A graph editor for the VTK/Graph

Nahomi Ikeda
MSc Distributed Multimedia Systems
2003/2004

Summary

The aim of the project is to create a tool called a "Graph editor" for the VTK Graph library which is an extension of VTK (Visualization ToolKit). The graph editor enables a user to edit a VTK graph by direct user manipulation. The editor provides a rendering window and Graphical User Interface (GUI) components to support editing functionalities.

Minimum functionalities developed are;

- Select Node
- Add Node
- Delete Node
- Move Node
- Add Edge
- Delete edge
- User interaction e.g. Rotate/Zoom/Pan
- Selection Feedback

Advanced functionalities developed are;

- Undo an action
- Load a file
- Save a file

All functions are written in Tcl scripting language, and the GUI components are supported by Tk libraries; an extension of Tcl. The user is able to edit a graph by interacting with 3-dimensional world coordinate space with user navigation supported by VTK.

This report describes in details how the editor was integrated with the VTK pipeline. It also describes work carried out to assess and evaluate the usability and design of the editor.

Acknowledgements

First of all, my sincere thanks go to my project supervisor, Professor David Duke, who has spent a lot of time helping and encouraging me. His continuous encouragement and invaluable advice enabled me to complete this project.

I would also like to thank Professor Roy Ruddle, my assessor, who gives me valuable advice on the Mid project report and on the project demonstration.

Thanks to my friends who helped to test the system for the evaluation. Their valuable opinions gave a great contribution to improve this project.

I would like to thank my sincere friend, Andrew Kaittany who always gives me warm encouragements and inspiration, as well as invaluable advice and help.

Finally, I would like to thank my family who let me have this great experience and for their continuous encouragement and love.

Table of Contents

Figures

Chapter 1: Introduction

Graphs have been used to model relational data in a wide range of fields and are applied in areas as varied as engineering, business and medicine. In bio-science, graphs are used to describe “Protein interaction networks” [Han et al, 2002] and “genetic maps”. In business, graphs are used to model “workflow” and various other processes. The elements of the workflow graph may include departments, customers, productions and their relationships. [Diguglielmo et al., 2002]. The structural properties such as connectivity and depth showed by a graph are more easily recognised visually than through textual description. [Newbery, 1988] Therefore, graphs can be, and are, used to solve modern complex problems.

Drawing a graph by hand can be tiresome and time-consuming work. It is infeasible to draw a graph with thousand of nodes and millions of edges by hand. Moreover, changing layout style can only be done by drawing another thousand nodes and edges. Graph visualization addresses these problems by automating the layout and depiction of graph structures. However, the problem with graph visualization is that when a graph drawer wants to manipulate the graph, he/she has to either implement low-level manipulation (programming) or change the graph data and repeat the visualization process. This is where a graph editor is wanted. The graph editor was invented to enable a user to manipulate a graph directly through the rendering window. It is more like a user interface or a plug-in to a graph visualization system.

1.1 Aim

This project aims to build a tool, called a graph editor for the VTK(Visualization ToolKit) Graph library. The editor must provide simple functionalities such as adding and deleting nodes and edges to modify a graph. Additionally, advanced functionalities such as undo actions are desirable if time permits. The tool developed in this project is intended to be used as a component of a sophisticated visualization library, and it is sensible to assume that users will be computer literate and familiar with direct manipulation interfaces.

1.2 Project Objective and Minimal Requirements

The editor should have a display showing a graph and Graphical User Interface (GUI) enabling a user to implement functions in order to manipulate the graph. The functions which are to be implemented have been divided into two sets. The editor should provide, at least, the following functionalities;

- **Adding new nodes and edges**
- **Deleting existing nodes and edges**
- **Pan and Zoom functions**

- **Colouring of nodes and edges based on attributes**
- **Initial 3D view**

If the time permits, the following advanced functions will be implemented.

- **Bend points in edges**
- **Constrain motion of points (Editing on a fixed-size grid)**
- **Grouping/Nesting nodes and edges**
  (Create/Work with groups of nodes and edges.)
- **Labelling (Adding / Editing text on nodes and edges)**
- **Rotation (3D)**
- **Copy/ Cut / Paste**
- **Undo/Redo for most editing changes**

In addition to the functional objectives, successful completion of this project should involve;

1. Greater familiarity with C++ programming language
2. An introduction to Tcl/Tk scripting language
3. Deeper knowledge of dataflow visualization systems
4. Experience of using VTK graph library
5. Experience of creating interactive computer graphics applications.
6. Better understanding of visualization system: projection

1.3 Key Challenges

The key point of this project is to understand how a data flow visualization pipeline works, the relationship between the graph data and visualization process, and the available components of the VTK and VTK Graph library. The ability to understand the VTK source codes written in C++ is essential. Since this project is implemented using Tcl script language, competence in Tcl/Tk is necessary. Although usability is not the primary objectives of the project, attention must be paid to designing the GUI.

VTK graphs are layed out in 3-dimensional world coordinate space (WCS). However, a user has to interact with the graph through 2-dimensional space, which is a projection screen of the 3D space. This means, a user has to be able to do a direct manipulation in 3D space through a 2D space. The key point here is how the system allows the user to interact with the object drawn in 3D space through the 2-dimensional projection onto the screen.

1.4 Project Methodology

While the aim of this project is to create a new tool, it is carried out based on an extension of existing software system; the VTK (Visualization ToolKit). This project is therefore implemented by following a combination of the standard software engineering waterfall model and the Component-based software engineering model mentioned by [Sommerville, 2004]. The Waterfall model takes the fundamental process activities which include *Requirements analysis and definition*, *System and software design*, *Implementation and unit testing* and *Operation and maintenance*. The Component-based software engineering model is based on the existence of a significant number of reusable components. [Sommerville, 2004] So, this approach includes *Component analysis* and *System design with reuse*. The model followed is shown in Figure 1.1.

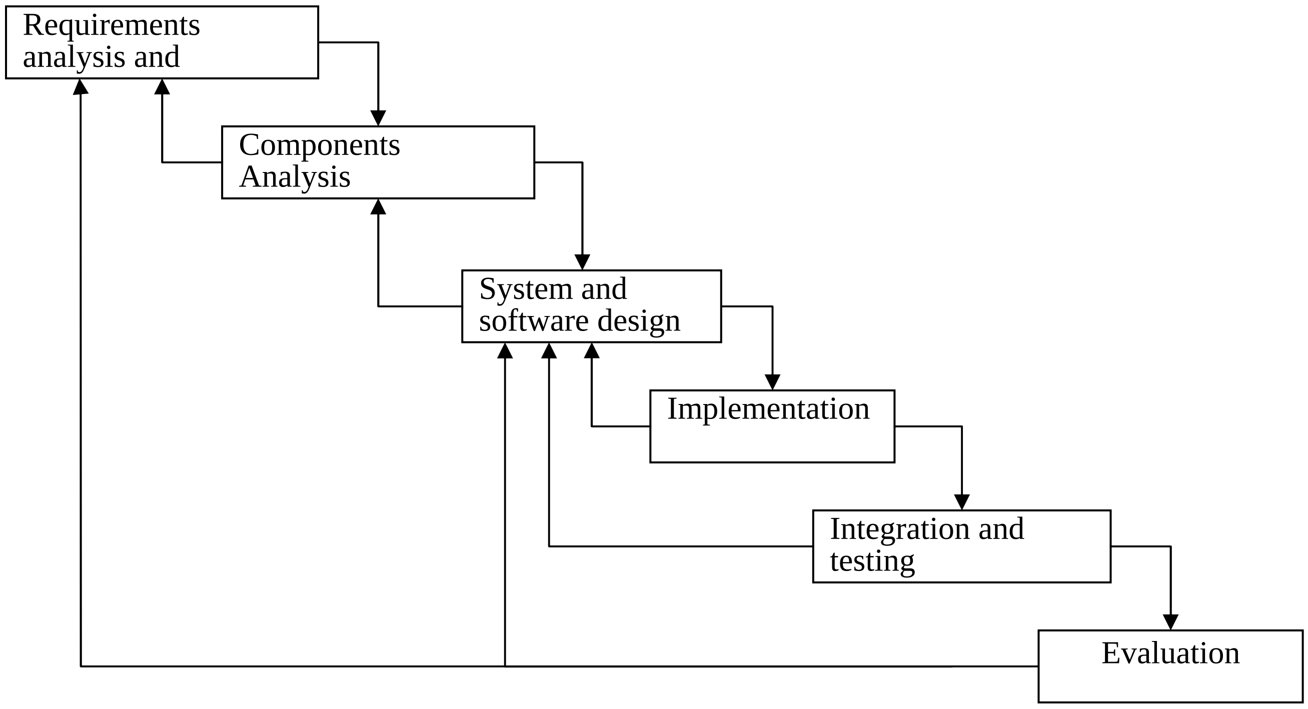


Figure 1. 1: The design of software engineering model

1. Requirements analysis and definition

This phase includes analysis of other existing applications (e.g. Graph drawing libraries and toolkits) and background research to give a clear definition of essential functionality for a graph editor.

2. Components Analysis

As the graph editor is established based on VTK Graph library which is an extension of VTK, analysis and study of existing classes has to be done in this phase.

3. System and software design

This phase will establish the overall system architecture. Software design involves identifying and describing the fundamental software system abstraction and their relationships, and including how existing components will reused and/or extended.

4. Implementation

During this phase, the software design is realised as a set of programs. [Sommerville, 2004] If any problems with requirements or reasons to change the original design are found at this stage, it is possible to go back to the third stage and re-design the system; “System and software design”.

5. Integration and testing

Programming may be divided into small pieces called units. However, it is expected that the software required here will take the form of a single script, so this will be a single unit. *“The individual program units are integrated and tested as a complete system to ensure that the software requirements have been met.”* [Sommerville, 2004] Again, if any problems are identified, it can go back to stage 2.

6. Evaluation

Alternatives or improvements can be found from the result of evaluation. In case where there are needs of re-design the system, or re-set the system requirements, the process is interacted with them deeply.

The practical developing process may not be clearly divided into these 6 stages. The processes interact with each other. “The software process is not a simple linear model but involves a sequence of interactions of the development activities.” [Sommerville, 1992]

As requirements are divided into two sets, “System design” and “Implementation” processes are repeated for each set.

Chapter 2: Background Research

This chapter begins by outlining graph theory and terminology, and follows this with a description and discussion of current available applications working with graphs. Then introduce VTK libraries and graph libraries which are used as a tool to build the system followed by the introduction of Tcl scripting language and its extension Tk.

2.1 Graph theory and terminology

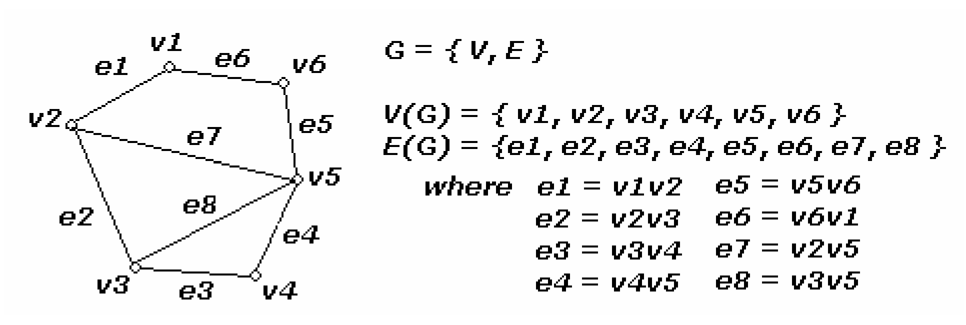
This section introduces mathematical graph theory and key terminologies as well as the layouts used to form a graph readable and understandable.

2.1.1 Graphs

A graph is an abstract structure that is used to model a relation. [Harel&Koren,2002] It is mathematically defined [Chartrand& Lesniak, 1996] as following.

"A ***graph*** G is a finite nonempty set of objects called ***vertices V*** together with a (possibly empty) set of unordered pairs of distinct[1] vertices of G called ***edges E***. "

The graph is typically written $G = \{V, E\}$ and the vertex set and edge set are referred to $V(G) = \{v_1, v_2, ..., v_n\}$ and $E(G) = \{e_1, e_2, ..., e_m\}$ respectively. The edge $e = \{u, v\}$ or $e = uv$ connects vertices *u* and *v*. If *e* is an edge of a graph $G(e \subseteq E(G))$, then *u* and *v* are *adjacent vertices*.[Chartrand& Lesniak,1996 (10)] The summary is shown in Figure 2.1



G = { V, E }

V(G) = { v1, v2, v3, v4, v5, v6 }
E(G) = {e1, e2, e3, e4, e5, e6, e7, e8 }
where e1 = v1v2 e5 = v5v6
e2 = v2v3 e6 = v6v1
e3 = v3v4 e7 = v2v5
e4 = v4v5 e8 = v3v5

Figure 2. 1: Graph *G*

Vertices are also called *Nodes*. In this project report, the term "Node" is used in preference to vertex.

[1] If the vertices are not distinct, the edge is called *loop* and a graph which has such loop is called a *loopgraph*. While a graph with an edge which has the same pair of vertices is called *multigraph*. In this report, a *graph* means a simple graph without any loop and multiple edges.

A ***subgraph*** of $G = \{V, E\}$ is a graph $G' = \{V', E'\}$ such that $V' \subseteq V$, $E' \subseteq E$ and for each edge uv in E', $\{u, v\} \subseteq V'$ [West, 2001]. "*A **cycle** is a graph with an equal number of vertices and edges whose vertices can be replaced around a circle*". [West, 2001] That is, if it is possible to come back to the start node through passing all nodes, the graph is a *cycle.* A graph which has a *cycle* as a *subgraph* is called a ***cyclic graph***, otherwise the graph is ***acyclic.*** Figure2.2 shows (*a*)a cycle, (b)cyclic graph and (c)acyclic graph.

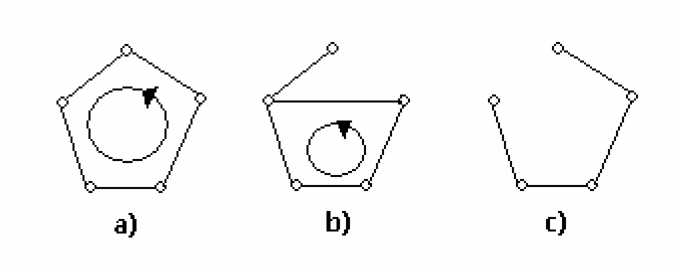


Figure 2. 2: a) cycle, b) cyclic graph and c) acyclic graph

Graphs can be divided into two groups by considering their edges; ***undirected graphs*** (Figure2.1) and ***directed graphs*** or ***digraphs***. *Undirected graphs* such as the shown in Figure 2.1, are used to model symmetric relations~~.~~ Their edges are not directed and are normally represented as straight lines. The edge of a *directed graph* is drawn with an arrow, and directed from one node, referred to as "***tail node***", to another, referred to as "***head node***". Considering the edges of a directed graph, a tail node is called "***source node***" and the head node is called "***target node***" or "***destination node***". Meanwhile, when a node is considered, the source nodes of incoming edges are called "***parent nodes***" and the target nodes of outgoing edges are called "***child nodes***". The parent node and child node of Node "v_n" are v_n's ***neighbours*** . The summary of directed graph is shown in Figure2.3.

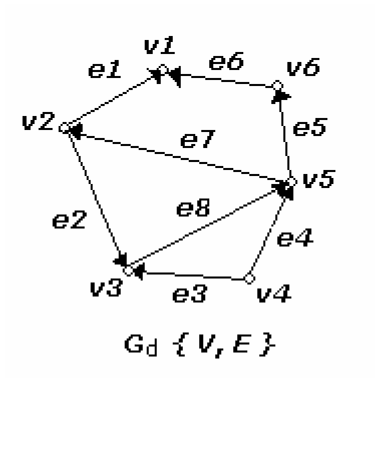


Figure 2.3: Directed Graph G_d

Nodes associated to an edge

Edge	Source	Target
e_1	v_2	v_1
e_2	v_2	v_3
e_3	v_4	v_3
e_4	v_4	v_5
e_5	v_5	v_6
e_6	v_6	v_1
e_7	v_5	v_2
e_8	v_3	v_5

Nodes associated to a node (Neighbours)

Node	Parent	Child
v_1	v_2, v_6	
v_2	v_5	v_1, v_3
v_3	v_2, v_4	v_5
v_4		v_3, v_5
v_5	v_3, v_4	v_2
v_6	v_5	v_1

In a *directed graph*, a *cycle* must be a graph all of whose vertices can be ordered on a circle. This means if it is possible to come back to the start node by following edge direction, the graph is *cycle* (Figure 2.4).

