

Visualizing QoS in Networks

By
Werner Grift

THESIS PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE AT STELLENBOSCH UNIVERSITY

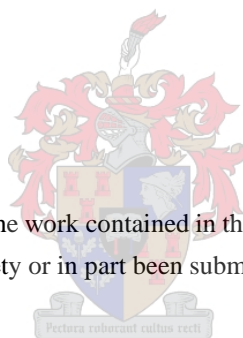


Supervised by: Prof. A.E. Krzesinski

December, 2006

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

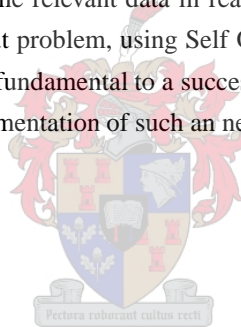


Signature:

Date:

Abstract

Network simulations generate large volumes of data. This thesis presents an animated visualization system that utilizes the latest affordable Computer Graphics (CG) hardware to simplify the task of visualizing and analyzing these large volumes of data. The use of modern CG hardware allows us to create an interactive system which allows the user to interact with the data sets and extract the relevant data in real time. We also present an alternate approach to the network layout problem, using Self Organizing Maps to find an aesthetic layout for a network which is fundamental to a successful network visualization. We finally discuss the design and implementation of such an network visualization tool.



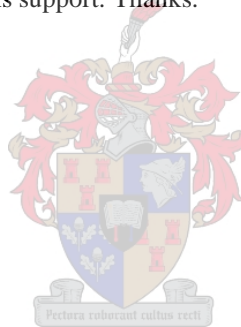
Opsomming

Netwerk simulaties genereer groot volumes data. Hierdie tesis stel voor 'n geanimeerde visualisering wat gebruik maak van die nuutste bekostigbare rekenaar grafika hardeware om die visualisering van groot volumes data te vergemaklik. Die gebruik van moderne rekenaar grafika hardeware stel ons in staat om sagteware te skep wat n gebruiker in staat stel om met die data te werk. Ons stel voor 'n alternatiewe benadering om die netwerk se uitleg daar te stel, met die hulp van tegnieke wat gebruik word in die studie van neurale netwerke. Ons bespreek dan die ontwerp en implementering van so 'n netwerk visualisering program.



Acknowledgements

I would like to thank my parents for all the support they have given me throughout the years. I would also like to thank Cobus Combrink who always provided me with the bigger picture of things. Lastly I would like to thank my supervisor, Professor Anthony E. Krzesinski, for all the inspiration and motivation he has given me over the past couple of years. This thesis would not have been possible without his support. Thanks.



Contents

Abstract	iii
Opsomming	iv
Acknowledgements	v
1 Introduction	3
1.1 Information Visualization using Modern Computer Graphics Hardware	3
1.2 Overview	4
1.3 Network Visualization Background	5
1.4 Research Motivation and Goals	8
2 Graph Visualization	9
2.1 Graph Layout Algorithms	9
2.2 The Spring Modeling Algorithm	10
2.3 The Self Organizing Map	10
2.3.1 The Neighbourhood Function	12
2.3.2 The Alpha Function	13
2.3.3 Refining the Neighbourhood Function with Observations	15
2.3.4 Implementing a Self Organizing Map, a Performance Analysis	16
2.3.5 Customizing Network Layouts with the SOM Algorithm	17

3	Program Design	19
3.1	The Design Paradigm	19
3.2	An Object Oriented Approach Using C++	20
3.2.1	The Class Diagram Key	22
3.2.2	The Proposed Class Design Model	23
4	Visualizing Network Data	29
4.1	Representing Network Nodes	30
4.1.1	Providing Network Engineering Tools for Simulator Testing	31
4.2	Link Visualization	33
4.2.1	Selecting an Appropriate Graphical Object to Represent a Link	33
4.2.2	Link Visualization Interpretations	34
4.3	Rendering Packets	35
4.3.1	Understanding Interactive Computer Graphics (CG)	36
4.3.2	Using the Vertex Shader to Improve Packet Animation Performance	37
4.3.3	Performance Evaluation	39
4.3.4	The Physics Involved in Packet Visualization	43
4.4	Visualizing Label Switched Path Networks	46
4.4.1	The Advantages of LSP Network Visualization	46
5	Data Interface	48
5.1	Classic Data Exchange Models	48
5.2	Using C++ Member Function Pointers to Simplify the Data Interface's Design and Implementation	49
5.3	Data Interface Implementation	51
5.3.1	The Standard Network Stream	52
5.3.2	The LSP Network Stream	53
5.4	An Example	54

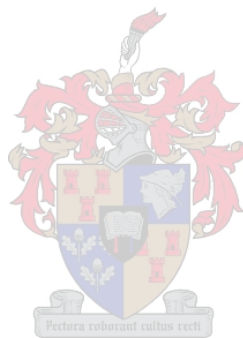
6 Conclusion	57
6.1 The Use of Network Visualization	57
6.1.1 Network Visualization as an Analysis Tool	57
6.1.2 Visualizing QoS in Real Networks	58
6.2 Aspects of Network Visualization	58
6.2.1 Graph visualization	59
6.2.2 Program Design	59
6.2.3 Visualizing Network Data	60
6.2.4 Data Interface	60
6.2.5 Network Performance Visualization	60
6.3 Final Remarks	61
A The Vertex Shader Program Used to Achieve Packet Animation	62
Bibliography	64



List of Figures

1	One iteration of the SOM algorithm. The nodes \mathbf{m}_i are trained on the random input sample \mathbf{x} by moving them closer to \mathbf{x}	11
2	The neighbourhood function calculated for node 4.	13
3	The function $\alpha(t) = 1 - 2 \arctan(t)/\pi$	14
4	The function $\alpha(t) = 1 - 2 \arctan(\sigma(t))/\pi$ with $\sigma(t) = Rt/T, T = 8000$. Different graphs are show for different values of R , where smaller values of R are used for large networks and vice versa.	15
5	The time t in seconds to achieve an aesthetic layout with the SOM algorithm for a network consisting of n nodes.	17
6	The effect of different probability density functions on the resulting layouts for the same network structure.	18
7	A suggested optimum class structure of our network visualization program.	21
8	The circular inherency problem encountered with a simple network class model.	24
9	The diagram indicates which classes are suitable for extension when writing a customized visualization for network data originating from a unique source.	28
10	Visualizing different types of nodes inside a network.	32
11	Visualizing network links with different bandwidths indicated by links widths.	35
12	The average frames per second FPS achieved when rendering n packets.	41
13	The average percentage gain achieved over the same number of packets n rendered as in Figure 12.	42

14	Illustration of the data interface used to transfer event data.	51
----	-------------------------------------------------------------------------	----



Chapter 1

Introduction

1.1 Information Visualization using Modern Computer Graphics Hardware

Recently the computer graphics (CG) industry has undergone a major transformation. State of the art CG hardware is now available to the consumer at a fraction of its previous cost, creating a market that will drive CG technology for years to come.

The introduction of *Graphics Processing Units* (GPUs), a dedicated programmable graphics chip, enables complex and information rich visualizations. The popularity of these GPUs has prompted several companies to write high level *Application Programming Interfaces* (APIs) to make these GPUs easy to program.

The goal of this thesis is to visualize *Multiprotocol Label Switching* (MPLS) networks. To achieve this goal we have to address a number of problems: defining an efficient input/output system, utilizing the graphics hardware properly and obtaining an appropriate metaphor to visualize MPLS networks.

Simulating a large computer network on a PC-class computer is likely to encounter scalability problems. The same holds true for network animation. For a network animation to be successful, a certain visual quality must be maintained. Since all the data cannot be displayed at once the user might also be required to interact with the animation to extract pertinent data.

Until recently PC-class computers were not capable of providing such functionality. While such computers are capable of determining the specifics of the animation, they cannot update the display fast enough for the animation to be of a satisfactory visual quality.

In the late 1990's specialized graphics hardware was designed to overcome this short-coming. Although this solved the problem of animation, it did not solve the aspect of real time interaction. A network animation might have hundreds of links transporting millions of packets. Being able to interact with such an animation requires the *Central Processing Unit* (CPU) to make millions of calculations per second which saturates the CPU's ability to perform other operations that render the animation.

The *Graphics Processing Unit* (GPU) solves this problem by freeing up the CPU from graphics calculations. This can be viewed as a form of a parallel processing; while the CPU determines what is to be drawn, the GPU computes the transformations and renders the scene.

Although this technology was mainly intended for computer games, other fields such as Information Visualization also benefited. GPU performance increases at approximately three times the rate of *Moore's Law* cubed [1]. The capability of visualization packages will therefore increase dramatically.

1.2 Overview

As stated above, our goal is to visualize the quality of service, or QoS, in MPLS networks. It is widely recognized that the MPLS network protocol can secure delivering guaranteed QoS in next generation networks and we therefore selected this protocol as our visualization goal. We will start with basic topology visualizations of networks and progress towards visualizing more complex phenomena within networks.

One could also have used a graphical metaphor to convey the current state of the network to a network engineer, but we feel that it could be difficult for someone not familiar with the metaphor's meaning to gain insight into a network. We therefore chose the node/link layout visualization as a basis to serve as an intuitive approach in visualizing network data. Chapter 4 explains how we expand on this idea to visualize network protocols.

This approach does not come without it's own set of problems. Large networks can quickly saturate screen space and create confusion about the connectivity of the network. Since current computer displays are relatively small this problem can never be avoided. However, one can employ a set of techniques that attempt to prevent such a scenario from occurring. The use of bigger or multiple displays, the use of layout algorithms (Chapter 2) and the use of 3D graphics are but to name a few. One could also divide the network into a hierarchical set of clusters if the network becomes too large to visualize. These clusters might expand or contract on user demand to retain all network information as needed [2].

There aren't any limitations concerning the origin of the visualization data. Since interfacing with the visualizer is accomplished through a set of C++ libraries, any program

with data that can be visualized with the node/link layout paradigm can be interfaced with the visualizer with minimal effort. We avoid designing complicated scripting languages to interface the data source with the data visualizer, as these scripting methods are tedious to implement, generally slow and result in huge trace files. We did however make use of C++ member function pointers to serve as an optimized low level connection between the data source and the visualization. We elaborate on this in Chapter 5.

Since network data vary over time, network animation plays an important part of the of the visualization software's design. For an animation to be successful the network displayed should at least update once every 30 *msec* to allow the user to interact with the visualization. Interaction can be viewed as a very important part of the data extraction process. For large networks users will be required to navigate hundreds of links and nodes to focus on parts of the network that are important. One of the key concepts behind information visualization research, is to enhance understanding of a domain by relying on the ability of humans to rapidly recognize and make sense of large volumes of visualized data.

Writing structured code that performs well enough to be used for an animation is difficult to achieve. Object oriented code tend to run slower than classic coding methods because of the overhead associated with member function calls. A short discussion on how we attempt to balance these two paradigms of coding in our project to achieve animation is discussed in Chapter 3.



1.3 Network Visualization Background

With the rapid increase of computational power Computer Scientists began using computers for more than their ability to process numbers. Users could start to interact with the computer in different ways one of which was to visualize complex phenomenon in such a way that a better understanding of the problem under investigation could be gained.

Companies such as Silicon Graphics [3] started manufacturing specialized graphics workstations that were able to process much more complex visualizations than any personal computer. It was on these specialized workstations that the development of the first network visualizations were based. Few people had access to these expensive systems and the lack of standards that are in place today hindered the progress of information visualization.

Although the basic groundwork of network visualizations began in the 1980's, for example the basic node/link representations and matrix representations by Bertin [4], we discuss the work done in the mid nineties on graphics workstations as this is more related to our current work. We will demonstrate how the problems that occurred in the mid nineties are solved by using modern day computer graphics hardware.

Becker, Eick and Wilks [5] introduced a visualization system called SeeNet which produced static 2D layouts of networks on geographic backgrounds to give more insight into the geographic relationships of the networks. The problem encountered was that geographic layouts are highly susceptible to display clutter, where many links are overdrawn in close proximity because of the particular geographic layout. To solve this they introduced an information culling system where links that are “interesting” or most overloaded are drawn last in such a way that they obscure the less interesting links below. Node maps are also introduced where information about all links connected to a particular node are aggregated into one parameter and visualized by the appropriate metaphor at the node’s location. It was suggested that this technique solves the cluttering problem, but maybe too aggressively as much link information is lost due to the aggregation of link data. Alternatively they suggest a network layout that has no geographic information at all, consisting of a network visualized as a matrix. Each node is assigned to a row i and column j and a link connecting node i to node j is then coloured at position (i, j) according to the link load from i to j . They then introduce the important concept of dynamic parameter focusing. This requires the user to adjust certain parameters that govern the visualization to produce a clearer view of the network data. The dynamic part entails display updates as the user changes a particular parameter. Users are therefor required to use their cognitive skills to adjust parameters in such a way as to obtain the optimum visualization.

Eick, Cox and Taosong [6] continued to develop SeeNet3D which uses 3D network displays to solve the display clutter problem. They introduce five novel ways of visualizing time varying network data. The first two consist of geographical arc maps connecting nodes located on either a sphere or a flat surface containing the earth’s continents which are then viewable in 3D. The rest are so called “drill down views” that focus on certain portions of a network for more in-depth information. They report that these techniques are suitable for ¹sparse networks, but for larger networks display clutter problems could only be avoided by using suitable graph layout algorithms. Since our network visualization visualizes hundreds to thousands of network nodes we do not retaining geographic information, but rather make use of graph layout algorithms to obtain an optimal layout. Their system also features parameter focusing by means of adjusting the viewing angle, zoom and translucency to obtain an optimum visualization. Updating the display fast enough to display these changes was difficult to accomplish because of the visual complexity of the scenes which compromised the technique. One of our goals is to use the latest computer graphics hardware available to avoid such problems. To illustrate how far computer graphics hardware has come over the past few years, a typical SeeNet3D scene took one second to render at the the time whereas today the same scene can be rendered hundreds of times per second on a standard personal computer. Although this suggests that interactivity and scalability issues are solved by today’s computational power however, depending on how complex the visualization one can still run into performance problems that compromises interactivity with the visualization.

¹A network whose connectivity is less than 0.1

Taosong and Eick [7] presented a object-oriented network visualization system written in C++. Although the system does not provide any new network displays, it addresses other issues involved with network visualization one of which is network data handling. Since time varying networks generate large volumes of data, the storage of this data could exhaust the main memory of computer systems. They propose storing the data in a database. This way certain data manipulating functions could easily be implemented through SQL queries, for example selecting all the links with certain properties. Although this causes other concerns, such as the delay caused by query performance when animating a network, we feel that this is a useful way of approaching the data mining aspect of network visualization and recommend further investigation in future work. They describe a network base class providing rich functionality for visualizing a network. One can use these base classes to derive specialized classes tailored for specific visualizations. We also followed the same design paradigm for our network classes, providing basic functionality and building upon them to provide more complex network representations.

More in accordance with our current research is VINT's Network Animator or Nam [8]. Nam was primarily designed to visualize trace files generated by ns² simulations, but can also be used to process data generated by a real network. Nam's main feature is the ability to animate packet flows over a network which is useful for network engineers when designing a network protocol or investigating the workings of existing protocols. Special mention is made about the invaluable roll Nam has played in protocol development by network engineers. Nam visualizes packet flows in 2D without using any graphics hardware acceleration which limits its potential. Interactivity is limited because of low frame rates and packets can sometimes appear to be moving backwards on links. Nam also provides statistical outputs in the form of 2D graphs to complement its visual system.

A part of our research involves implementing such a packet animator but on a large scale using the best technology available. We then answer questions such as how big a network can be animated using this technique and what insight can be gained through it. We extend this idea to visualize next generation protocols such as the MPLS protocol to provide network engineers with the same type of visual support systems.

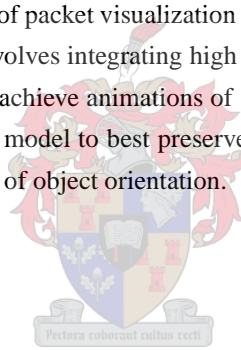
The National Laboratory for Applied Network Research or NLANR [9] developed a visualizer called Cichlid [10] to aid visualizing the large data sets generated by numerous network monitoring projects at NLANR. Cichlid was designed to be a distributed system consisting of a server side that gathers user data sending it over a TCP connection to a client side visualization. Although our system also makes use of distributed sources of data, it is implemented with less complexity. Cichlid gathers data converting it to an internal data structure called a Data-Set, sending it in compressed form to the client side visualizer. The client then decompresses the data for processing. Our system makes use of a byte stream sent from the server to the client with all of the data processing handled by the client. This way we ensure simplicity at the server side which makes the system easier to

²ns is a network simulation package that simulates network protocols on a packet level.

use. This enables Cichlid and our visualization to perform real time network visualization with different data sources as input. Cichlid uses bar charts and topology maps to visualize network data in interactive 3D using graphics libraries such as GLUT[11]. While the use of such libraries speeds up development time of an application the infrequent updating of such libraries causes it to be of little value in accessing and using the latest graphics hardware features. This causes Cichlid to perform sluggishly on performance systems which again compromises the interactivity of the visualization.

1.4 Research Motivation and Goals

The goal of this thesis is to develop an network visualization tool that can assist network engineers using their cognitive skills to understand occurrences of certain phenomena inside networks. At worst this might involve investigating as far down as the packet layer. Our research involves finding the limits of packet visualization and reporting on the insight gained by such a visualization. This involves integrating high end graphics programming with suitable network object models to achieve animations of sufficient performance. We strive to design such an object oriented model to best preserve data abstraction while not compromising the performance because of object orientation.



