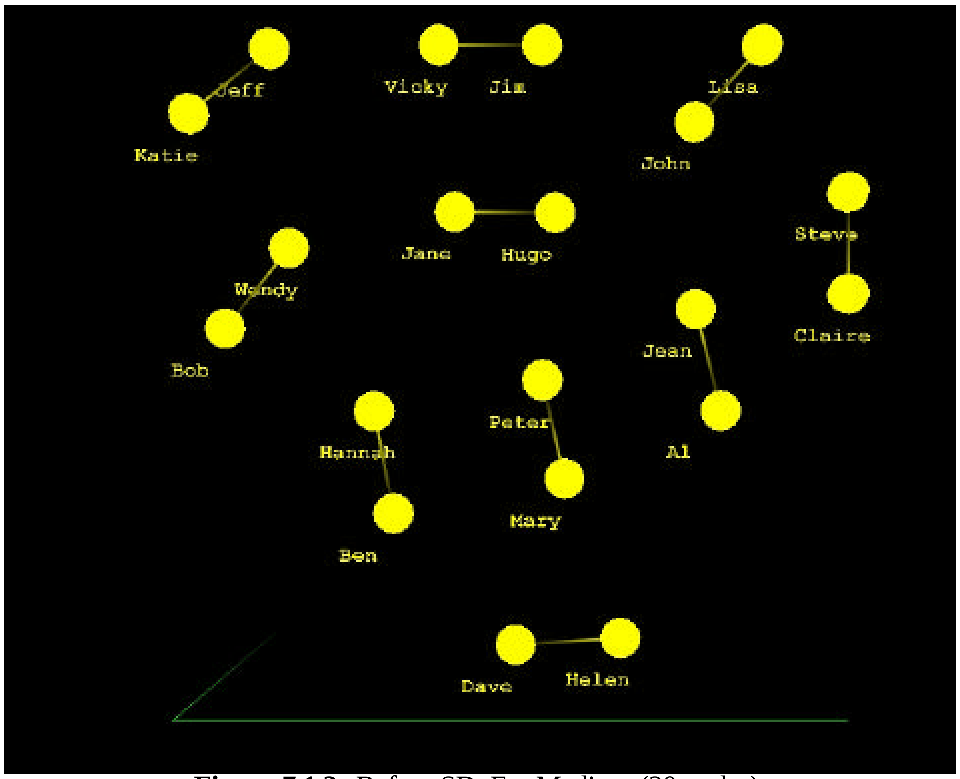


Figure 7.1.3: Before SDoF – Medium (20 nodes).

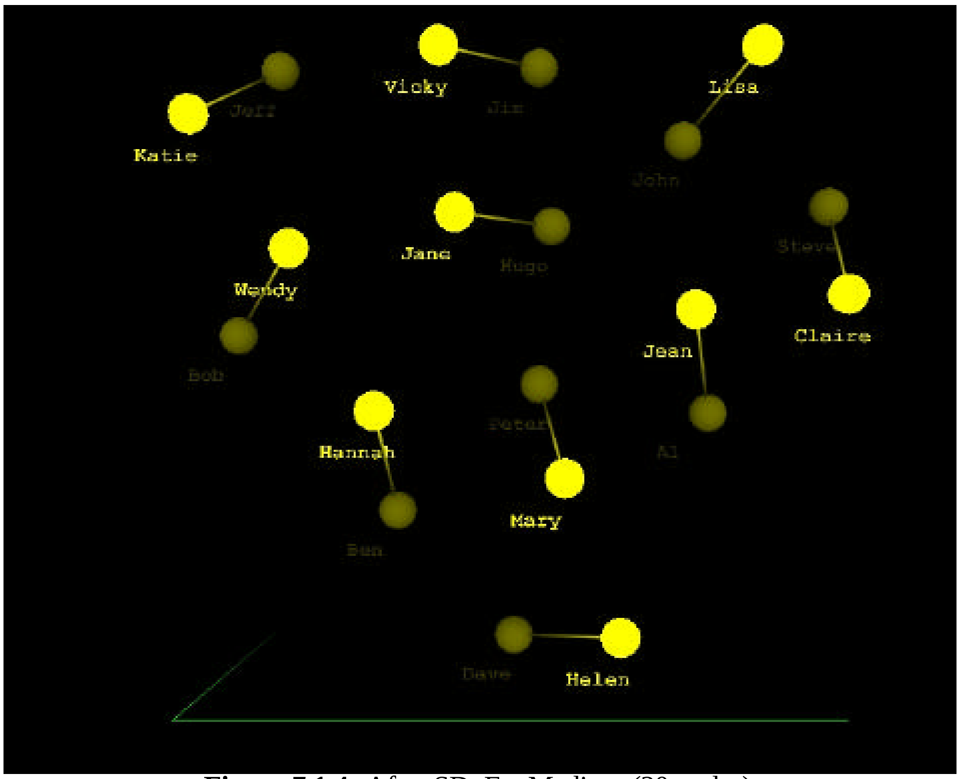


Figure 7.1.4: After SDoF – Medium (20 nodes).

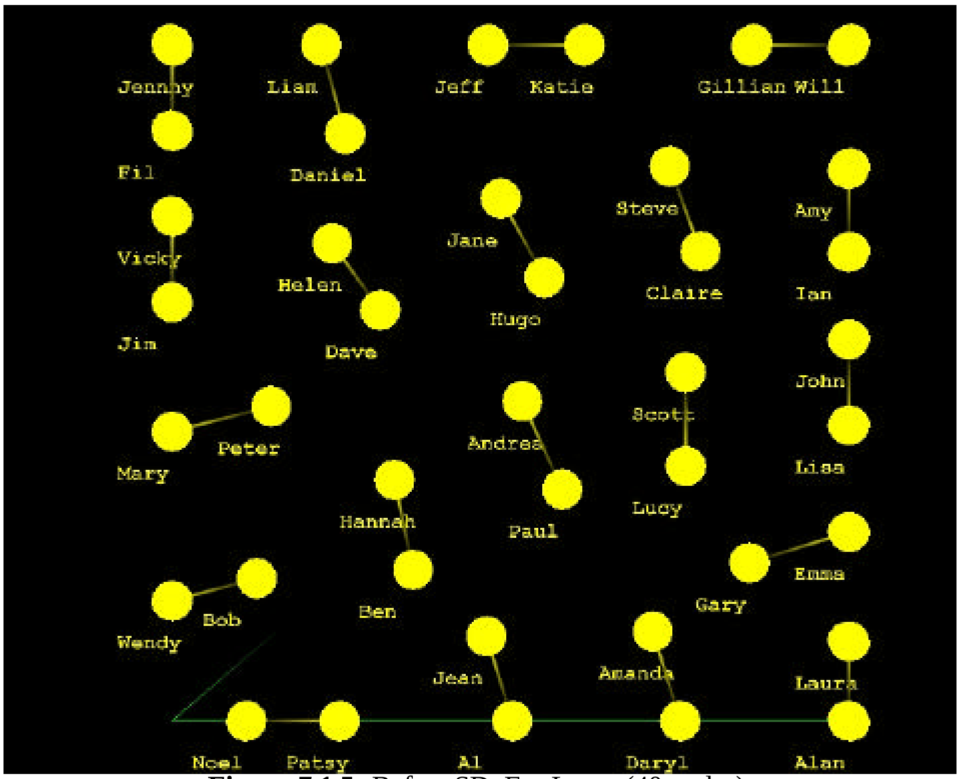


Figure 7.1.5: Before SDoF – Large (40 nodes).

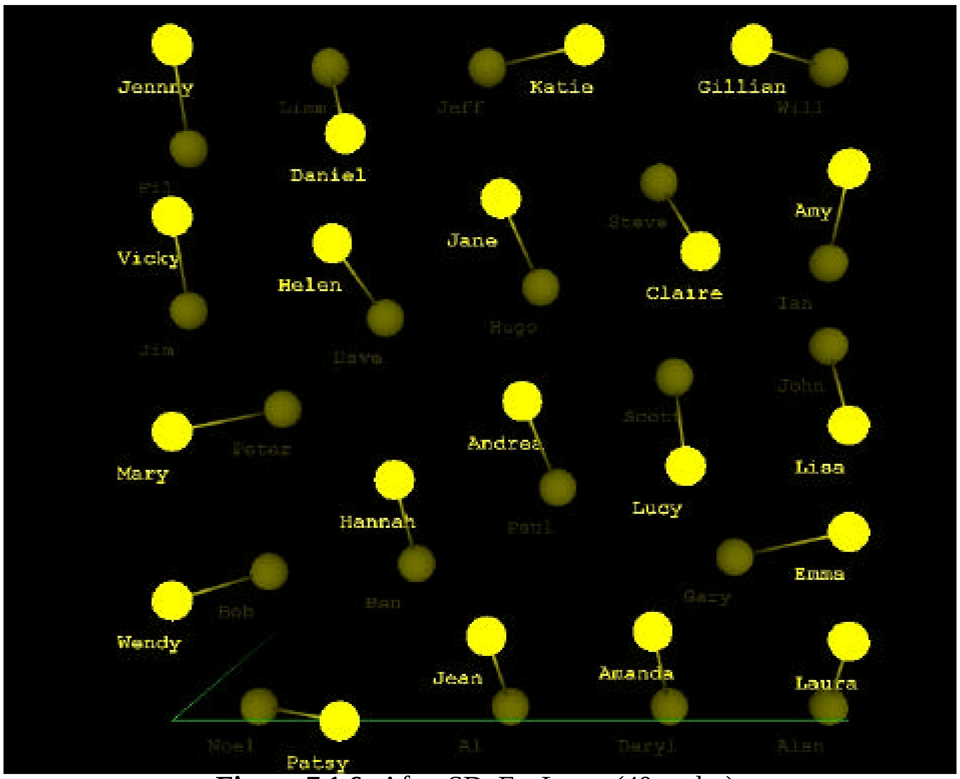


Figure 7.1.6: After SDoF – Large (40 nodes).

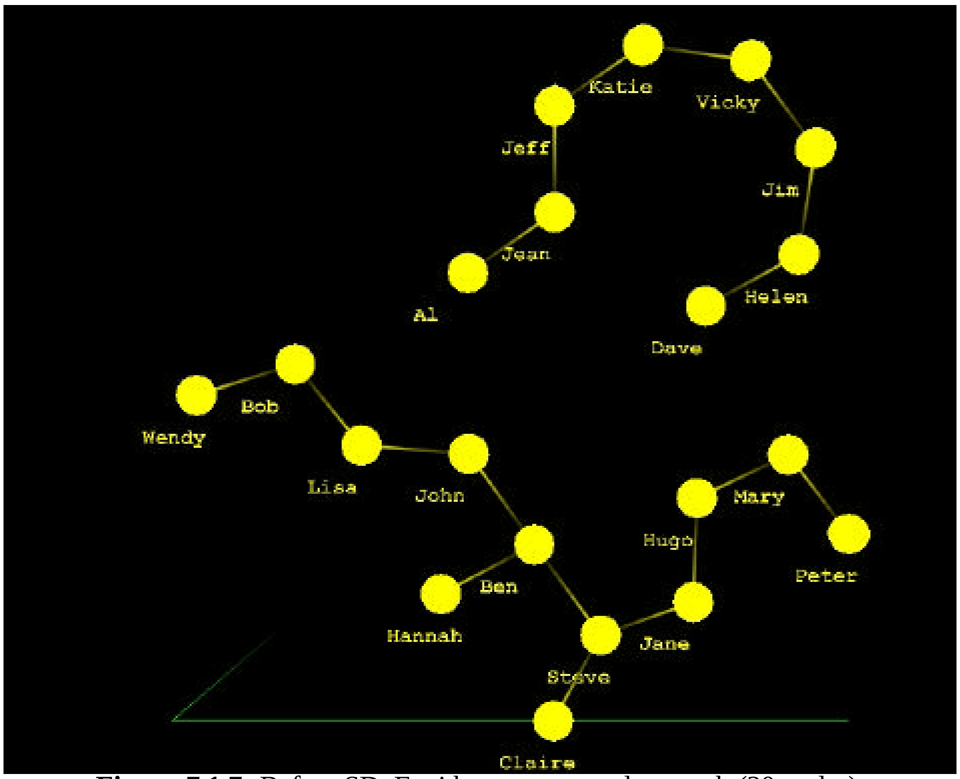


Figure 7.1.7: Before SDoF with a more complex graph (20 nodes).