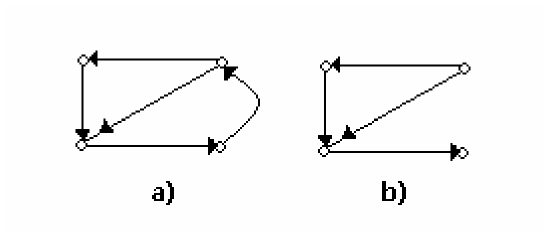


Figure 2. 4: a) Directed cycle, b) Directed acyclic graph

In practical terms, nodes represent *items* in an application (e.g. state, an activity, a program module). The edges represent the relationships between nodes. (e.g. state transition, activity duration, procedure invocation) [Newbery, 1988]. A famous example of a graph representation is the London underground railway map by Harry Beck. The nodes represent the stations, and edges represent the connections between the stations.

2.1.2 Data Structure

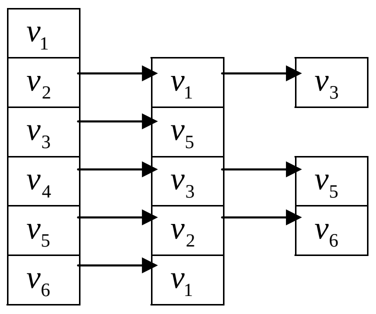
The graph G_d (Figure 2.3) can also be described by means of matrices and lists. One of the matrices is called the *Adjacency matrix.*(Figure2.5 a)). An adjacency matrix is a $n \times n$ matrix where *n* is the number of vertices. If there is an edge from vertex v_i to v_j, the value of a_{ij} = 1, otherwise a_{ij} = 0. Another way of describing a graph is the *Adjacency list* (Figure2.5 b)). This is a list of neighbouring vertices. The size of the structure is *O(n+e)*, where *e* is the number of edges.

From

To

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	1	0	0	0	1
v_2	0	0	0	0	1	0
v_3	0	1	0	1	0	0
v_4	0	0	0	0	0	0
v_5	0	0	1	1	0	0
v_6	0	0	0	0	1	0

a)



b)

Figure 2. 5: a) Adjacency Matrix, b) Adjacency List

The graph implementation in the VTK graph library is based on adjacency lists, but is more complicated. Further details are explained Chapter 3.2.1.

2.1.3 Graph representation and layout

Although graphs are mathematical objects, it is often easier to understand by looking at a drawing of it; this requires a graph layout. Layout is the placement of nodes and edges within some coordinate systems. Many algorithms have been invented to draw graphs more effectively. Here, three basic classes of layout are introduced.

Tree Layout : *"A tree is a connected acyclic graph."* [West, 2001] An undirected connected graph is a tree if it has no cycle. The identification of parent/child relationships depends on the choice of a special node called the *root* node which has no incoming edge. A directed connected graph is a tree if all nodes have one parent apart from the *root* that has no parent. In Figure 2.6, (a) and (d) are trees. (b) is a cycle and (c) is not a tree because one of the nodes has more than one parent.

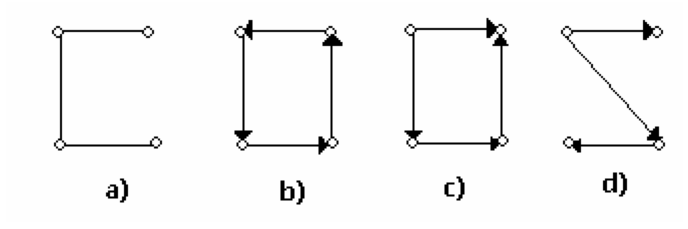


Figure 2. 6: Tree layout

A commonly used tree layout in a computer science positions a *root* on the top and children nodes "below" their ancestor. It can be drawn top-down as well as left-to-right. [Herman et al., 2000] The typical example of a tree layout is the hierarchy of a folder (directory) system in a computer. However, there are many other ways of drawing trees, beyond the scope of this project; see Herman et al for references.

Planar Layout : A planar graph is a graph which does not have the intersection between edges (edge crossing). In Figure 2.7, both (a) and (b) have the same structure but (a) is drawn using Planar layout while b) is not.

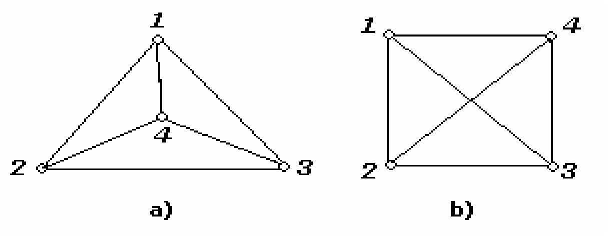


Figure2. 7: Planar layout

<u>Orthogonal layout:</u> Edges of a graph are drawn along x or y coordinates. This means, edges are drawn by either horizontal or vertical line.[Papakostas & Tollis,1998] (Figure 2.8 (a))

<u>Grid Layout :</u> Nodes of a graph are positioned at points with integer coordinates. [Herman et al.,2000] (Figure2.8(b))

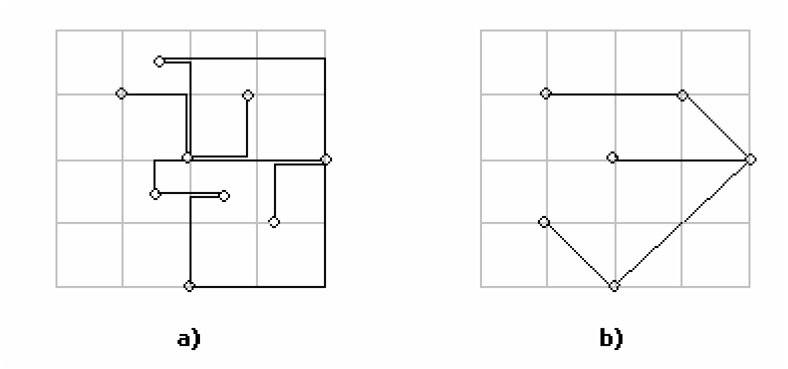


Figure 2. 8: a) Orthogonal layout
b) Grid layout

Although layout algorithms usually treat nodes as mathematical points (i.e. of no size) some layout algorithms also need to consider node size. In addition to layout, graph drawing techniques may also take into account scale, styles and colours. For example, nodes may be represented as dots, circles or cubes. They may be clustered or evenly distributed. Edges may be of varying thickness, length and may be straight or curved. [Herman et al., 2000] Good layout improves the aesthetical representation and usefulness of a graph.

2.2 Working with Graphs

There are many toolkits and libraries available for drawing and manipulating graphs. Although many tools contain a range of functionalities, they can be categorized into Graph Drawing, Graph Visualization or Graph Editing toolkits/libraries by considering their primary focus.

Graph Drawing is concerned with the geometric representation of graphs and networks. It deals with relatively small-medium graphs which can, for example, fit on a page. The focus of Graph drawing is on layout. This usually involves static, traditional node/edge views. The graph community develops new technologies such as layout algorithms and toolkits and also investigates theoretical results on efficiency and limitations of drawing techniques. A good source of these results is the proceedings of the annual symposium on Graph Drawing published by Springer-Verlag.

Graph visualization deals with medium- large graphs which cannot be understood entirely in a single view. For this reason, graph visualization techniques employ a combination of dynamic interaction techniques and multiple views. Recent key issues in Graph visualization are **size** and **usability** of graphs.

[Herman et al.,2000] How should a graph be visualized if it has billions of elements? Is it possible to get viewability and usability at the same time? It is said that 3D layout could be one of the possible solutions, but the problems still remain.

Newbery (1988) defined a **Graph Editor** as follows: "*A graph editor will display the graph on the screen and allow the user to modify the graph by manipulating it directly with a mouse. The user can alter the structure of the graph as well as alter information associated with it. An updated drawing of the graph can then be displayed.* "

Sometimes, a graph editor is considered as a visualization system with direct manipulating functions.

2.2.1 Available applications

A number of Graph visualization libraries and toolkits with editing functions are examined. Some of them are freely available on the Internet and some of them are commercial packages. A summary of each toolkit is given below followed by a discussion of editing capabilities.

Graphic Visualization libraries

- **Swan** [http://simon.cs.vt.edu/Swan/Swan.html]

"**Swan** is a data structure visualization system developed as part of Virginia Tech's NSF Educational Infrastructure project." [Shaffer, 2001] It provides C/C++ program code to be annotated to support visualisation of graphs, including arrays, lists, trees and general graphs with supports for general layout algorithms such as hierarchic layout and circular layout as well as manual layout. The Swan window provides editing functionalities e.g. for adding and deleting nodes and edges and their attributes in 2D coordinate space. It is useful for relatively small graphs.

- **da Vinci** [http://www.informatik.uni-bremen.de/daVinci/]

Developed by Micheal Fröhlich, and Mttieas Werner at Universithy of Bremen, "***da Vinci** is a generic visualization system for automatic generation of high-quality drawings of directed graphs.*"[Fröhlich& Werner,2004] *da Vinci* supports multi-view and multi-graph which allows a user to work on the same graph in different coupled views (windows) or to load many graphs in several uncoupled windows at the same time. The graph editing functionalities are supported by *da Vinci API.* Since the graph data structure of the editor is independent from that of *da Vinci*, only the graph editor is capable for modifying a graph. A wide range of manipulating functions can be implemented in 2D space by click- and- drag interaction.

Graph drawing libraries

- **AGD** (Algorithms for Graph Drawing) [http://www.ads.tuwien.ac.at/AGD/]

This is an object-oriented library implemented in C++ language by using LEDA, a toolkit, providing algorithmic mechanisms in the fields of graph and network problems.[LEDA,2004] Layout algorithms supported by the library can be divided into three types: those for planar graphs, those for hierarchical graph drawing and algorithms within the planarization method. Available platforms are Linux and Windows 95/98/NT. Pre-compiled binaries are available from the web page free of charge.

- **GDtoolkit** (Graph Drawing Toolkit) [http://www.dia.uniroma3.it/~gdt]

GDtoolkit is a graph drawing toolkit composed of the Graph Application Programming Interface (GAPI) written in C++ and the Batch Layout Generator (BLAG). It mainly focuses on drawing entity-relationships and data-flow-diagrams. GAPI and BLAG enable GDtoolkit to provide functions to draw several types of graphs such as trees, flow networks and planar graphs and to automatically draw them according to many different aesthetic criteria and constraints.

Graph editors

- **Tulip** [http://www.tulip-software.org/]

This was created by David Auber, as part of his doctoral work. *"Tulip software is a system dedicated to the visualization of huge graphs.*"[Auber, 2003] The system "*supports graphs with a number of elements (nodes and edges) up to 500.000 on a personal computer*."[Auber,2003] It provides 2-dimensional and 3-dimensional graph editing with user interaction and direct manipulation. i.e. rotate, zoom clustering selecting and adding and deleting nodes and edges. It supports several kinds of graph layout in 2D and 3D.

- **yEd** [http://www.yworks.com/en/products_yed_about.htm#]

A powerful graph editor written entirely in the Java programming language. Graphs are viewed in 2-dimensional space and the editor provides high-level Graphical User Interface which enables the user to modify a graph, e.g. clicking to create a node and dragging to create an edge. yEd makes use of **yFiles library** written in Java and supporting various layout algorithm and functions. It is equipped with powerful features such as automatic label positioning, creating groups of nodes, supporting all file formats and attributes that are customisable to user's need.

- **WilmaScope** [http://www.wilmascope.org/]

This is a Java Based Graph editor invented by Tim Dwyer. Graphs are drawn as colourful images in 3D and can be manipulated directly. Typical functions are all supported such as add/delete node/edge, move, set label, set colour, hide, cluster etc. No graph layout tools are provided.

2.2.2 Graph Editing Functions: A summary

Papakostas et al.(1998) set initial functions for direct graph manipulation as following,

"Software supporting interactive graph drawing features should be able to:
Give the user the ability to interact with the drawing in the following ways:

- *Insert an edge between two specified vertices,*
- *Insert a vertex along with its incident edges,*
- *Delete edges, vertices, or blocks of vertices,*
- *Move a vertex around the drawing,*
- *Move a block of vertices and edges around the drawing."*

Furthermore, most existing graph editor support the same or similar functions. Figure 2.9 is a summary of the functions they provide.

Change the structure	**Change the appearance**	**Change view**	**Layout**
Copy / Cut/Paste	Customize font	Browsing	Tree
Add / Delete	Customize colours	Contracting	Grid
Undo / Redo	Smoothness	Zoom in/out	Hierarchical
Move	Labelling (Naming)	Snap	Orthogonal
Selecting	Various shapes of nodes	Rotating (3D)	Symmetric
Reverse direction	Resize (Shrink/Grow)		Circular (Radial)
(Directed graph)	Width/Length		Spanning
	Grouping(Clustering)		

Figure 2. 9: A summary of functions commonly supported by graph editors

From this investigation, the minimum functional requirements for the graph editor are identified with the following considerations.

1. Can the function be achieved with some combination of other functions?

   E.g. Reversing direction of an edge can be done by deleting and adding an edge.
2. Is the function crucial for graph manipulation?

   E.g. Changing appearance is not a necessary function.

2.3 VTK (Visualization toolkit)

VTK is an open-source, object-oriented software system for computer graphics, visualization, and image processing. [Scheroeder et al,1998] Professional support and products for VTK are provided by Kitware, Inc. [http://www.kitware.com/] Kitware Inc.(2004) describes the advantage of VTK compared with other graphic libraries as follows: " VTK *provides the graphics model at a higher level of abstraction than rendering libraries like OpenGl or PEX. This means it is much easier to create useful graphics and visualization applications."* The core characteristics of VTK are ;

- **Object-oriented modelling and design**

  VTK is an Object-Oriented (OO) toolkit, though its approach is unconventional in that it departs from the strict integration of data and processing that underpins the OO paradigm. That is, while conventional OO paradigm combines data stores and processes into a single object, VTK tends to deal with these separately by defining distinct hierarchies of *Data object* and *Process object*. Both approaches have pros and cons. As a result, a small set of critical operations in VTK is implemented by the former approach. However, the design and implementation of VTK is strongly influenced by OO principles. [Scheroeder et al,1998] This means it takes the advantages of the OO paradigm such as encapsulation, inheritance properties and software reuse, fault containment and complexity reduction. OO modelling also allows the software to incorporate future innovations by having a great flexibility in order to "*support the addition of new material without a significant impact on the existing system*". [Scheroeder et al,1998]

.

- **Using interpreted language interface and compiled language**

  VTK consists of two subsystems: a compiled C++ class library and an "interpreted" wrapper layer which lets a user manipulate the compiled classes using the interpreted language. This structure is used in order make the most of the advantages of both compiler language and interpreted language.

  Generally, compiled languages are higher performing than interpreted language.[Schroeder et al.] So VTK uses a compiled language to create core computational objects. C++ is chosen for this because, firstly, it is widely used and supported by a large selection of development tools and compilers. Secondly *"C++ is strongly typed language, it constrains correct connectivity between process and data object in the visualization network."* [Schroeder et al.] The advantage of using interpreted languages is that they are very simple to write, but yet allows the creation of higher level applications. Interpreted applications can be built significantly faster and have greater flexibility than compiled applications. [Schroeder et al.] VTK supports a number of possible interpreted languages such as Tcl, Java and Python. [Schroeder et al.] That is, “*the advantage of this architecture is that it is possible to build efficient (in both CPU and memory) algorithms in the compiled language C++, and retain the rapid code development features of interpreted language (avoidance of compile/link cycle, simple but powerful tools, and access to GUI tools).*” [Scheroeder et al, 1998-2001]

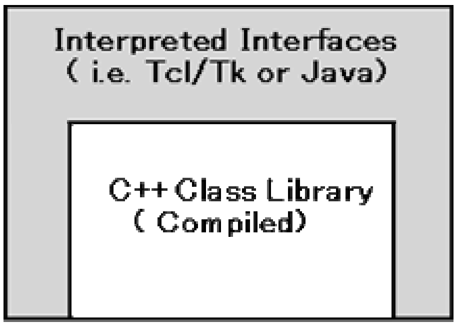


Figure 2. 10: VTK System Architecture

Because wrapping over one hundred thousand compiled classes manually is infeasible, a simplified C++ parser was built to do this automatically. The parser controls the exchange of information between interpreted language and VTK’s C++ method. Now, over 90% of the public methods are wrapped in this way.