Chapter 2

Graph Visualization

2.1 Graph Layout Algorithms

Several node layout algorithms [12, 13, 14, 15, 16] attempt to approximate an aesthetic graph layout. The most aesthetic layout could be defined as a layout that produces the least amount of overdraw, where links obscure one another because they are drawn on top of each other. Although no algorithm exists that minimizes the amount of overdraw occurring in a particular layout, certain graph layout algorithms produce the desired effect. Given enough nodes and links there exists a limit to the least amount of overdraw that can occur. If a display has a rendering surface area A then it is guaranteed that overdraw will occur if the rendering area taken up by the nodes, $N = n_1 + n_2 + \dots + n_i$ added to the area taken up by the links, $L = \ell_1 + \ell_2 + \dots + \ell_j$ is bigger than the rendering surface area A . Ensuring that $N + L \ll A$ we keep the components n_i and ℓ_j small enough to minimize the effects of overdraw whilst keeping the components large enough to be visible on a computer display system. The other primary attribute of an aesthetic layout is to convey as much information about the structure of the network as to make sense of certain phenomena pertaining to bandwidth allocations and link flows inside a network.

The spring modeling algorithm [16] is one such algorithm. This algorithm regards the network as a set of interconnected springs, with each spring having its own potential energy. The overall energy of the network is calculated as the aggregate of all the spring energies, which the spring modeling algorithm then minimizes. The system oscillates to reach this low energy state which is an equilibrium that represents the final layout. Early implementations of this heuristic approach were done in 2-dimensions, but 3-dimensional implementations are also common.

2.2 The Spring Modeling Algorithm

We implemented the spring modeling algorithm to explore the utility of such an heuristic approach to achieve aesthetic layouts. A node is set to have a repulsive force with respect to all other nodes that is inversely proportional to the distance between them. All connected nodes are then given a spring force which acts as a weightless force between them. The system is then “released” to converge to a state of low energy. This is achieved by calculating the average force vector for each node and then translating each node in the direction of its average force vector by a distance proportional to the force. This process is repeated until a steady state is reached.

This simple approach to solving the spring modeling algorithm, yields an aesthetic layout. This method however leads to a number of problems pertaining the way the spring modeling algorithm solves the layout problem. The respective input parameters for the algorithm include spring length, spring stiffness, initial configuration and the repulsive forces between nodes. The layout achieved is highly dependent on these parameters and no relationship could be found between these parameters and a network’s size and connectivity.

A large sparsely connected network will span across a large volume since the total repulsive force between all nodes is too large for the springs to coalesce the network. In the same way a large strongly connected network would tend to implode forcing different nodes close to or on top of each other which causes overdraw. The problem becomes more acute when a network has an uneven connectivity distribution where some parts are sparsely connected and others are strongly connected. The resulting layouts of such networks have shapes that are elongated in different areas which makes viewing the network as a whole difficult. Selecting the correct parameters for the quadratic force functions with respect to different sized networks and their connectivity is a difficult problem

Although an aesthetic layout can often be achieved by constantly adjusting these input parameters and evaluating the resulting layout, the process is time consuming and tedious. We therefore considered another approach known as the *Self-Organizing Map* [17] (SOM) from the field of neural networks.

2.3 The Self Organizing Map

A SOM can be viewed as a simple type of regression, where a function is fitted to a distribution of input samples. The following example involves fitting a set of *discrete reference vectors* to a distribution of vector input samples. The set of discrete reference vectors represents the locations of the network nodes inside a network layout.

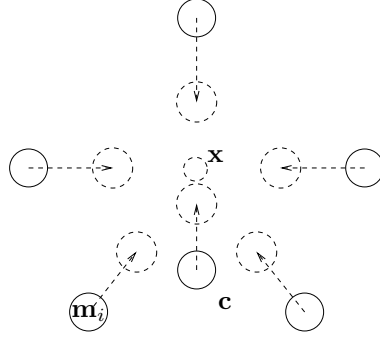


Figure 1: One iteration of the SOM algorithm. The nodes \mathbf{m}_i are trained on the random input sample \mathbf{x} by moving them closer to \mathbf{x} .

Consider Figure 1 where the nodes $i \in \mathbb{N}$ are represented by a set of reference vectors

$$\mathbf{m}_i = (\mu_{i1}, \mu_{i2}, \dots, \mu_{in}) \in \mathbb{R}^n.$$

The random input vector at time t

$$\mathbf{x}(t) = (\xi_1, \xi_2, \dots, \xi_n) \in \mathbb{R}^n$$

is compared with the reference vectors \mathbf{m}_i and the location of the best matching node, c_t , is defined as the *response* node $c \in \mathbb{N}$ at time t .

Let $\mathbf{x}(t) \in \mathbb{R}^n$ be a random vector. The SOM is the nonlinear projection of an arbitrary probability density function $P(\mathbf{x})$ onto a set of reference vectors \mathbf{m}_i . There are many ways of defining the closest match for an input sample $\mathbf{x}(t)$. In most cases the minimum Euclidean distance $\|\mathbf{x} - \mathbf{m}_i\|$ between the sample \mathbf{x} and the reference vector \mathbf{m}_i is used to find the response node. Other methods include matching criteria based on the dot product of \mathbf{x} and \mathbf{m}_i . In Figure 1 let c denote the response node sought, then

$$\|\mathbf{x} - \mathbf{m}_c\| = \min_i (\|\mathbf{x} - \mathbf{m}_i\|).$$

The process by which the nonlinear projections are formed is referred to as the learning process. In the classic SOM, nodes that are *topographically*¹ close to the response node c will activate and also learn from the input sample \mathbf{x} . This results in a local relaxation of the nodes in the neighbourhood around the input sample, which if repeated leads to a global ordering. This iterative process can be described by the equation

$$\mathbf{m}_i(t+1) = \begin{cases} \mathbf{m}_i(t) + \alpha(t) (\mathbf{x}(t) - \mathbf{m}_i(t)), & i \in N_c(t) \\ \mathbf{m}_i(t), & i \notin N_c(t) \end{cases}, \quad \text{where } t \in \mathbb{N} \quad (1)$$

for arbitrary $\mathbf{m}_i(1)$. $\alpha(t)$ represents the learn rate as a function of time and is covered in more detail in section 2.3.2. The function $N_i(t)$ is referred to as the *neighbourhood*

¹Some nodes are connected to other nodes in a neighbouring relationship that forms the topology.

function and plays a central role in the relaxation process. It is required for convergence that $\alpha(t) \rightarrow 0$ when $t \rightarrow \infty$, which results in $\mathbf{m}_i(t+1) \approx \mathbf{m}_i(t)$ for large values of t . The neighbourhood function can be used to control the “strictness” by which the reference vectors approximate the input data.

The next sections describe how we modified the classic neighbourhood function to achieve a global ordering that best represents the structure of the network.

2.3.1 The Neighbourhood Function

The SOM algorithm has many diverse applications, one of which consists of the reordering of nodes on a plane according to certain criteria. One such application reorders nodes, initially randomly located within an arbitrary volume, to approximate a certain shape. This is accomplished by training the nodes on a set of randomly generated input vectors with a probability density function resembling the desired shape resulting in the reordering of nodes randomly located within the shape.

In the SOM algorithm each random input sample is compared with all nodes and the response node is identified. This node and all its neighbours learn from the input sample by moving closer to the response node. The algorithm begins by defining the neighbouring nodes as the nodes that are connected, directly or indirectly, to the receptor node. A node’s neighbours are therefore the nodes that are reachable through the network from that node.

Applying the SOM algorithm to achieve a network layout with this definition of a neighbourhood function results in directly linked nodes residing closer to one another than indirectly linked nodes. Nodes that are indirectly linked from a response node learn in proportion to their distance (hop count) from the response node. The response node therefore always learns the most with its neighbours learning to a lesser degree and so on.

We therefore start the process by taking each node and calculating the hop counts to all of its neighbouring nodes. We calculate this before the main SOM algorithm starts by running Dijkstra’s algorithm on each node, with the link weights set to one. This yields the shortest paths and distances to each node’s neighbours which we use to set up the neighbourhood function. If a response node c ’s most distant neighbour hop count is denoted by D_c , define the neighbourhood function $h_{c,i}(t)$ for a neighbour i with a hop count $d_{c,i}$ as

$$h_{c,i}(t) = \frac{D_c - d_{c,i}}{D_c}.$$

Figure 2 illustrates a simple 12 node network with the neighbourhood function calculated for all of node 4’s neighbours. The resulting SOM is described by the equation

$$\mathbf{m}_i(t+1) = \mathbf{m}_i(t) + \alpha(t)h_{c,i}(t)(\mathbf{x}(t) - \mathbf{m}_i(t)), \quad \text{with } t \in \mathbb{N} \quad (2)$$

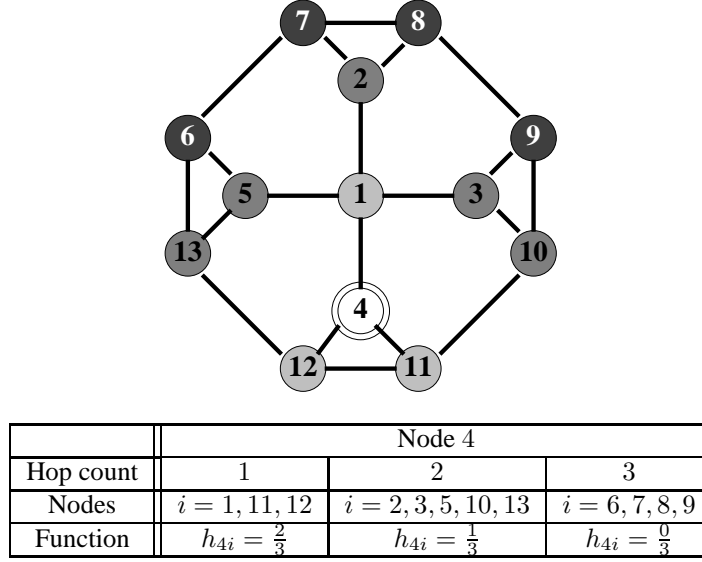


Figure 2: The neighbourhood function calculated for node 4.

2.3.2 The Alpha Function

Since $h_{c,i}(t) \rightarrow 0$ as $t \rightarrow \infty$ we introduce a monotone decreasing function $\alpha(t) \rightarrow 0$ as $t \rightarrow \infty$ to guarantee that convergence takes place. Observing the animation process of the layout with a simple linear alpha function allowed us to investigate the qualities an alpha function should have. We require a function that starts off with a “large” alpha value and quickly converges to zero after some 30% of the total number of iterations have been reached. We quantify a large alpha value by observing the layout process while displaying the current alpha value. We determined that the initial alpha value should be large enough to cause sufficient reordering of the initial node layout but small enough that this initial shuffling does not take up most of the iterations. We started with the function

$$\alpha(t) = 0.5 - \arctan(t)/\pi \quad (3)$$

and determined experimentally that a suitable initial value would be $\alpha(1) = 0.5$. Since $\alpha(1) = 0.25$ in Equation 3 we multiply by two and obtain

$$\alpha(t) = 1 - 2 \arctan(t)/\pi \quad (4)$$

which is shown in Figure 4.

Figure 3 shows that after a few of iterations the alpha value becomes small well before the 30% iteration mark, since the algorithm typically runs for thousands of iterations. The animation process shows that the layout converges after about 50 iterations. To achieve a network layout with the alpha function described in Equation 2, the SOM algorithm

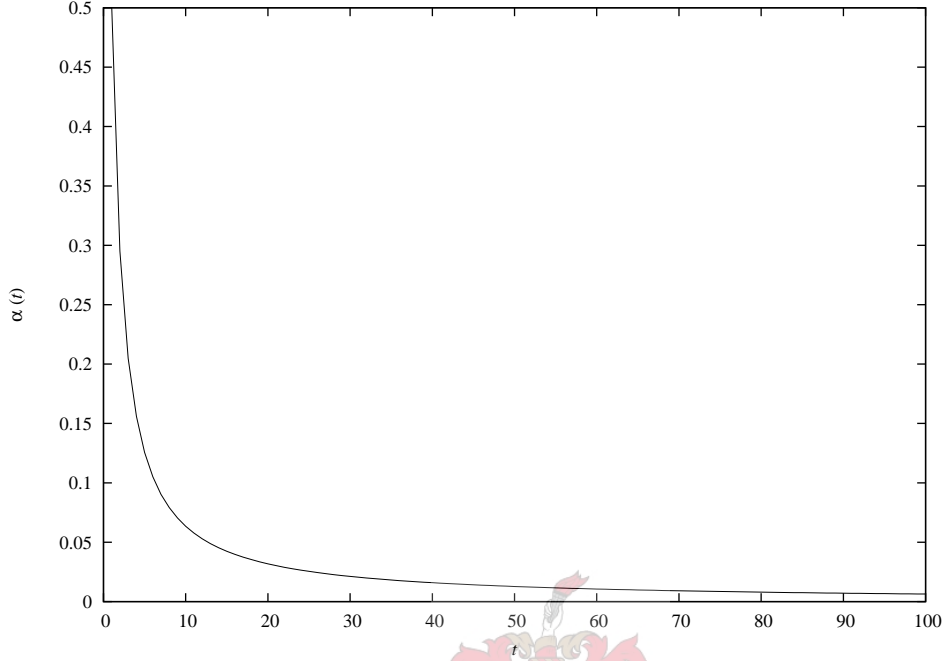


Figure 3: The function $\alpha(t) = 1 - 2 \arctan(t)/\pi$.

would have to run for many iterations as each iteration only induces a small learning effect. We therefor scale the x axis such that when the algorithm reaches its last iteration T , the corresponding alpha value is 0.5×10^{-1} . As stated above, this value has also been obtained experimentally as values lower than this causes little or no learning by the nodes. Since the original alpha function reaches this order of magnitude at some point $t = R$ (the actual point is $t = 15$) we scale the new alpha function to have the same shape as the original alpha function up to $t = R$, but with arbitrary range T . We accomplish this by scaling the iteration number t as a percentage of the total number of iterations T and using the result as input to the alpha function. The function

$$\sigma(t) = Rt/T$$

where R denotes the range scaled over is the function we used to scale the range to acquire the desired affect. The resulting graph shown in Figure 4 has the same form as the original graph up to $t = R$, but with arbitrary range 8000. The range scaled over R can then be adjusted to achieve better results for different sized networks. Choosing $R = 7$ is a good value for large graphs while smaller graphs do better with $R = (60, 120)$, although this is a matter of fine tuning the alpha function for better performance as any value between 7 and 90 can produce the desired effect.

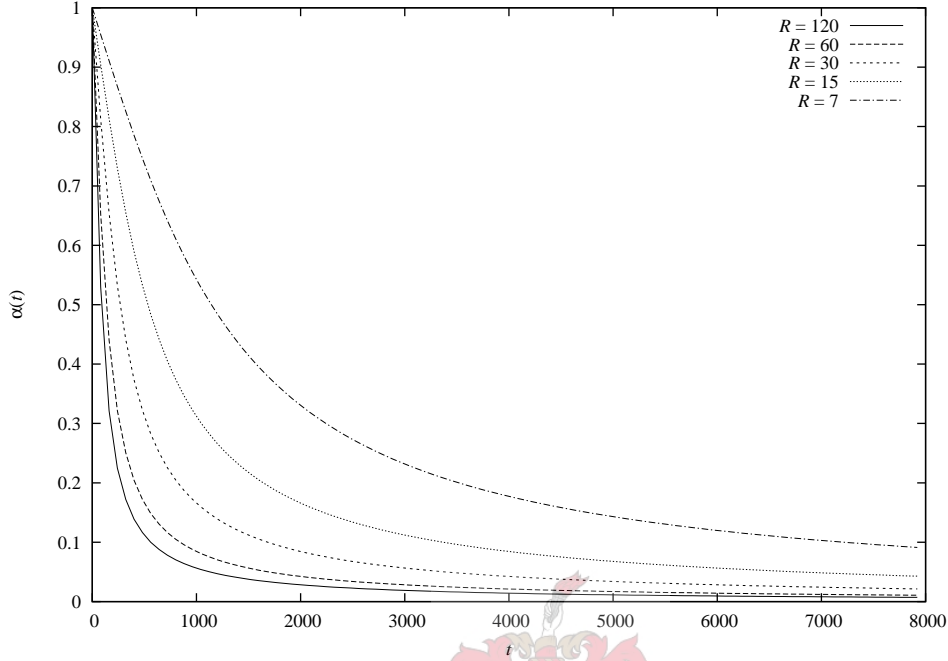


Figure 4: The function $\alpha(t) = 1 - 2 \arctan(\sigma(t))/\pi$ with $\sigma(t) = Rt/T$, $T = 8000$. Different graphs are show for different values of R , where smaller values of R are used for large networks and vice versa.

2.3.3 Refining the Neighbourhood Function with Observations

In the previous section we discussed how we used observations to fine tune the alpha function. We do this by animating the layout process from the first to the last iteration and observing the result. To test our algorithm we first created a network with a known layout, which we then shuffle and fed back to the SOM algorithm as input. This reference network consists of a network with nodes evenly distributed over a $n \times n \times n$ cube structure which is the optimum layout². This enables us to compare the layout algorithm's aesthetic abilities with the known network layout and also to test the performance of different variations of the neighbourhood function, as certain variations require more iterations to reach an aesthetic network layout.