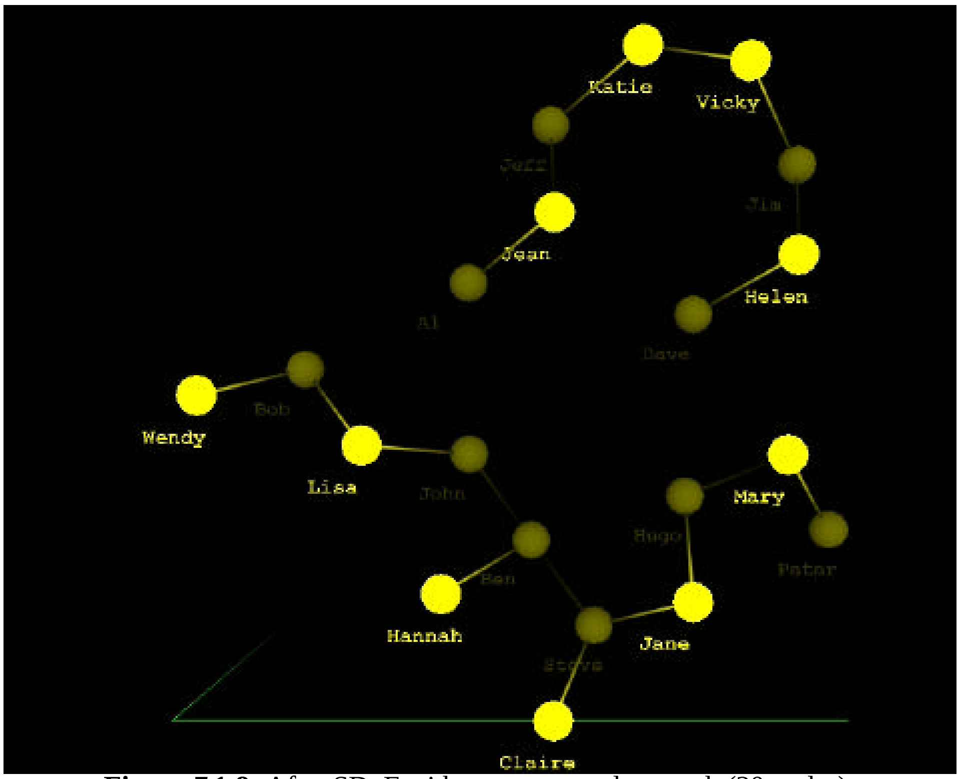


Figure 7.1.8: After SDoF with a more complex graph (20 nodes).

7.2 Emissive Lighting

These are more examples of the results obtained using the same test data. The implementation of the emissive lighting allowed for the graph to be in three dimensions.

Although this chapter was designed to demonstrate the full effect of the visualisation techniques, the screen-shots are of high quality and this may not be reproduced in print. It is recommended that, if possible, when reading this chapter the reader consults the actual document, available on the CD that accompanies this dissertation.

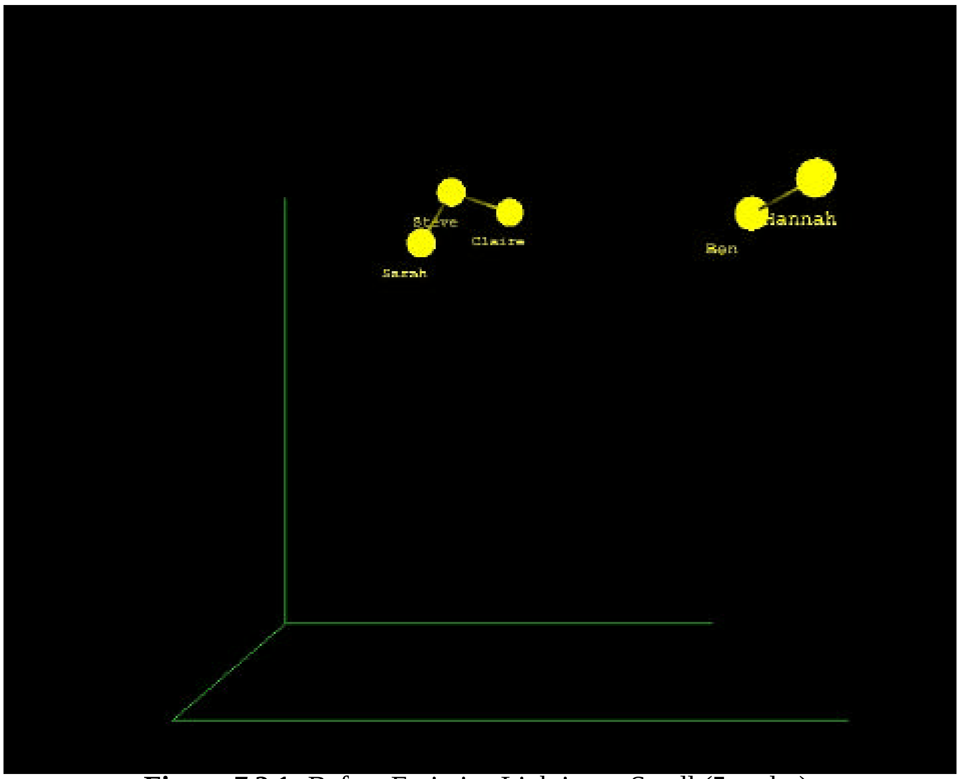


Figure 7.2.1: Before Emissive Lighting – Small (5 nodes).

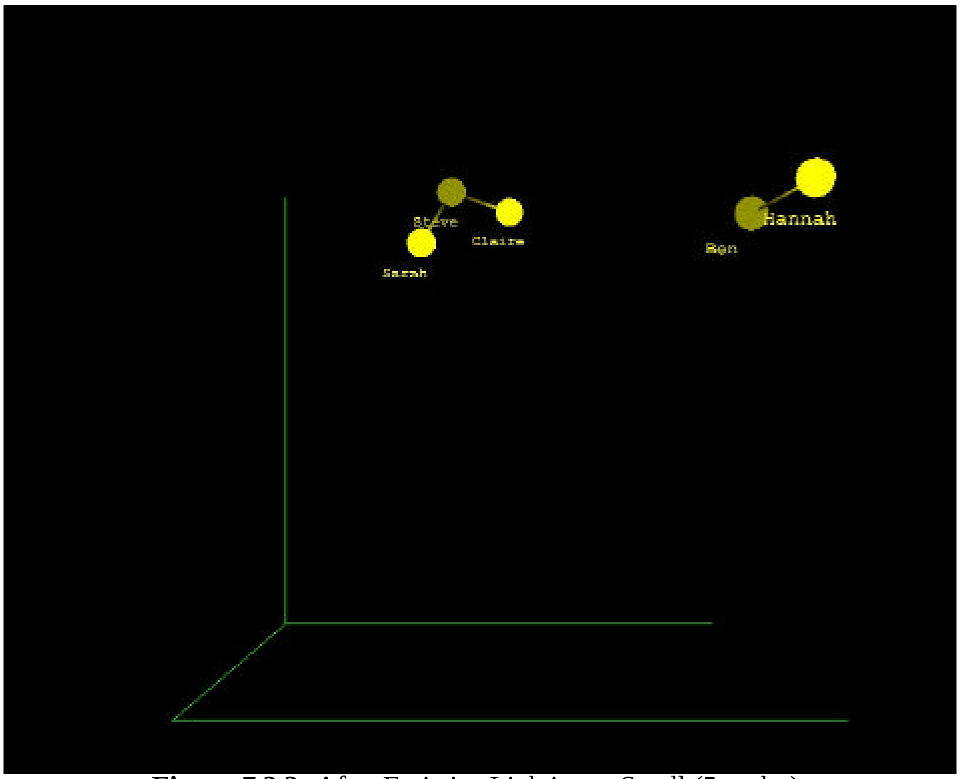


Figure 7.2.2: After Emissive Lighting – Small (5 nodes).

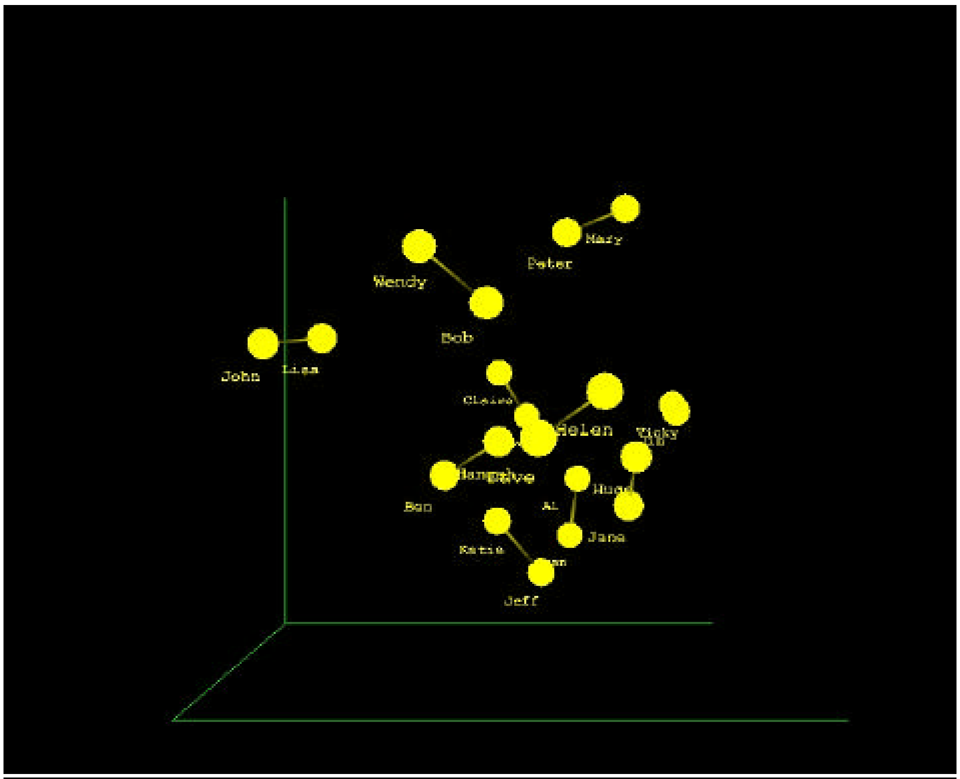


Figure 7.2.3: Before Emissive Lighting – Medium (20 nodes).