- **Multi-platform**

  VTK has been tested on nearly every Unix-based platform, PCs(Windows 98/ME/NT/2000/XP), and Mac OSX Jaguar or later.[Kitware Inc., 2004]

- **Standards based**

  In order to make it easier for others to adopt the system, VTK uses standard components and languages. Thus, C++ was chosen as the implementation language, with scripting support provided via bindings to Tcl, Pythan and java.

- **Portable**

  An application created by using VTK can be run in any platform. VTK provides “*high-level abstraction for 3D graphics that would be independent of new incarnations of graphics libraries.*”[Schroeder et al.]

  The core VTK system” *is independent of windowing systems*”[Schroeder et al.]

- **Freely available**

  VTK has been improved by dozens and perhaps hundreds of developers and users. Free availability of software encourages people to contribute bug fixes, new ideas, algorithms and/or applications. The number of lines of compiled code was 100,000 when it started, and it has, since then, rapidly expanded to nearly 700,000. The VTK library contains nearly 900 classes. It can be seen that the library has developed dramatically. Online-documentation is also freely available.

2.4 VTK Graph library

A fundamental requirement for the project was the use of the VTK graph library. This has been developed as an extension to VTK. As introduced in chapter 2.2, many graph drawing applications have been developed. Most of them are developed on their own libraries. However, combining graph drawing with other visualization techniques and working within a modular framework gives advantages such as software reusability, robustness and scalability. The VTK graph visualization library is an implementation of this model. The library supports manipulation methods a number of filters that provide various layout operations. The source code and documentation are freely available from [www.comp.leeds.ac.uk/~djd/graphs.html]. The VTK Graph library includes the following categories of supports:

2.4.1 Basic support

vtkGraph is the main class and foundation of VTK Graph library. This class supports the storage and methods to manipulate a graph by providing:

"(i) A high-level interface to graph structure, and
(ii) A means of uniquely assigning IDs to nodes and edges while managing efficient storage of the graph and associated data".

[Duke, 2004]

Manipulation of a graph includes adding/deleting node/edge, creating a bend of an edge, copying a graph, etc. Each node and edge is identified by its unique id. A user can access the properties of the nodes and edges e.g. the number of children, the ids of incoming edges and the number of bends.

2.4.2 Iterators

An Iterator is an object that provides access to a collection of values via two operations;

HasNext --- Is there another item in the collection
GetNext --- Return the next item from the collection.

Different implementations (subtypes) of an iterator class allow uniform access to a range of collections. In the graph library, iterators allow access to the nodes and edges of the graph, the neighbours, parents and children of a node and to the in-and out-edges.

2.4.3 Layouts

There are 8 different layouts supported. They all support either 3D or 2D layouts, or both. Layout filters assign a position in geometric space to each node of their input graph.

2.4.4 Geometry Filters

These filters take a graph as an input and output polydata that forms *"a renderable representation of the graph."* [Duke, 2004]. Nodes and edges are processed by different geometric filters. After these filters, the edges are mapped to polydata lines and nodes are located at the points extracted by the filter.

2.4.5 GML(Graph modeling language) file format in VTK/Graph

The GML format was invented to enable graphs to be exchanged between different programs. This was mainly developed for Graphlet drawing system. A GML file contains two discrete sets, one for the nodes and another for the edges. The edges have references to the source node and the destination node. [Rodgers,1998] The "GML" format used in VTK/Graph library for input and output does not completely match the original GML. The major difference is "*that node and edge attributes have to be declared explicitly, and there is provision of a node and edge count and node degree information as well as the specification of the type of graph layout.*" [Duke, 2003] The VTK Graph editor assumes that the files dealt with are in this extended GML format. However, there are various graph description languages available. [Eppstein,2004] It would be possible to support other formats with minimal change to the editor.

2.4.6 Summary

These functions give the following advantage over VTK library graph representation.

1. *"Provide methods to access properties and to have simple means of traversing parts of the graph structure."*
2. *"Determine when two nodes or edges are the same by allocating points and cell IDs "*

[Duke 2004]

2.5 Tcl/Tk

"Tcl (Tool command language) is a simple scripting language for controlling and extending applications".[Ousterhout,1994] In this project, Tcl was used to develop the libraries. Tk is a toolkit which extends Tcl. *"Tcl/Tk is a powerful development environment offering portable GUI development on Unix and Windows systems."* [Ousterhout,1994] The advantages of using Tcl are ;

1. Less code to write

The reason why Tcl was chosen is because of its simple writing style. Figure 2.11 shows comparison of Tcl code to C++ code. As can be seen, Tcl takes much less code to write a program rendering a cube than C++ code does. The bigger the program is, the greater the difference is.

2. Simple

Tcl was originally written in C language. However, a Tcl/Tk application called *wish* provides a high-level interface to GUI programming and Tcl/Tk users do not need to write any C code at all. This hides many of the details faced by a C programmer.[Ousterhout, 1994]

3. Interpreted language

Because it is an interpreted language, the application can be developed and executed without recompiling or restarting the application. [Ousterhout, 1994]

4. String based

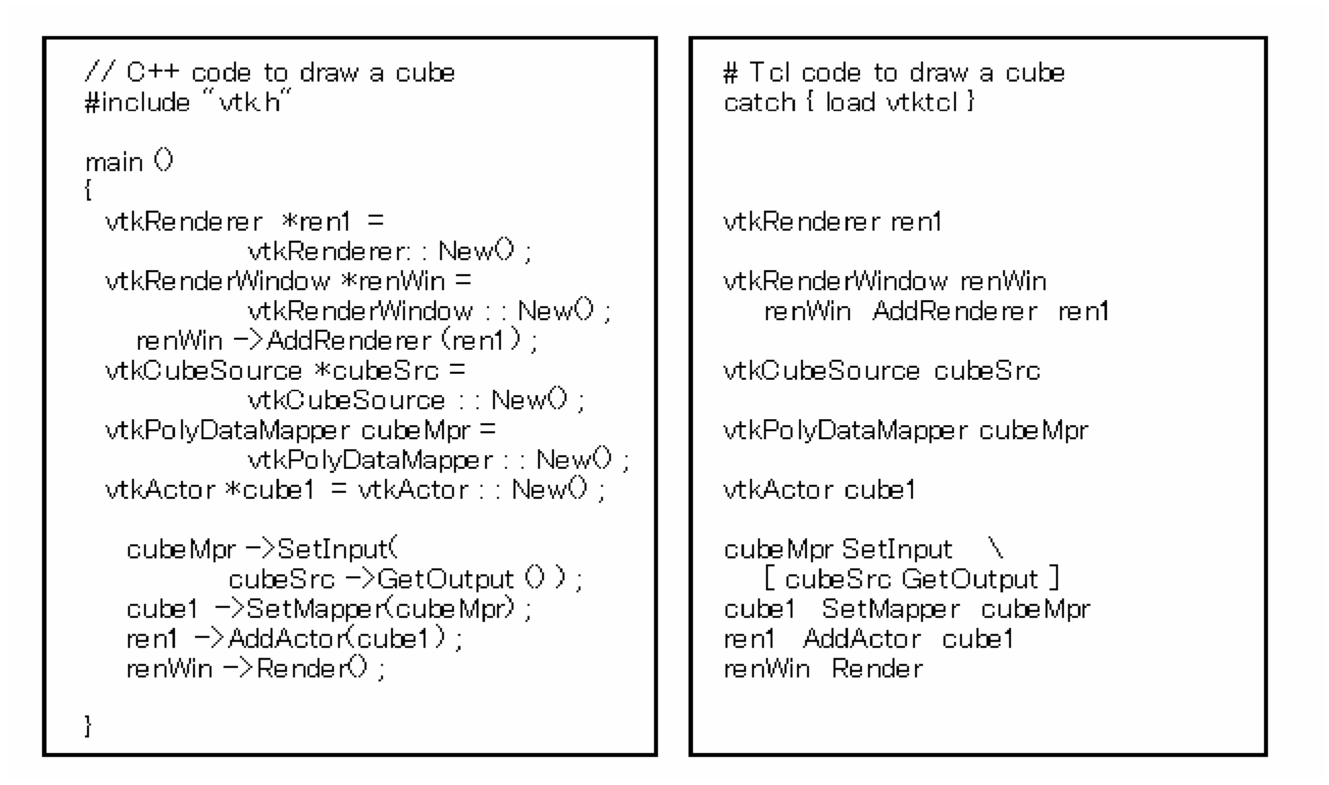
Tcl tends to treat everything as a string. Compared to other programming languages which have various data types such as Int, Float etc. this is convenient as it allows rapid prototyping and construction of commands without the need to implement auxiliary data structures. The advantages of dealing strings are;

- Uniformity --- anything can be constructed from strings, even program!
- Simplicity --- no need to worry about types
- Exploration --- easy to debug/explore program

5. Tk toolkit

Tk toolkit allows creating a powerful GUI application by providing common components such as push-buttons, text widgets, scroll bars etc. Another advantage of Tcl is that the Tk library is well integrated; access to Tk components makes much use of Tcl's data handing. This allows an application to have rapid support for GUI. Tk is window system independent. Thus, applications built in Tcl/Tk are computer-platform independent.

Because of the differences between C++ and Tcl, not all of the methods that are available in C++ are accessible in Tcl and vice versa. In terms of pipeline management, there is nearly a 1-1 correspondence between codes needed in C++, Tcl, Pythen and Java.



```
// C++ code to draw a cube
#include "vtk.h"

main ()
{
  vtkRenderer  *ren1 =
              vtkRenderer: : New() ;
  vtkRenderWindow *renWin =
              vtkRenderWindow : : New() ;
    renWin ->AddRenderer (ren1 ) ;
  vtkCubeSource *cubeSrc =
              vtkCubeSource : : New() ;
  vtkPolyDataMapper cubeMpr =
              vtkPolyDataMapper : : New() ;
  vtkActor *cube1 = vtkActor : : New() ;

    cubeMpr ->SetInput(
             cubeSrc ->GetOutput () ) ;
    cube1 ->SetMapper(cubeMpr) ;
    ren1 ->AddActor(cube1 ) ;
    renWin ->Render() ;

}
```

```
# Tcl code to draw a cube
catch { load vtktcl }



vtkRenderer ren1

vtkRenderWindow renWin
   renWin  AddRenderer  ren1

vtkCubeSource cubeSrc

vtkPolyDataMapper cubeMpr

vtkActor cube1

cubeMpr SetInput   \
   [ cubeSrc GetOutput ]
cube1  SetMapper  cubeMpr
ren1  AddActor  cube1
renWin  Render
```

Figure 2. 11: A comparison between the C++ and Tcl code to render a cube

The figure is from [Scheroeder et al, 1998]

Chapter 3: System Design

Since the VTK Graph editor must interface with the VTK pipeline, it is necessary to understand something of how this pipeline operates in order to make sense of design decision. This chapter starts by discussing how a visualization data flow system works and explains how interactive actions are transformed and data is updated. An algorithm demonstrating 3D user interaction in the VTK graph editor is described. Then the VTK Graph library is introduced; its graph data structure is described, and the process through which nodes and edges are represented within images is explained. In the end of the chapter, overall system structure is described and each step in the pipeline will be detailed.

3.1 VTK Visualization Pipeline

3.1.1 Data visualization

Visualization consists of four processes; **Acquiring data**, **Filtering, Mapping** and **Rendering**. As VTK is object-oriented, these processes are implemented by objects called *process objects* and the data generated or transformed by process objects called *data objects*. Process objects can in turn be classified as *source objects*, *filter objects* and *mapper objects*.[Scheroeder et al.,1998] They carry out transformation of the data into graphic image. Process objects may have inputs and/or outputs. A pipeline is formed when the input of one process object is connected to the output of another.

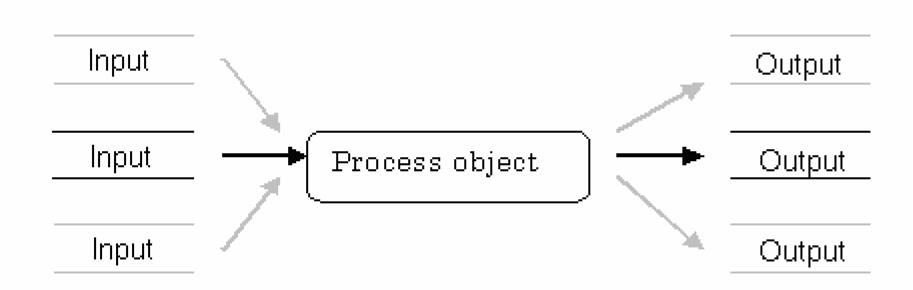


Figure3. 1: Process object; Input and Output may be single or multiple

In the first step of the visualization pipeline, data is acquired by source objects. An example of these objects is *reader* which reads data from specific types of file. Next, filter objects process the input and generate output. Filter objects always require at least one input and output data. Then, the visualization data is converted into graphics data organized into components called *Actor* (a geometric representation such as polygons or points) by *mapper objects* so that it can be rendered and displayed as a graphic image. The Rendering process is done by Graphics Pipeline which implements lighting, viewing and geometry. That is, *mapper objects* provide the interface between the visualization pipeline and graphics pipeline. *Mapper objects* are also used to write out data to a file as well as terminate the visualization pipeline. [Scheroeder et a.,1998] Figure 3.2 shows how data flows through the visualization pipeline.

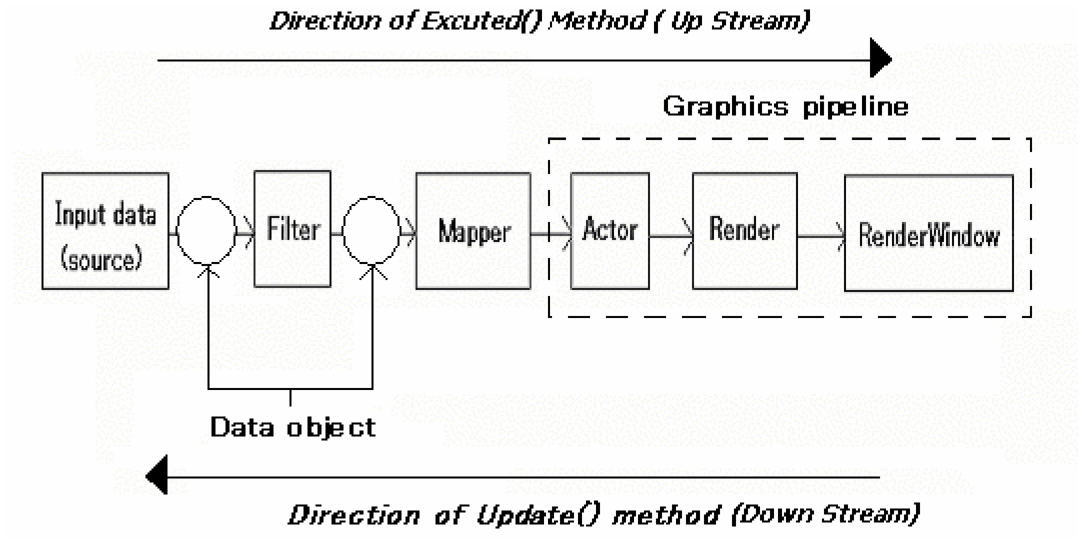

Direction of Excuted() Method (Up Stream)
Graphics pipeline
Input data (source)
Filter
Mapper
Actor
Render
RenderWindow
Data object
Direction of Update() method (Down Stream)

Figure3. 2: Data flow of visualization pipeline

3.1.2 User interaction and data Update

VTK provides a class called *vtkRenderWindowInteractor* for user interaction. This allows a user to manipulate the properties of a scene directly through the rendering window by translating mouse and keyboard events into the relevant operations. The typical examples of user interaction are rotating/zooming/panning the view and actors(objects)as well as picking the actors.

To implement the GUI, the rendering window has to provide the view with the illusion that the user is directly manipulating the objects drawn. This involves translating low-level graphics events (i.e. picking) into higher-level operations (e.g. selecting a node). The management of the visualization pipeline is a complicated process and is not a main concern in this project, therefore it is only briefly described here.

When a render window is interacted, the render window is redrawn. Then, a message is passed down the pipeline, down stream, (Figure3.2.) toward the source objects asking for them to determine whether they need to be modified. At the end of the pipeline, process objects compare their time of last execution with time of last modification, and if they are “out of date” re-execute to regenerate their data. This regeneration cascades back up the pipeline, up stream (Figure 3.2.), until the render window is reached, at which point an up-to-date representation can be redrawn.