Inserting the alpha function as computed in Equation 4 with the neighbourhood function $h_{c,i}(t)$, we observe a resulting layout that approximates the reference layout's structure to an acceptable degree. The degree of acceptability is determined visually as the reference layout is easily recognized. Although the SOM algorithm works well and achieves an aesthetic layout, it takes a considerable time to compute³. Although we tried to reduce the execution time by using advanced coding techniques, the algorithm proved to be inherently

²Refer to the cube network structure on the top right hand side of Figure 6 on page 18 for an display of the optimum network layout.

³Some 30 seconds is required for a network of 100 nodes and 300 links on a P4 2.8GHz processor.

slow. With the current definition of our neighbourhood function a network consisting of n nodes would train ⁴ $n - 1$ neighbouring nodes on each random input sample \mathbf{x} .

The literature suggests [17] decreasing the radius of the neighbourhood function over time, but we found this to have little effect as initially the algorithm is still slow. This led us to recognize only *directly* connected nodes as neighbours. We thereby improved the algorithm by implementing a trivial neighbourhood function and at the same time substantially⁵ reduced the execution time of the SOM algorithm since only the direct neighbours of the response node learn from each random input sample. With reference to Figure 2, if node 4 generates a response then only the neighbours 1, 11 and 12 learn from it, instead of all the nodes directly or indirectly connected to node 4. Let $N_c = N_c(t)$ denote the set of neighbours of node c at time t . Define the new neighbourhood function to be $h_{ci} = 1$ if $i \in N_c$ and $h_{ci} = 0$ if $i \notin N_c$.

This neighbourhood function requires slightly more iterations to achieve the correct network layout, but since each iteration takes much less time to compute the total execution time is much less. It is also not necessary to calculate Dijkstra's algorithm on each node at the beginning of the algorithm which saves execution time for large networks.

2.3.4 Implementing a Self Organizing Map, a Performance Analysis

In the following algorithm let $\mathbf{x} \in \mathbb{R}^3$ and $\mathbf{m}_i \in \mathbb{R}^3$. Let I denote the number of nodes in the network. The algorithm runs for T iterations where it is observed that $T \sim 750I$ is sufficient for convergence.

Algorithm 1 SOM

1. Initialize $m_i \quad \forall i \in \{1 \dots I\}$
 2. **for** $t = 0 \dots T - 1$ **do**
 3. $\mathbf{x} \leftarrow x(t)$
 4. $c \leftarrow \min_i (\|\mathbf{x} - \mathbf{m}_i\|) \quad \forall i \in \{1 \dots I\}$
 5. $\alpha \leftarrow \alpha(t)$
 5. **for** $i = 1 \dots I$ **do**
 6. **if** $i \in N_c(t)$ **OR** $i = c$ **then**
 7. $m_i(t + 1) = m_i(t) + h_{ci}(\mathbf{x} - m_c(t))$
 8. **end**
 9. **end**
 10. **end**
-

The algorithm has time complexity $O(I^2)$ because of the lookups necessary on line 4. A naïve implementation will cause the algorithm to execute inefficiently. We implemented a 3D hash function and utilized Intel Pentium 4's SSE2 [18] instruction set to speed up the best match on line 4 as well as lookup tables for the functions $\alpha(t)$ ⁶ and $N_c(t)$, reducing

⁴Refer to Figure 1 on page 11.

⁵The amount depends on the connectivity of the network.

⁶C++'s arc tan function executes slowly because of the high numerical accuracy it maintains.

the execution time to $O(I \log I)$.

Our test system running a Pentium 4 2.8 *GHz* with 512 *MB* DDR 400 memory running in dual channel mode was used to measure the execution time of the SOM algorithm. Our test network consists of the network described in section 2.3.3, with the dimensions of the network increased in increments of one from a $2 \times 2 \times 2$ network to a $15 \times 15 \times 15$ network for each sample run. The time t taken for the SOM algorithm to produce each desired reference network consisting of n nodes is shown in Figure 5.

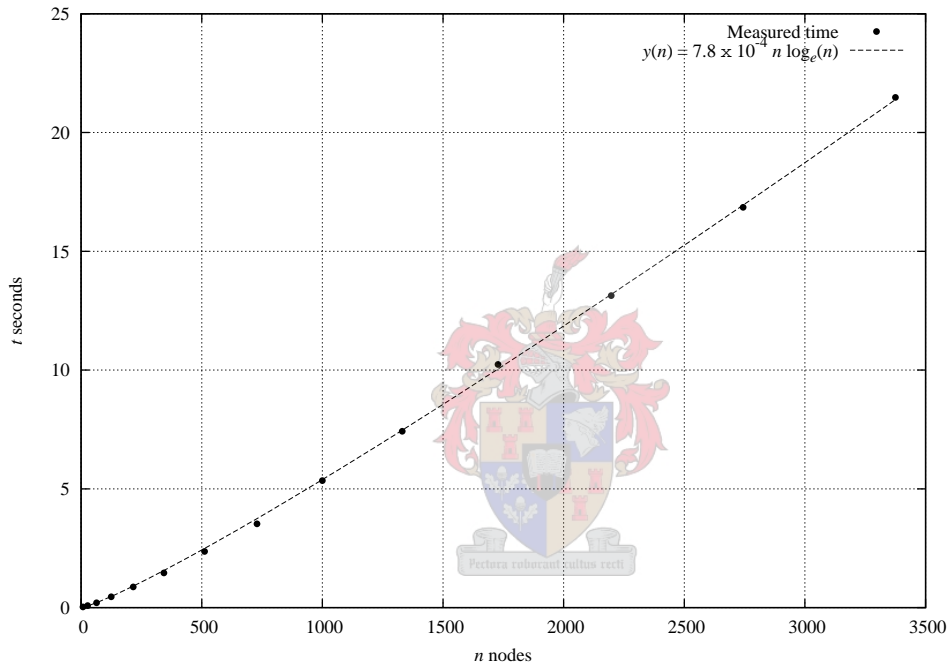


Figure 5: The time t in seconds to achieve an aesthetic layout with the SOM algorithm for a network consisting of n nodes.

The points on the graph show the time taken for the SOM algorithm to complete each network layout using $750 \times n$ iterations. The data can be approximately described by the function

$$y(n) = 7.8 \times 10^{-4} n \log(n).$$

The long execution times usually associated with neural network implementations have therefore been avoided by providing a simple but effective implementation that is optimized to provide a scalable solution to the graph layout problem.

2.3.5 Customizing Network Layouts with the SOM Algorithm

It has been shown that a the SOM algorithm can be used to create layouts for different sized networks without having any user input. The SOM algorithm does however provide

a different way of customizing a network layout. If a user has an idea of what the network might look like, he can “instruct” the SOM algorithm to approximate this shape, which can be 1, 2 or 3-dimensional taking on any size or volume. It is in this way that we set up our test network described in section 2.3.3. We generated a network topology that is known to have an optimum layout in the shape of an cube. We then configure the SOM algorithm to generate the random input vectors within the interior of a cube, thereby forcing the network to approximate the shape of a cube with nodes evenly distributed therein.

Changing the manner in which the input samples are generated has a marked effect on the resulting layout. A set of nodes can be ordered differently according to the different probability densities of the random input samples. This makes the SOM algorithm very flexible. Figure 6 illustrates the effect of random input samples with different probability densities on the final network layout. In the first case the random input variables were generated within the interior of a cube which results in the network layout approximating a cube. The second network was created by generating random vectors within the interior of a sphere. In principal the same network structure can be made to approximate any area or volume with non-homogeneous densities, by generating random input samples from the appropriate probability distributions.

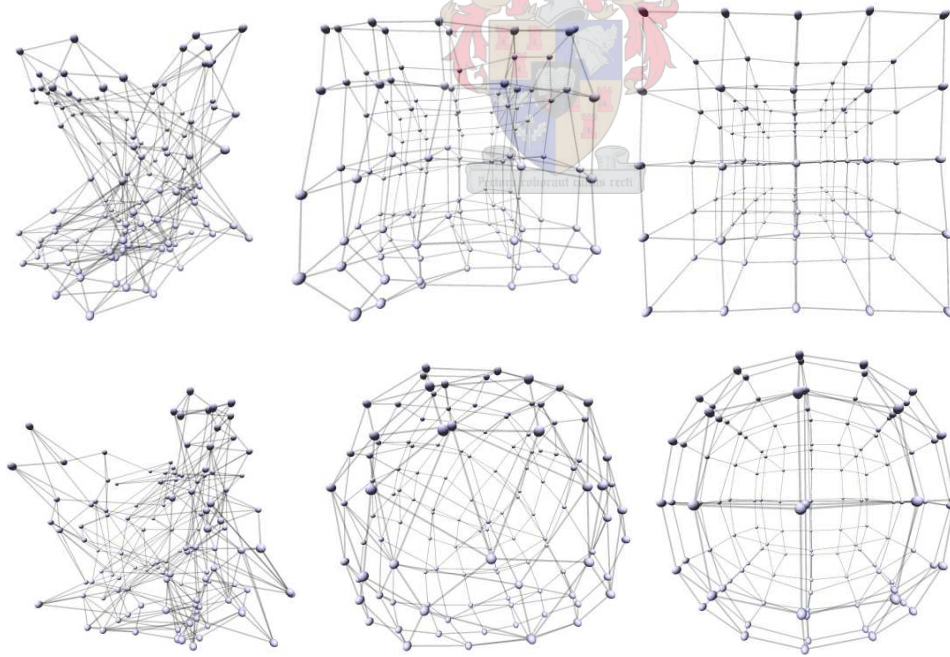


Figure 6: The effect of different probability density functions on the resulting layouts for the same network structure.

Chapter 3

Program Design

3.1 The Design Paradigm

Although there exists no general rule as to what constitutes as a proper program design, we discuss the design for building a network visualization application that worked for us. Before designing a program one has to determine which language to use for the implementation. This is important since although different languages can accomplish the same task, some languages are more suitable for to certain types of programs than others. Since we are developing a performance application our choices are limited to the Delphi and C++ programming languages. We chose C++ on a Linux platform as our implementation language as our expertise lies mostly therein.

Before choosing an implementation language one needs to be fully acquainted with the functionality of such a language and in particular its *class* operation. Designing a class inheritance model without fully understanding a language's capabilities is bound to lead to design problems later on [19]. Although designing a program before it is developed is the best way to start, it does not always lead to the desired results. When designing a program, the future functionality of the program is never as clear as its immediate application, which can lead to a premature design compromising the program's extensibility later on. A solution is to be flexible about the entire design paradigm. There exists no right or wrong design for any particular program and a design must be allowed to evolve to incorporate the better understanding gained from working with the program and language over time.

The TCP/IP network protocol has been deeply embedded into today's networks. Designing a network visualization application solely for the purpose of visualizing such protocols has its disadvantages. Writing a visualizer to visualize only one protocol would compromise its ability to keep up to date with networking technology. It is our goal to prevent this from happening. Since it would be a daunting task to write code specially tailored for

each possible network protocol, our design philosophy is based on a design that can be extended to visualize any protocol with minimal extra effort. We must therefore identify the commonality between many network protocols, for instance the fact that they all use nodes, links, packets and routes for their operation, and use these objects as building blocks to visualize different network protocols. We therefore set out to develop a visualizer that visualizes these different network components (nodes, links and packets) according to certain inputs. One can argue that this is an obvious way to develop a network visualization tool, but scalability issues can be solved by developing code that is designed to capture the essence of a protocol and thereby automating protocol behaviour to such an extent that very little input is required. We stated in Chapter 1 that our goal is to visualize a MPLS network, which does not mean the program is designed to do only that. On the contrary, we typically create more complex generic classes (nodes, links and packets) and extend them into lightweight MPLS versions to provide for MPLS protocol visualization. The following sections describe how we used a object oriented language to accomplish this.

3.2 An Object Oriented Approach Using C++

Object orientation is a way of producing simple understandable code that can be easily understood and extended. Programs developed with an object oriented code design produce the same results as programs developed in a non object oriented environment, but with certain advantages and disadvantages. Performance losses are incurred due to data hiding, also known as data encapsulation, which is commonly encountered with developing object oriented code. Since we are writing a graphics application, certain decisions have to be made relating to the degree of strictness in which we will adhere to data encapsulation. If we keep too strictly to data encapsulating practices, all data exchange between classes has to be mediated through a member function call. Sometimes the performance overhead caused by these member function calls does not justify the small amount information retrieved by them, especially when in our case they are called thousands of times per second. On the other hand, if all class variables are publicly accessible, the debugging of such code can become troublesome and would negate the advantages gained from data hiding. We attempt to strike a balance between the two models by encapsulating all class variables by default and unhiding them when it is clear that retrieval of such a class variable is frequent enough to induce an unacceptable performance penalty. These performance threatening variables do not occur very often, hence we feel that it is a good practice to follow when designing an application with potential scalability problems. Figure 7 presents our class structure in its final form as evolved over time and we discuss its meaning.

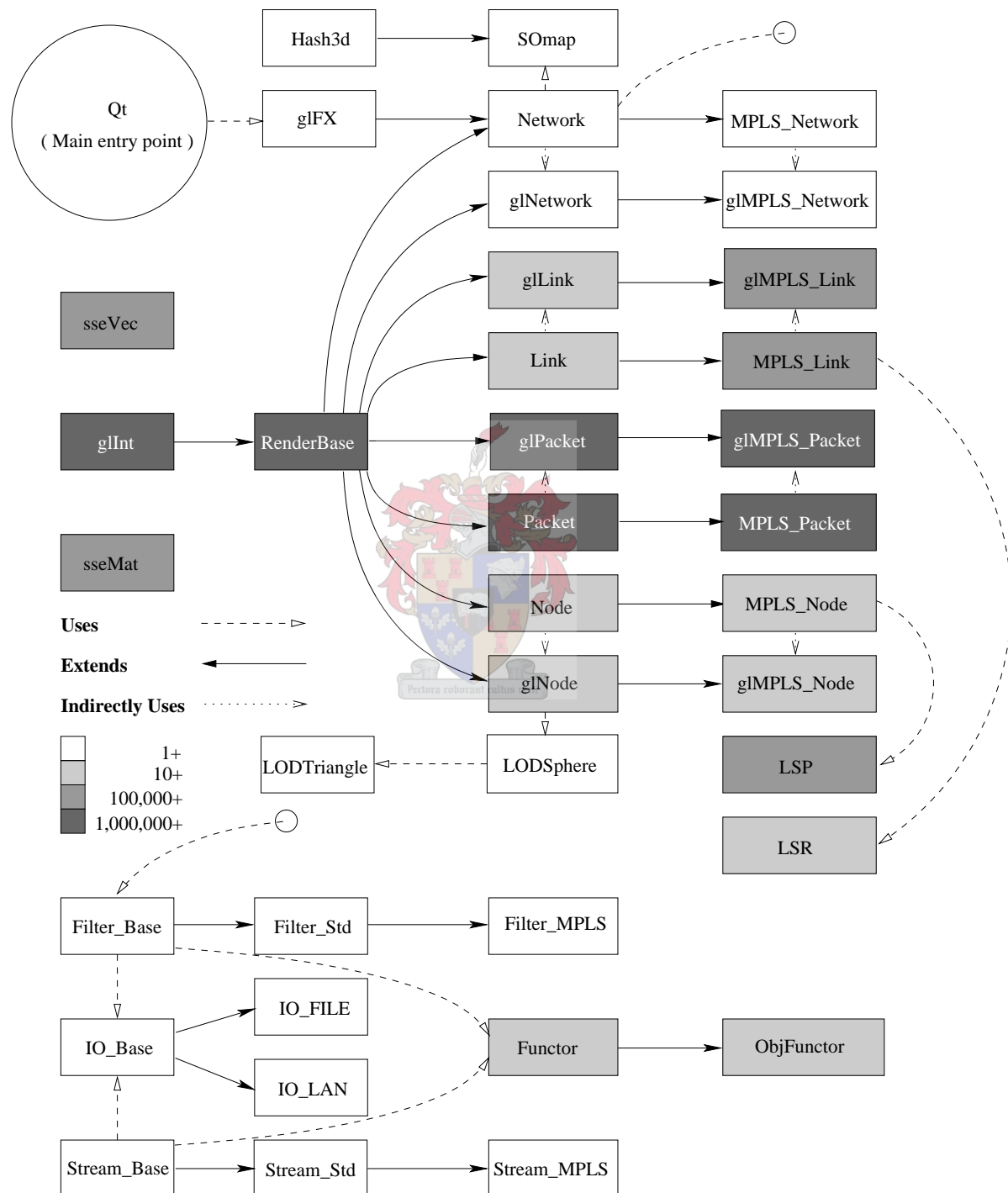


Figure 7: A suggested optimum class structure of our network visualization program.