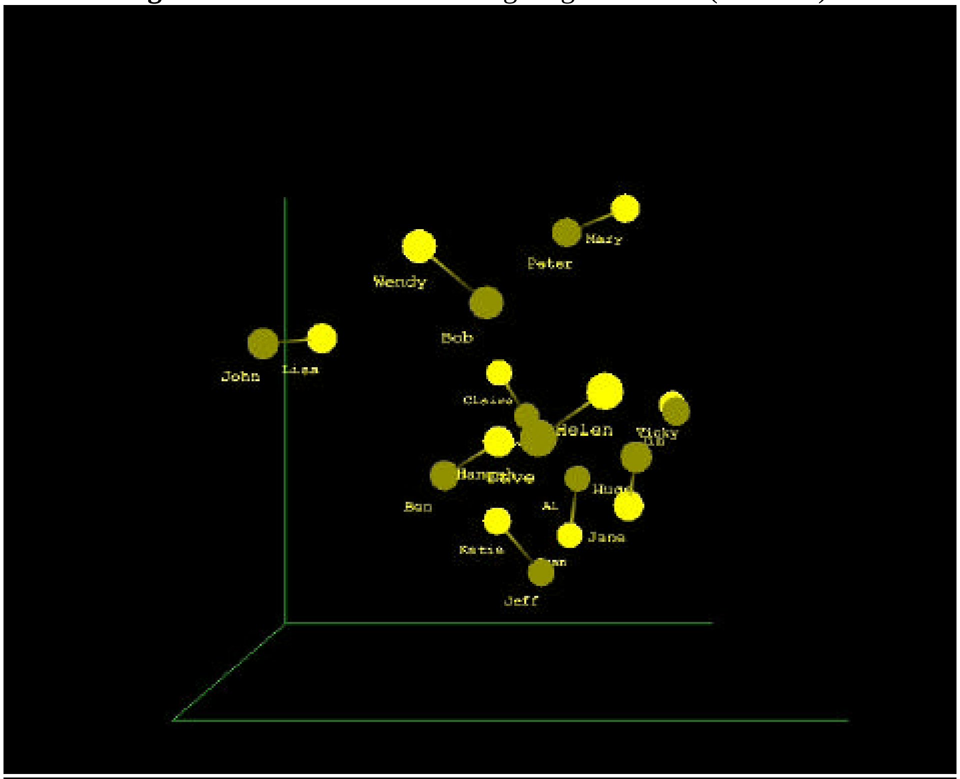


Figure 7.2.4: After Emissive Lighting – Medium (20 nodes).

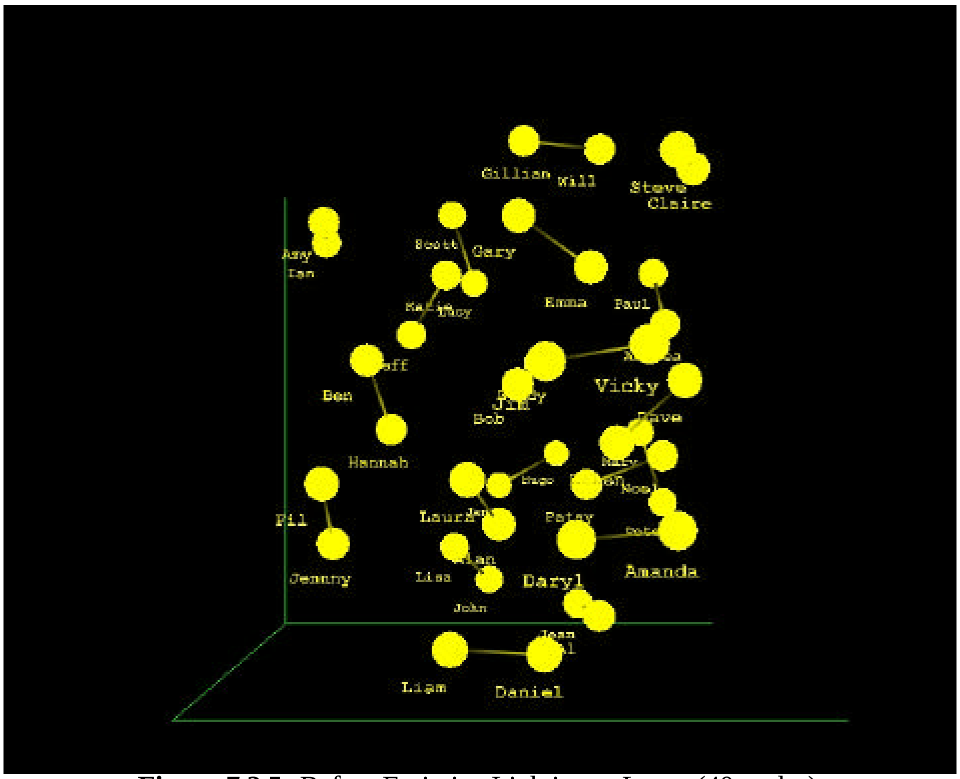


Figure 7.2.5: Before Emissive Lighting – Large (40 nodes).

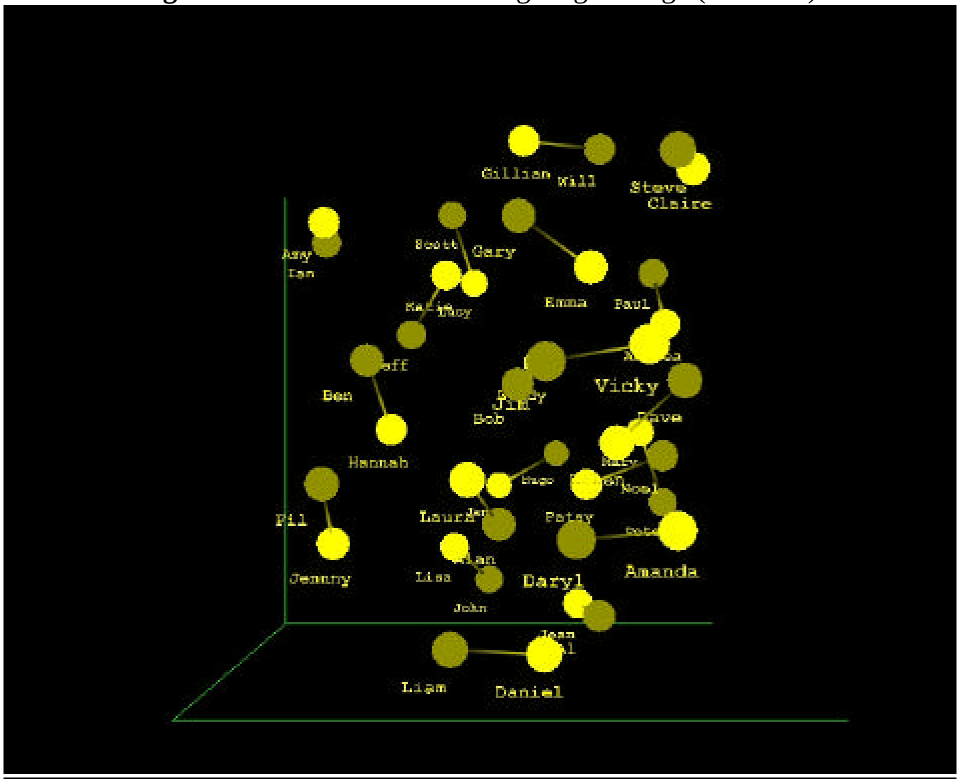


Figure 7.2.6: After Emissive Lighting – Large (40 nodes).

The 3D system allows viewing from different distances and angles;

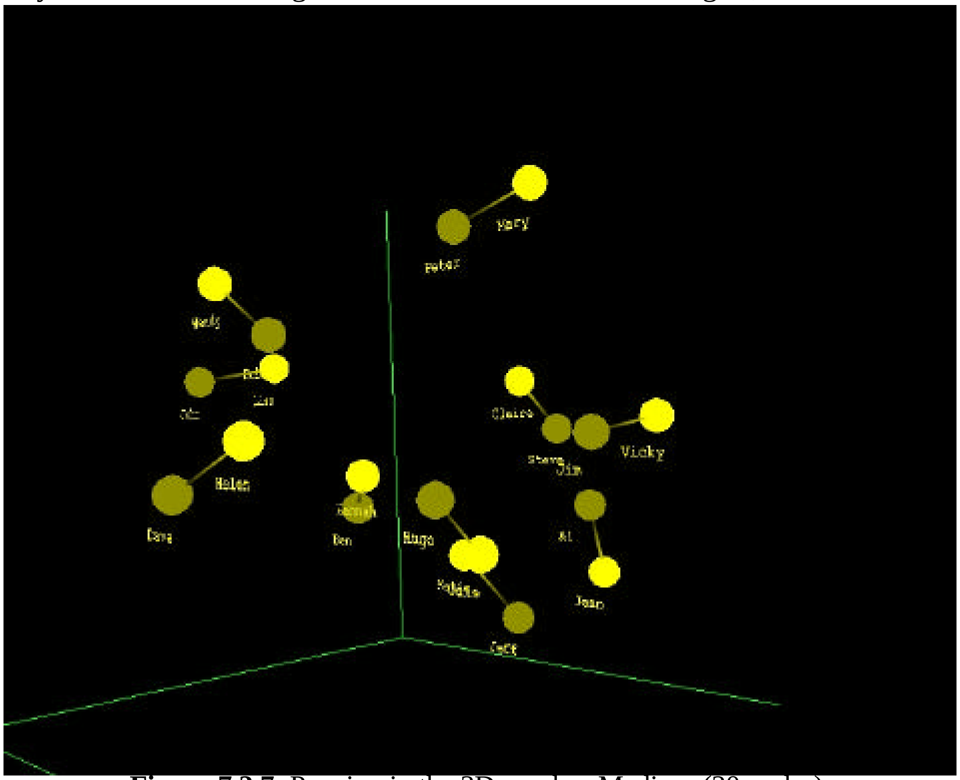


Figure 7.2.7: Panning in the 3D graph – Medium (20 nodes).

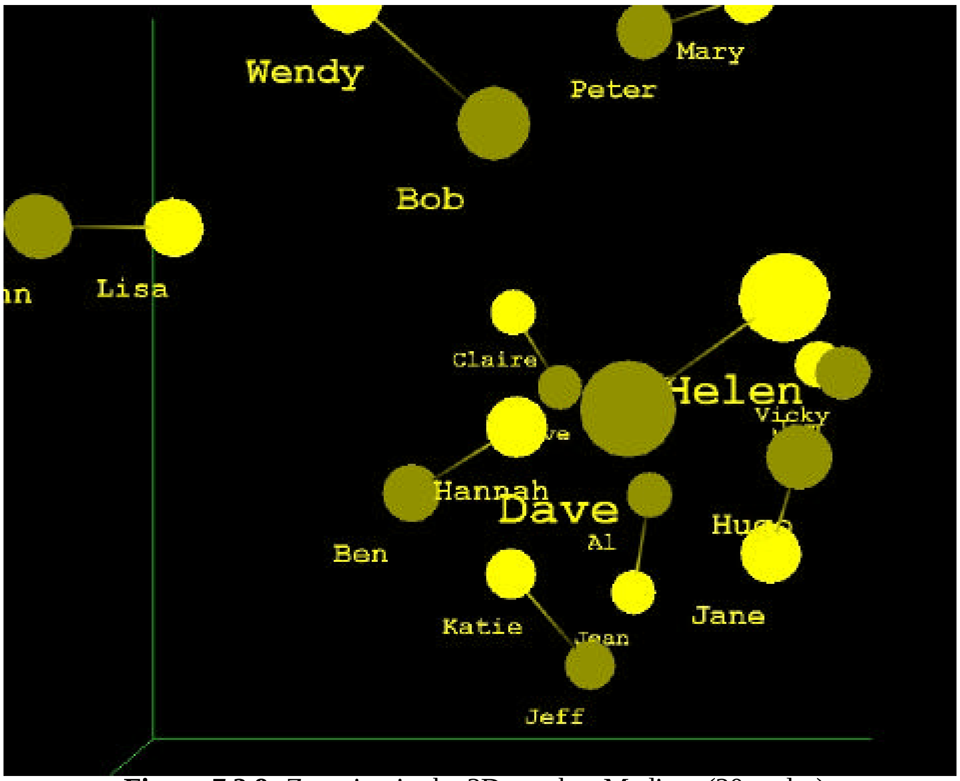


Figure 7.2.8: Zooming in the 3D graph – Medium (20 nodes).

7.3 Other Results

There are numerous other aspects of the application that do not fit neatly under the previous headings. The screen-shots on the following few pages demonstrate;

1. **The effect of grouping nodes together.**
   Grouping works in 2D (Fog) or 3D (Emissive Light) and simply groups nodes together according to an 'area code' that is attached to the name of the node. There are four such area-codes (1, 2, 3, and 4) that refer to the top-left, top-right, bottom-left and bottom-right quadrants of the screen respectively. The grouping is ignored once motion is turned allow for the graph layout algorithm to arrange the nodes in a more presentable fashion.