3.1.3 3D direct manipulations

Even though a graph is located in world-coordinate space (WCS), the user always has to interact with it through a 2-dimensional projection screen. In the initial plan, it was assumed that the graph functions would be implemented in a 2D WCS formed by projecting the graph onto a plane (Figure3.3). Thus there would be a direct match between the movement in WCS and screen coordinate space (SCS), therefore, allowing the user to avoid dealing with depth control. The original design of this system is as follows; there are two render windows. One shows an overview of the graph. The other shows a part of the graph which is obtained from the projection screen, called the "manipulate screen", of the 3D WCS. A user is able to change the position and orientation of the view point of "manipulate screen" by using a particular VTK class called "vtkPlaneWidget". So the view point can be set wherever the user wants.

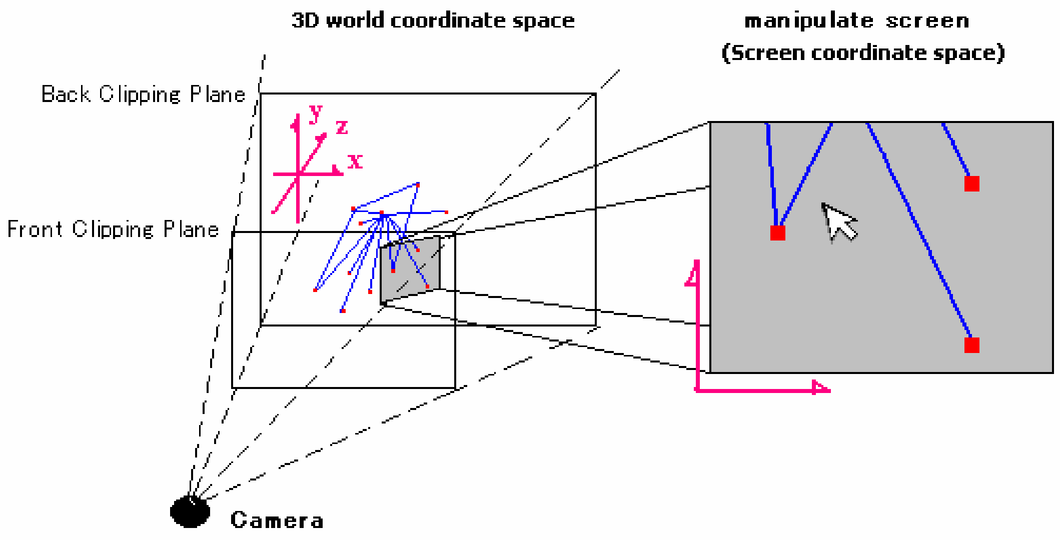


Figure3. 3: Initial plan for user direct manipulation

However, as the project developed, it was found that it is possible to pick a point in the 3-dimensional WCS directly by using another VTK class called vtkPointWidget. This provides the function to translate 2D mouse movement on the screen into 3D movement within world coordinate space.

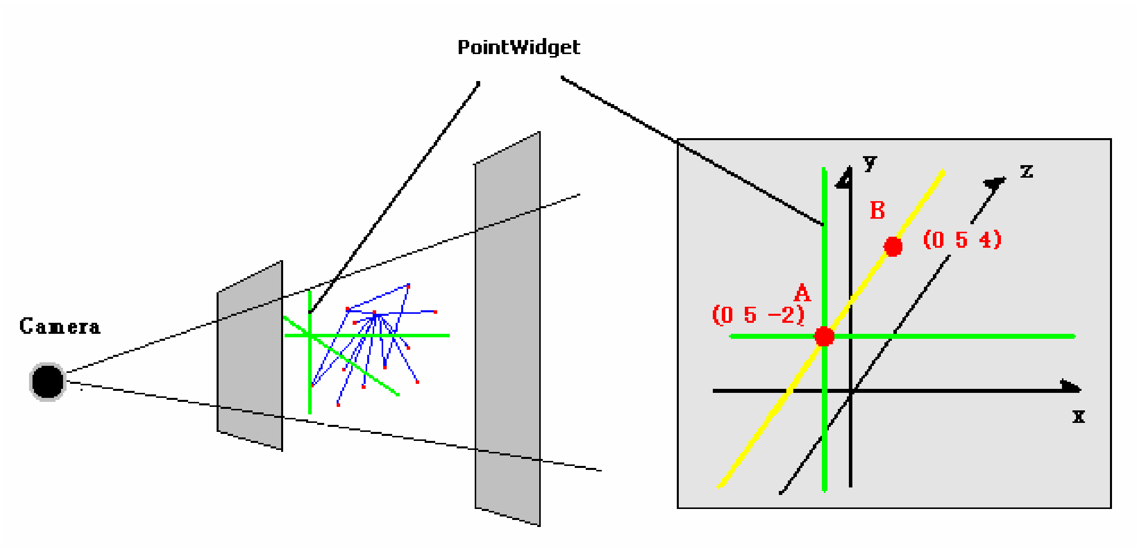


Figure3. 4 : 3D manipulation using PointWidget

The PointWidget lines can be moved in any direction by grabbing a part of the widgets. This means, the user can set the centre of the widget at any point in 3D WDS. However, in terms of usability, exact positioning along a single axis is very difficult. If a user wants to increase or decrease the value of a coordinate of a particular point, the widget has to be slide *along* the relevant axis. For example, when a user wants to move a node from A to B (increasing the value of the z coordinate)(Figure 3.4), he/she must drag the mouse as tracing the yellow line which is parallel to z axis, otherwise it will also change the x and y coordinates. This could be improved by i.e. sticking other axes. However, it is not a critical problem and the outward simplicity of working directly in the 3D WCS was seen to outweigh other concerns.

The PointWidget is a component that works with other interactor styles. This means that while operating PointWidget, the user can also change the view point and manipulate the object (i.e. through picking).

3.2 VTK Graph library and Visualization Pipeline

3.2.1 VTK Graph library

The structure of the VTK graph data object is complex. It consists of a number of inter-linked arrays, in which arrays are used in place of lists to provide an efficient encoding of an adjacency-list representation of a graph. Each node is associated with a collection of edge IDs, i.e. those edges that enter and/or leave the node. Auxiliary data structures provide the source and target node for each edge, the location of each node, data associated with nodes and edges, and for edges that bend, the location of each bend point. As the project involved use of the library, and not modification to it, further detail of the internal implementation is beyond the scope of this report.

3.2.2 Graph Visualization pipeline

Figure 3.5 shows the VTK Graph Visualization pipeline to which VTK Visualisation Pipeline has been applied. The source object is *Reader* which reads graph data from particular file formats such as GML or GXL and stores it into a graph object which internally will represent the graph using array and lookup tables that implement a form of adjacency list. The layout of the graph data is determined by layout filters and then positioned into WCS by geometry filters that map the structural organization of the graph into geometric entities (VTK polydata). Since nodes and edges have different rendering attributes i.e. representation and/or geometric definition, their data are processed separately.

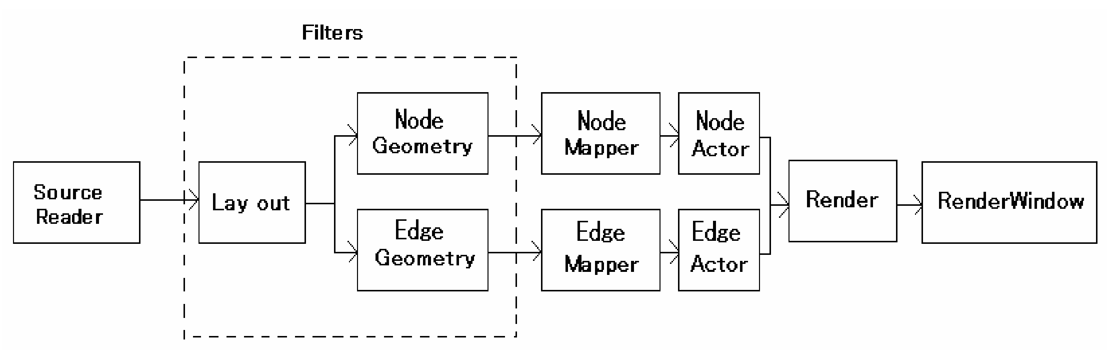


Figure3. 5 : VTK graph Visualization pipeline

3.2.3 User interaction and data update

The algorithm for data update with user interaction is the same as explained in section 3.1.2. The key point for this project is that, if graph data is modified in the pipeline, it must explicitly be marked as modified, and then the render window must be instructed to redraw so that the change to the graph is forced onto the screen. For instance, when the user selects the representation of a node, say, “Node19 ” in the rendering window and issues the ‘Delete’ command, the graph data will be instructed to delete the information about “Node19”. Then the visualization pipeline will be re-executed and the graph will no longer have “Node19”.

It has to be remembered that even though a graph is manipulated, the data file remains as it was. In order to save the change, it is necessary to write up the data modified into a file by process objects such as vtkGML*writer*.

3.2.4 Data structure and representation of Nodes

Node data are visualized by the following processes. Each node has a unique integer ID in order to distinguish it from other nodes. Because Node IDs can span a wide range of values, the graph data object provides efficient storage by automatic mapping from node ID to node Index (Figure 3.7) which is a value within a compact range of 0 to N – 1 where N is the number of nodes. When the graph is transformed into geometric data (Polydata), the graph node index will become the point ID in the corresponding field. The NodeGeometry filter assigns coordinates to each node (Figure 3.7) by mapping node coordinates into point positions so that the nodes can be located in the WCS. Next, Glyph3D filter maps a graphical representation form, in this case cube, to each point. The cube is represented by 8 vertices which are linked to a point ID. The glyph filter can be instructed to generate an additional data array, called InputPointId, giving the point ID of the input that is represented by this glyph, pid. In the end, nodes are represented as cubes (In the table below, the cubes are assumed to be of size 2x2x2 in the world coordinate space).

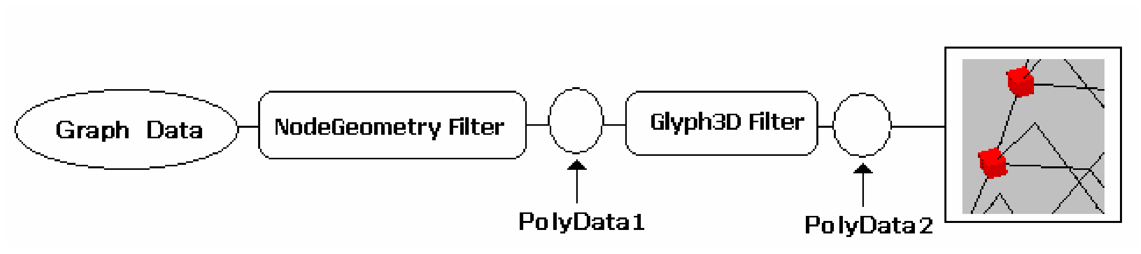


Figure3. 6 : Using cubes to illustrate nodes

Graph Data

Node ID	Node Index
980	0
35	1
180	2
530	3
789	4
3	5
3000	6
…	…
4	N-1

PolyData 1

Point ID	Coordinate
0	(0,0,0)
1	(10,10,10)
2	(8 ,1 ,9)
3	(3, 2, 7)
4	(1, 2 ,5)
5	(9, 0, 2)
6	(6, 1 ,6)
…	…
N-1	(1, 1, 9)

PolyData2

Polydata Index	Coordinate	Ipid
0	(-1, -1, -1)	0
1	(-1, 1, -1)	0
…	…	…
7	(1,-1,1)	0
8	(9,9,9)	1
9	(9,11,9)	1
…	…	…
15	(11,9,11)	1
…	…	…
8 x N-1		N

Figure3.7 : Node data

- **Graph Data**

Node ID … User’s handle for a node.

Node Index … System‘s location for a node --- Node Index may change as nodes are added/removed.

N = the number of Nodes as well as points.

- **PolyData 1 (Representing nodes as points within space)**

Point ID = = Node Index

Each point ID is assigned the coordinate of a point in world coordinate.

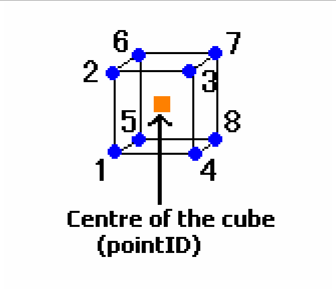
The maximum of point ID number is the number of nodes.

- **PolyData 2 (Representing nodes as glyphs)**

A node is represented as a cube with 8 vertices. The glyph filter can be instructed to generate an additional data array called input pointID, giving, for each point defining the geometry of the glyph the point ID of the input that is represented by this glyph pid.

The coordinates of the vertices are;

1	(-1, -1,-1)
2	(-1,1,-1)
3	(1, 1, -1)
4	(1,-1,-1)
5	(-1, -1, 1)
6	(-1, 1, 1)
7	(1, 1, 1)
8	(1,-1, 1)



VtkCubeSource
X = size 2
Y = size 2

Figure 3.8: Glyph3D for node representation

3.2.5 Data structure and representation of Edges

Edges are also identified by unique IDs. This is again dealt with by linking the edge Ids to edge Indexes. “Technically a graph is directed“ [Duke, 2004], so each edge has source Node and target Node. However, VTK graph library enables each node to recode incoming and outgoing edges, and enables each edge to recode source and target nodes, edges are treated as an undirected edge and represented as straight lines. Edge is also identified by its parent node and child node IDs.

3.3 Overall system structure

The functional model of the structure of the VTK Graph editor is displayed in Figure 3.9.

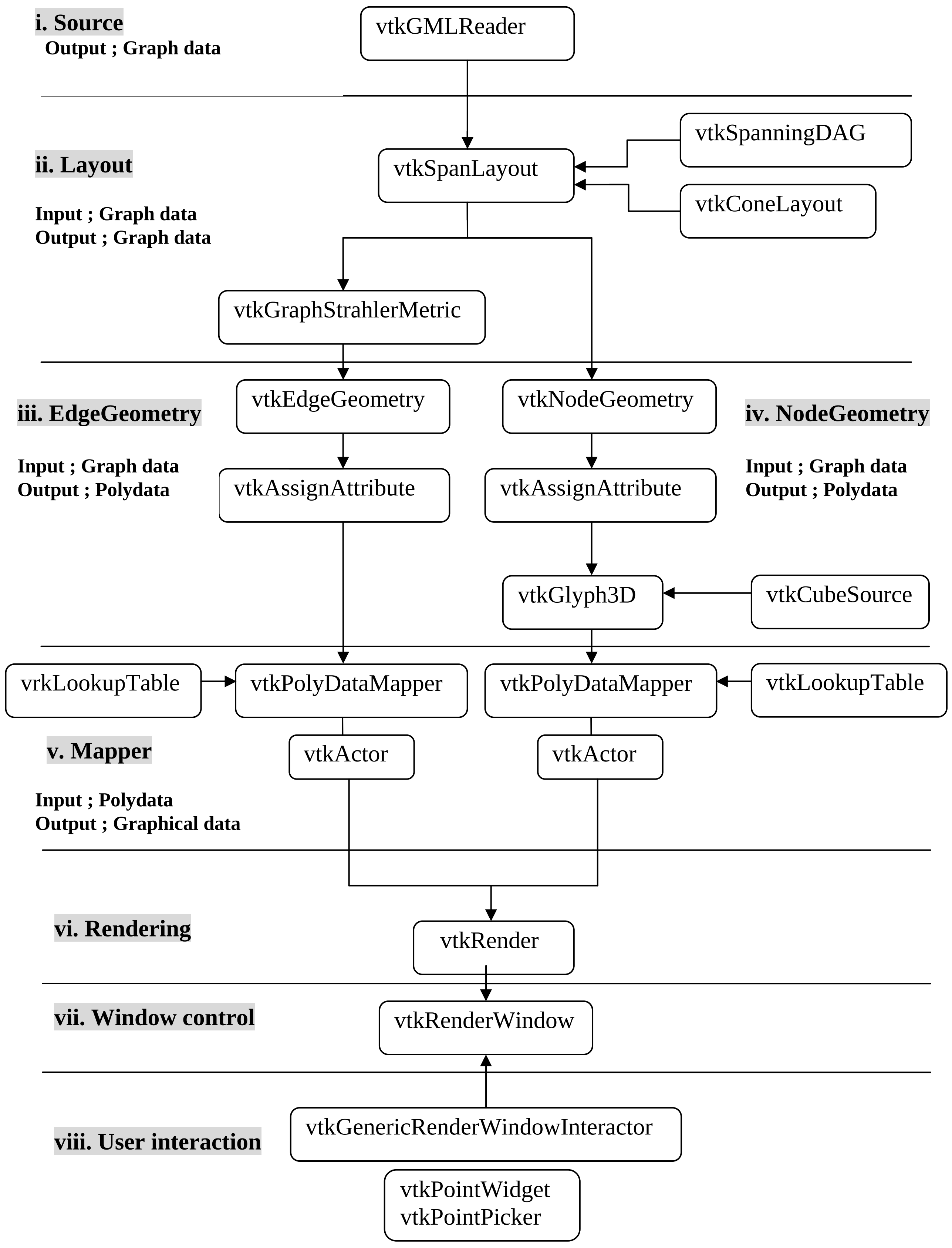


Figure 3.9: Overall system structure

i. Source ; The system starts reading a file by Source Data "vtkGMLReader". This source object supports GML file format only.

ii. Layout ; Then, the layout filter is applied to the graph data. The filter assigns a position to each node of the graph data. Input data for the filter is numeric, graph data and output data is also graph data but now the nodes have coordinate and the graph has "shape". (Figure3.10) SpanLayout was chosen for now this layout filter requires the help of two further tkSpanningDAG(Directed Acyclic subGraph) and ConeLayout .