3.2.1 The Class Diagram Key

Before we discuss the individual classes we first explain how to interpret the class diagram. We used our own class diagram format which has the same look and feel as standard class representation models, such as UML[20], but with a few modifications. We first explain the diagram key located in the middle left hand side of the class diagram. The different arrows indicate how to interpret the relation between different classes. The relation is revealed by reading the class name on the left hand side of the arrow followed by the the arrow translation, followed by the class name on the right hand side. We next explain the individual meanings of the different translations by the arrows:

- The **Uses** relation: This relation indicates that the left hand class relies on functionality obtained from the right hand class to perform its duty. The left hand class typically contains a pointer to the right hand class, which it then uses to make member function calls through. The left hand class can also have a container of pointers pointing to many instances of the right hand side class. The different shades of grey, as seen in the key, provides an indication as to which relation is used.
- The **Extends** relation: This relation is the more familiar of the three relation types. The class that is pointed to by the arrow extends the class that is not pointed to by the arrow. This relation represents C++'s derived class concept, whereby an extended class consists of the functionality of the base class, but with extra functionality added to it.
- The **Indirectly Uses** relation: This relation indicates two related classes that are not related in the same way as classes with the **Uses** relation, but that are dependent on each other. We added this relation to clear up the confusion caused by classes using other classes through a parent interface class. Note that no two classes can have this of type relation if their parent classes do not have a **Uses** relationship between them. This dependency relationship lies at the core of the design.

The other piece of information we added to the diagram is the colour key. This key gives an estimate of how many instantiations a certain class is expected to have in a typical execution of the program. We find this information vital to understanding the roles different classes play in a program. We have four different instance ranges which represent educated guesses of the typical instance count expected. Note that a class that is only instantiated once can be expected to have a fat interface consisting of rich functionality, while classes that are instantiated many times can be expected to be small and optimized with little interface functionality. The instance count therefore gives a good indication of the role played by certain classes inside a program.

3.2.2 The Proposed Class Design Model

We next discuss how we obtained our design model and how we interpret it in terms of visualizing network protocols. As mentioned before, our design is based on a code reusability model where we build a set of tools and extend them to different specializations that represent visualizations of different protocols. To do so we require a model that allows a class to be extended many times with the least amount of code duplication. We discuss the common pitfalls encountered when seeking such a design and our solution to this problem. Second, we show how we use this solution to provide the optional ability to visualizing highly customized network systems. We now give an overview of a few selected classes that contributed to the core design.

RenderBase

Although the **RenderBase** class is one of the smallest classes, it is conceptually one of the most important. This class alone allows us to provide a code reuse environment that is fundamental to our design goals. Since we did not find this design by accident we give the background as to how we chose this design model.

To understand why this model works we first explain why other, more widely practiced models, do not work. Consider the class **Network** in Figure 7. Such a class would typically contain all the functionality associated with building, storing and managing a network structure inside a program. Member functions such as `InsertNode()` and `InsertLink()` are typical functions associated with such a class. Adding code that renders the network represented by the **Network** class, a typical approach would be to extend **Network** into a graphical object class which we call **glNetwork** (this is not the class structure depicted in Figure 7, but merely an example). In doing so **glNetwork** inherits all the necessary data, the link and node lists etc., to visualize the network. Although this rendering code can be placed inside **Network**, it is not desirable to have a mixture of code implementing network management with code implementing the rendering path. Having them apart saves compile time and eases debugging considerably.

So far the implementation is clean, effective and can be viewed as a suitable design. However, when extending the classes to the next level certain problems arise. Say for example we desire a specialization of the **Network** class, a label switched path network called **MPLS_Network**, that includes the functionality contained within **Network** combined with the functionality required to manage a label switched path network. The current model proves to be unusable with such a configuration. When we extend **Network** to **MPLS_Network** there exists no way to reuse the code, written inside **glNetwork**, to visualize the network represented by the **MPLS_Network** class which is for the most part the same. The class **glMPLS_Network**, which renders the extra features represented by the **MPLS_Network** class, would then have to be rewritten and since it would contain most

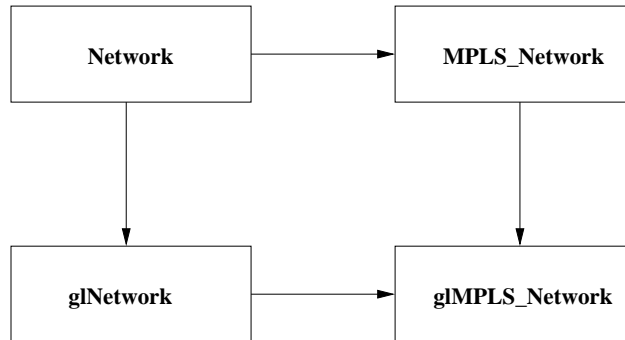


Figure 8: The circular inherency problem encountered with a simple network class model.

of the code written inside **glNetwork**, would violate the principle of code reusability. The circularity of this problem is shown in Figure 8.

To solve the problem we have no choice but to break the circular dependency. We choose to break the derivation from the network objects to their respective graphic counterparts, which causes two adverse affects. Firstly, since we broke the inheritance between the graphical objects and the network objects, the graphical objects lose access to the data structures necessary to visualize the network. The second problem is similar to the first, but in reverse. Network classes now need pointers to instances of their respectable graphics classes in order to render themselves. We need to find a way to ease the development of such classes because extending these base classes would require the user to know exactly which functions to overload.

We solve this problem by introducing a simple and lightweight¹ interface class called **RenderBase**. **RenderBase** contains one variable, a pointer `m_pRenderer` to a **Render-Base** interface *implementation*. It also contains a set of member functions (these denote the rendering interface) commonly associated with objects that are graphically represented, such as the `Render()` member function call. All the member functions inside the interface are *virtual* and have only one purpose, to redirect a requested interface function call to the *instance* pointed to by `m_pRenderer`. To give an illustration we give a small code extract of the **RenderBase** class:

```

class RenderBase :glInt{
Protected:
    RenderBase* m_pRenderer;
Public:
    void RenderBase();
    void ~RenderBase();
    //....

```

¹A class that consumes very little or no memory that is normally associated with interface classes.


```

virtual void RenderBase::Render()
{
    m_pRenderer->Render();
}
//.....
};

```

Every network related² class is set to derive from the **RenderBase** interface. This allows some other class to call upon them to render through the standard interface. To give an example of how this works, refer to the class diagram in Figure 7 where the first rendering call for each frame originates from the window manager³, which is represented by the large circle. The window manager calls the **Network** object to render itself through the heavier interface class **glFX**, which is necessary to encapsulate the complexity of the window manager. The **Network** class then iterates through its lists of network component classes⁴ and calls them to render themselves. Now because the component classes extend the **RenderBase** interface, the **RenderBase** interface automatically intercepts and redirects these calls to the appropriate rendering object pointed to by the respective `m_pRenderer` pointers. These pointers are set to point to the appropriate rendering classes for each different network object, for example the **Node** class will set its `m_pRenderer` pointer to point to **glNode**, which implements the rendering interface. The **glNode** class therefore contains the appropriate overloaded member functions, defined as virtual in the interface, to render the **Node** object.

Media classes

The media classes combined with the the classes derived from the **RenderBase** interface include all the classes needed for extension to make a specialized visualization possible. Referring to the diagram in Figure 7, the media classes consist of the cluster of classes at the bottom of the page. We stress that it is not always necessary to extend classes to visualize every network, but we feel that it would be ill advised to create a model that does not allow for extension when they are definitely needed. Different models of a network are rarely the same and hence special modifications have to be made to visualize them.

We will explain some of the media classes in order to explain how the different classes fit together in visualizing a new protocol. We begin with the **Stream_Base** and **Filter_Base** classes. These two classes encode the data input and output of the system. We do not make use of trace files as many other visualizers do when acquiring data, because of their encumbering size. **Stream_Base** is the class that provides the functionality of streaming

²Such as nodes, links and packets for example.

³We use a windows manager called Qt in Linux.

⁴Such as the nodes, links and packets.

data from the source to the visualizer through a local area network or a file. **Filter_Base** reads this data and presents it to network classes for interpretation.

Converging Media, Network and Graphical classes

We conclude this chapter by explaining how we extend existing classes to provide specialized visualizations. To repeat, our design defines a set of classes to create a basic network visualization which are then extended to form a more complex visualization if needed. It is not necessary to extend the model for every new visualization, but if an existing implementation lacks the means to handle certain unique circumstances, such extensions are unavoidable. We create the opportunity to augment the program directly instead of writing a complex⁵ scripting system ourself that would not be as well defined and functional as the C++ language.

We create a specialized visualization in three steps. First we extend the class **Stream_Base**, or any of the classes derived from it containing functionality suitable for reuse. This new class is used to stream customized data to the filter classes. We therefore extend the **Filter_Base** class to create an extended class capable of interpreting such a data stream. This step allows us to adapt a standard data stream to contain one or more extra network events that are normally not associated with a standard⁶ network. We add the member functions necessary to stream such events, through the use of a derived class. Since the corresponding filter is now incapable of interpreting these extra events inside the data stream, it is extended to read in these events. The extended filter class is therefore tasked with interpreting the extra events and passing these events to the network core classes.

It is clear that the network classes must be adapted to handle the new data sent to it by the filter class. We proceed to extend the network classes to manage this data so that it can be visualized. It should be clear that the media classes only handle the transport and delivery of data streams to the network classes. It is at the delivery part where the extended network classes are needed. If an extended network class does not exist then the filter class extracts the extra events from the data stream and discards them. In practice this would never happen but it demonstrates the granularity of the system. Different components can be put into place and connected to augment the standard visualization. This eases the development phase and helps with version control.

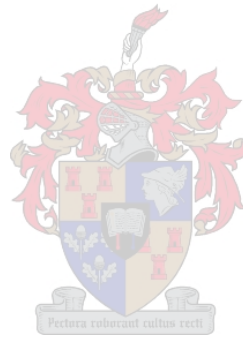
Once the data is inside the network classes the graphical classes can be extended. These extensions are optional as the extra event data given to the network classes might effect already visualized network phenomena. The optionality is achieved by the extended network classes setting their renderer pointers to point at graphical classes that already exist. This is the most complex part of the procedure since most developers know the C++ programming language but few have experience in coding complex graphical routines in them.

⁵Which involves defining and interpreting a scripting language.

⁶The standard network visualizer provides for common network events.

There are no shortcuts around this problem. We have implemented many helper functions to aid graphical development, but one needs to understand the graphics API OpenGL [21] to be able to make graphical contributions. The advantage however is that most network visualization packages have no support for such customizations.

This process is illustrate in Figure 9. The diagram shows the classes written for visualizing a fictitious network “My Network”. Although it appears as if various classes need extending, the extended classes are mostly empty and contain only a few extra functions since they make heavy use of their parent classes’ functionality.



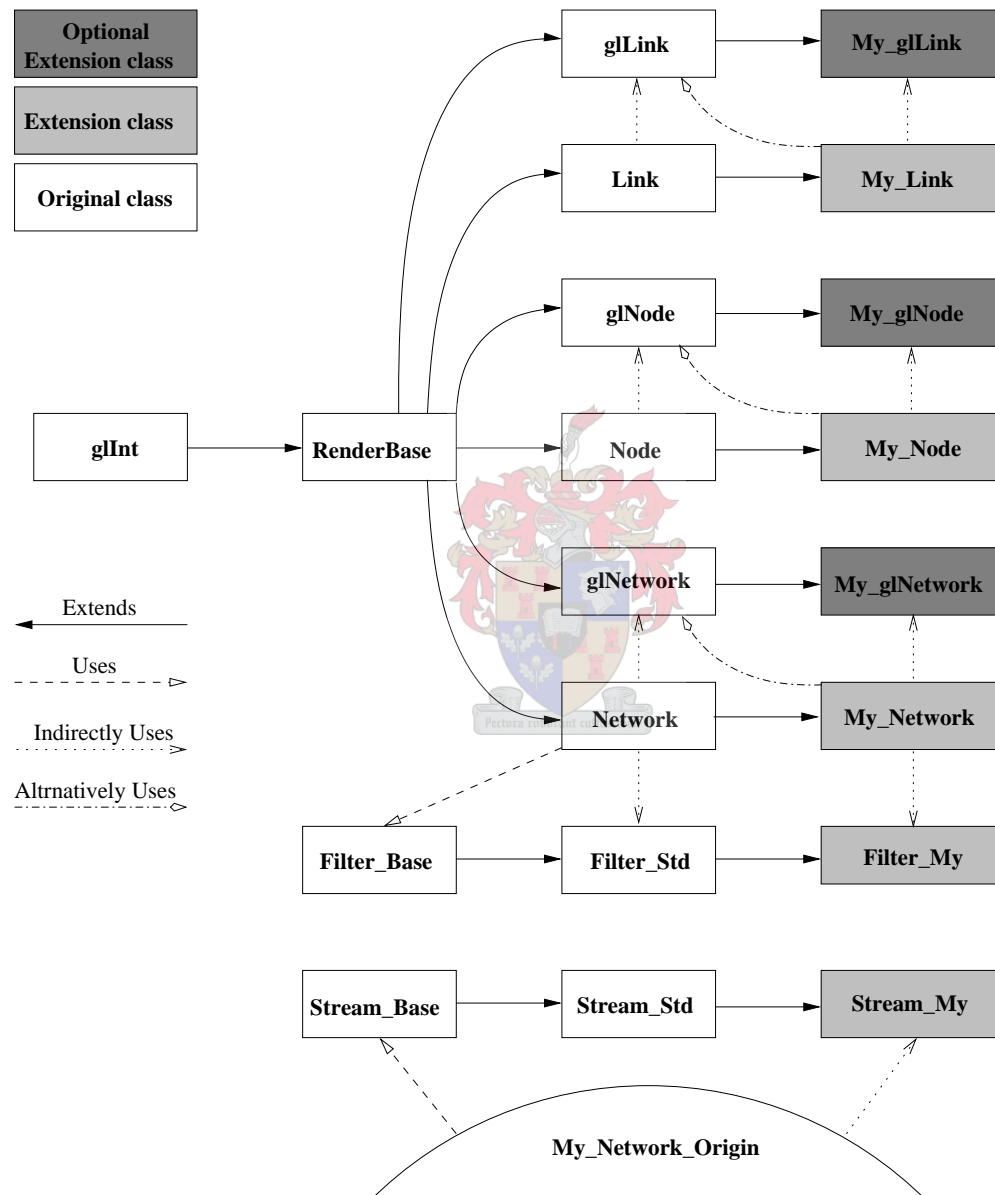


Figure 9: The diagram indicates which classes are suitable for extension when writing a customized visualization for network data originating from a unique source.

Chapter 4

Visualizing Network Data

Having an overall view of the network's physical structure is helpful in identifying bandwidth allocation and bottlenecks within a simulation. However, having snapshots of the entire state of a network at different time instants can be even more helpful. This requires the simulator to send detailed information to the visualizer in order for the visualizer to display these snapshots. One way of achieving an animation is to make the simulator send the event information with an accompanying *event time*. The visualizer then animates the network on a frame by frame basis, adding new visualizations at the appropriate event times. This is a simple process which can become complicated with regards to visualizing different phenomena inside a network. If, on average, event times are far apart, then simple implementations can be used to synchronize the event times with the visualized effects. With regards to packet visualization though, difficult decisions have to be made about the granularity of different event times and in which format these event times are transmitted to the visualizer. As we will see later, these decisions have a significant effect on the numerical stability and scalability of the system. Since there can be only one time system within the visualizer, the optimum implementation needs to be found.

It would be ideal if the visualizer could run in the same execution thread as the simulator. This would enable the simulator to make direct calls to the visualizer which is the most efficient way of transferring event information to the visualizer. In practice this is not possible, because of insufficient computational power to execute both processes at the same time. We therefore construct a second network structure between the simulation process and the visualization process, with the first network structure residing inside the simulator. In this way the second network, and therefore the visualization, is driven by a stream of events from the simulator instead of by the computationally intensive algorithms that drive the simulation process. The second network serves as an abstraction layer between the simulation and the visualization of a network. A simulated network now only needs to

communicate with the cloned network instead of the graphics process directly. The graphical process in turn uses the second network's state information to visualize the network on screen at hundreds of frames per second, completing the visualization process.

This approach has several advantages. Not only does it take most of the complexities away from the simulator, which results in improved user friendliness, but also creates a visualization platform that can be automated and equipped with features. For example, the simulator does not need to send millions of packet events to the visualizer, but rather sends one packet event that indicates a packet departure on a certain Origin Destination (OD) pair. The consequences of this event, for example rendering the packet flow on a certain route, is then handled by the second network process. Other examples include the colouring of nodes or links according to some value transmitted through an event by the simulator, not being necessary anymore. This does however require a sophisticated network process running on the visualization side.

In Chapter 3 we discussed how we extended the application to visualize different types of networks. We extended the standard network classes to more specialized versions to achieve custom visualizations where needed. In this chapter we discuss the visualization of these standard network classes. The idea is to write visualizations for as many standard networks as possible, thereby simplifying the process of making custom network visualizations. We implemented node, link, packet and LSP visualizations as our standard set of network graphical components and discuss them in the following sections.



4.1 Representing Network Nodes

The nodes inside the visualization can have many different interpretations depending on the sort of network being visualized. Since the network data can originate from many different types of simulators and even real networks, nodes can either represent countries, autonomous systems, routers or IP switches. If more than one type of node exists inside a network then we need to visualize these with different shapes. The standard shapes currently supported consist of a sphere, a cube and a tetrahedron. If more shapes are needed they can easily be added by extending the **Shape** interface class. Shapes can be used to visualize different node entities by altering their size, colour and their rotational properties. Although these parameters can be customized to visualize *any* aspect concerning nodes, we propose a set of intuitive meanings and implement them as a default.

- The **type** of graphical object used to visualize a node (also known as a glyph) might be used to indicate different types of nodes inside the network. Setting this parameter depends upon the user's interpretation of the different types of nodes existing inside the network. By default a sphere is used to visualize a node, but other types can be defined and implemented.