2. **The effect of the graph layout algorithm.**

3. **The importance of the number of relevant nodes.**
   This is a crucial issue with Semantic Depth of Field that appears to have been avoided in the publications so far. The reason, perhaps, is that it is a general visualisation issue. The user must give careful consideration to exactly what they wish to visualise. The effects of techniques such as Semantic Depth of Field are far more impressive when visualising less than half the total number of nodes.

4. **The use of different colour schemes.**
   This aspect of the program demonstrates that it is possible to use a wide variety of colours to illustrate the benefits of Semantic Depth of Field. This is an obvious advantage over colour-based visualisation systems as it allows for black and white rendering as well as allowing for the needs of colour-blind users. Experimentation revealed that the results are most pleasing when using bright colours on a dark background (hence the yellow on black scheme used in this Chapter).

5. **The perceptual benefit of hiding labels.**
   Another conclusion of this project is that, when it comes to visualisation, the less on-screen distraction the better. These shots demonstrate the improvement in visualisation achieved by hiding the name labels. Up until now the labels have been visible to demonstrate that the correct nodes are being visualised.

6. **A more accurate implementation of blur.**
   This screen-shot shows the result of a more accurate implementation of blur that uses multiple spheres of differing transparencies to draw blurred nodes. The problem with this approach and the reason fog was used instead is because of the overhead associated with using multiple objects. It is accepted in the world of Computer Graphics that a true blur effect is a challenging implementation.

7. **Edge thickness according to edge weight.**
   The use of an algorithm to determine the width of an edge based on the weight of that edge, this can prove useful when visualisation requires that edge labels be turned off.

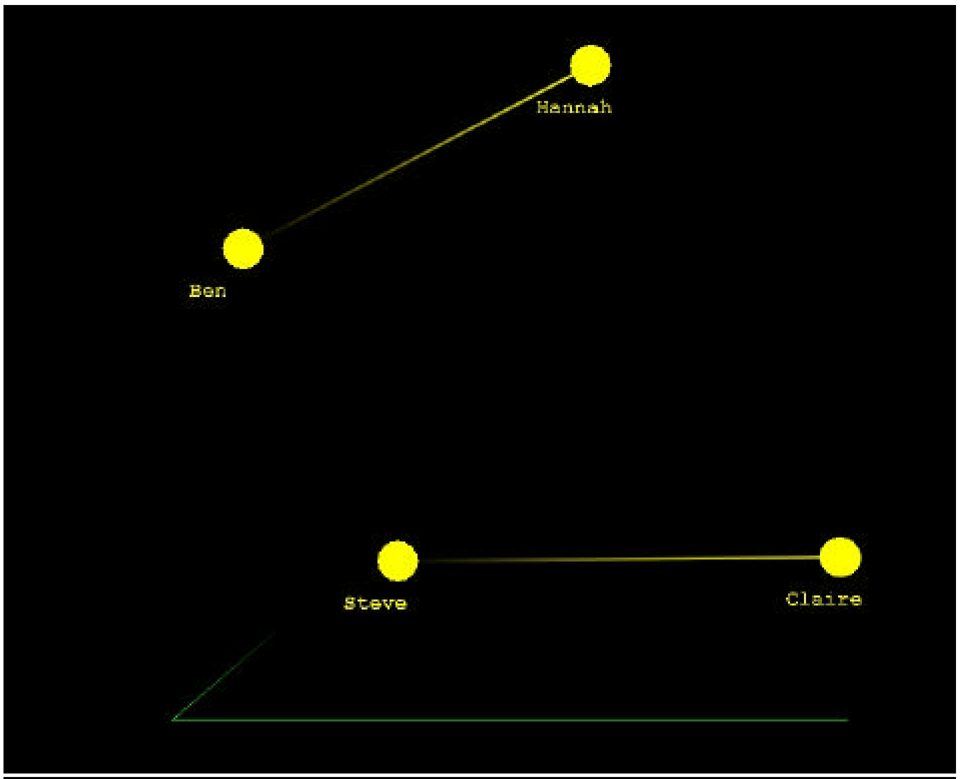


Figure 7.3.1: The effect of allocating each node a group (2D).

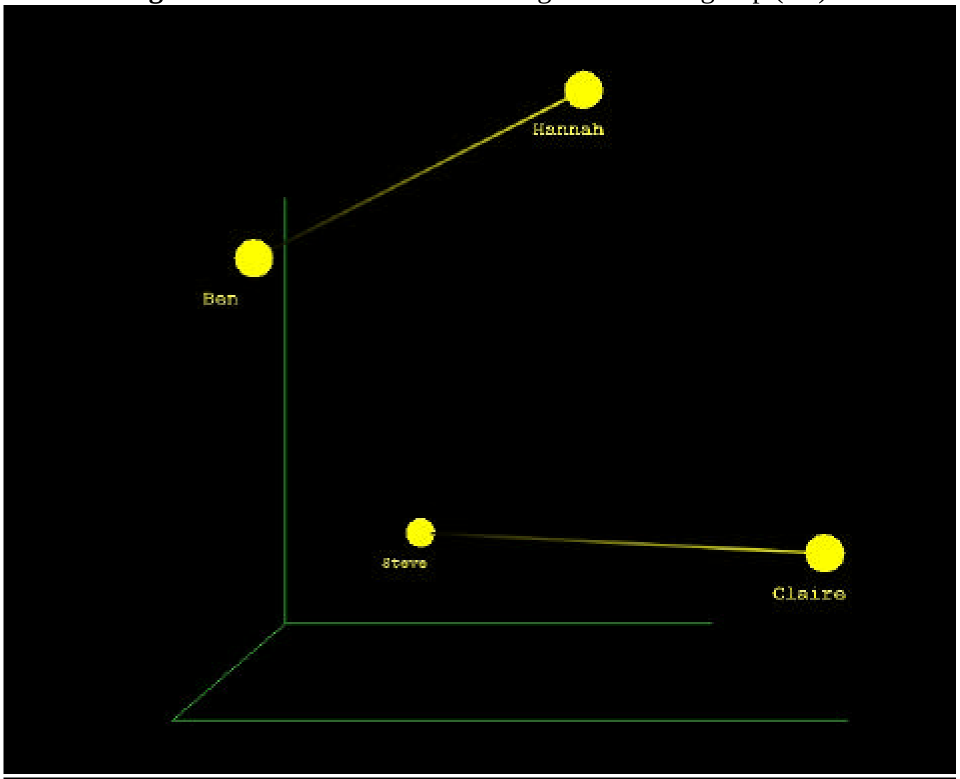


Figure 7.3.2: The effect of allocating each node a group (3D).

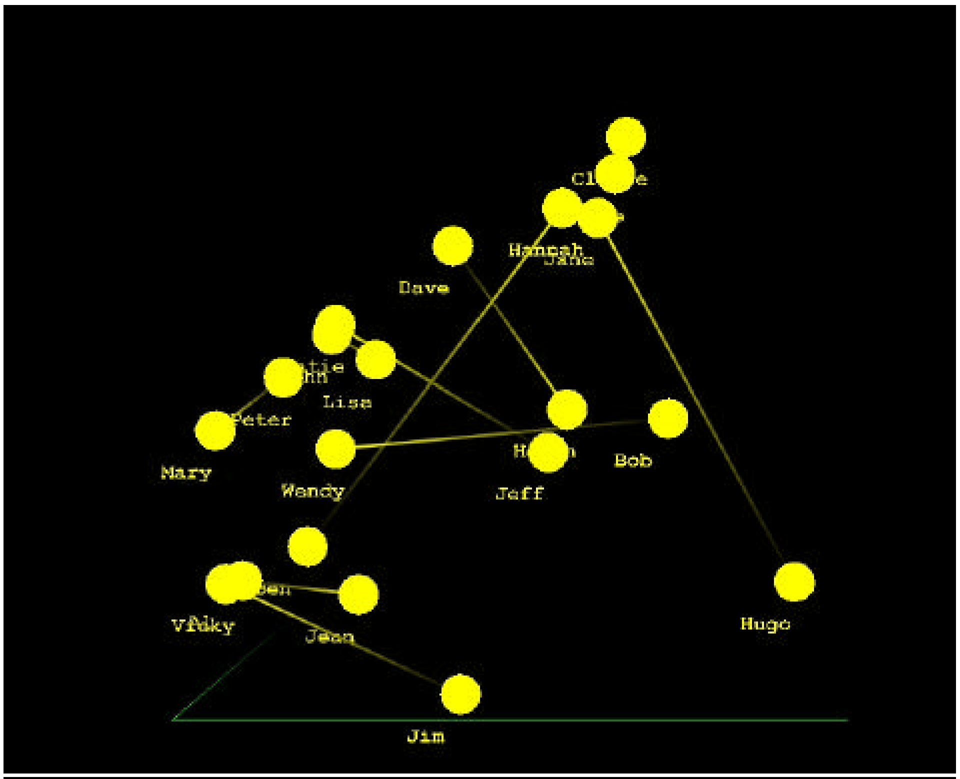


Figure 7.3.3: Before the Graph Layout takes control (2D) – Medium (20 nodes).

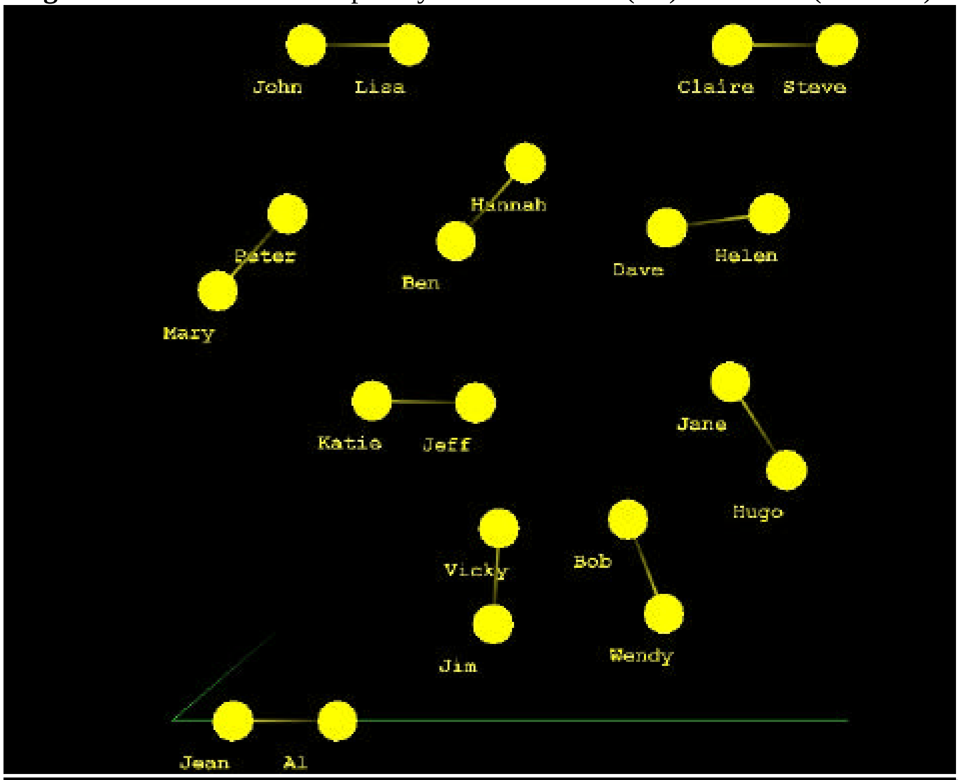


Figure 7.3.4: After the Graph Layout takes control (2D) – Medium (20 nodes).