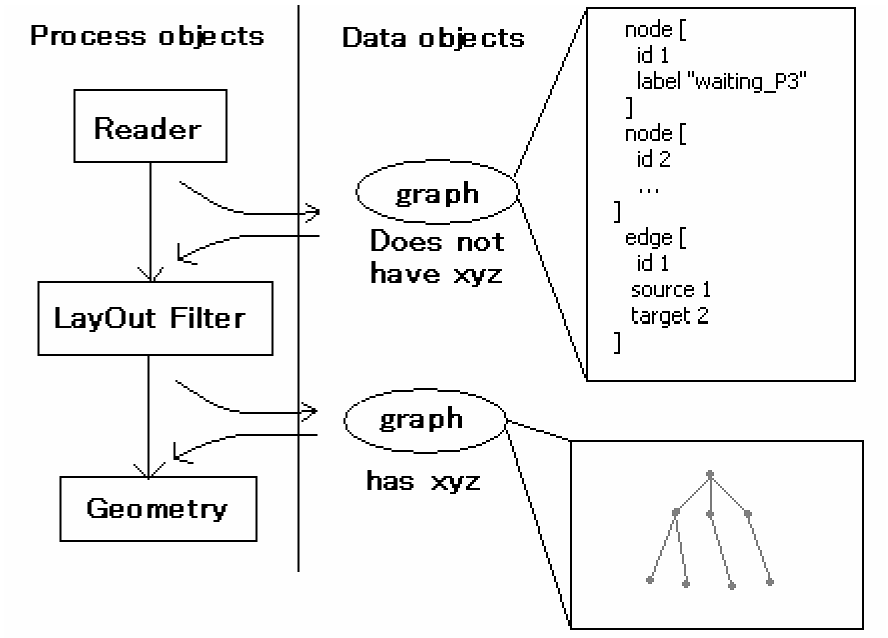


Figure3.10 : Layout Filter

vtkShrahlerMetric is used to vary the degree of the nodes. The Strahler metric is a value assigned to each node of a tree that, in a sense, captures the structural complexity of the sub-tree rooted at that node.[Dukde, 2004] The values are not actually used in the editor, but the filter is included to illustrate how application-dependent properties of nodes can be applied to the graph.

iii. EdgeGeometry ; This part is responsible for drawing the edges as polydata lines. The EdgeGeometry takes graph data as input and produces polydata which has world coordinates.

iv. NodeGeometry ; This part is responsible for positioning the nodes and representing them as a cube in WCS. The detail of the process is described in section 3.2.4.

v.Mapper ; All actors are mapped into WCS. vtkLookUpTable allows attributes data (from either points or cells in the input polydata) to be mapped to colour values (RGB triples) that are then used to colour the rendered geometry.

vi.Rendering ; The actors are rendered. The graph whose edges are represented as straight lines and nodes are represented as cubes portrayed in the rendering window.

vii. Window control ; Window properties such as size are set.

viii. WindowInteractor ; vtkPointPicker is used to select a point by shooting a ray into a graphics window and intersecting with actor's defining geometry - specifically its points.

3.4 Graphical User Interface (GUI)

The GUI is intended to provide direct access to editing functionalities. It decreases the effort required to achieve basic capabilities with the tool and reduces the number of errors the user can make. The graph editor is equipped with GUI components to support the functional implementation and provide information to help user navigation. A Simple Graphic User interface is created using Tk widget set and all GUI components are designed by standard base so that the novice user can adapt to the system smoothly.

3.5 User Navigation and Interaction

Navigation and user interaction is supported by vtkRenderWindowInteraction. This provides functions to allow a user to control camera and actor (object) positions with two interaction styles; Joystick mode and trackball mode. Supported functions are ;

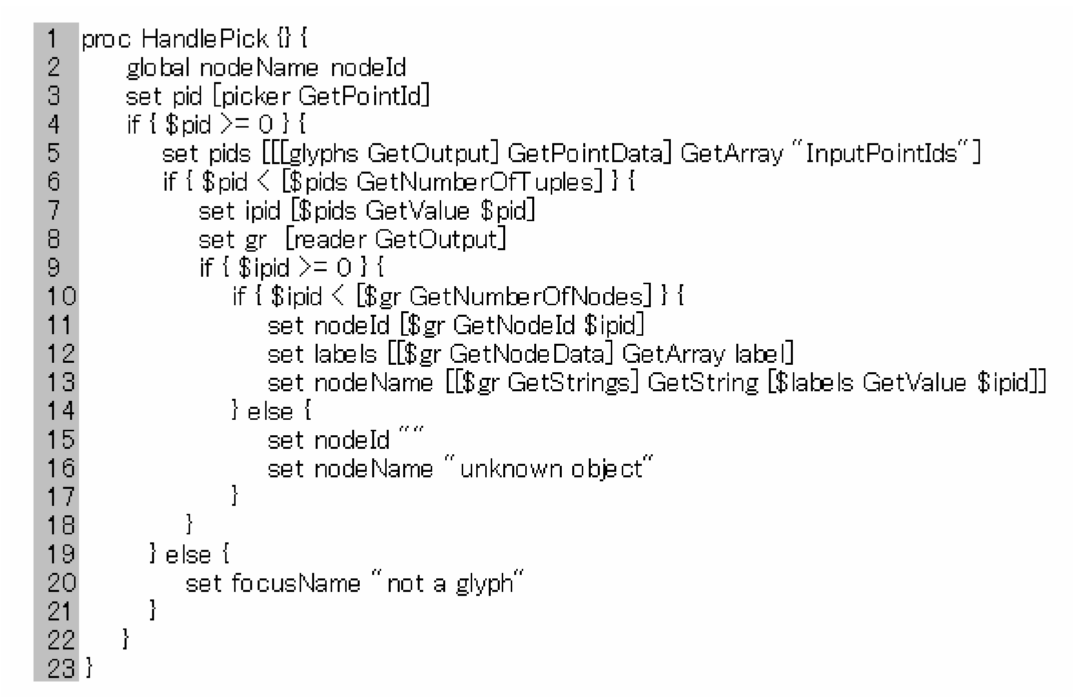
- **Zoom**
- **Pan**
- **Rotate**
- **Pick**

Chapter 4: System Implementation and Testing

This chapter demonstrates the technical aspects of the functions of the VTK Graph editor implemented and indicate the problem faced. The functions require interaction with specific nodes of the graph. For instance, when creating an edge, two nodes must be selected through the rendering window. Selecting a node by user interaction is done by a function called *Picking*. The chapter starts from the algorithm of Picking followed by the other graph manipulating functions and at the end, the implementation of GUI is described.

4.1 Select Node using Picking

What this function does is to enable a user to select a node by direct interaction through the rendering window. In other words, this function connects the node represented in the world coordinate as a graphical image, in this case cube, and the source data (graph data) of the graph so that whenever the graph is changed, the source data is updated. Once the node is selected, a function is applied to only the node selected. This means, this function allows the user to choose a particular node to modify the graph.



```
proc HandlePick {} {
    global nodeName nodeId
    set pid [picker GetPointId]
    if { $pid >= 0 } {
        set pids [[[glyphs GetOutput] GetPointData] GetArray "InputPointIds"]
        if { $pid < [$pids GetNumberOfTuples] } {
            set ipid [$pids GetValue $pid]
            set gr  [reader GetOutput]
            if { $ipid >= 0 } {
                if { $ipid < [$gr GetNumberOfNodes] } {
                    set nodeId [$gr GetNodeId $ipid]
                    set labels [[$gr GetNodeData] GetArray label]
                    set nodeName [[$gr GetStrings] GetString [$labels GetValue $ipid]]
                } else {
                    set nodeId ""
                    set nodeName "unknown object"
                }
            }
        } else {
            set focusName "not a glyph"
        }
    }
}
```

Figure 4.1: Select Node

When a pixel on the display is picked up (by press “P” as set by VTK), it must be automatically connected to a component of the graph represented in the display. This means that the screen coordinates of the selected point must be mapped to a node ID through the following process.

1. Pick is applied to current mouse position.
2. If a point is nearby, ***pid*** will be returned.
3. Check whether a point was picked
4. Returned i*pid* is the *pid* of some point in the polydata in the output of glyph3D (Figure 3.7 Polydata Index)
5. First, find what the pid of the point that was input to produce this glyph (Figure3.7 Ipid)
6. Now, the input pid == node index (Figure 3.7 Node Index)
7. So ask graph what nodeId is for this pid (Figure3.7 NodeId)

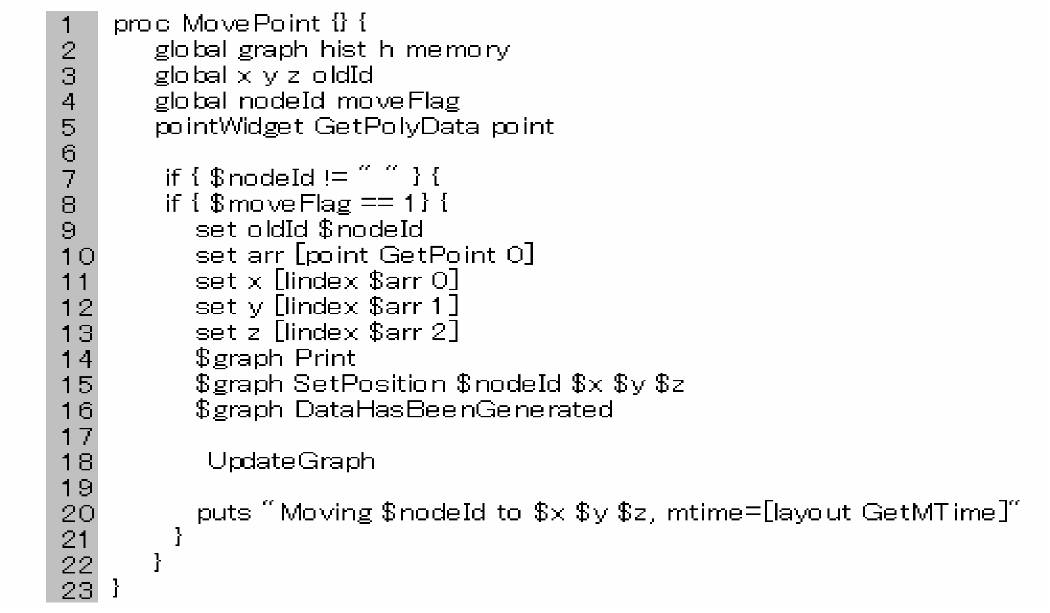
NodeId bar give a reference to a picture that shows this i.e. a screen dump.

4.2 Implementation of minimum requirements

4.2.1 Move Node

"Move Node" enables a user to move a selected node in 3D WCS using PointWidget.

1. Select a node. This returns *nodeId* of the node being picked.
2. Get the new x y z coordinates in the world coordinate space by using PointWidget. Use an array to store the information of new position.
3. Update the position of the node data by means of replacing the coordinate information from the old one to the new one obtained in step 2.
4. Mark the graph as modified and tell the render window to render, so that the pipeline is then updated automatically.



```
proc MovePoint {} {
    global graph hist h memory
    global x y z oldId
    global nodeId moveFlag
    pointWidget GetPolyData point

     if { $nodeId != " " } {
     if { $moveFlag == 1} {
        set oldId $nodeId
        set arr [point GetPoint 0]
        set x [lindex $arr 0]
        set y [lindex $arr 1]
        set z [lindex $arr 2]
        $graph Print
        $graph SetPosition $nodeId $x $y $z
        $graph DataHasBeenGenerated

         UpdateGraph

        puts "Moving $nodeId to $x $y $z, mtime=[layout GetMTime]"
      }
    }
}
```

Figure 4.2: Move Node

4.2.2 Add Node

When the $\boxed{\text{AddNode}}$ button is pressed, a new node is added. This method is created by using the *AddNode* method of the vtkGraph class.

1. The point (xyz coordinate) in the world coordinate space is picked up by using PointWidget. This is where a new node is going to be located. (Practically, place the centre of PointWidget where the new node will be positioned)
2. Press $\boxed{\text{AddNode}}$
3. The graph data is updated with a new node.
4. Id of the new node is generated automatically.
5. A default label for the node is created, so that if the node is selected, a label will be available to display.
6. The new data will be automatically visualized and rendered into the render window

4.2.3 Delete Node

There were two ways to implement "Delete Node" function. The First option was to pick up the node and then press $\boxed{\text{Delete}}$. The second option was to press $\boxed{\text{Delete}}$ followed by picking up the nodes. The First option has an advantage that the user can change the node being picked before implementing "Delete Node". By this operation, the user can avoid deleting a wrong node by mistake. However, if the user wants to delete 10 nodes at once, the user has to repeat picking and deleting 10 times whereas the second option enables the user to delete a number of nodes pressing $\boxed{\text{Delete}}$ only once. Although with the second option the user does not have a chance to re-choose the right node, this problem can be overcome by an undo function. Therefore, it can be said that the second option provides better performance.

The following is the algorithm of "Delete Nodes". This method is created by using the *DeleteNode* method of the vtkGraph class.

1. The node is picked up by handle picking function. This gives *nodeId* of the node being picking.
2. If the node exists, it will be deleted from the graph data and so will edges related to the node. Any associated data for the node is removed.
3. The graph data is updated and reshuffled in order to retain compactness. The new data will be automatically visualized and rendered into the render window.

4.2.4 Add Edge

"Add Edge" function enables a user to create an edge between specific two nodes, source node and target(destination) node. This method is created by using the *AddEdge* method of the vtkGrpah class. The algorithm follows the paint program algorithm which described by [Angel, 2000]

1. The first node is selected by “Select Node”. This node is treated as a Source Node and stored into an array.
2. The second node is selected. If second node is not selected properly, the process goes back to step 1 with the message “There is no edge ”
3. If the second nodeID is different from the first nodeID, it is treated as a target Node and an edge from sourceNode to target Node is created, otherwise the process goes back to step 1.
4. The new data will be automatically visualized and rendered into the render window.

4.2.5 Delete Edge

The algorithm is almost the same as “Add Edge”. When two nodes are selected, check if the edge between these two exists and if so delete it, otherwise the process goes back to step 1. All associated data is deleted from the data set except the nodes associated with the edge. The complexity with deleting edges is that vtkGraph deals with edge as a directed edge. So it is necessary to check the reverse order of source/target node Ids.

```
if { [$graph HasEdge $srcNode $desNode]} {        # check if there is an edge from sourceNode to targetNode
    $graph DeleteEdge $srcNode $desNode

} elseif { [$graph HasEdge $desNode $srcNode]} {   #check if there is an edge from targetNode to sourceNode
    $graph DeleteEdge $desNode $srcNode

} else {                                            #There is no edge between sourceNode and targetNode
    set menustatus "It does not have any edges with $srcNode. Try again"
}
```

Figure 4.3: Delete Edge

4.2.6 Selection Feedback

The default colour of a node is red. When a node is selected, its colour is changed from red to yellow so that the user can see which node has been selected and/or is active. The algorithm of “Selection Feedback” uses an array implemented in the VTK library by *vtkIntArray* to store the colour information of each node. The value is either 0 or 1. If the value = 0, the colour of node is assigned as red, while it is yellow if the value = 1.