- Different **sized** nodes are used to indicate nodes carrying different loads. This load parameter can represent the total aggregate flow going through a node, or the number of label switched paths (LSPs) going through a node, depending on which type of network is visualized. The size parameter is implemented as a dynamic parameter which means that as the values change while the network is being animated, a change in size is observed.
- **Colouring** a node can be used to visualize the current utilization of a node. While the size indicates the total bandwidth available, the colour visualizes the current utilization of that bandwidth. When visualizing an MPLS network we change the interpretation of a node and call it a label switched router or LSR. Since there is no notion of a total amount of LSPs that are allowed to flow through a LSR, or the fact that if such a limit existed would it ever be reached, we choose to visualize the “utilization” of a LSR as the number of LSPs flowing through it. We do this by finding the LSR with the most LSPs flowing through it and then colouring all other LSRs relative to that LSR. A red colour indicates the LSRs with the most LSPs flowing through them and a green colour indicates few LSPs flowing through a LSR.
- A node may be rendered so that it **spins** around its own center of gravity. Although a spinning node has no intuitive meaning, spinning a node at certain speeds might be used to attract attention to that node. A node can be configured to have some dampening force working against some spinning force. Such a spinning force might be equal to the rate of change of some node parameter. To give an example, if a LSR’s number of LSPs flowing through it changes rapidly, a LSR node might be made to spin to attract the users attention. Other uses can also be implemented.

Figure 10 illustrates node visualization inside a network. For clarification, this example visualizes nodes that are sized according to a load factor. Since our test network is not driven by a simulation process, all of our examples are generated by a simple artificial network event generator. We use this artificial network instead of a simulated network because it allows us to control the visualization to illustrate ideas.

4.1.1 Providing Network Engineering Tools for Simulator Testing

One of the goals of this thesis is to provide an environment that allows a network engineer to query certain phenomena inside a network. To use an analogy, in the same way as computer programmers use integrated development environments (IDEs) to aid in debugging source code, we provide a debugging environment for network engineers. This functionality can be useful in finding errors and algorithm flaws inside network simulators. To give an example, we implemented an extra visualization feature concerning nodes that can aid a network engineer in “debugging” the simulation process. This feature consists of an All

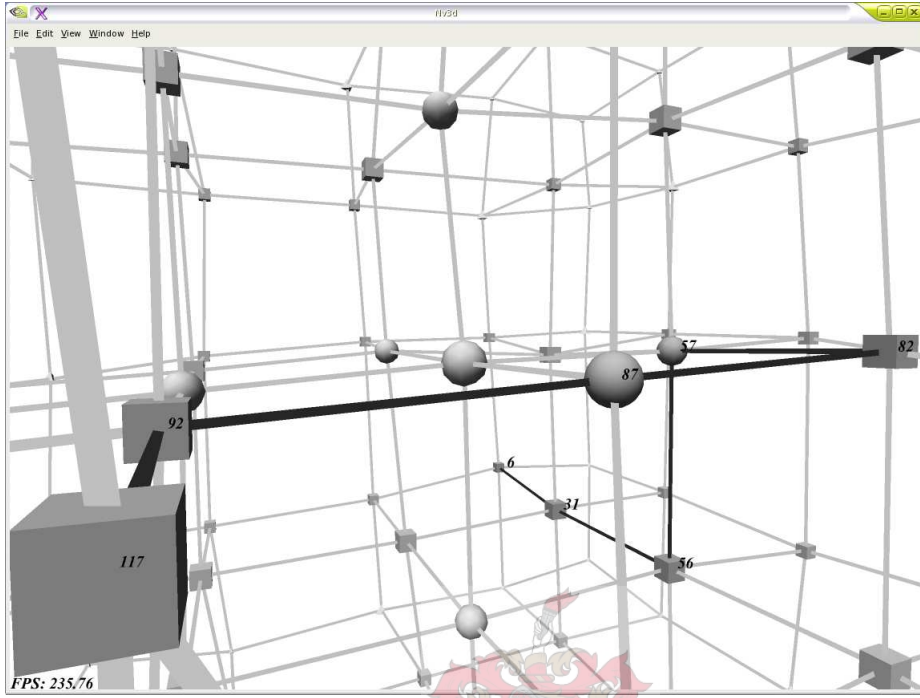


Figure 10: Visualizing different types of nodes inside a network.

Pair Shortest Path algorithm that computes and highlights nodes and links traversed along the shortest path between any OD pair. A possible use might be as follows.

Firstly, upon closer examination of Figure 10 one observes the shortest path between node 117 and node 6. The path is depicted by the darker coloured links along the traversal of nodes 117, 92, 87, 82, 57, 56, 31 and 6. To acquire this shortest path the user selects two nodes and request the shortest path between them. An engineer can then set the visualizer to only draw packets flowing between this OD pair and observe the optimality of the flow directly, as indicated by the links, and hypothesize why a certain flow, flows as it does. The shortest path visualization can also be updated every t simulation seconds to indicate changes in the route of the path over time, if so desired. It is important to realize that these functions can be executed in real time which means that the simulator need not send specialized event information to prompt the visualization of a shortest path between a certain OD pair, or even the set of LSPs that exist between a OD pair, at a certain point in time. Furthermore, a shortest path at time t can be computed by using the residual bandwidth at time t , or any other desired metric, as link weights. This way a realistic shortest path at time t can be obtained without the help of the simulator.

Although we implemented only one such feature, many other engineering tools can be trivially implemented, but because it does not add anything new to the concept we leave such ideas for further studies.

4.2 Link Visualization

The visualization of links is slightly more complex than the visualization of nodes. First, having a non-optimal link visualization system would compromise the interactivity of the visualizer which makes it difficult to navigate the network. Second, it must be kept in mind that the visualized links will later serve as packet carriers. Links have to be rendered in such a way that visualizing packets on them creates the least amount of confusion.

4.2.1 Selecting an Appropriate Graphical Object to Represent a Link

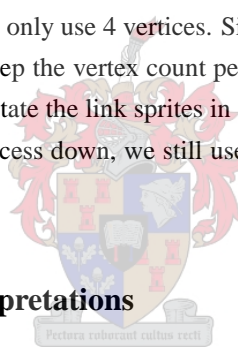
Cichlid [10] provides a network visualization mode called “Vertex-Edge Graphs” which attempts to give real time packet visualization with the use of OpenGL. Nodes are drawn with edges connecting them with packets flowing over the edges. When the Cichlid visualization is run with the demo provided, the shortcomings become immediately apparent. The visualization slows down to an unacceptable degree. This makes navigation impossible and confusing packet flows are displayed that appear to move backwards in time. We discuss some of the reasons why such shortcomings are encountered and the solutions for overcoming the shortcomings.

First, computer graphics hardware was not available to justify the creation of a packet animation system of that scale. The first GPUs were only released in August 1999 which means the earliest implementations could only have come in the year 2000. Even then the GPUs were not nearly as powerful as those of today. Second, even if the hardware was available at the time, the software drivers for Linux were not. They only became available from April 2000. All these factors mean that at the time of implementation only old techniques were used to achieve visualization. The fact that this type of visualization did not scale at the time meant devoting resources elsewhere to find appropriate ways of visualizing network data.

Other lessons to be learned from Cichlid concern the kind of glyphs they used to visualize the network. Of particular interest to us are the link visualizations. Cichlid used 3 dimensional cylinders to visualize links between nodes. Although they look pretty they are an impractical surface to visualize network packets on. Cichlid opted to use small cubes flowing in the path of a link to represent packets. This is not a good idea if more complicated and realistic packet visualization is to be attempted. Large packets would obscure the links they are flowing on. Furthermore, having to translate, rotate and scale thousands of cubes would require a huge amount of processing power, which at the time was provided solely by the CPU. Each packet visualized requires a $[4 \times 4][4 \times 1]$ matrix-vector multiplication for each of the 4 vertices of the cube. Even on today's GPU hardware this is a expensive operation if done thousands of times per frame. Bidirectional flows of packets would also create confusion as they flow on top and over each other.

The other more familiar packet animator Nam [8] uses a better system. The links are visualized in 2 dimensions by rectangles with bidirectional packets flowing on either side of the rectangles. In this way the packets do not obscure each other and create no confusion about where a packet is headed. We use a similar system, but in 3 dimensions. Because of the third dimension certain issues arise. Since we want to use a flat surface to represent links, viewing problems arise when navigating around such a flat surface in 3 dimensions. Extending the rectangle to a rectangular prism would solve the problem of a rectangle appearing flat from certain angles, but causes the same problems that result from using cylinders. Using 3 dimensional shapes to represent links requires complex 3 dimensional packets to flow inside them which is too expensive to render because of the numbers of packets to be rendered.

We therefore use rectangles but we rotate them in such a way that they are always perpendicular to the viewing angle. A flat surface rotated to always face the viewer is known as a sprite, which is a widely used technique in 3D graphics. This way the links only use 4 vertices but more importantly, packets only use 4 vertices. Since there will be thousands of packets flowing, it is necessary to keep the vertex count per packet as low as possible. Although the calculations required to rotate the link sprites in the right direction are fairly complex which slows the rendering process down, we still use them because of the gains made by the resulting packet model.



4.2.2 Link Visualization Interpretations

The visualized links symbolize the physical connections between two network nodes. Although it would be possible to draw “virtual” links between OD pairs, it would not be practical. There are many more virtual links in a network than there are physical links and rendering them all would clutter the display area to such an extent that nothing would be gained from it.

As with nodes, the link visualization can be altered to convey state information. Links can be altered in colour and in size and we give possible intuitive meanings for each.

- The **colour** of a link can be controlled by many factors. A utilization indication would be the first obvious choice. Since links may be bidirectional, two independent colours are needed on each link to visualize incoming and outgoing bandwidth utilization. We do so by splitting the link in half, with each side representing incoming bandwidth utilization. The entire link can also be coloured according to its overall average utilization or some parameter based on the number of LSPs flowing through it. Many possibilities exist and are easily implemented. The advantage of this system is that the colours change in time according to the current network state and require no information from the simulator’s side.

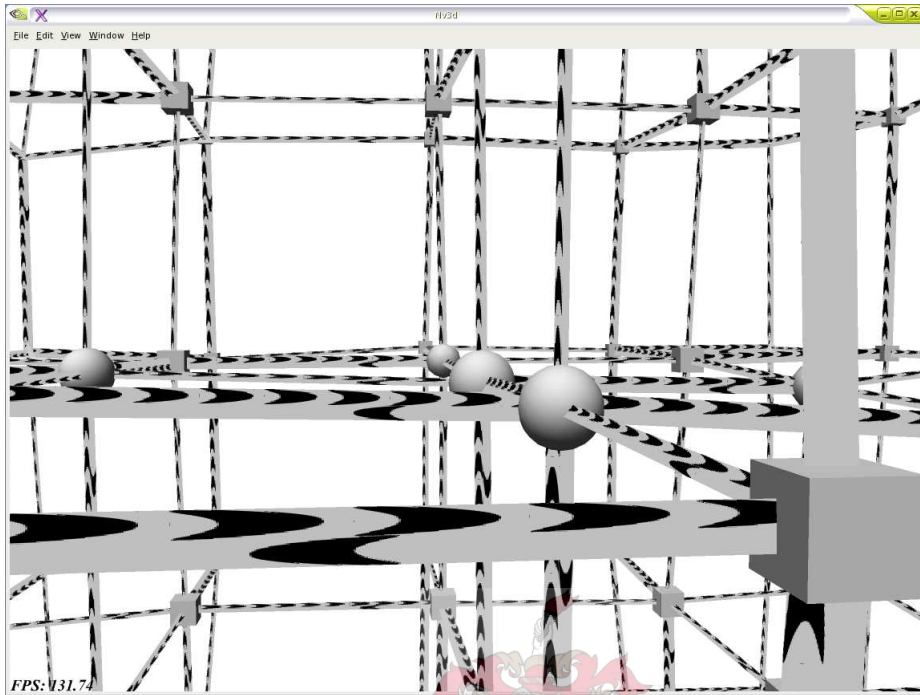


Figure 11: Visualizing network links with different bandwidths indicated by links widths.

- The **width** of a link is adjusted according to the bandwidth it carries. Other parameters might also be made to influence the width of a link but a bandwidth indication is an intuitive one. Having a bidirectional link with different bandwidths is handled by adjusting the width of the link ends independently.

Figure 11 illustrates the link sprites used to represent network links. Notice how different links are rotated to appear as if they were 3 dimensional objects. The mechanisms in place to govern these packets are explained in the next Section.

4.3 Rendering Packets

No network visualization tool is complete without some form of packet visualization. The most widely known network visualization tool that achieves packet-level animation is VINT's¹ network animator Nam[8]. Nam consists of a 2 dimensional network view with packets flowing on links. Nam packets have a rectangular shape extruded on the one side to form a point that indicates the direction of flow on the link. Our packets are based on a similar concept using texture mapping. Figure 11 illustrates packets flowing on network links in both directions. The packets are rendered as a rectangular polygons with an arrow

¹Virtual InterNetwork Testbed.

shaped texture mapped onto it. This visualization method helps to identify back to back packets.

The main feature that sets our system apart from Nam is in that we use modern computer graphics hardware allowing us to achieve packet animation on a larger scale. To understand the differences and the advantages gained by using this technology we first explain how different graphical systems work.

4.3.1 Understanding Interactive Computer Graphics (CG)

In its most basic form CG can be created by writing low level routines that access a memory buffer inside the video card. Upon accessing this memory buffer, colour values can be placed inside the buffer which results in pixels being displayed on screen. This was the first technique used to provide basic computer graphics to end users in the form of games such as “Space Invaders”. Essentially the technique calculated each pixel’s colour value for each frame, with successive frames calculated as fast as possible to achieve animation. This method was only useful at very low resolutions such as 320×280 ($= 89600$ pixels). Executing a loop 89600 times to calculate each pixel’s colour value in under 0.033msec per frame was no easy task with the processing power available at the time. However, this technique can be used to display mostly static graphical objects, which is still used to produce today’s window-like operating systems. The standard GDI² Nam uses this GDI to provide its network animations, which works as long as computer processors are fast enough to “paint” the animation fast enough.

It became clear that this system (which has been dubbed software rasterization) had no future if more complicated CG was to be attempted. A new system was developed that shifted the rasterization (paint) process, which was done by software (CPU) at the time, to video hardware. Therefore, instead of sending the video card an entire set of pixels to represent some shape (for example a rectangle), only the vertices or corners are sent with the hardware rasterizer painting the pixels in between. This revolutionized the interactive computer graphics industry. Complex graphical scenes could now be constructed at high resolutions running at high frame rates.

Although scenes grew in visual quality and complexity a new problem emerged. Before the shapes that the millions of vertices represent can be painted, they (the vertices) need to be translated into place. This requires the construction of a so called 4×4 “world” matrix for each object, which comprises translation, scaling and perspective information. The respective world matrices are multiplied with each vertex of an object, to translate an object into place. This is an expensive operation that can saturate today’s fastest CPUs. The solution to this problem is much the same as the solution to the software rasterization problem: move the computations to specialized video hardware. Because of marketing reasons or

²Graphical Device Interface.

because of the complexity involved in creating this new form of hardware, vendors decided to give the chip that performs these computations a name: the Graphics Processing Unit (GPU). Because the GPU was specially built to perform Hardware Transform and Lighting (known as Hardware T&L), it could perform these duties much faster than the non specialized CPU (known as Software T&L) could, while in the process also freeing the CPU to do other calculations.

The last piece of information necessary to understand how we implemented the packet animation system is a concept known as vertex shading. In most 3D applications vertex shading would not be used. Normally objects to be rendered are mostly static and few in number. By static we imply that the vertex data do not change on a frame by frame basis. The rectangular prisms used to visualize packets are an example of a static object. When a packet is created its vertices are created in such a way as to convey its size and thus never change while the packet is flowing down a link. A dynamic object would represent something like a water surface with waves. Vertex data change constantly according to a sine wave function. Implementing such vertex changes on a frame by frame basis is expensive and hence the necessity for a vertex shader. A vertex shader effectively changes the position of static vertex data (the flat water surface) according to some function programmed inside a vertex shader program. Just as a C++ compiler is used to generate code for CPU's, a vertex shader compiler compiles a vertex shader program for the GPU.

4.3.2 Using the Vertex Shader to Improve Packet Animation Performance

Now that we know how the basic vertex shader process works we explain how we use it to gain a significant performance improvement when animating packets. We mentioned that the vertex shader helps improve performance of dynamic objects, but noted that the packet objects are static. Furthermore, we claimed that vertex shaders are normally only used when inter vertex movement of the same object does not occur in unison, because if they did they could all be translated using a common world matrix. So why then use a vertex shader to translate a packet over a link? The reason lies in the fact that packets occur in large numbers, so many that the normal method of translating each individual packet into position would be too costly. Furthermore, closer inspection reveals that visualizing packets is indeed required to be dynamic, if only for a short period of time. When a packet reaches the end of a link or is located at the beginning of a link, its length needs to be reduced otherwise it would extrude outwards beyond the ends of the links. This poses an optimization issue because the packet vertex data now also need to be updated on a frame by frame basis. When rendering more than ten thousand packets per frame, this vertex updating severely impacts upon the performance of the system.

Consider the number of operations needed to translate a packet that is $P\%$ complete at

the correct location on a link using the conventional method. Each packet's vertices are defined to be located at the link where they have completed 0% of their journey.