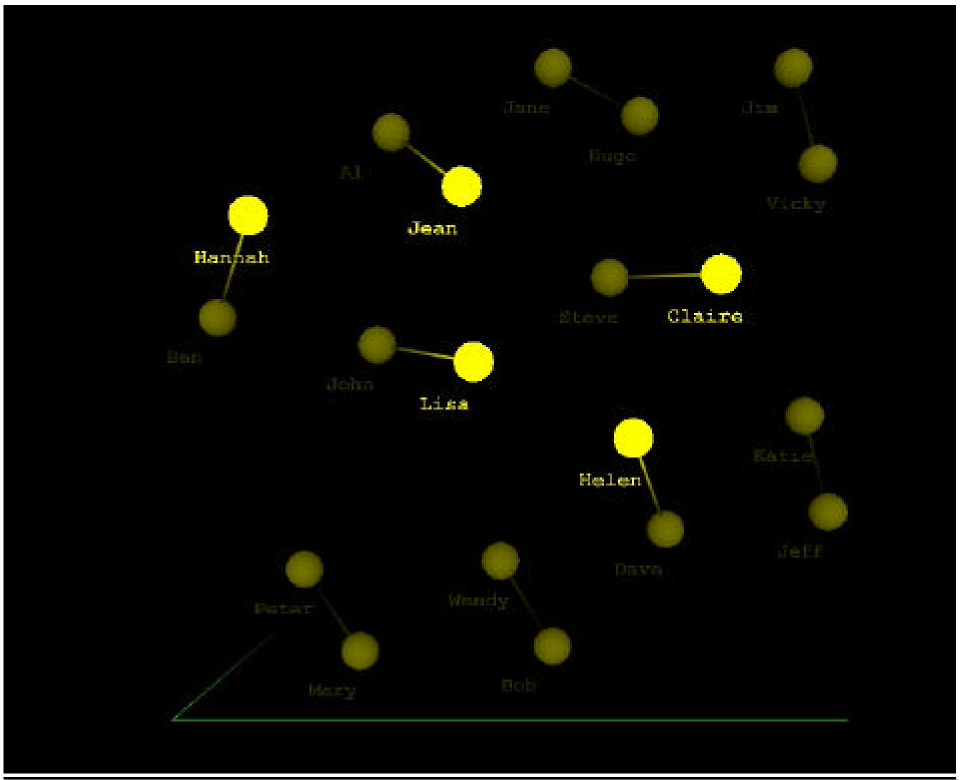


Figure 7.3.5: Visualising ¼ of all nodes – Medium (20 nodes).

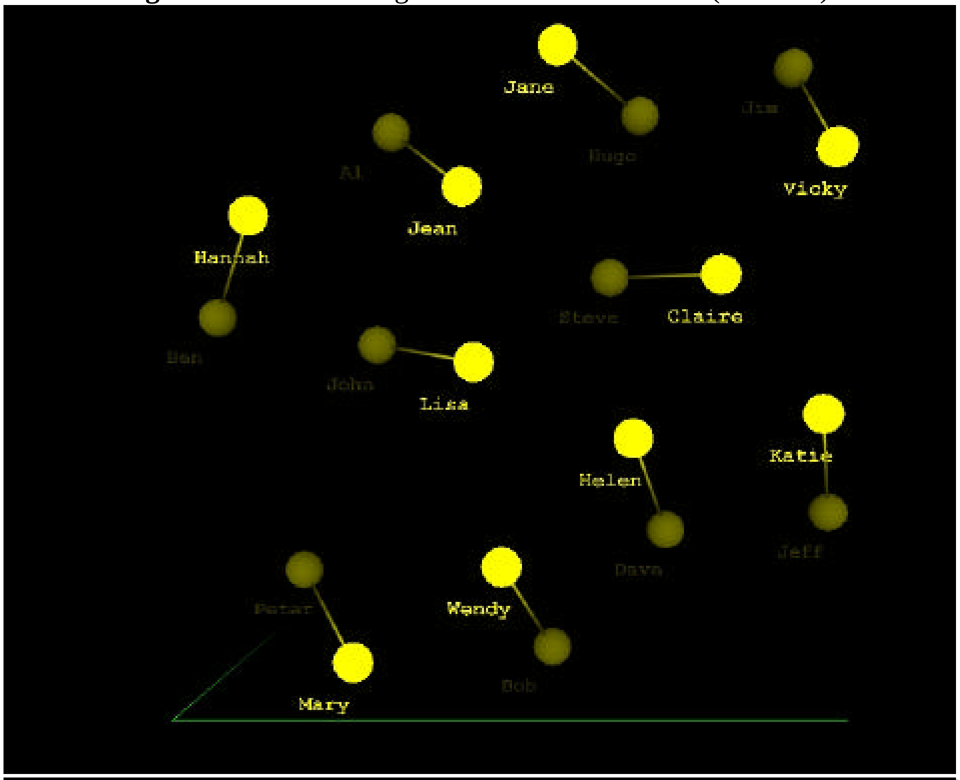


Figure 7.3.6: Visualising ½ of all nodes – Medium (20 nodes).

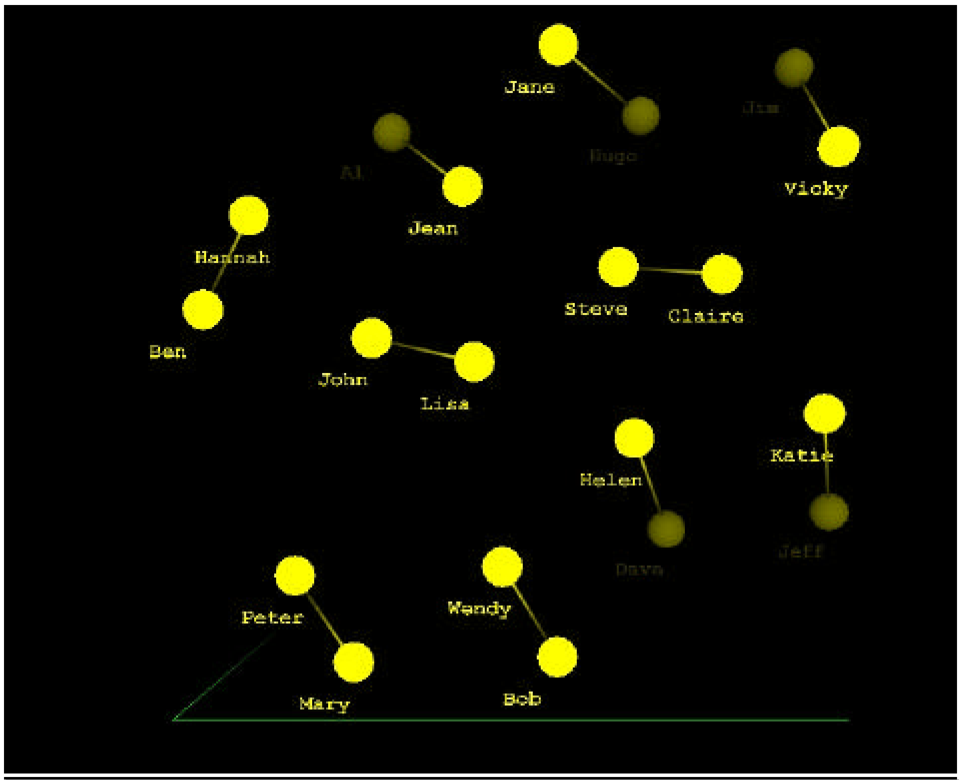


Figure 7.3.7: Visualising ¾ of all nodes – Medium (20 nodes).

This dissertation suggests a simple solution to this problem; employ reverse visualisation. In the example shown in figure 7.3.7 it would, perhaps, be more sensible to be visualising those exact nodes that are being used as context at the moment. That way the demonstration would clearly show that the nodes were in the minority, more so than the current view does. For example, if the highlighted nodes represent those people who like the taste of chocolate, and the purpose of the demonstration were to show that many people do indeed like chocolate then perhaps it would make more sense to visualise those people who do not like chocolate.

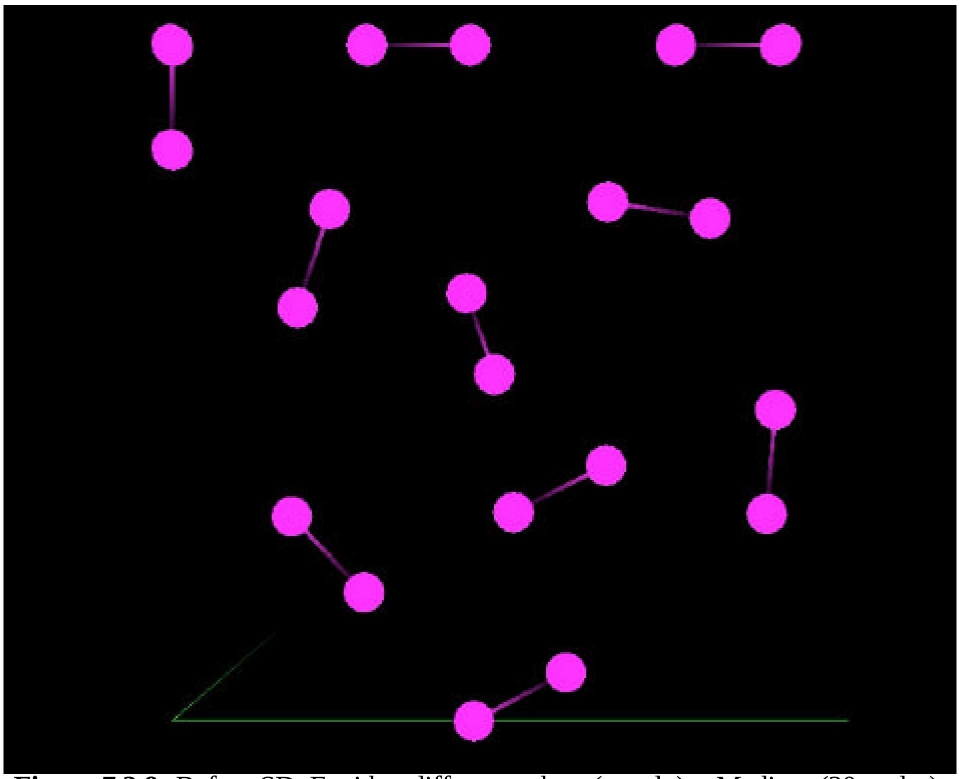


Figure 7.3.8: Before SDoF with a different colour (purple) – Medium (20 nodes).

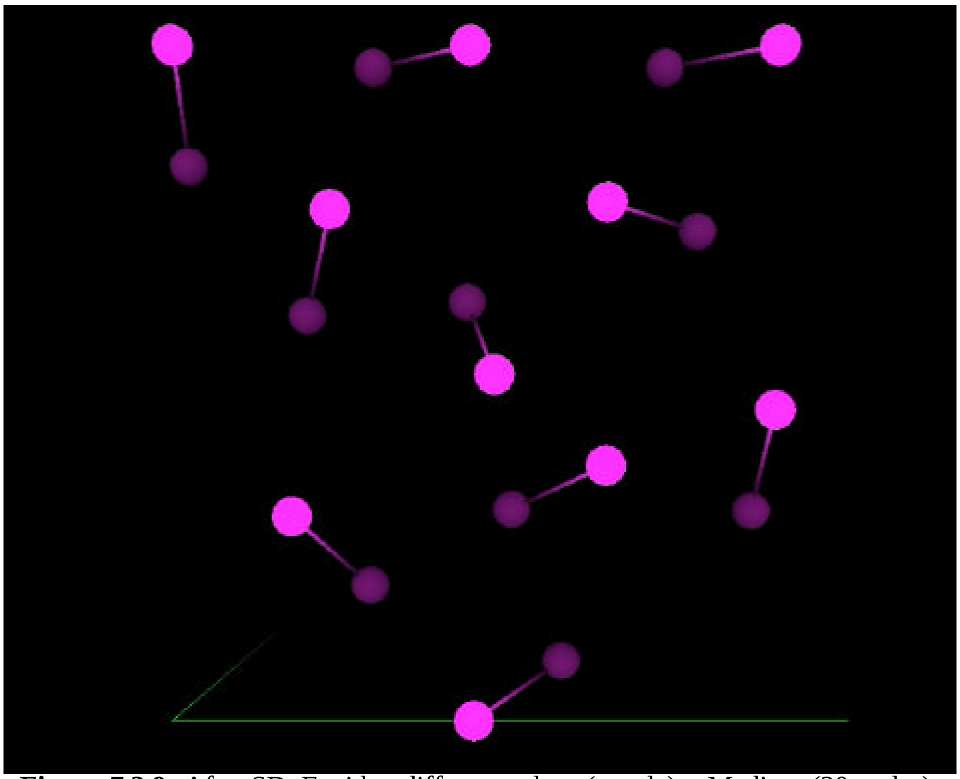


Figure 7.3.9: After SDoF with a different colour (purple) – Medium (20 nodes).