Colour

Node ID	Colour scalar
1	0
2	1
3	0
4	0
…	…

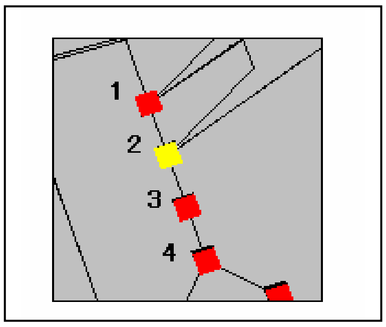


Figure4.4: Change the colour of the node activated

In “Select Node” procedure, when the node is selected, the colour value of the node is changed from its default value 0 to 1. When the user picks another node, the former selected node turns back to red automatically as its value changes back to 0. The program below demonstrates this process.

```
set oldSelection $nodeId
set carray [[$graph GetNodeData] GetArray colour]
                    $carray SetValue [$graph GetNodeIndex $nodeId] 1
            if {$oldSelection != " " && [$graph HasNode $oldSelection]} {
                    $carray SetValue [$graph GetNodeIndex $oldSelection] 0
             }
```

Figure 4.5: Selection Feedback

The same algorithm can be used to change the default colour or shape(cube, sphere etc) of nodes and edges.

4.3 Implementation of Advanced functions

4.3.1 Undo

Undo functions execute the opposite function to the most recently applied function. It has to return the model to the state before the last command. There are several ways of doing this, for example, keeping a snapshot of the state before a command. But this is expensive in terms of memory space. The approach here is to apply an inverse operation to the model to cancel out the effect of the last operator. That is, when “Add Node” is carried out, undo function implements “Delete Node” and when an edge is added, undo function executes “Delete Edge”. To do this, the program needs to store all the information associated with the node or edge modified. For example, in order to retrieve a node deleted from the graph, its nodeId and all edges connected to the node are required. Undo of “Move Node” requires the old position of the node that was moved. The program uses a Tcl list to store this information. Tcl allows the storage of all values including strings and other lists into a list without any special treatment being necessitated. The following code shows how the information of history is stored.

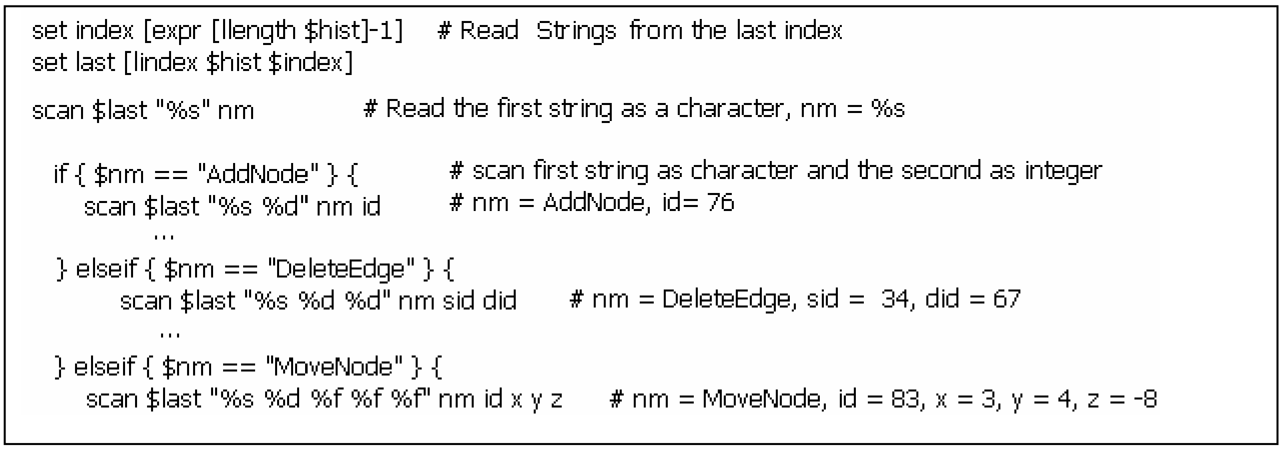
```
set hist [list ]            # Create empty list

lappend hist "AddNode $nodeId "
lappend hist "MoveNode $nodeId $x $y $z "
lappend hist "DeleteEdge $srcNode $desNode "
```

List “hist”

	History
0	AddNode 76
1	MoveNode 83 3 4 -8
2	DeleteEdge 34 67

Figure 4.6: History storage for Undo

When Undo is pressed, the system reads the last index of the list and scans the strings stored. The strings are then identified by means of assigning variables. Figure 4.7 shows the process of scanning and the result after assigning.



```
set index [expr [llength $hist]-1]    # Read  Strings  from the last index
set last [lindex $hist $index]

scan $last "%s" nm              # Read the first string as a character, nm = %s

  if { $nm == "AddNode" } {          # scan first string as character and the second as integer
     scan $last "%s %d" nm id        # nm = AddNode, id= 76
          ...
  } elseif { $nm == "DeleteEdge" } {
        scan $last "%s %d %d" nm sid did      # nm = DeleteEdge, sid =  34, did = 67
          ...
  } elseif { $nm == "MoveNode" } {
     scan $last "%s %d %f %f %f" nm id x y z      # nm = MoveNode, id = 83, x = 3, y = 4, z = -8
```

Figure 4.7 Implementation of Undo

“Undo” of “Add Node”, “Add Edge” and “Delete Edge” functions are relatively straight forward. It basically involves carrying out the corresponding negation of the action; hence, “Delete Node”, “Delete Edge” and “Add Edge” respectively will undo the 3 functions above. However, “Undo” of “Delete Node” and “Move Node” need more work.

- **<u>Undo of “Delete Node”</u>**

  When deleting a node, the edges associated with it are also deleted. This means, when recovering a deleted node, not only the node but also the deleted edges have to be retrieved. Again, Tcl lists are used to store all the information of neighbours of the node, and vtkIdIterator is used to obtain the neighbour Ids. In order to distinguish edge direction, neighbouring nodes are stored in two lists, one for Parents NodeIds and the other for Children NodeId. When retrieving the edges, nodes stored in Parent’s list and Child’s list become source nodes and target nodes respectively. The algorithm for undoing a “Delete Node” action is;

1. Get the neighbours’ Id
2. Store parent’s Ids to ParentList, children’s Ids to ChildList
3. Store the ParentList to the list of source nodes (sourceList), and ChildList to the list of target nodes (targetList). (Figure4.8)
4. Pass sourceList and targetList with the node Id to Undo procedure (method) to retrieve the edges.

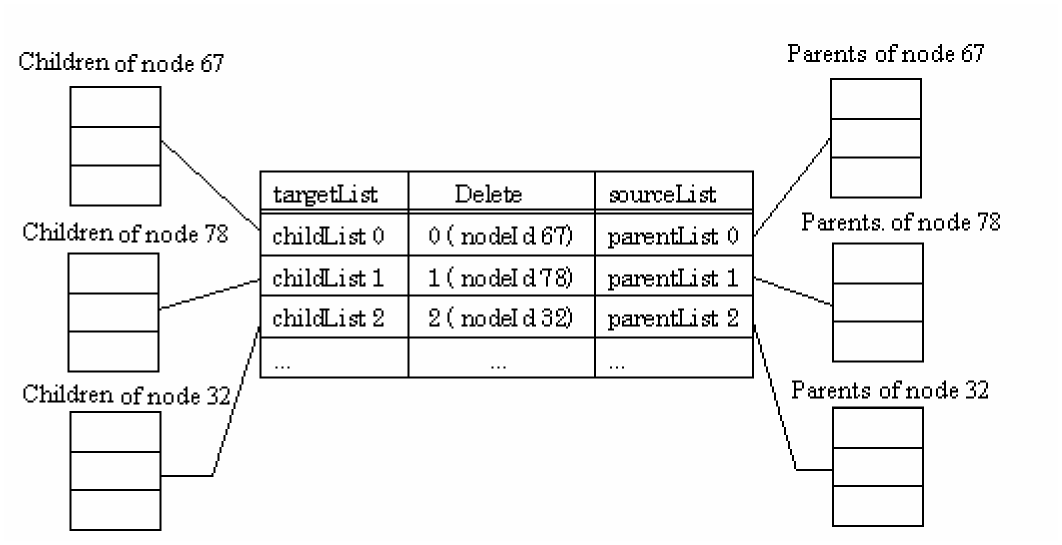
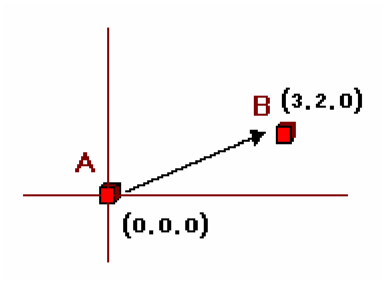


Figure 4.8: The structure of dealing the information for Undo a "Delete Node"

- **<u>Undo of "Move Node"</u>**

A node can be moved by grabbing and dragging from a start point to an end point. The problem with the Undo for "Move Node" is how to identify the endpoint. Figur4.9 shows the example to explain more details. The node N has been moved from point **A** to point **B**. It can be seen visually that the process is one step from A to B. In practice, however, the node is passed every pixels from A to B. Figure4.9 shows the record of the movement. If "Undo" retrieve one step before the end point, the node is moved just one pixel backward.



```
Moving 0 to 0.074718 0.074718 0, mtime=387
Moving 0 to 0.149436 0.074718 0, mtime=387
Moving 0 to 0.224154 0.074718 0, mtime=387
Moving 0 to 0.37359 0.149436 0, mtime=387
                    ...

Moving 0 to 2.83539 1.86406 0, mtime=387
Moving 0 to 2.83539 1.93683 0, mtime=387
Moving 0 to 2.98094 1.93683 0, mtime=387
Moving 0 to 2.98094 2.00961 0, mtime=387
Moving 0 to 3.05371 2.00961 0, mtime=387
```

Figure 4.9: The actual steps of "Move Node"

The solution was to check whether the node being held is the same as the node previously moved. That is, it is assumed that as long as the same node is being held, the movements are counted as one step. Under this condition, even though the user stops the node in several points before reaching the end point, "Undo" moves it back straight to the start point.(Figure4.10)

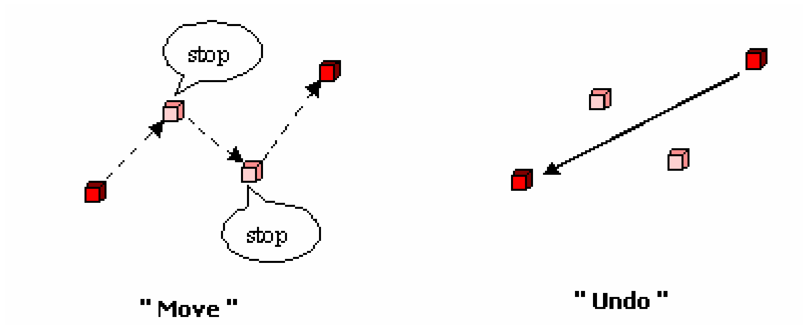


Figure 4.10: Undo a "Move Node"

4.3.2 Open File

Loading a file and saving data functions are not in initial project plan, but these functions were added to make the system more practical. *Tk_getOpenFIle* procedure automatically pops up a dialog box to select a file. The following pseudo-code demonstrates how "Load file" was implemented in Tcl.

```
OpenFile
Obtain file name using tk_getOpenFile widget
    If filename is valid
        Set the reader file to the chosen filename
        Set the source of the edited graph to be the output of the reader
        Remove all existing actors from the render window and reset the view
        Update the links between the graph data object and the filters that visualize it.
        Ask the render window to update
```

The file may or may not have layout specification. In case where the graph data has layout specification, the process has to avoid layout filter to be applied, otherwise the graph data will have its shape by being processed by layout filter.

4.3.3 Save File

As mentioned previously, even when a graph is visually modified, its data written in a GML format is not manipulated. It is necessary to save the change to the graph into a file before the program is terminated. This may not be a critical requirement for a graph editor but it is a fundamental function for any editor, because saving changes allows a user to continue the task from the form last modified. "SaveFile" is created by using the vtkGMLWriter method of the vtkGraph class. An extensibility issue with "SaveFile" is that after data is saved, the instance *writer* is deleted so that in each session modified data is dealt by a new *writer*. The writer could be deleted outside of "test 2" . However, deleting inside allows the software to expand to deal with various formats without any program modification. The following pseudo-code demonstrates how "Save file" was implemented in Tcl.

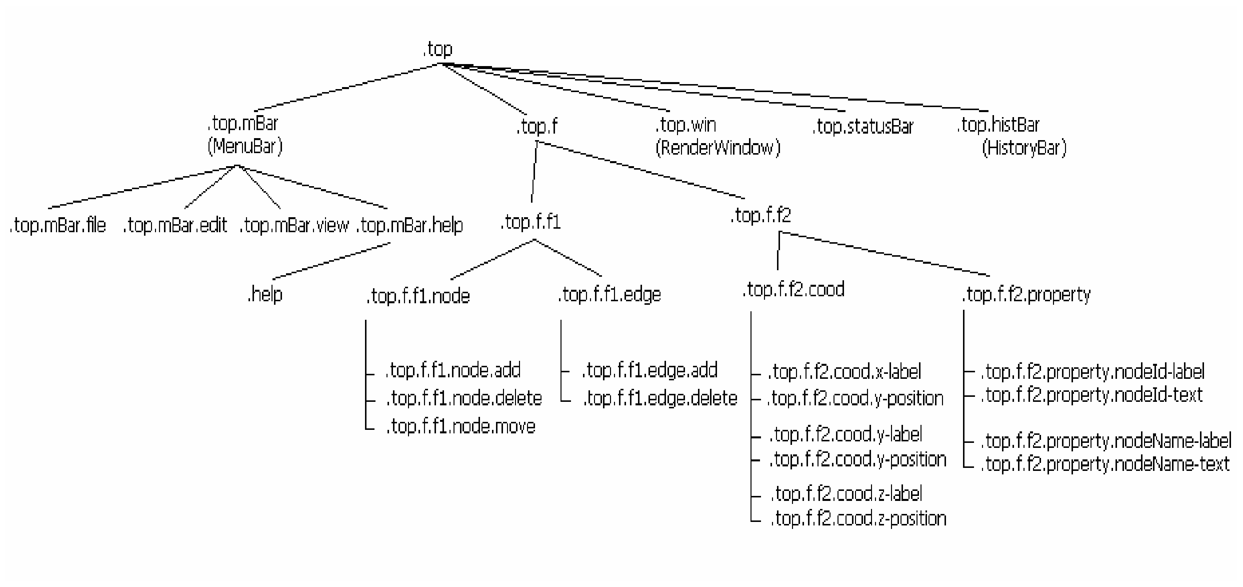
Save File

```
Obtain file name using tk_getSaveFile widget
    If filename is valid /* test 1 */
     If the file format is GML file  /* test 2 */
        Create an instance of the GML writer class
            Assign the filename and the graph to the writer
            Instruct the writer to write the graph to the file
            Terminate the writer instance
     Else if file format is not GML, return error "Can't write this file"
       Return no value
```

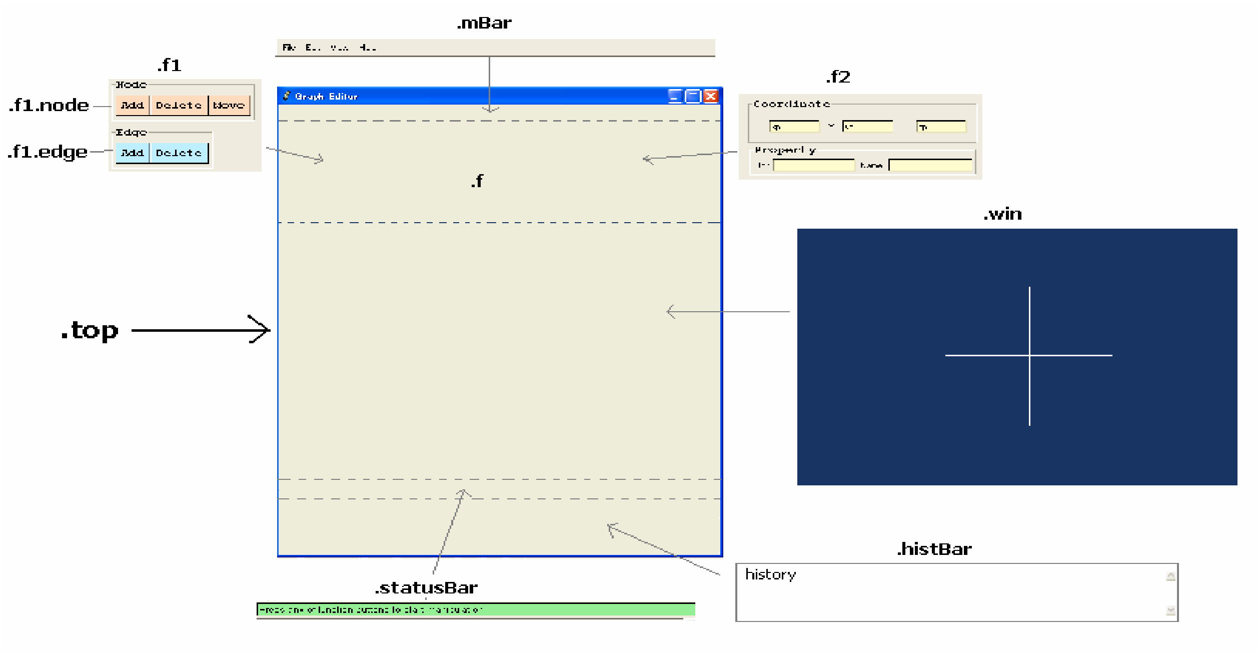
4.4 GUI and User Navigation using TkWidgets

4.4.1 Tk widgets

The GUI is written in TK, an extension to Tcl. Tk provides support for common user-interface components such as push-buttons, text widgets and scroll bar.[Scheroeder ,1998] Widgets are arranged hierarchically. The graph editor consists of 6 widgets. The hierarchical structure of the widgets is shown in Figure4.11 (a), and an exploded view of the screen is shown in Figure 4.11(b) to clarify the widget structure.