1. **Save the current world matrix** ($[4 \times 4]$ **matrix copy**). This step ensures that the next packet to be put in place has a fresh world matrix containing no translation information of the previous packet. The world matrix currently only contains world rotation and perspective information, not translation.
2. **Create the initial packet translation vector** ($[4 \times 1]$ **vector copy**). This vector is created by setting it equal to the direction vector of the current link the packet is traveling on. This direction vector has a magnitude equal to the current link's length and because nodes may be moved by the user, a link's direction and length might change. Such changes therefore need to be updated into the translation vector for every frame.
3. **Create the final packet translation vector** ($3 \times$ **floating point multiply**). We now multiply each of the initial packet translation vector's components with P , the completion status value that represents the actual position of the packet on the link.
4. **Multiply translation information into the world matrix** ($[4 \times 4][4 \times 4]$ **floating point matrix multiplication**). When specifying a translation vector to be added to the world matrix OpenGL converts this vector into a matrix and multiplies it with the world matrix. The setup of the world matrix is now complete and unique to this packet on this link.
5. **API Render call (hardware performs a $[4 \times 4][4 \times 1]$ floating point matrix vector multiply 4 times)**. The hardware translates and rotates each of the packet's 4 vertices into place. It does so with the use of the computed world matrix. It then proceeds to render the packet onto the screen.
6. **Restore the world matrix** ($[4 \times 4]$ **matrix copy**). Restore the world matrix saved in step one for the next packet to use.

These 6 steps are not expensive to compute once, but executing these 6 steps for each packet on each link for each frame for many frames per second adds up to a large number of instructions. The most costly steps are steps 4 and 5, but they all contribute to make the entire process very slow.

The question presents itself, can we do better? The answer is that we are in fact doing better. We mention in step 5 (computationally the most expensive step) that the matrix vector multiplications are done using graphics hardware, in other words the GPU. This takes much of the workload from the CPU. Acquiring such functionality is already a major step forward and requires complicated mechanisms to acquire the the correct set of circumstances necessary for such hardware acceleration to take place. However, with the help of a vertex shader program we can do better.

With our vertex shader method the number of operations required on a per packet basis by the CPU is 0. Steps 1,2,4 and 6 are eliminated with steps 3 and 5 handed to the GPU for processing, with the aid of the vertex program. The idea behind this is that we instruct a link to render all the packets traveling on it at once, with the individual packet translation information embedded within each vertex of every packet. For rendering purposes, each packet's vertices consist of four floating point values, three to specify its position and one to be used later by the transformation process to store 3D perspective division information. We propose to piggy back in this fourth value the P (the position of the packet on the link) value described earlier in this section. This provides the necessary information to position each unique vertex of each unique packet at the right location on the link it is traveling on. Without a vertex shader program to intercept this P value the rendering process would break down since the fourth value for each vertex is required to be equal to 1 by the renderer. This P value is also meaningless if we do not know in which direction to translate the vertex into and that is why we stop at the link level when rendering packets (in other words we cannot render all the packets flowing inside the network with one API render call). Before every link makes the API call to render the packets flowing on it, it specifies a constant value to be input to the vertex shader program. This constant value is the direction of the link. When the rendering call is made, the vertex shader program is run with every packet vertex as input which in turn collects the P value from the vertex's fourth value and translates the vertex $P \times linkDir$ into that direction. It then resets the fourth value to 1 and completes by multiplying the translated vertex with the world matrix. The entire program can be viewed in Appendix A.

Although there are further subtleties involved in rendering the packets flowing on a link with one API render call, they are of a technical nature. They involve maintaining an index buffer for each link for each frame, which the API then uses to determine which packet vertices are to be drawn for a particular frame. The index buffer is updated every time a packet enters and leaves a link. The CPU and GPU are also working in parallel. While the GPU is busy transforming and rendering the thousands of packets, the CPU is continuing to read the next set of events from the trace file and setting up the index buffers for the next set of rendering calls. This results in a large performance increase.

4.3.3 Performance Evaluation

In the previous sections we described how our packet visualization system works. We complete this discussion with a performance evaluation of the system. We focus mainly on performance reports concerning the packet visualization system, as the node-link visualization does not present the type of scalability issues that the packet visualization does. A question such as "How many packets can be visualized at any one time without compromising interactivity?" is the type of question we answer here.

Our test system is running on Pentium 4 2.8 *GHz* CPU with 512 *MB* DDR 400 memory running in dual channel mode. The system also runs a *Nvidia* GForce FX 5900³ series graphics card, which at the time of the benchmark was about the 4th fastest card on the market and by no means a slow GPU. The vertex shader and therefore the packet visualization performance is directly dependent on the speed of the GPU.

Our test network being simulated consists of a 125 node 300 link cube-like network seen in most of the figures. Because we incorporated complex culling techniques which speed up the performance if links are not visible, we made sure that the entire network is visible at all times. Every link on the network is injected with a constant packet stream in both directions. We inject the packets and start taking performance readings only when the network reaches its maximum packet count. We start off with only a few packets per link, which results in an overall packet count for the network of 750, until we saturate the links with as many packets as they can carry resulting in a total packet count of 12000 visualized on screen. The links are set up to visualize a link speed of 1*Gbps* in both directions with a link latency of 0.25*msec*⁴. When all these links are fully utilized we obtain 12000 packets flowing in the entire network. Performance is quantified by the average frames per second (FPS) achieved over a visualization period of 60 seconds.

Figure 12 shows two graphs depicting the average FPS achieved for visualizing x packets where x is 750 to 12000. The solid line graph shows the FPS achieved when using the vertex shading packet visualization method, with the second graph showing the FPS achieved when using conventional techniques which make use of standard GPU transformation power. The error bars represent the minimum and maximum FPS obtained for a certain test visualization, with the average somewhere in between. The second graph in Figure 13 shows the average percentage gain over the amount of packets visualized.

The results are promising for both techniques, with the less efficient of the two techniques starting to fail when more than 6000 packets are visualized. The goal is to obtain a rendering speed of more than 30 FPS, as FPS below this compromises the interactivity of the network visualization. Figure 13 shows that under heavy load our vertex shading rendering path achieves as much as a 350% increase in performance.

With a larger network than the one we tested here the performance would not necessarily drop in normal use. In reality only parts of the network would be in view at any one time. Because of the culling techniques in place, invisible links together with the packets flowing on them are culled to increase performance. This means that while navigating through a network, frame rates of 40 FPS can be expected when there are 12000 packets currently in view. This is a promising result.

Another factor that influences the performance of the system is the rate at which the

³Graphic card performance is currently increasing at a phenomenal rate. Within 2 years this graphics card might be as much as 30 times slower than the leading edge product.

⁴0.25*msec* is the time taken for one bit to propagate through a fiber of length 50*Km*.

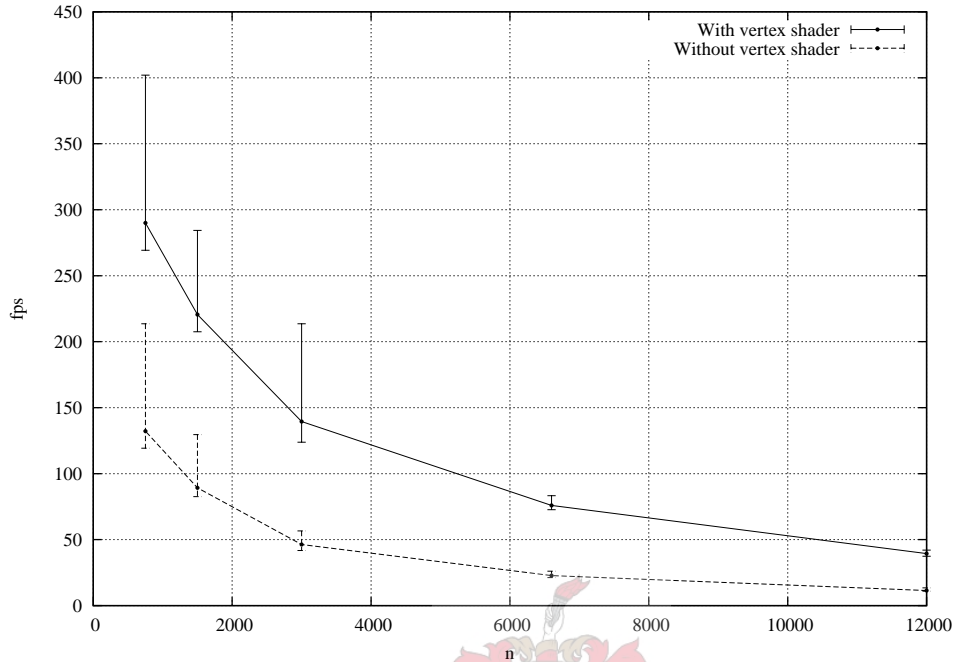


Figure 12: The average frames per second FPS achieved when rendering n packets.

visualization animates the network. These performance measures were taken when visualizing the network at 1/15000th of the actual speed. In theory this has an influence on the performance, as injecting and removing packets at high rates requires more bookkeeping in terms of setting up the correct index buffers for each link etc. by the CPU. With the non-vertex shader technique this is indeed the case with performance dropping rapidly as the speed of the visualization is increased.

However, because of the optimized parallel nature of our vertex shader solution we found the speed factor to have no effect on the performance. Our explanation of this observation is the fact that the GPU must be lagging behind the CPU while trying to finishing its rendering tasks. Therefore, while the GPU is still rendering the current frame's batch of packets the CPU has had enough time to setup the next frame's rendering state. We set out to confirm our theory with experimental results acquired when visualizing the network with 12000 packets.

Firstly we discuss the background of the test conditions. In the render path, before the next frame's batch of packets can be rendered it is necessary to instruct the GPU to finish with its current rendering queue, thereby synchronizing the GPU with the CPU. This is necessary for every frame because a situation can arise where vertex data are being changed before they are rendered. To achieve synchronization a special API function is used called `glFlushVertexArrayRangeNV()`. This instructs the GPU to finish with its current queue of render calls while blocking the CPU. We set out to measure the time before and after this

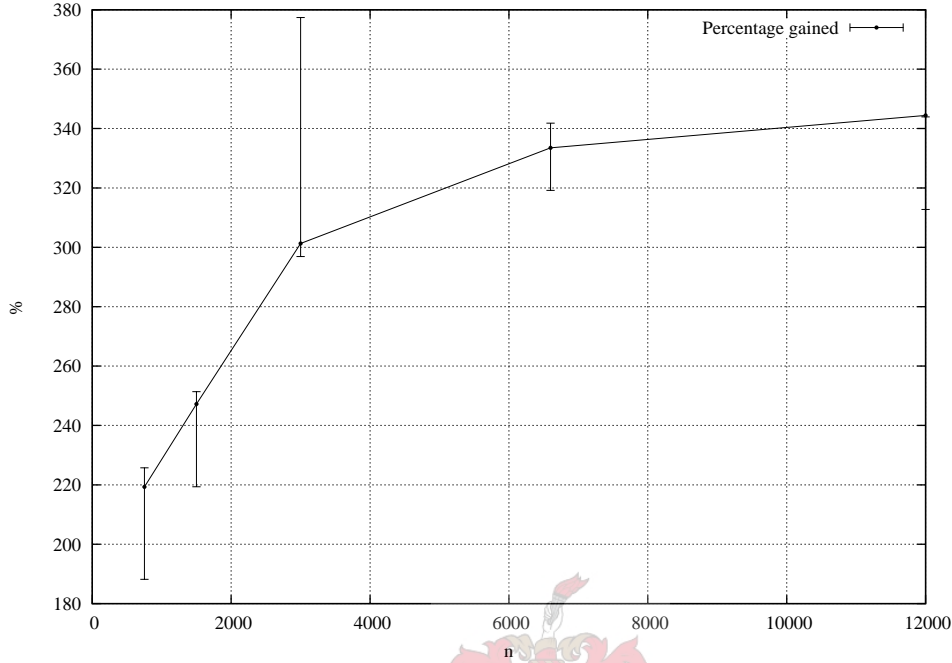


Figure 13: The average percentage gain achieved over the same number of packets n rendered as in Figure 12.

function call to get an idea of how long on average the CPU waits for the GPU to finish its rendering queue in extreme conditions.

For the non-vertex shaded implementation we find that on average the CPU waits $4.1msec$ for the GPU to finish with its rendering tasks. When increasing the speed of the visualization to 1/1000th of the actual speed we find a large performance drop with an average waiting time of $1.1msec$. This indicates that the CPU and GPU finish with their tasks in more or less the same time. Because of the very low FPS acquired in this test we also deduce that both the CPU and GPU are being overworked. Installing a faster GPU will gain no additional FPS, and only when installing a faster GPU and CPU together can an increase in the frame rate be expected.

For the vertex shaded implementation the situation is different. We find the average waiting time to be $15.4msec$ and dropping to $12.25msec$ when the speed is increased, with no apparent FPS loss. This indicates that the CPU still has $12.25msec$ left to do more bookkeeping duties. If a faster GPU is installed this average waiting time would come down and the average FPS would go up until the CPU becomes the bottleneck again.

4.3.4 The Physics Involved in Packet Visualization

In the previous sections we discussed the bookkeeping required for packet visualization. It mainly referred to the process of keeping track of which packets are currently flowing on a link and which packets are in a standby buffer. Although special techniques must be applied for optimization reasons, it is implementation specific and up to the programmer to implement correctly. Now we will discuss the physics behind the packet system as it dictates what the simulator needs to send to the visualization system.

We conclude that there are no abstractions we can use to visualize packets with less accuracy, that would make it easier to use for a simulator. Furthermore, if the physics is not implemented correctly it would provide very little useful information to the user. The disadvantage is that if a simulator wants to use packet visualization, it needs to implement a packet engine to generate data fit for visualization. Unfortunately we found that few network simulators go to this trouble, with the exception of the ns [22] simulator. Most simulators work with aggregate flows and therefore cannot benefit from raw packet visualization.

The requirement placed on the simulator is that as it generates packet departure events, the event times must be realistic. We argue in the next section why such realism might help a network engineer gain more insight into a protocol's failure or success. The possible violation of the packet physics would be a scheduled packet departure event on a link when a previous packet event is still being signalled onto the link. Other issues would also arise if a packet does not compensate for the latency incurred when packets propagate over a link. This might cause the visualization of packets that are out of sync. We therefore explain the calculations used to visualize packets in order to clarify the requirements necessary to generate events that can utilize the packet visualization system.

Minimum Inter Packet Departure Times, Collisions and Queuing

Network simulators can use inaccurate inter packet departure times for simulation purposes and still generate useful data from such models. These do however present a problem when generating event data for the visualizer. The problem is that in order for the visualizer to do most of the work it needs examine the packet event times and accurately place the packets on a link in such a way that the packets do not overlap. If they do overlap then graphical anomalies are produced which are distracting. The time taken for a packet to be transmitted on a link is

$$\tau_{\ell}(s) = 8s/\beta_e$$

where $\tau_{\ell}(s)$ represents the transmit time in seconds of a packet of size s bytes onto the link, and β_e represents the bandwidth of the link in bits per second. Since the formula

is not computationally intensive we can calculate these values when simulating an event stream and reject those packets with incorrect event times. The simulator is required to do the same calculations and thereby stream correct event times. This double checking of event times can be useful for debugging simulators, if such an event is accompanied by an error message. In practice this functionality might indicate occasional packets generated with incorrect event times by the simulator. If the visualizer has to reject packets too often the visualization would slow down considerably because of output error text overhead.

The simulator can also request that the visualizer enable an automatic visualization queuing system. This is only recommended for label switched path network simulators that have a relaxed or simplified packet physics system. Such simulators would typically inject packets at ingress node and leave it to the visualizer to visualize the correct behaviour of the packet flow until the packets reach the egress router. In such a mode a packet is inserted into a priority queue when they reach a congested link. If all packets have the same priority this queue would become a FIFO queue. The visualizer can then place packets on the link when it is ready to send another packet. If the queue becomes full packets are dropped. If however the simulator keeps track of queueing events, it can request a different queueing mode where it then can send queue and drop events for the visualizer to visualize. This way queueing strategies such as Random Early Detection can be correctly visualized.

The next step is to calculate the packet lengths correctly.

The Packet Length, Propagation Delay and Link Visualization Length Dilemma

The next step is to size the packet visually in proportion to its length on a link. To render the length of a packet on a link, we use the percentage of the link's storage capacity used by a packet to calculate its length. A link's storage capacity refers to the number of packets that can be in transit on the link. If a link ℓ has a storing capacity of β_ℓ bytes and a graphical representation vector $\mathbf{v}_\ell = \mathbf{p}_{\ell,o} - \mathbf{p}_{\ell,d}$, then the length of the packet is calculated by

$$\rho_\ell(s) = \frac{s}{\beta_e} \times \|\mathbf{v}_\ell\|$$

where

$$\beta_\ell = \beta_e \rho_e$$

with ρ_e equal to the propagation delay on link ℓ . $\mathbf{p}_{\ell,o}$ and $\mathbf{p}_{\ell,d}$ represent the position vectors of the two endpoints of the link.