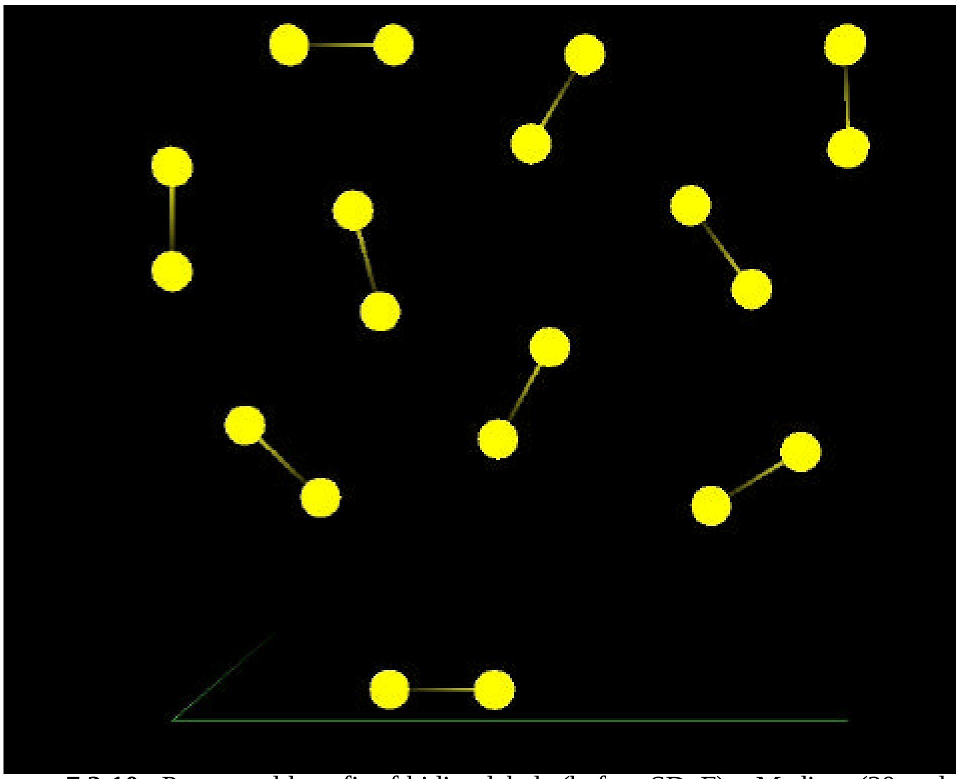


Figure 7.3.10: Perceptual benefit of hiding labels (before SDoF) – Medium (20 nodes).

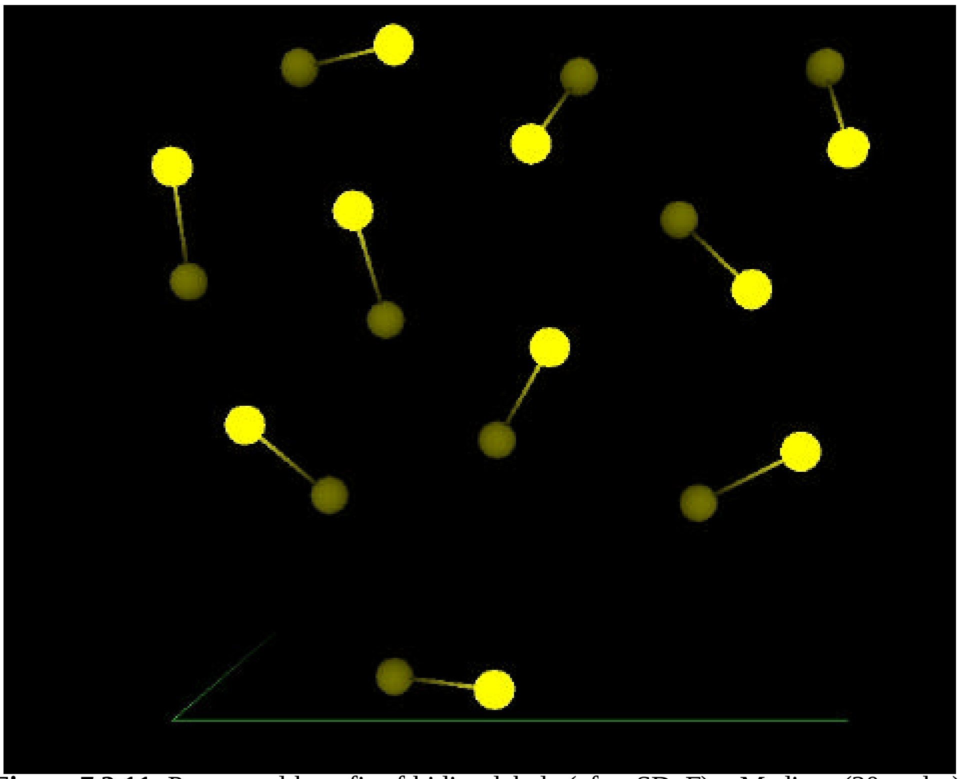


Figure 7.3.11: Perceptual benefit of hiding labels (after SDoF) – Medium (20 nodes).

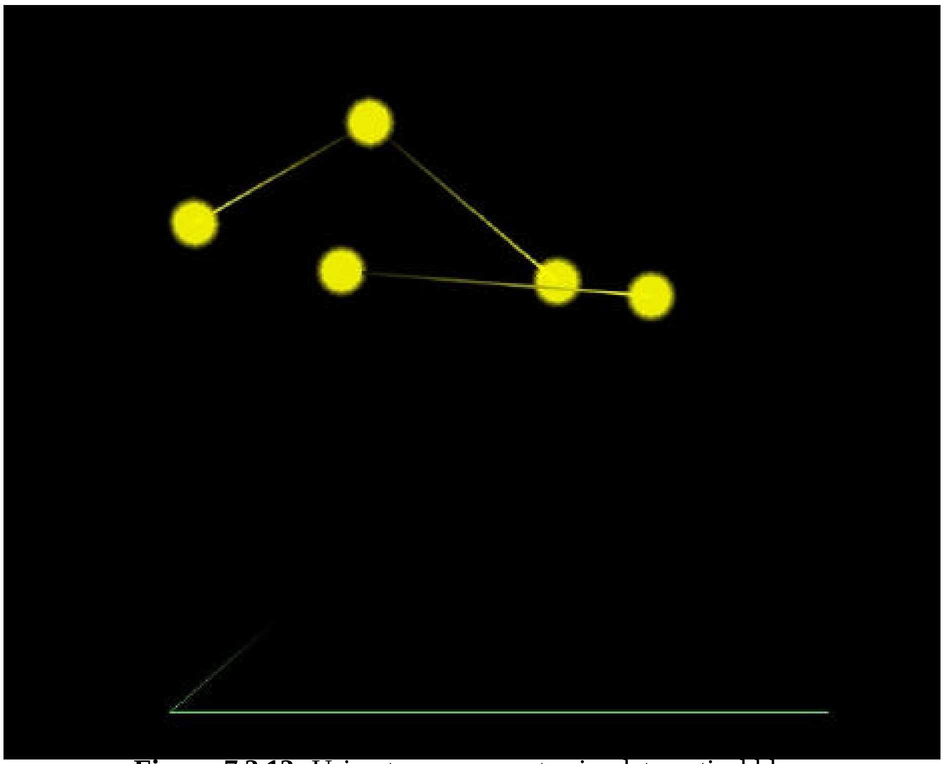


Figure 7.3.12: Using transparency to simulate optical blur.

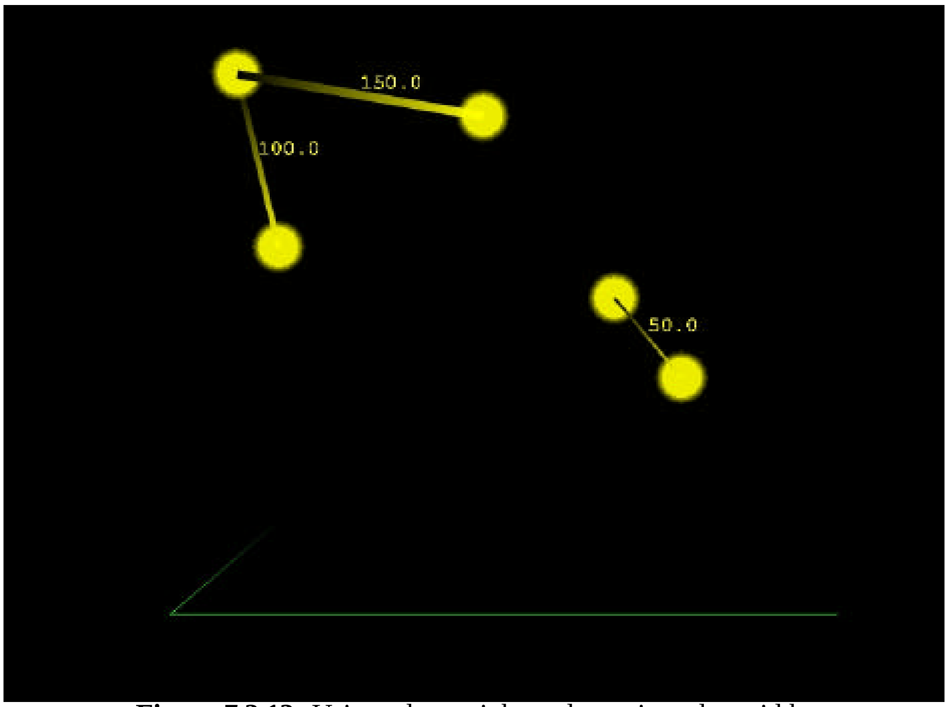
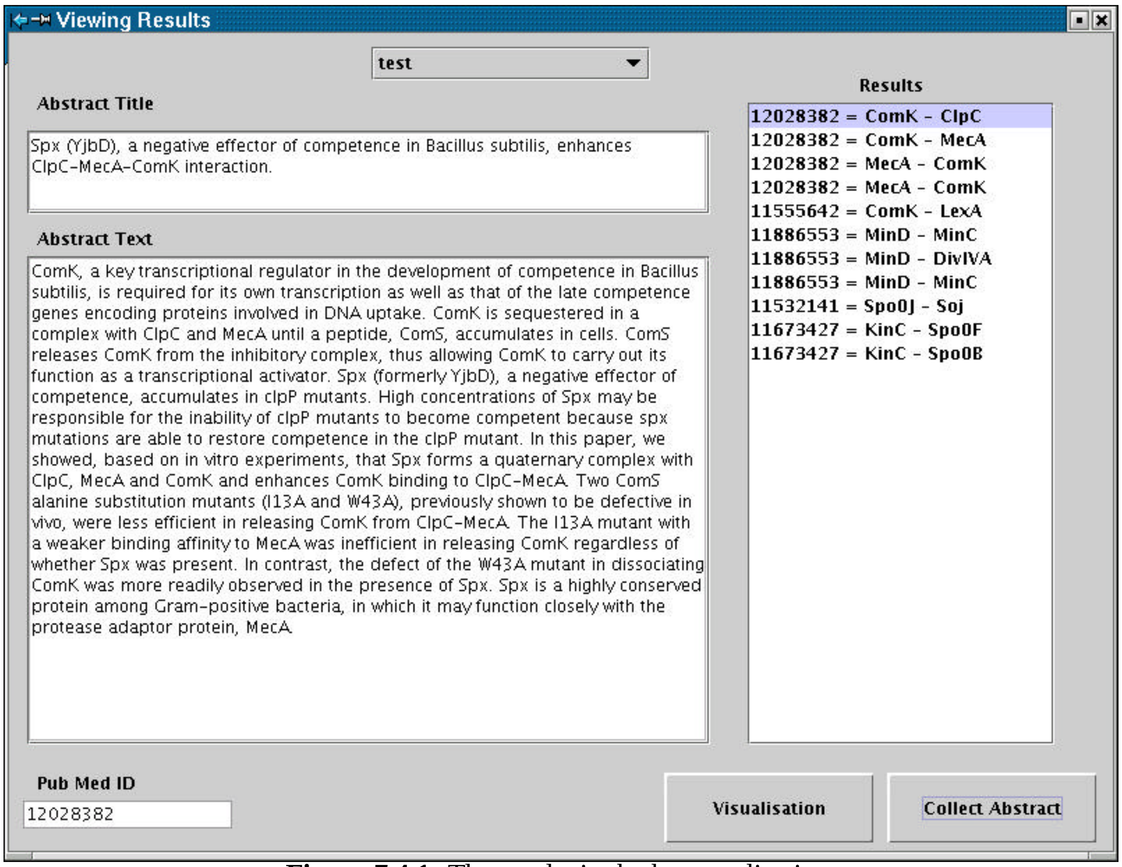


Figure 7.3.13: Using edge weight to determine edge width.