(a) The hierarchical structure of the widgets of the graph editor window



(b) An explode view of the graph editor window

Figure 4.11: The structure of GUI of the VTK Graph editor

4.4.2 Main MenuBar (.mBar)

This main menu is pull-down style as most software.

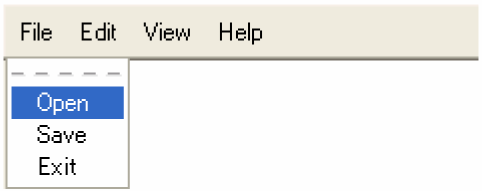


Figure 4.12: Menu Bar

- **File**

It contains operations to load a file and save the data into a file.

- **Edit**

There are two functions available, "Undo" which retrieves one previous form of the graph, and "Clean" which clears the rendering screen.

- **View**

This is one of the user navigation functions. Pressing "front" resets the view point, therefore, it is always possible to come back to the initial view point.

- **Help**

Simple help information comes up in window describing step by step the instruction for all functions the system covered.

4.4.3 Function Buttons (.f.f1)

By pressing the buttons, the user is able to implement functions to modify the graph. The buttons are dependent on each other so that there are no more than one function carried out at any one time. When a button is pressed, the function mode is available and the button appears sunken. The mode can be changed either pressing other function buttons or pressing it again. When the mode of the function is disabled, the button appears raised.

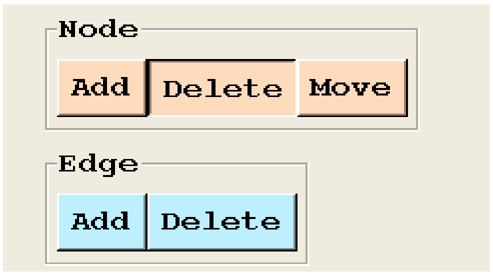


Figure 4.13: Function Button

4.4.4 Information Bar (.f.f2)

This includes coordinate bars and property bars. The coordinate bars, x y z, show the coordinate of the node picked. When the node is being moved, the coordinate bars show the position movements so that the user can know which direction to drag the new node. The property bars contain node Id bar and node label bar. These shows the property of node currently being picked.

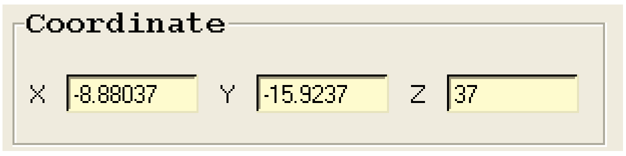

Coordinate
X -8.88037 Y -15.9237 Z 37

Figure 4.14: Coordinate Bar

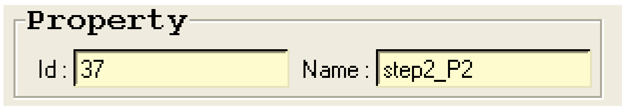

Property
Id : 37 Name : step2_P2

Figure 4.15: Property Bar

4.4.5 Status Bar (.statusBar)

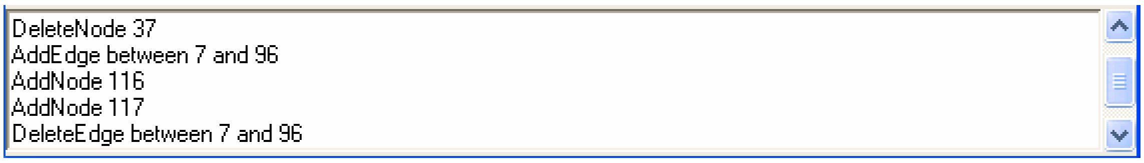
The message of the status bar specifies current situation and also leads a user to carry out the function properly, i.e. “Select a node to delete “.

Click the node you want to delete

Figure 4.16: Status Bar

4.4.6 History Bar (.histBar)

This shows the recode of functions implemented with node IDs.

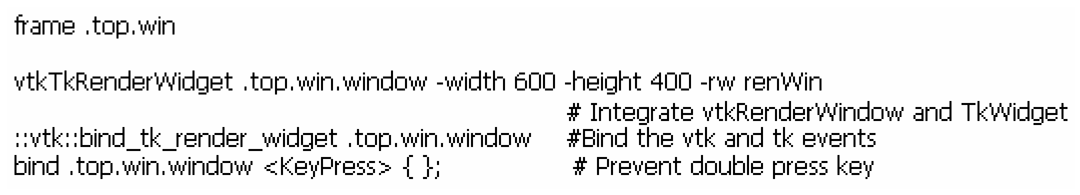


```
DeleteNode 37
AddEdge between 7 and 96
AddNode 116
AddNode 117
DeleteEdge between 7 and 96
```

Figure 4.17: History bar

4.5 User Interaction

While the application window and GUI are created in Tcl/Tk, user interacting functions are still supported by vtk. There were two issues concerned during implementation process. In order to achieve the integration of Rendering Window which displays a graph and Tcl/Tk which provides GUI into a single window, a special object called vtkTkRenderWidget is used. This allows Tk widgets to interface with vtk events. However, TkRenderWidget has different management of vtkRenderWindowInteractors **from vtkRenderWindow**, there is therefore a need to bind vtk and Tk events. The code below shows events binding. The last code is to create own interactor style in order to avoid double press key action. VTK considers "Press down" and "Release" as individual actions, "Keypress" or "Click" actually generates two actions. The new intoractor style returns one callback from pressing and releasing a key. This allows a user to avoid holding key down while carrying one action.



```
frame .top.win

vtkTkRenderWidget .top.win.window -width 600 -height 400 -rw renWin
                                              # Integrate vtkRenderWindow and TkWidget
::vtk::bind_tk_render_widget .top.win.window    #Bind the vtk and tk events
bind .top.win.window <KeyPress> { };            # Prevent double press key
```

Figure 4.18: Binding VTK and Tk events, and Creating new interactor

Chapter 5: System Evaluation

The purpose of this project is to build a tool called "graph editor" which enables a user to modify a VTK graph. The system assumes users are able to do direct manipulation such as "click and drag". Since the system is an interactive tool and its purpose is editing, the evaluation has been done by mainly concentrating on functional usability. The evaluation methods are;

1. **Participative Evaluation**
2. **Comparative Evaluation**

The nature of this project is that design requirements and preferences were clearly set out, and so evaluation is mostly appropriate as a summative tool, to assess what has been achieved and to identify areas for future work.

5.1 User Participation

The purpose of involving users in evaluation is "*to find out what users want and what problems they experience*".[Preece, 1994] Preece describes user-centred design in which the user is involved in all phases of development. In requirements capture, user studies may be carried out to understand what the system should do, and during the design, consultation and formative evaluation via prototypes may help guide design choices.

There are slightly different concerns about usability because users' needs are dependent on their ability and knowledge. For example, novice users tend to prefer "easy to use" functions, which more, smaller incremental steps and therefore do not mind if the whole process takes longer. While experienced users like to use "quick" steps even though this means the process may be a little more complex. In this graph editor, the implementations of the functionalities involve 3D interaction. The aim of this project did not extend to statistically significant usability studies. Instead, it was more appropriate to conduct limited studies to gain insight into where the design was successful and where there may be scope for future improvements.

5.1.1 The characteristics of the participants

Seven participants consist of five males and two females. Subjects were asked to complete a questionnaire covering their background and in particular their experience of 3D graphics. The backgrounds are found to range over; Chemistry, English literature, Culture and Computing (Information systems and Computer Graphics). Information on 3D graphics experiences was requested in order to define whether users' 3D experience affects what they find about the usage of the system. That is, this is to distinguish between people who find the task hard because it simply is a hard task if user has no 3D graphics experience, and those who find the task is hard because the interface is not ideal.

In order to measure the users' ability to deal with computer 3D world, three questions were asked ;

- How often do you play 3D computer games?
- How often do you use 3D modelling tool?
- How often do you use 3D navigation environment?

The users were asked to select the most appropriate answer from : Often-O, Sometimes-S, Rarely-R and Never-N

		3D game	3D modelling	3D navigation
1	Male	O	S	S
2	Male	O	S	S
3	Male	S	N	S
4	Male	O	S	S
5	Male	N	R	S
6	Female	N	N	R
7	Female	N	N	R

Figure 5.1 The characteristic of participants

- **Participants constrain**

There was a limitation of finding participants who have a different level of experience. The subjects are students (postgraduate and undergraduate) and as a result, have some level of experience with computer use. It has noted that in general males tend to have a greater interest in computer games than females do, and it was useful in this study to have more male participants, who had 3D game experience, than females.

5.1.2 Evaluation methods

Preece(1994) introduced several methods for participative evaluation. The methods used here are;

- **Observing and monitoring usage**

  This is to determine how the users interact with the system in carrying out realistic tasks. There are a number of techniques for collecting and analysing data, such as video recording [Preece at el,.1994] of user actions and keeping a history of what the user has done. In this project evaluation, *direct observation* was carried out with the author of this report acting as an observer and making notes. The purpose of this evaluation is to gain better understanding of user behaviours and their needs by watching user behaviour as it unfolded in real time.

- **Collecting users' opinions**

  The users' opinions were gained by through the use of a questionnaire which was completed immediately after experiencing the system. The questionnaire includes non-functional evaluation which focuses on the available functions but not to find the needs of additional functionalities. The main objective of this valuation was to determine the usability of the specific editing functions. However, the usability of user navigation was also involved as all functions may require some level of navigation within a scene. The evaluation criteria are

  - Functionalities --- ease of use
  - User navigation --- clarity and efficiency
  - Complexity of 3D interaction

5.1.3Guideline for evaluation process

The participants were given an empty screen and asked to draw a graph by using any and all functions available. Help documentation showing how to implement the functions was provided and supports for dealing with 3D view were given where necessary. After 10 minutes, they are asked to draw particular shape of a graph showing Figure5.2 , which consists of 9 nodes with 8 edges.

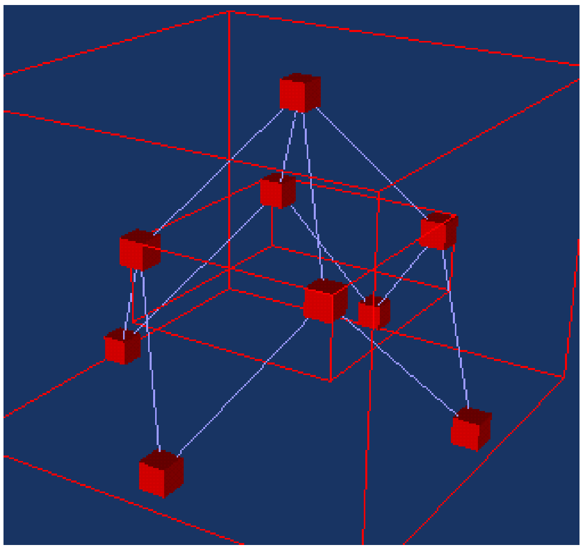


Figure 5.2: Test model

5.1.4 Evaluation result and analysis

Unless otherwise indicated, numerical entries in the table represent an assessment of the difficulties where 1 = easy, 5 = difficult and N = Never use

Participants		1	2	3	4	5	6	7
Functional usability	AddNode	1	1	1	1	3	1	2
	DeleteNode	2	1	3	2	2	1	2
	MoveNode	2	2	1	1	2	3	4
	AddEdge	2	1	2	3	3	4	4
	DeleteEdge	2	2	2	3	2	2	3
	SelectNode	4	3	5	4	4	5	5
User Navigation	Help	N	1	N	N	N	N	N
	Coordinate	1	1	1	1	3	N	4
	Property	2	1	2	N	3	N	3
	Status Bar	1	1	5	2	1	2	2
User Interaction	View manipulation	3	1	3	1	2	1	4
	Control PointWidget	1	3	2	2	1	3	2
	Mouse Interaction	1	3	3	1	3	3	3
Overall	System explore	2	3	3	3	3	4	4
	Navigation	3	2	4	2	2	2	4
	Functionalities	2	2	2	1	2	2	2
	Speed	2	1	2	1	1	1	1
	Testing completed (time in minutes) * = Never completed	20	*	20	8	18	*	30

Figure 5.3: The result of Questionnaire

From the table, some initial observation can be made in particular concerning the shaded entries.

5.1.5 Observing and monitoring

- **Difficulty with "Select Node"**

It can be noted that all users, regardless of background, found "Select Node" to be difficult. It often happened that the user cannot choose the right node even though their cursor is pointing correctly. The solution is to set a view point to the node closer and/or than any other nodes. The reason for this problem is to do with the operation of the VTK point picker, which finds all points that lie within a given threshold of a ray cast through the picked point.
Experience user may have made unwarranted assumptions about how the system worked. User with less experience had no such background to confuse them. That is, it seemed that the non-3D experienced users took more time to understand this logic than the users who are experienced.

- **Interface consistency**

The users started from "AddNode". From their experience with this function, the users invariably assumed that an operation is completed when the button is pressed. However, "Add Node" works differently from the 4 other functions; for these, the relevant button must be pressed first, and then selecting a node or nodes completes the operations. This inconsistency confused the users. Most users press a button and expected something to happen. There is a need for consistency in function implementation in order to remove confusion, or change the appearance of the button so that it reflects its different behaviour.

5.1.6 Users' Opinion

During the evaluation, the following suggestions were given by participants.

- Improve picking
  The system implements picking node by pressing "p". However, because mouse motion is very sensitive, it often happens that the cursor has swerved from the node before pressing "p" and point cannot picked accurately.
  Solutions
  1. In the longer time, it would be useful to re-organize the functionalities associated with mouse buttons. This however requires changes to the bindings for standard VTK events, and this beyond the scope of the project.

2. On screen labels could be provided allowing a user to select a node by entering its label into text box. The disadvantage of this approach is that a large number of labels might crowd screen.

- Positioning and moving a node
  It is useful if a node can be moved or positioned by specifying coordinates explicitly. This would allow users to choose an exact position for a node. Some available software already covers this function.
  Solutions
  1. Use text box to enter the coordinate. The disadvantage of this is discussed later.

- Indication and system feedback
  When the function mode is switched, any selected node was still marked as selected; this has since been corrected.
  - Separation of choosing node and confirming operation. This improvement is likely to appeal more to novice users, as it increases the steps to complete the operation. e.g. Press function button, choose nodes, and press $\boxed{\text{ok}}$ to create edge.

- Selecting a group of nodes
  There were cases where the user wanted to move several nodes at the same time or to create edges from one node to several nodes at one time. This was considered in the project plan but there was not time to implement.