Although this algorithm calculates the length of a packet correctly, we use a different method of visualizing packet length. Because of the extruding packet problem, the calculated length needs to be adjusted at the ends of a link. Since we make use of the vertex shader we not only propagate the vertices of a packet over a link, but we dynamically adjust

them to portray the correct length. In the case of a platform not capable of vertex shading, the previous method is employed in conjunction with scissor boxes. We start by creating all 4 packet vertex positions at the edge of a link with the head of the packet's two vertices overlapping with the two tail vertices. When each individual packet's P value is then set for each vertex we start by adjusting the head vertices and then after a time $\tau_l(s)$ we adjust the tail vertices. When reaching the end of a link we stop adjusting the head vertices and continue adjusting the tail vertices. We therefore bypass the calculation of adjusting the packet's length according to the packet's position and because the renderer cannot know when such changes might start altering a packet's geometry, it must update the packets geometry for every frame and hence the performance penalty.

The final step is to create a realistic packet system where packets on different links can be compared by their lengths to gauge their size. This might be helpful if a network carries different sized packets or has different bandwidths on different links. However, a link's propagation delay is proportional to its length, while in visualized form it is not. The SOM algorithm achieves a layout without any compensation for link latencies. In the current system, a link with a large latency might be visualized as short and therefore result in packets appearing shorter on that link. Thus we can compare different packets on the same link, but not between links. A possible solution to this problem might be to size the links in proportion to their latencies after the SOM algorithm completes its layout. This is a unexplored problem as using latency information might result in unpredictable layouts being generated on artificially generated networks. We have not implemented such a system as we are not sure how insightful it would be to compare sizes of packets flowing on alternate routes.

Packet Position and Removal

A packet p 's position $P_{p,\ell}$ on link ℓ is easily calculated. We compute the total time Δt_p a packet has been flowing on a link by subtracting the packet's service time μ_p (a possible queuing delay) from the packet's event time $t_{p,e}$ and then subtracting that from the current simulation time t_S (this is the time used by the visualizer to do its own simulation calculations and has no relation to the simulator's time)

$$\Delta t_p = t_S - (t_{p,e} - \mu_p).$$

We then divide the total time by the propagation delay of the link the packet is flowing on

$$P_{p,\ell} = \Delta t_p / \ell_{pg}.$$

We then remove the packet from the link if the packet's total time in the system is greater than its maximum time in the system $\Lambda t_{p,\ell}$ where

$$\Lambda t_{p,\ell} = \ell_{pg} + \tau_\ell(s) + \mu_p$$

4.4 Visualizing Label Switched Path Networks

The basic idea behind visualizing a label switched path network is the creation of a system whereby the simulator need not calculate predictable values for the network visualizer. Therefore, concepts such as sending a packet down a route need not be decomposed into packet/queue events for each link by the simulator. If the visualizer is aware of a route then it could send the packet down successive links itself. Because the system was designed to be extensible, this functionality can be written by a user who wants to visualize a label switched path network, but because next generation network engineering will use such functionality we use this as an example of how to implement such an extension.

We start by defining the network classes that would be used by such a network. Since we have already written a class structure of a basic network visualization, very little needs to be done for the new set of classes. For instance, we defined a class called **LSR** which represents a label switched router. This router has the functionality of a basic network node but with extensions, such as keeping track of which routes flow through it. We therefore derive the **LSR** class from the **Node** class and reuse the code written for the class **Node**. This includes inserting nodes etc. We need not write any code that renders a **LSR**, since this is inherited from the **Node** class. If we derive **LSR** from **Node** and add no extra code, **LSR** would behave exactly like the **Node** class because of the **Renderbase** interface that does everything automatically. If the **LSR** class does not override any of the interface calls, the **Node**'s implementation of the interface would be used. If the **LSR** class does override an interface call, it may choose to use or supplement the **Node** class's interface, or bypass it and define its own implementation.

The second extension works in the same way as the first. This includes the media classes used to stream network information to the visualizer. We extend the classes to send extra information to the visualizer which can be used by the extended network classes. Basic functionality is retained through extension.

Since a label switched path functions differently from an administrative point of view with no rendering changes we generally need not write new graphics code to visualize the network. Depending on what needs to be visualized a derived class such as **LSR** is allowed to change the colour field of its parent class's renderer object with information pertaining the number of routes going through the LSR. In this way a single point of failure might be identified. We derived a set of classes that functions in this way which can be seen in Figure 7 in Chapter 3.

4.4.1 The Advantages of LSP Network Visualization

The LSP network classes allow us to handle events such as adding multiple LSPs to an OD pair and removing them later on, visualizing the changes as they happen. Substantial

optimization can be achieved with packet rendering. Instead of many events inside the trace stream for packet departure times we can now only have one packet event per route. Because of the network structure in place, the visualizer knows that when a packet finishes on one link it needs to be forwarded to the next link along the LSP until its destination is reached. Packet information can be copied from the one link's packet buffer to the next, thereby saving the reload needed from the network event stream. This results in significantly smaller trace files.

In this mode the standard packet priority queueing model explained above is always used. It would be impossible to code every different queueing scenario and if the basic queueing functionality is not sufficient, the simulator's queue functionality needs to be implemented inside the visualizer so that packets are animated properly.

Other advantages are that ideas can be quickly tested with this type of network visualization. If a network engineer wants to test an idea, s/he can simply instruct the visualizer to create an LSP between an OD pair and not specify a path. The network code embedded within the visualizer would then select the shortest path at that time between the OD pair according to the respective link weights. The path can also be controlled by changing link weights by the simulator throughout the animation, or an explicit path can be specified. Packets can be sent down this path. This can be achieved without writing any simulation code, but by using the visualizer's standard behaviour.

An OD pair can specify a maximum latency QoS value for packets traveling over some of its LSPs. Packets flowing along these LSPs can be coloured as they flow according to their aggregate latency built up by propagating links and experiencing queueing or other delays. Thus, although a flow algorithm might compute a network configuration that achieves maximum aggregate flow, the packets themselves might exceed acceptable latency limits. The deployment of voice over IP in next generation networks will require setting up routes with the least latency which will become a bigger priority than generating revenue per unit bandwidth sent. Using tools such as the packet visualization will allow analysts to explore the QoS of a network more effectively than sifting through large amounts of packet data.

Many other types of visualizations are possible by adjusting the colours of rendered objects. We implemented a few to convey the idea, but in reality many of the visualization effects would be configured by the user.

Chapter 5

Data Interface

In this thesis we have examined the data transfer between a simulator and the visualizer. In practice, network data can have any number of sources and we therefore designed an interface to abstract the underlying complexities of encoding and decoding network data. This chapter does not discuss the more commonly associated meaning of a user interface, which is the interface used to interact with the visualization. We have not implemented such an interface because it does not add any scientific value to this thesis. It is a time consuming coding exercise to extract data inside the visualization and display it using different graphs or graphical effects. Such a user interface will be designed when it has been determined that the underlying visualization system is complete and scalable.

5.1 Classic Data Exchange Models

We start by explaining commonly used methods of providing network event data as input for a visualization program. Most implementations use a text data format to specify event data, for example NAM [8] uses such a trace file format. Not only is the data stored in text format, but it is also readable. This has advantages and disadvantages. Having a readable trace file format permits the developer of the visualization system to debug trace files that do not work properly, by being able to read the line that caused the problem. Test cases can be developed with minimal effort. Normally this sort of trace file format is convenient when providing input to a program. One can read the inputs given to the system and gain a better understanding of how the system works.

With most systems a readable trace file would be the preferred method of transferring data. However, because of the volumes of data associated with network trace data we claim that this is not an appropriate way to transfer data. Computers can read more obscure formats than humans can and creating trace files that are not readable can save space. All

the descriptive names and spaces between different values etc. can be substituted by a set of serialized bytes.

ASN1 [23] is a well known encoding-decoding standard. It takes a set of variables, data structures and events from one computer system, encodes it into the ASN1 encoding format, then sends it to a second computer system for decoding and processing. The ASN1 coding-decoding rules are simple and can encode a general set of variable length data units in one stream. The data unit consist of a header and a payload. The header describes what kind of data unit this packet represents with the accompanying payload consisting of a variable number of bytes as data. The decoder looks up the packet type in a table from which it determines the nature of the data. The size of the payload is decoded from the header and the payload is extracted and processed. Because of the generality of the ASN1 format, we implemented our own encoding format that works in a similar way, but is optimized in terms of event data storage capacity. We do not need an encoding-decoding standard that can encode-decode any number of data structures with arbitrary sizes in one stream. This approach is too general and complicates the encoding-decoding stages of the data transfer. We only need a system that can encode-decode simple network event data structures of which the nature of the payload size and meaning are known. Using this type of data encoding in conjunction with C++ function pointers we created a system that is optimized to cater for the amount of network data we are working with.

5.2 Using C++ Member Function Pointers to Simplify the Data Interface's Design and Implementation

To properly demonstrate the effectiveness of this method let us consider how to implement a data interface in an ideal situation. As stated above an ideal situation (given enough processing power) would be to have the visualizer and the simulator contained inside the same application. The simulator would make procedure calls to the visualization subsystem. Since in reality these two subsystems, the simulator and the visualizer, do not exist inside the same program such a simple and effective method is not possible. Such a solution would require the source code of the simulator to be combined with the source code of the visualizer. Such a feat is possible and would work, but it would not be practical. Hardware requirements for the simulator and the visualizer differ radically and combining the two would limit the functionality of the two programs. The simulator would typically require a fast computer with possibly multiple CPU's and a large amount of memory. The visualizer requires a different kind of computer where state of the art graphics hardware is a key element and CPU processing power is not as important. Since the computers with the biggest processing power available do not have the capability to receive graphical hardware, one starts to realize that such a combination of source codes would not be possible.

Using procedure calls to drive the visualization from the simulator is therefore not possible and traditionally some form of message transfer has been used to communicate between the two systems. Messages would be written to a file and the visualization would parse these messages and act accordingly. It is in this parsing step where we make our optimization by using a different system than that which is normally used. Our system can be described as a type of remote procedure call interface. Traditionally a message can be represented as:

```
<eventNr> <eventId> <time> <ingress> <egress> ...
-----
1         packet    2673   20         50         ...
2         packet    2678    4         20         ...
3         packet    2685   35         4          ...
```

which would have to be scanned, parsed and acted upon. A scanner would convert the “packet” string into a number or token which is then input to the parser that combines different tokens to decode the meaning of the message. Our system eliminates this scanning and parsing stage. Instead of writing readable messages we write numbers embedded inside a byte stream to indicate which procedure of the visualizer we want to call. These numbers serve as indexes into an array of pointers to member functions¹. The entire scanning parsing system has therefore been substituted by simply reading 4 bytes from a stream and interpreting those 4 bytes as an integer value which is used as an index into an array of functions.

It only remains to pass parameters to these functions. With reference to our system we would make function calls with parameters to drive the visualization. We have described how we select which function is to be called but not how the parameters are handled. Solving the parameter problem is trivial as in addition to the function index embedded within the byte stream we also add parameter values. The function index is extracted, the function at that index is invoked and the function itself reads the correct number of parameters from the stream. Figure 14 illustrates the concept.

Although the byte stream also contains network setup information etc. 99% of the trace file consists of event data. When simulating a network with 125 nodes and 300 links all running at one gigabits per second for a tenth of a second with all the links running at full capacity our system generates a trace file with a size of 77Mb. Compression schemes might be used in future implementations to reduce the size of the files. Doing a standard compression of the file yields a file size of about 10Mb which is manageable.

In the previous chapter we explained that although our visualizer can visualize data that specifies packet event data per link, a more advanced use would be to specify packet event data per LSP. This would reduce the size of the trace file considerably. Instead of the

¹A member function is the name given to function that is associated with a C++ class construct.

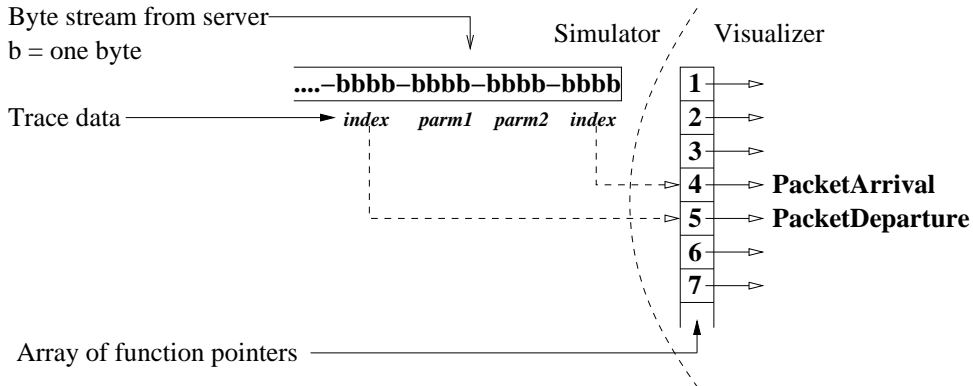


Figure 14: Illustration of the data interface used to transfer event data.

simulator sending packet event data for each packet traveling along each link along an LSP, it would be more advantageous if it could only send a packet event per LSP. Since our visualizer supports an LSP network and does its own physics calculations on the packets (for rendering purposes) this feature was not difficult to implement. Therefore when using a simulator that generates packet flows over a LSP network, a simulation can be visualized over a much larger time frame and on a bigger network without being encumbered by huge trace file sizes.

5.3 Data Interface Implementation

The above section describes the concept used to transfer data between the simulator and the visualizer. The implementation of this concept is more complex because we have different network types that build on each other and because we designed the data interface to be extendible with the least effort. These complexities involve C++ specific issues and do not add any value to the thesis.

To hide the technical complexity of the way that data is transferred we have written a lightweight library that is separate from the visualizer. This library is linked together with the simulator and is used to generate the trace file output. A simulation software package to output visualization data can therefore be modified without being concerned with technical issues involving the generation of a trace file that can be read by the visualizer. The library consists of procedure calls that the simulator can use to visualize its network. The library converts the sequence of function calls and their respective parameters into a trace stream that conforms to what the visualizer expects.

5.3.1 The Standard Network Stream

The following functions are provided by the standard network stream:

```
void Begin( unsigned int gNoNodes, unsigned int gNoLinks )
```

The network topology setup phase is started by making a call to this function. The parameters **gNoNodes** and **gNoLinks** specify the number of nodes and links in the network. These parameters serve as a hint to the visualizer for performance optimization reasons. If the exact numbers are not known then specifying zero values would be sufficient albeit at a minor initial performance loss.

```
void End()
```

This function is called when the network topology setup is complete. This function invokes post processing functions by the visualizer such as network layout etc. No further changes to the network topology can be made after this function has been called.

```
unsigned int NewNode( unsigned int gType, float gSize, unsigned  
int gColor = 0 )
```

This function can only be invoked between an *Begin()/End()* sequence. This function adds a network node to the topology. The parameter **gType** specifies the type of the node ranging from $1 \dots n$ where n is the number of types implemented. Specifying different type values would result in different visualizations of a node. The current implementation supports 3 types (sphere, cube and pyramid) but more can be added. The parameter **gSize** is used to scale the visualization of the node to be either bigger or smaller. The **gColor** parameter is used to specify the color of the node which is interpreted as an array of 4 bytes each specifying r,g,b,a values. The **return** value is the ID assigned to this node. This ID will later be used to connect two nodes with a link. IDs start at 0 and end at $n - 1$ where n is the number of calls to *NewNode()*. The return value is provided for convenience and need not be explicitly stored.

```
unsigned int NewLink( unsigned int gNode1, unsigned int gNode2,  
unsigned int gCapReceive, unsigned int gCapSend )
```

This function can only be invoked between an *Begin()/End()* sequence. This function creates a link between two nodes. The **gNode1** parameter specifies the ID of the source node and **gNode2** the destination node. The IDs are obtained from the *NewNode()* function or can be specified manually as long as it is between zero and the number of nodes minus one. **gCapReceive** is this link's receiving capacity in bits per second, that is traffic from **gNode2** to **gNode1**, and conversely for **gCapSend**. The **return** value is the ID of this link. As with nodes, IDs start at 0 and increment as more links are created.

```
void PacketEvent( float fTime, unsigned int gDirection, unsigned
```

```
int gLID, unsigned int gSize )
```

This function can only be called between a *BeginOfSimulation()/EndOfSimulation()* sequence. This function triggers a packet event at time **fTime** on the link with ID **gLID**. To indicate whether a packet was sent from **gNode1** to **gNode2** or vice versa the **gDirection** is set to either indicate a send or a receive. Zero indicates receive and anything else indicates a send. **gSize** specifies the size in bytes of the packet.

```
void BeginOfSimulation()
```

This function must be called before simulation data is streamed and is necessary to signal to the visualizer that a new simulation is starting.

```
void EndOfSimulation()
```

This function must be called when the simulation has finished.

5.3.2 The LSP Network Stream

The LSP network stream is built on top of the standard network stream. The two streams designs are similar with the standard network stream being a subset of the LSP network stream. In addition to the functions specified in the standard network stream the LSP network stream provides the following functionality:

```
void NewLSP( unsigned int gLSPid, unsigned int* aLSRs, unsigned
int gLength, unsigned int gBwReceive, unsigned int gBwSend )
```

This function creates a new label switched path within the network. Since label switched paths can be created and torn down in the course of a simulation this function is handled as an event and must be called between a *BeginOfSimulation()/EndOfSimulation()* sequence. The parameter **gLSPid** is a simulator assigned ID to the LSP it created. With the creation and tearing down of LSPs it is required that the simulator keeps track of different LSPs and their IDs. **aLSRs** is a pointer to an array of label switched router (which are nodes) IDs which describes the path through the network. The two parameters **gBwReceive** and **gBwSend** are the capacities assigned to this LSP in bits per second.

```
void RemoveLSP( unsigned int gLSPid )
```

This function removes LSP with ID **gLSPid** from the network and must be called between an *BeginOfSimulation()/EndOfSimulation()* sequence.

```
void PacketEvent( float fTime, unsigned int gDirection, unsigned
int gLSPid, unsigned int gSize )
```


This function signals that a packet of size **gSize** is to be sent down the LSP with ID **gLSPid** at time **fTime**. As with the standard network stream the **gDirection** parameter

specifies if the packet is being sent or received.