7.4 Results of Software Integration

One of the aims of this project was to implement a graph visualisation system that could be easily integrated with another program. The results of that successful integration are demonstrated in this section. The application is designed to cope with various forms of data and so it was not difficult finding another project to test the integration with. Mr Jon Bainbridge was developing a piece of software designed to make information extraction easier. As Chapter One had discussed the perceived importance of combining visualisation techniques with data-mining methods this seemed ideal. The host application involves the collection of literature from a database and then the extraction of relevant information from that literature. The software is connected to the Medline medical database and is currently set-up to extract information on protein-protein interactions. This meant that Jon could follow the simple steps in the user manual for JVS and set up a data model with the edges representing these protein interactions. The 'bolt-on' classes can then generate graphical representations of that data and the user of the combined system then has access to the visualisation techniques that are at the core of this project. It was a thoroughly successful integration and proves that this project developed a piece of software that adheres to the principles of modular design and software integration. What follows are the results of just one of the integration-testing sessions that were carried out:



Abstract Title

Spx (YjbD), a negative effector of competence in Bacillus subtilis, enhances ClpC-MecA-ComK interaction.

Abstract Text

ComK, a key transcriptional regulator in the development of competence in Bacillus subtilis, is required for its own transcription as well as that of the late competence genes encoding proteins involved in DNA uptake. ComK is sequestered in a complex with ClpC and MecA until a peptide, ComS, accumulates in cells. ComS releases ComK from the inhibitory complex, thus allowing ComK to carry out its function as a transcriptional activator. Spx (formerly YjbD), a negative effector of competence, accumulates in clpP mutants. High concentrations of Spx may be responsible for the inability of clpP mutants to become competent because spx mutations are able to restore competence in the clpP mutant. In this paper, we showed, based on in vitro experiments, that Spx forms a quaternary complex with ClpC, MecA and ComK and enhances ComK binding to ClpC-MecA. Two ComS alanine substitution mutants (I13A and W43A), previously shown to be defective in vivo, were less efficient in releasing ComK from ClpC-MecA. The I13A mutant with a weaker binding affinity to MecA was inefficient in releasing ComK regardless of whether Spx was present. In contrast, the defect of the W43A mutant in dissociating ComK was more readily observed in the presence of Spx. Spx is a highly conserved protein among Gram-positive bacteria, in which it may function closely with the protease adaptor protein, MecA.

Results

```
12028382 = ComK - ClpC
12028382 = ComK - MecA
12028382 = MecA - ComK
12028382 = MecA - ComK
11555642 = ComK - LexA
11886553 = MinD - MinC
11886553 = MinD - DivIVA
11886553 = MinD - MinC
11532141 = Spo0J - Soj
11673427 = KinC - Spo0F
11673427 = KinC - Spo0B
```

Pub Med ID

12028382

Figure 7.4.1: The results in the host application.

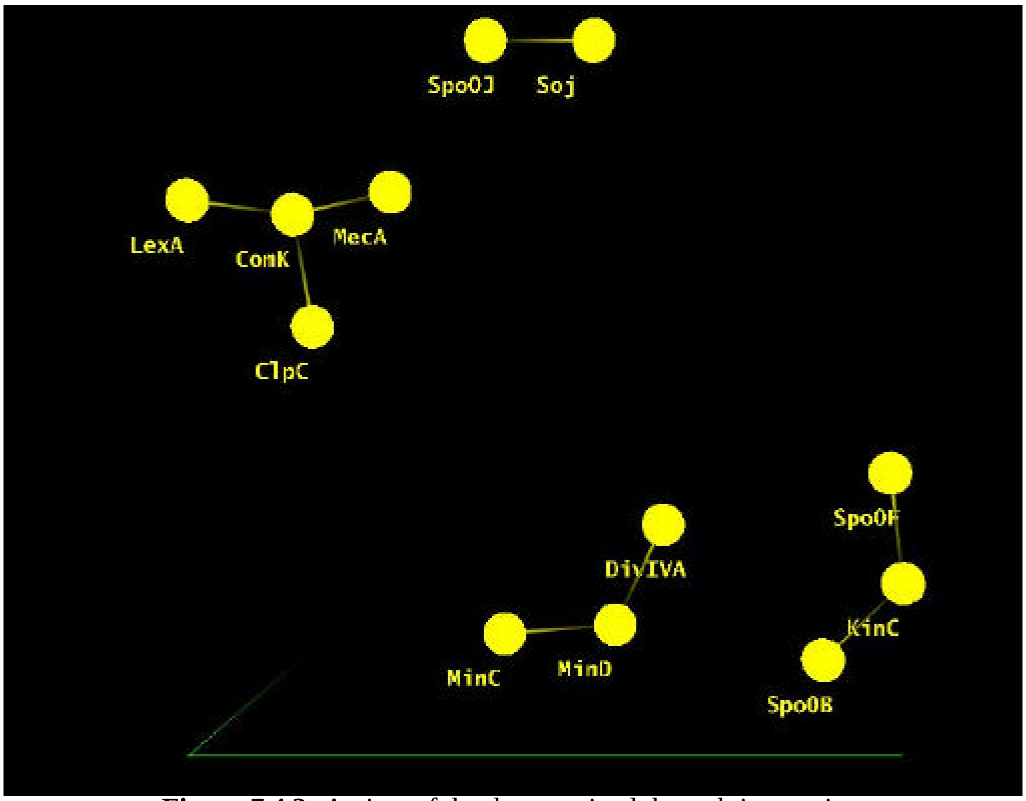


Figure 7.4.2: A view of the data received through integration.

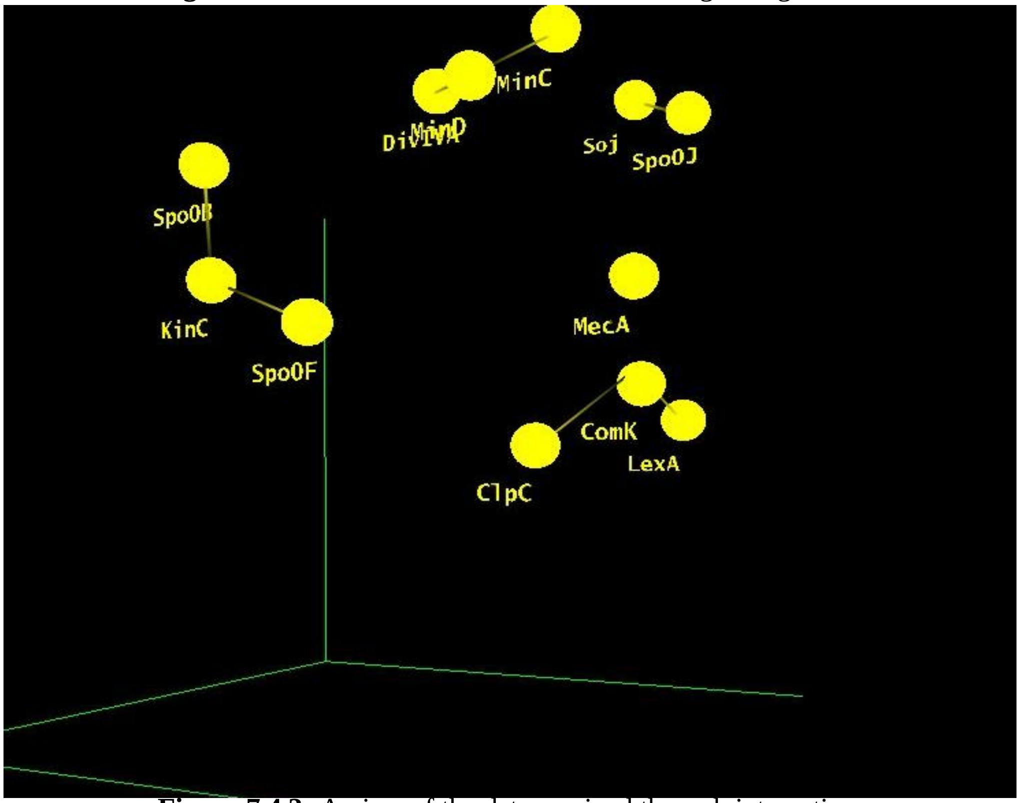


Figure 7.4.3: A view of the data received through integration.

Chapter Eight

Discussion

8 Discussion

There will be both positive and negative conclusions drawn from the work that was carried out and a retrospective look back on what has been learned. This chapter will also provide a discussion on the various aspects of the program that could be developed in the future.

8.1 Summary of Program Features

In summary, the application provides graph visualisation using two distinct techniques, both of which generate presentable, clear results. Perhaps just as importantly, the application was successfully integrated with a data-mining application. Due to the flexible nature of the data model this means that the visualisation system could be applied to literally any data set that involves a configuration of nodes and edges, or indeed just nodes. Summary of the key features of the application:

• **Semantic Depth of Field;**	realisation of concept & implementation
• **Emissive Lighting;**	generation of concept & implementation
• **Proven ease of integration;**	incorporate visualisation into existing system
• **Dynamic layout algorithm;**	far more presentable than random positioning
• **Test suite (with GUI);**	experiment with techniques, manipulate test data
• **Error handling (on model);**	robust system of handling external data input

8.2 Completion of Objectives

This section will discuss the completion of objectives making a distinction between what has been achieved and what was planned. To compare the finished application with its original specification (Chapter 1), most of the original objectives were met.

The primary objective of this project was to implement Semantic Depth of Depth and evaluate its ability to provide high-quality focus and context visualisation of data. While there is room for argument over whether the fog implementation is technically Semantic Depth of Field, the results clearly show that it has a similar effect. The key aim was kept in mind throughout the course of the project; the result is a system that has achieved, to a high degree, what was expected of it. The basic idea of any 'focus & context' approach to visualisation is to enable the user to have the primary object of interest presented in detail while at the same time having an overview (or context) available.[25] The software allows the user to decide which objects are of primary interest and then presents a view of that data, clearly highlighting those objects while retaining the others as context.

The secondary objective was to further improve visualisation, or at least attempt to, by implementing an additional technique. This technique was developed during the research phase of this project and named Emissive Lighting. Again, the technique was successfully implemented, although the results did not match those of the SDoF implementation. Aside from the actual method used to visualise the nodes, the technique was implemented in the same way as SDoF, with the aim of generating clear, concise and consistent source code.

Another objective was to produce a system that could model different types of data and be easily integrated with another application if necessary. The simple way in which the design models nodes and edges ensures that the completed application could indeed be applied to various forms of data. Given the intense focus on design patterns at the design and implementation stage, the software can easily be integrated with another application. This was proven by a successful integration with a data-mining application.