5.4 An Example

We proceed to give an example of how the data interface can be used by a simulation process. The example we give here is not driven by a simulation process but rather a simple loop that causes all the nodes to send a constant byte stream to neighbouring nodes. The code presented here is C++ as currently the data interface libraries can only be linked to by an C++ compiler. It is possible to write a data interface in any language which can be used with a simulator written in that language.

The example we give here is the code that was written to create the network which consists of a $n \times n \times n$ node network, which is presented in several Figures in thesis. The code starts off by creating all the nodes contained within the network followed by a simple loop to connect the nodes in a mesh like structure as illustrated in the figures. The last part consists of generating event data for visualization which in this case is a simple loop that causes all the nodes to send packets down the links at full capacity.



```

#define dimension 5
//A simple function mapping 3 coordinates into an array
int gd( int i, int j, int k)
{
    return i + j * dimension + k * dimension * dimension;
}

//The test network
void Media::TestNetwork()
{
    //Variables
    int size = dimension;
    int noNodes = (unsigned int)pow( (float)size , (float)3 );
    int noLinks = size * size * ( size - 1 ) * 3;
    Stream_Std ui( Stream_Base::FILE, "netTracer1.tfl" );

    //Start of the network definition phase
    ui.Begin( noNodes , noLinks );

    //Create all the nodes
    for( int c = 0; c < size * size * size ; c++ )
        ui.NewNode( 5, rand()%4 + 2 );

```

```

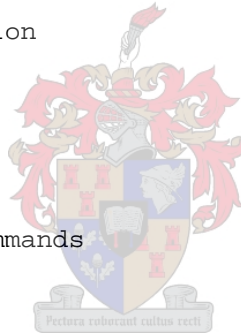
//Create all the links
for( int i = 0; i < size; i++ ){
    for( int j = 0; j < size; j++ ){
        for( int k = 0; k < size; k++ ){
            //Create a link with a bandwidth of 1000000 * 1000 bps
            if( i + 1 < size )
                ui.NewLink( gd(i,j,k), gd(i+1,j,k), 1000000, 1000000 );
            if( j + 1 < size )
                ui.NewLink( gd(i,j,k), gd(i,j+1,k), 1000000, 1000000 );
            if( k + 1 < size )
                ui.NewLink( gd(i,j,k), gd(i,j,k+1), 1000000, 1000000 );
        }
    }
}
//End of network definition
ui.End();

long double t = 0.0;

//Begin of simulation commands
ui.BeginOfSimulation();

//Simulation lasting 0.1 sec
for( t = 0 ; t < 0.1; t+=0.000012)
{
    //For all the links
    for(int c = 0; c < noLinks; c++ )
    {
        //Send and recieve a packet of size 1500 bytes
        ui.PacketEvent( (float)t, Receive, c, 1500 );
        ui.PacketEvent( (float)t, Send , c, 1500 );
    }
}
//End of simulation commands
ui.EndOfSimulation();
}

```



The code is simple and is easy to understand and illustrates the entire sequence of function calls necessary to create a network visualization. The example network and its simulation have been kept simple to keep the code as compact as possible. In practice the function calls would be inserted in different places of an simulator's implementation to extract the trace data needed for visualization.



Chapter 6

Conclusion

Visualizing large and complex systems to some degree of abstraction is of great use to scientists today. Although it might not always be used as a tool to prove tangible results, it serves to initiate investigations into complex systems. Analyzing large simulations of networks is one of the many fields where quantifying results in terms of a single number is very hard to achieve, but yet is sometimes done to simplify results. A simulation might reveal that a network that performs well in certain aspects, such as bandwidth utilization and the number of dropped calls, but only for the network as a whole. These numbers are sometimes obtained by aggregating performance results from different areas of a network over time and combining them into a single value. The problem with such a technique is that although the network might be performing well, some subsets of nodes and links might not be performing according to the QoS requirements. This can skew the outputs of simulation runs so that they look more favourable than they really are.

6.1 The Use of Network Visualization

6.1.1 Network Visualization as an Analysis Tool

Our visualization attempts to identify situations where a part of a network might during some intervals not be performing well, by visualizing the performance of the network using graphical constructs. A network engineer can with little effort identify problematic areas and launch an investigation by examining the raw data of the simulation and drawing conclusions from there.

Network visualization can also be used as a tool to test whether or not certain algorithms function as intended when building network simulators. It may not always be possible to predict the effect an algorithm will have on traffic patterns within a large network, but

concepts can be tested on smaller networks that can be visually inspected. Once the visualization of a small network shows the predicted traffic flows, one can simulate larger networks with greater assurance that the algorithm is working as intended. Other problems can immediately be identified such as certain links not carrying any traffic, large queues at certain nodes or packets whose end to end delays are higher than they should be.

6.1.2 Visualizing QoS in Real Networks

Another matter worth investigating is the visualization of the QoS in real networks, and in particular, MPLS networks. The idea is as follows.

First, it should be realized that a real time visualization would not be practical as the overhead of acquiring the performance data from different parts of the network in real time might be too large. However, insight into how the network was performing in the recent past can be obtained. Apart from fluctuating packet flows inside a network caused by faulty hardware resulting in packet loss, packet flows follow fixed routes inside a label switched path network and hence can be reproduced to a certain degree of accuracy. Detailed information will be necessary to determine the packet service times on each hop and how large router queues can get before they start dropping packets. Information about router scheduling together with the algorithms used to prioritize packet flows of different service levels will also be needed.

Second, it would be necessary to record the utilizations of all the ingress/egress pairs over time. Since today's networks have programs that monitor link utilizations such information is available and can be used as input data for a visualization. The visualizer can process this information to produce a good approximation of the packet flows inside the network at any given point in time.

A powerful computer would be needed to do the calculations, especially for large networks, and the computation would be feasible for label switched networks with OD pairs and routes ranging in the thousands. For large networks one can keep the computations manageable by working with aggregate flows by discarding detailed packet information altogether.

6.2 Aspects of Network Visualization

A goal of this thesis was to justify the validity of visualizing network data. Information visualization has been used in other fields to successfully present large result sets, to enhance understanding of certain problems and find the answers to them. For example consider the visualization of the air pressure around the surface of a new type of wing. Although the visualization of the result will not give a definitive answer as to whether or not the

configuration of this wing would provide more lift and less drag than other configurations tested, it could serve as a good indication. With time and experience scientists would be able to predict which visualizations represent better results than others and thereby save a considerable amount of time sifting through raw data.

We aim to create the same visualization for network engineering, not to give engineers a definite yes or no answer, but to give them an overall picture of what is happening inside the network. We believe that such visualization software can serve as a valuable tool to develop different algorithms that manage flows inside a network. We implemented several basic simulations to test the visualizer and many times the visualization quickly showed when mistakes were made in the simulator logic, or when certain things we expected to see were not being shown. The visualizations served as a valuable tool to initiate investigations into the different phenomena observed.

The areas of network visualization focused on in this thesis concern the visualization of large result sets as opposed to how to use different graphical constructs to highlight problematic areas within a network. To put things in clear perspective the following section summarizes the aspects of visualization we did and did not cover in this thesis

6.2.1 Graph visualization

The first problem that we encountered was creating a suitable visual representation of an input network. We need to present a system whereby the connectivity of a network could be used to generate a graph that could be visualized in three dimensions. Since most networks do not contain geographical data we had to artificially create the geographic locations to best convey the structure or connectivity of the network, while also being suitable for a three dimensional representation that is not confusing.

The Self Organizing Map algorithm is a useful tool in this regard. The algorithm is flexible and can be configured to create a layout of a large network within seconds. The algorithm can also deal with different sized networks much better than other conventional layout algorithms such as the spring modeling approach where a considerable effort has to be put into creating a visualization of each network.

By coding an optimized graph layout system we are confident that we can generate flexible layouts for variable sized networks in an acceptable amount of time.

6.2.2 Program Design

Many platforms and languages exist that can be used to write network visualization software. While the platform is not all important, the code base needs to be well designed to

simplify the development of such a visualizer. By implementing a proper set of base network classes we have shown that creating visualizations of more advanced network constructs can easily be achieved by building on existing functionality. Network visualization in particular lends itself to an object oriented design which if done correctly can provide substantial advantages over more conventional coding techniques.

Although a simpler design would be quicker to implement and can provide the same visual results as our object oriented design, it would not result in a pragmatic code base and would most likely be useless. Extending the functionality of a program built upon a simple design would require a large amount of code duplication if not a complete rewrite. It is important to note that we have shown that with a proper object oriented design the visualizer can be extended to to visualize a new type of network with little effort.

6.2.3 Visualizing Network Data

The visualization of the nodes and links of a network can be done in many ways using modern graphical languages. The challenge is to create a balance between the quality and the practicality of the graphical objects used. One has to consider that networks can be large and a poor choice of graphical objects could compromise the scalability of the visualization.

We have chosen the representation of nodes, links and packets to retain graphical viewing quality while not sacrificing the scalability of the system. We have shown that by using our choice of graphical objects acceptable performance is achieved, even when visualizing large networks.

6.2.4 Data Interface

Trace data that can serve as input to a visualization program can become cumbersome to work with. It is not uncommon for large trace files to be generated by network simulators. To address this issue we devised a novel way of transferring data between the simulator and the visualizer. To hide the complexity of this data transport model we implemented a set of libraries that can be integrated with a simulator to transfer data to the visualizer with minimal complexity. Since not all simulators can link with C++ libraries a different library needs to be created for each different language/platform.

6.2.5 Network Performance Visualization

One area that warrants more investigation is what we refer to as “Network Performance Visualization”. We touched on this subject in Chapter 4 where we explained what certain

colours mean on nodes, links and packets. There may be other aspects that one would like to visualize when monitoring the performance of a network of which we only implemented visualizations for nodes, links and packets. These performance criteria might differ from simulation to simulation and therefore we cannot implement a global metric that encapsulates the performance of all possible networks. For example some performance metrics might be based on how much flow each link carries, while others base it on how many paths traverse each link that is capacitated with a certain amount of bandwidth.

Different types of networks require different performance criteria and although implementing them all would properly demonstrate the power of using a network visualization tool, it would be a daunting task. One would have to develop a user interface that is highly customizable so that a user can configure what he wants to see, be they paths through a node or bandwidth carried on a link. We feel that although this can be accomplished it would require a considerable amount of work to make such a system sufficiently configurable to be of proper use, and we left it for future studies.

6.3 Final Remarks

Today's networks deliver guaranteed quality of service to their users. Users pay to use a network and they expect to receive the service they paid for. With the deployment of 3G networks, and the quality of service they promise, network engineers are turning to more advanced methods and algorithms to make sure that they are using the network infrastructure to its fullest. We are convinced that network visualization can be used to aid in the design and management of such networks, although the use of network visualization to the level expressed in this thesis is still unusual.

When these types of networks become more popular, network operators will have to use more advanced tools to effectively identify problem areas inside a network as bandwidth demands increase. A visualization tool can greatly assist in identifying such network deficiencies.

Appendix A

The Vertex Shader Program Used to Achieve Packet Animation



```
struct VertexIn { // Define our input vertex format
    float4 position :POSITION;
    float4 colour    :COLOR;
    float4 texCoord  :TEXCOORD0;
};
struct VertexOut { // Define our output vertex format
    float4 hPosition :POSITION;
    float4 colour    :COLOR;
    float2 texCoord  :TEXCOORD0;
};
// Main vertex program starts here.
// Returns      : A vertex with format VertexOut
// Parameter 1: The input vertex with format VertexIn
// Parameter 2: The translation direction for the packets
// Parameter 3: The current world matrix
VertexOut main( VertexIn vtxIn,
                uniform float3 linkDir,
                uniform float4x4 worldMatrix ) {
    VertexOut vtxOut;
    float3    vertexTranslation;

    // Setup the translation vector.
    vertexTranslation = linkDir * ( vtxIn.position.w );
```

```
// Translate the input vertex into position
vtxIn.position.xyz = vtxIn.position.xyz + vertexTranslation;

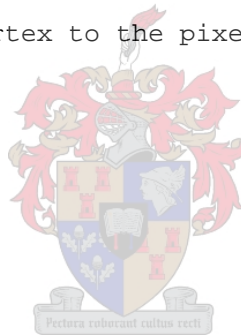
// Restore the piggy back value to 1.0f
vtxIn.position.w = 1.0f;

// Do world transformation, rotation and perspective correction
vtxOut.hPosition.xyzw = mul( worldMatrix, vtxIn.position );

// Set output vertex colour
vtxOut.colour = vtxIn.colour;

// Set output vertex texture coordinates
vtxOut.texCoord = vtxIn.texCoord;

// Return the output vertex to the pixel shader
return vtxOut;
}
```



Bibliography

- [1] Cg Toolkit. [Online]. Available: <http://developer.nvidia.com/cg>
- [2] F. N. Paulisch, "The Design of an Extendible Graph Editor," 1993.
- [3] Silicon Graphics. [Online]. Available: <http://www.sgi.com/>
- [4] J. Bertin, "Graphics and Graphic Information Processing," *Walter de Gruyter, Berlin*, 1980.
- [5] R. A. Becker, S. G. Eick, and A. R. Wilks, "Visualizing Network Data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 1, pp. 16–28, 1995. [Online]. Available: <http://citeseer.nj.nec.com/becker95visualizing.html>
- [6] K. C. Cox, S. G. Eick, and T. He. (1996) 3d Geographic Network Displays. [Online]. Available: <http://citeseer.nj.nec.com/article/cox96geographic.html>
- [7] T. He and S. G. Eick, "Constructing Interactive Network Visual Interfaces," *IEEE Computer Graphics and Applications*, vol. 16, no. 2, pp. 69–72, 1996. [Online]. Available: <http://citeseer.nj.nec.com/eick96aspects.html>
- [8] D. Estrin, M. Handley, J. Heidemann, S. McCanne, Y. Xu, and H. Yu, "Network Visualization with the VINT Network Animator Nam," *Technical Report 99-703, University of Southern California*, 1999. [Online]. Available: <http://citeseer.nj.nec.com/article/estrin99network.html>
- [9] National Laboratory for Applied Network Research. [Online]. Available: <http://moat.nlanr.net/>
- [10] J. Brown, A. McGregor, and H.-W. Braun. Network performance visualization: Insight through animation. [Online]. Available: <http://citeseer.nj.nec.com/brown00network.html>
- [11] GL utility tool. [Online]. Available: <http://www.glut.org/>
- [12] I. Bruss and A. Frick, "Fast Interactive 3-D Graph Visualization," *Lecture Notes in Computer Science*, vol. 1027, pp. 99–150, 1996. [Online]. Available: citeseer.ist.psu.edu/bruss96fast.html

- [13] P. Gajer, M. Goodrich, and S. Kobourov, "A fast multi-dimensional algorithm for drawing large graphs," 2000. [Online]. Available: citeseer.ist.psu.edu/gajer00fast.html
- [14] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, "Surface reconstruction from unorganized points," *Computer Graphics*, vol. 26, no. 2, pp. 71–78, 1992. [Online]. Available: citeseer.ist.psu.edu/article/hoppe94surface.html
- [15] B. Monien, F. Ramme, and H. Salmen, "A Parallel Simulated Annealing Algorithm for Generating 3D Layouts of Undirected Graphs," in *Proc. 3rd Int. Symp. Graph Drawing, GD*, F. J. Brandenburg, Ed., no. 1027. Berlin, Germany: Springer-Verlag, 20–22 1995, pp. 396–408. [Online]. Available: citeseer.ist.psu.edu/article/monien96parallel.html
- [16] A. Kumar and R. Fowler, "A Spring Modeling Algorithm to Position Nodes of an Undirected Graph in Three Dimensions." *Technical Report Department of Computer Science, University of Texas – Pan American*, 1996.
- [17] T. Kohonen, *Self-Organizing Maps*, 1st ed., ser. Information Sciences. Springer, 1995, no. 30.
- [18] Intel Pentium 4 SSE2 Instruction set. [Online]. Available: <http://www.intel.com>
- [19] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Addison-Wesley, 1997, no. 30.
- [20] A. S. Evans, "Reasoning with UML class diagrams," in *Second IEEE Workshop on Industrial Strength Formal Specification Techniques, WIFT'98, Boca Raton/FL, USA*. IEEE CS Press, 1998. [Online]. Available: <http://citeseer.ist.psu.edu/92968.html>
- [21] M. Segal and K. Akeley, "The OpenGL Graphical System: A specification," 1999. [Online]. Available: <http://www.opengl.org>
- [22] K. Fall, "Network Emulation in the VINT/NS Simulator," *Proceedings of the Fourth IEEE Symposium on Computers and Communications*, 1999. [Online]. Available: citeseer.ist.psu.edu/fall99network.html
- [23] D. Tantiprasut, J. Neil, and C. Farrell, "Asn.1 protocol specification for use with arbitrary encoding schemes," *IEEE/ACM Trans. Netw.*, vol. 5, no. 4, pp. 502–513, 1997.