The implementation of a dynamic graph layout algorithm was another objective that was successfully achieved. The algorithm itself is not particularly revolutionary. The important point is that it demonstrates that one can be easily implemented along side visualisation techniques that themselves make changes to the physical layout of the entities in the graph. The system can simultaneously re-arrange the nodes in two dimensions (to make them more presentable) and make changes to the position of nodes on a third axis (to move them into fog).

The final two objectives, as specified in Chapter One, were probably not so important. They were both achieved; the application can provide two visualisations (based on an individual model) along-side the original Sun Graph-Layout applet. That is useful in terms of demonstrating the importance of data visualisation, and the lack of understanding of data that can result from implementing no visualisation technique (as in the Sun applet). The web-page for this project served to update staff on the progress of the work, as well as providing assistance for use of the application (such as the API).

8.3 Future Work

Although the application did achieve all of its objectives (to some degree), as the work progressed there were an increasing number of areas of the program that deserved more attention than was possible. This is a summary of the work that would ideally be carried out on the program in the future, the list is roughly ordered according to the perceived importance of work in a particular area;

Two issues regarding the implementation of SDoF;

- Blurring of edges – “things that do not need to be blurred should not be” figure 5.1.8
- Unreadable fogged labels – “blurred text should remain readable”

- Picking to allow for ‘dragging’ and ‘dropping’ of nodes in the graph.

- Collision Detection to avoid colliding nodes.
- Investigation of the performance implications of using different shapes to represent nodes.
- A review of the implementation of edges, in particular, arrow directions – pattern masking lines for arrow effects.
- Introduce a custom universe specific to this program that would make it easier to understand the current methods of setting up views and fog.
- Low-performance mode for low-spec systems.
- Textual view of data model in 'bolt-on' mode (limited importance).

- Aspects of Human-Computer-Interaction (HCI). In particular, Usability Engineering – evaluation of the user interface, and user task analysis (GOMS).[9]
- Complete the conversion to an applet to allow for on-line demonstration of the system.
- Produce a more thorough demonstration showing just how vast the applications of this technology are. The program could easily be used along-side a file management system (e.g. windows explorer), for example, to improve information visualisation.

Even though an interactive system may be properly designed, it will not necessarily be completely intuitive to use or understand. There should be no assumption that a well designed system should come without help and documentation.[9] Help and documentation are a crucial aspect of any software project. Regrettably there was little time built into the project plan for generation of such documents. There is plenty of scope for future work in this area.

At the present time one of the limitations of the application is the number of nodes that it can handle. This is due to the size of the addressable coordinate field (which can be grown in the future). The number of nodes is currently kept relatively small due to the performance implications but any future work could look into the exact capacity of the system.

Had there been more time available, this project would have undertaken a more thorough investigation into the possibility of using exponential fog which could have resulted in the combination of a **fog blending factor** with a **relevancy function**, it would have also allowed for a three-dimensional graph with Semantic Depth of Field. The use of a relevancy function would enable the visualisation system to implement multiple levels of relevance; however, it is known that SDoF only works effectively with two or three groups. The use of transparency to simulate blur would also have been given some further attention.

8.4 Conclusions

The use of Semantic Depth of Field is still in its infancy, yet this application has proven that an improvement in visualisation can be achieved by using the technique, or something similar. No doubt over the next few years there will be further attempts to implement the effect in different surroundings and, hopefully, some thorough analysis of the results in terms of perceptual psychology. Following personal communications with Robert Kosara (Head of Visualisation Research, VRVis) it was decided, at a fairly late stage, to include a third visualisation in addition to the 'fogging' and 'emissive lighting'. As has already been discussed earlier, the approach taken in tackling Semantic Depth of Field involved the use of fog, which is not technically the same as blur. There was the assumption that a true blur effect was not possible, but this was not proven. There is now evidence in the results section (Chapter 7, Figure 7.3.12) that blur has been investigated, but the conclusion is that the use of multiple transparent spheres to represent one blurred node is not currently viable in a graph system that may wish to model hundreds of nodes. Although the use of transparency is perhaps the only thing closer to blur than fog (in Java 3D), it is still not technically a correct implementation, i.e. a picture taken of a sphere that is out of focus would look different. On top of that, the implementation of transparency in Java 3D is actually very similar to the way in which fog works, blending object colour with background colour.[6] The advantage with fog is that there is no need for the creation of multiple *Shape3D* objects, which it is known has serious performance implications in Java 3D.

The discussion above lead directly to the real conclusion of this work; in order to satisfy the aims of Semantic Depth of Field, and indeed other visualisation techniques, it is necessary to ignore any pre-conceptions about how to go about achieving such goals and be truly flexible in any approach to implementing a solution. It is recognised that in order to realise the essential ideas of any 'focus & context' technique in new situations of computer use, it is necessary to move beyond the traditional notions of these techniques.[25] It still seems fair to say that the system does provide a viable alternative (to true SDoF) in the short-term until technological advances allow for the implementation of a truer blur effect, without hampering performance.

The use of Emissive Lighting is an even more recent proposal than Semantic Depth of Field. To the best of this writer's knowledge this potentially useful visualisation technique has not been recognised elsewhere. The initial results, although not as impressive as those of the SDoF implementation, are very promising. With more work on the method used to darken the colours the result could be even more pleasing. One of the requirements of any visualisation system is that a very large network may be displayed on the screen and zooming-in to any selected part of it is allowed to make that part legible.[30] The Emissive Lighting effect, because of its lack of dependence on 'depth-cueing', does make this possible.

There is the belief that an individual may need to write their own program to visually represent the specific data that they have collected.[30] A further conclusion of this project, although it may seem like an ambitious statement, is that this is unnecessary providing the design of the data model is flexible and intelligent. It is the closeness of mapping between the data model implemented in this software (node & edge) and the fundamental forms of data (entity & relation) that is the key to the success of the integration and, ultimately, the use of the system to visualise data.

Bibliography

Books;

[1] Tufte, E.R. (1990) Envisioning Information. Connecticut, Graphics Press.

[2] Schroeder, W., Martin, K., Lorensen, B. (1998) The Visualization Toolkit. 2nd ed. New Jersey, Prentice Hall.

[3] Schneiderman, B. (1998) Designing the User Interface: Strategies for Effective Human-Computer Interaction. 3rd ed. Addison-Wesley.

[4] Wilson, L.B., Clark, R.G. (2001) Comparative Programming Languages. 3rd ed. Harlow, Addison-Wesley.

[5] Goldstein, E.B. (2001) Sensation and Perception. Wadsworth Publishing.

[6] Walsh, A.E., Gehringer, D. (2002) Java 3D API Jump-Start. New Jersey, Prentice Hall.

[7] Angel, E. (2000) Interactive Computer Graphics: A Top-Down Approach with OpenGL. 2nd ed. Addison-Wesley.

[8] Ware, C. (2000) Information Visualization: Perception for Design. London, Academic Press.

[9] Dix, A., Finlay, J., Abowd, G., Beale, R. (1993) Human-Computer Interaction. Hemel Hempstead, Prentice Hall.

[10] Di Battista, G., Eades, P., Tamassia, R., Tollis, I. (1999) Graph Drawing: Algorithms for the Visualization of Graphs. New Jersey, Prentice-Hall.

[11] Gross, J., Yellen, J. (1999) Graph Theory and its Applications. London, CRC Press.

[12] Horstmann, C., Cornell, G. (2003) Core Java 2: Fundamentals. California, Sun Microsystems Press.

[13] Bailey, D.A. (1999) Java Structures: Data Structures in Java for the Principled Programmer. Singapore, McGraw-Hill.

[14] Jorgensen, P.C. (2002) Software Testing: A Craftsman's Approach. 2nd ed. Boca Raton, Florida, CRC Press.

[15] Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F., Phillips, R.L. (1994) Introduction to Computer Graphics. Addison-Wesley.

Publications & Technical Reports;

[16] MacCormack, A., Kemerer, C.F., Cusumano, M., Crandall, B. (2002) Flexible Models of Software Development: Must we trade off efficiency & quality? IEEE Software [submitted].

[17] Kosara, R., Miksch, S., Hauser, H. (2001) Useful Properties of Semantic Depth of Field for Better F+C Visualization. VRVis Research Centre, Vienna, Austria. Technical Report, VRVis-028.

[18] Potel, M. (1996) Model-View-Presenter: The Taligent Programming Model for C++ and Java. Taligent Inc. {since merged into IBM}. Technical Report.

[19] Baudisch, P., Good, N., Stewart, P. (unknown date) Focus Plus Context Screens: Combining Display Technology with Visualization Techniques.

[20] Munzner, T. (2002) Information Visualization. IEEE Computer Graphics and Applications, 22 (1) January, pp.20-21.

[21] Shneiderman, B. (2002) Inventing Discovery Tools: Combining Information Visualization with Data Mining. Information Visualization, 1 (1) March, pp.5-12.

[22] Cockburn, B. (2002) Application of Semantic Depth of Field to Visualisation Problems in Bioinformatics. MSc. dissertation, University of Newcastle-upon-Tyne.

[23] Dogrusoz, U., Feng, Q., Doorley, M., Frick, A. (2002) Graph Visualization Toolkits. IEEE Computer Graphics and Applications, 22 (1) January, pp.30-37.

[24] Bruß, I., Frick, A. (unknown date) Fast Interactive 3-D Graph Visualization.

[25] Björk, S., Redström, J. (unknown date) Redefining the Focus and Context of Focus+Context Visualizations.

[26] Kosara, R., Miksch, S., Hauser, H. (2002) Focus & Context Taken Literally. IEEE Computer Graphics and Applications, 22 (1) January, pp.22-29.

[27] Kosara, R., Miksch, S., Hauser, H. (2001) Semantic Depth of Field. VRVis Research Centre, Vienna, Austria. Technical Report, VRVis-001.

[28] Ware, C., Purchase, H., Colpoys, L., McGill, M. (2002) Cognitive Measurements of Graph Aesthetics. Information Visualization, 1 (2) June, pp.103-110.

[29] Kosara, R. (2001) Semantic Depth of Field – Using Blur for Focus & Context Visualization. Ph.D. thesis, Vienna University of Technology.

[30] Cawkell, T. (2001) Progress in Visualisation. Journal of Information Science, 27 (6) December, pp.427-438.

[31] Chen, C. (2002) Information Visualization. Information Visualization, 1 (1) March, pp.1-4.

[32] Song, M. (2000) Visualization in Information Retrieval: a three-level analysis. Journal of Information Science, 26 (1) February, pp.3-19.

Web-Pages;

[33] Unknown Author (2001) Model-View-Presenter Framework [Internet] Object Arts Ltd, London. Available from: <http://www.object-arts.com/OldStuff/Overviews/ModelViewPresenter.htm> [Accessed 20 October, 2002].

[34] Burbeck, S. (1992) Applications Programming in Smalltalk-80: How to use Model-View-Controller. [Internet] Department of Computer Science, University of Illinois at Urbana-Champaign. Available from: <http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html> [Accessed 24 September, 2002].

[35] Wheeler, S. (1996) The Model-View-Controller Architecture. [Internet] CERN, European Laboratory for Particle Physics. Available from: <http://rd13doc.cern.ch/Atlas/Notes/004/Note004-7.html> [Accessed 24 September, 2002].

[36] Baray, C. (1999) The Model-View-Controller Design Pattern. [Internet] Department of Computer Science, Indiana University. Available from: <http://www.cs.indiana.edu/~cbaray/projects/mvc.html> [Accessed 22 September, 2002].

[37] Hagen, J. (1996) Graph Layout; examples. [Internet] Sun Microsystems, Santa Clara, California. Available from: <http://java.sun.com/applets/jdk/1.0/demo/GraphLayout/index.html> [Accessed 23 September, 2002].

[38] Campione, M., Walrath, K. (1997) About the Java Technology. [Internet] Sun Microsystems, Santa Clara, California. Available from: <http://java.sun.com/docs/books/tutorial/getStarted/intro/definition.html> [Accessed 2 October, 2002].