5.2 Comparative Evaluation

There are various ways of implementing the functionalities provided by available toolkits. Comparison of design was done in the early stages of the project. However, it is worth discussing again after practical development. This is because some modification of initial design was necessary during the development due to technical constrains faced. The VTK/Graph editor will be compared with available 3D graph editors, WilmaScope and Tulip. The evaluation is focused on the functionalities developed. The implementation process of the following five functions in each system is compared.

- **Summary**

	VTKGraphEditor	WilmaScope	Tulip
AddNode	1. Move PointWidget 2. Press AddNode User manual positioning in 3D	1. Press AddNode System automatic positioning in 3D	1. Press AddNode 2. Click the position User manual positioning in 2D
DeleteNode	1. Press Delete 2. Select a node	1. Select a node 2. PressDelete	1. Press Delete 2. Select(click) a node
MoveNode	1. Press Move 2. Select a node 3. Moveable in 3D	1. Select a node 2. Set fixed position to the node 3. Select the node fixed 4. Moveable in 2D	1. Select a node 2. Press Move 3. Moveable in 2D
AddEdge	1. Press AddEdge 2. Select source node 3. Select target node	1. Press AddEdge 2. Select 2 nodes 3. Press Ok to confirm	1. Press AddEdge 2. Click a source node 3. Drag to a target node
DeleteEdge	1. Press DeleteEdge 2. Select source node 3. Select target node	1. Select an edge 2. Press Delete	1. Press Delete 2. Select (Click) an edge

Figure 5.4: Summary of functions

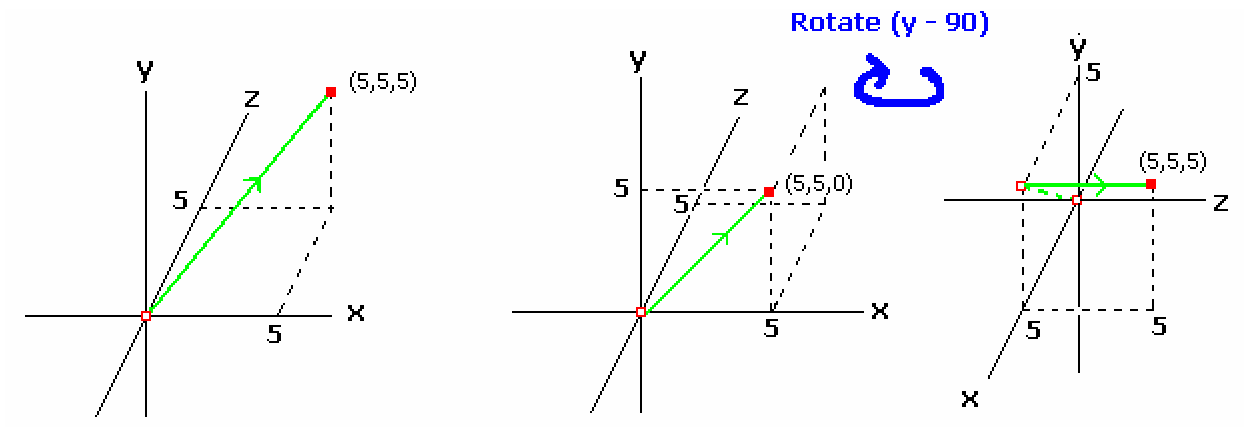
- **Discussion**

The number of steps to complete an operation is more or less, almost the same. The significant difference probably is that Tulip and WilmaScope make use of “Click”. The VTK/Graph editor uses “P” key instead. However, this is an unavoidable choice in order to keep availability of both view control and user interaction in terms of picking at the same time: both require use of mouse click to operate.

The advantage of VTK/Graph editor compared with WilmaScope and Tulip is that a user is able to interact in 3D WCS. That is, VTK/Graph editor supports positioning (AddNode) or moving a node in 3D space, while WilmaScope and Tulip allow a user to interact in 2D WCS by providing projection screen of the 3D WCS. The following example explains this in detail.

Assume that a user wants to move a node from A (0, 0, 0) to B (5,5,5). VTK/Graph editor enables the user to move a node directly from A to B by “Click and drag” using PointWidget.(Figure5.5 A). The approach of WilmaScope and Tulip is similar to the initial plan of this project, discussed in Chapter 3 “3D manipulation”. The node is first moved from (0,0,0) to (5,5,0) in xy coordinates (Figure5.5 B). Then, the user has to rotate the view point -90 degree so that the controllable area changes to yz coordinates. At the and, the node is moved from (5,5,0) to (5,5,5) This process is described in Help documents of WilmaScope as follows,

"When you move a node it moves to a position at the same depth from the view plane, but directly underneath the mouse pointer. You can still rotate the whole scene by dragging with the left mouse button so if you want to move the node to a new depth then first rotate the graph so that you get a side on view, and the drag it to the new depth."



A B C

Figure 5.5: Different implementation of 3D manipulation

Both WilmaScope and Tulip also provide as alternatives, an "entry operation" to specify the coordinate explicitly in order to position at a node exact place. However, this requires a user to calculate the coordinate in 3D (from existing objects) and this may be difficult if user does not have 3D graphic knowledge as it may require computing one position relative to the positions of other nodes. In principle, however, it would be quite simple to support this functionality in the VTK Graph editor.

Chapter 6: Conclusion and Future direction

6.1 Conclusion

The VTK Graph editor provides a rendering window showing a VTK graph and GUI components to enable a user to edit the graph by direct manipulation. The available functions are Adding Node, Deleting Node, Moving Node, Adding Edge and Deleting Edge. The graph can be interacted with and manipulated in 3-dimensional world coordinate space with view control functions such as Rotating, Zooming and Panning. The system provides Selection feedback to indicate a selected node, and text bars to specify current state, property of a node and current point coordinates in order to help user navigation. The mechanism that provides for selection feedback can be readily extended to support more general node colouring base on attributes.

Initial evaluation has shown the functionality is usable, while the same time, pointing to the opportunities for further improvement in particular in nodes selection.

Achieving the editor involve the significant amount of experimental work firstly to understand how the VTK pipeline operates, secondly to learn the use of the VTK Graph library, and thirdly to develop skills at implementing software with Tcl/Tk.

The difficulties with the development of the project were;

- The studying the VTK library took more time to understand than expected in particular pipeline operation; there is a shortage of documentation and also the module I took hadn’t covered enough depth.
- There is also steep curve for learning Tcl/Tk, although by the end of the project, sufficient confidents have been obtained to produce a significant piece of software.

The implementation was sufficiently thorough that is led to the identification of two undiscovered errors in the Graph library.

According to the minimum requirements set in the early stage of the project, it can be said that the development of minimum functionalities has been achieved.

6.2 Future direction

From the evaluation result and analysis, further development can be achieved by the following improvement.

6.2.1 Refinement of the existing functions

- **Position and move node by specifying a coordinate**

  One of the possible improvements for "Add Node" and "Move Node" is to make it enable a user to locate a node by specifying the coordinate explicitly.

- **Use of "Click and Drag "**

  WilmaScope and Tulip enable a user to select a node by clicking. Furthermore, Tulip allows creating an edge by clicking a source node and then dragging out an edge to the target node. In the VTK/Graph editor, due to technical constraints, selecting a node is currently done by pressing the "P" key, and creating an edge requires selecting source and target nodes in this way. Most of the participants wish to carry out the "Add Edge" task by using "Click and Drag" which seems more intuitive.

- **Undo + bend**

  For now, "Undo" does not support recovering the bend of edges. When "Undo" is used to recover deleted edges, straight lines are created between the source and target nodes, even if the edges had bends before they were deleted. In principal, this can be done in the same way as retrieving source and target nodes by undoing the "Delete node" action.

6.2.2 Adding advanced editing functions

Due to the time limitation, not all advanced functions have been developed. Here, further desirable functionalities are briefly introduced. This list is created from the outcome of the background research.

- **Clustering** (Nesting, Grouping)

  "Clustering is the process of discovering groupings or classes in data based on a chosen semantics".[Herman et al., 2000] This treats a number of nodes as a group, and apply the operation such as moving and deleting at the same time. It makes it also possible to hides a

particular group of nodes. The advantage of clustering is to reduce the number of visible elements being viewed as well as the number of operations implemented since it can treat a group of nodes as one node. [Herman et al., 2000] This means, clustering makes the graph look less complex and easy to manage.

- **Create bend to an edge**

- **Copy, cut and paste**
  Copy, Cut and Past a part of a graph with its all information and attributes.

- **Support various file format**
  Another file format VTK/Graph library supports is a compact format used for storing an unstructured grid. It is desirable to support various file formats such as GraphML[Graph ML team, 2001-2004], GraphXML and GXL , which are also widely used.

6.2.3 Enhancement of User Interaction

User interaction can be improved by customizing the following.

- **Labeling**
  Labeling means putting strings on some or node and edges in the rendering window, so that a user can distinguish one from another visually. Each label may be obtained from an attribute and/or ID of the object to which it refers.

- **Customizing appearances** e.g. Shape/Width/Font/Size of nodes and edges

6.2.4User navigation

- **Having multiple viewports**
  In order to see whole graph and a detail, set two multiple view points e.g. one show overview of a graph and the other show the part of a graph being manipulated.

Bibliography

Angel E. (2000) ***Interactive Computer Graphics A top-down approach with OpenGL.*** 2nd Edition. Addison Wesley Longman, Inc.

Achilleas P. & Tollis L. G. (1998) ***Interactive Orthogonal Graph Drawing*** IEEE Transactions on computers, Vol. 47, No. 11, 11/1998

Caldwell C. (1995) ***Graph Theory Glossary***
http://www.utm.edu/departments/math/graph/glossary.html [Final accessed 01/09/04]

Chartrand G. & L. Lesniak (1996) ***Graphs & Digraphs* 3rd Edition**
Chapman & Hall/CRC

Dogrusoz U., Feng, Q. and Madden B., Doorley, M. & Frik Arne (2002) ***Graph Visualization toolkit*** IEEE Computer Graphics

Duke D. ***Graph Visualization Libraryv1.20*** http://www.cs.bath.ac.uk/~did/README.html [Accessed 05/04/04]

Duke D. (2003) ***vtkGMLReader.h*** Graph visualization library for VTK

Diguglielmo G., Eric Durocher, Philippe Kaplan, Georg Sander, & Adrian Vasiliu(2002) ***Graph Layout for Workflow Application with ILOG JViews*** Graph Drawing 10th International Symposium, GD 2002 Irvine, CA, USA, August 26-28, 2002 Revised Paper M.T. Goodrich & S.G.Kobourov(editor) pp362-363 Springer-Verlag Berin Heidelberg Germany

Eppstein D. (2004) ***Geometry in Action Graph Drawing***
http://www.ics.uci.edu/~eppstein/gina/gdraw.html [Final accessed 06/09/04]

Fröhlich, M. & Werner M. (2004) **da Vinci** http://www.informatik.uni-bremen.de/daVinci/ [Final accessed 06/09/04]

Gansner E. R. and North S. C. (1999) ***An open graph visualization system and its applications to software engineering*** John Wiley & Sons, Ltd.
http://www.research.att.com/sw/tools/graphviz/GN99.pdf [Final accessed 01/09/04]

Goodrich M.T. & S.G.Kobourov(2002) ***Graph Drawing*** 10th International Symposium, GD 2002 Irvine, CA, USA, August 26-28, 2002 Revised Paper
Springer-Verlag Berin Heidelberg Germany

Graph editor instructions
http://loki.cs.brown.edu:8081/geomNet/gds/doc/ged/ged-help.html
[Final accessed 06/09/04]

Graphlet ***The GML File Format***
http://infosun.fmi.uni-passau.de/Graphlet/GML/ [Final accessed 06/09/04]

Graph ML team (2001-2004) The Graph ML File Format
http://graphml.graphdrawing.org/ [Final accessed 06/09/04]

Han K., Ju Byong-Hyon, & Park Jong H. (2002) ***Inter Viewer: Dynamic Visualization of Protein-Protein Interactions***
in Graph Drawing 10th International Symposium, GD 2002 Irvine, CA, USA, August 26-28, 2002 Revised Paper M.T. Goodrich & S.G.Kobourov(editor) pp364-365
Springer-Verlag Berin Heidelberg Germany

Harel D. & Koren Y. (2002) ***Graph Drawing by High-Dimensional Embedding*** *in* Graph Drawing 10th International Symposium, GD 2002 Irvine, CA, USA, August 26-28, 2002 Revised Paper M.T. Goodrich & S.G.Kobourov(editor) pp207-219
Springer-Verlag Berin Heidelberg Germany

Helio A. S., Lima Filho & C. M. Hirata(2002) ***GroupGraph: a Collaborative Hierarchical Graph Editor Based on the Internet.*** Dept of Computer Science Instituto Technologico de Aeronautica IEEE

Herman I. & Melancon G. & Marshall M. S. (2000)
Graph Visualization and Navigation in Informatino Visualization: Survey IEEE Transaction on Visualization and Computer Grpahics, Vol. 6, No. 1

Kaufmann M. & Wagner D. (Eds.)(2001) ***Drawing Graphs (Methods and Models)*** Springer-Verlag Berlin Heidelberg Germany

Kitware Inc. ***The Visualization toolkit*** www.vtk.org [Final accessed 06/09/04]

LEDA Algorithmic Solutions Software GmbH
http://www.algorithmic-solutions.com/enprodukte.htm [Final accessed 06/09/04]

Liotta G. (2004) ***Graph Drawing*** 11th International Symposium, GD 2003, Perugia, Italy, September 2003 Revised Paper
Springer-Verlag Berin Heidelberg Germany

Merris R. (2001) ***Graph Theory*** Jone Wiley & Sons, Inc. Canada

Newbery F. J. (1988) "***An interface Language for graph editor***"
TH0229-5/88/0000/0144 IEEE

Ousterhout J. K. (1994) ***Tcl and the Tk Toolkit*** Addison-Wesley

Papakostas A & Tollis L. G. (1998) ***Interactive Orthogonal Graph Drawing*** IEEE Transactions on computers, Vol, 47, No. 11, (PDF: 0073644)

Preece, J. (1994) ***Human-Computer Interaction***
(with) Yvonne Rogers, Helen Sharp, David Benyon, Simon Holland, Tom Carey
Addison-Wesley

Rodgers P. J. (1998) ***A Graph Rewriting Programming Language for Graph Drawing*** Proceedings of the IEEE Symposium on Visual Language
IEEE Computer Society

Sander G. (1997) ***Graph Drawing Tools and Related Work***
http://rw4.cs.uni-sb.de/users/sander/html/gstools.html [Final access 1/09/04]

Scheroeder W. J., K. M. Martin & W. E. Lorensen ***The Design and Implementation Of An Object-Oriented Toolkit For 3D Graphics And Visualization***
GE Corporate Research & Development

Scheroeder Will J., Ken Martin, Bill Lorensen (1998) "***The Visualization Toolkit" An Object-Oriented Approach to 3D graph***" Pertice-Hall,Inc. A Simon & Shcuster Company USA

Scheroeder W. J., Lisa S. Avila, William A. Hoffman C. Charles Law, K. M. Martin(1998-2001) "***The Visualization Toolkit User Guide***" ISBN 1-9390934-04-1 Kitware, Inc

Shaffer C. (2001) A Data Structure Visualization System ***Swan.***
Virginia Tech Information. http://simon.cs.vt.edu/Swan/Swan.html [Final accessed 01/09/04]

Spence R. (2001) ***Information Visualization*** The ACM Press, A Division of the Association for Computing Machinery, Inc.(ACM), London

VTK FAQ KitwarePublic. The free encyclopedia. http://www.vtk.org/Wiki/VTK_FAQ [Final access 06/09/04]

West D. B. (2001) ***Introduction to Graph Theory 2nd Edition***
Prentice-Hall, Inc. Upper Saddle River, NJ 07458

Zaslavsky T. (1998) ***Glossary of Signed and Gain Graphs and Allied Areas***
http://www.combinatorics.org/Surveys/ds9.html [Final access 01/09/04]

Appendix A: Project Reflection

One of the benefits I gained through this project was to experience whole process of “Project development life cycle”.

Through the background research, I became aware of the great possibility of Graph visualization and investigating available tools let me perceive the efforts and successful contributions which have been created by developers and researchers who have attempted to solve modern complex problems. It was meaningful experience for me to be one of them.

There were several difficulties I faced.

Understanding visualization data down/up flow algorithms took me a lot of time. Although I had learnt some of the techniques through MSc modules, it required deeper knowledge. Exploring VTK library supported by over 900 classes was enjoyable to find its great capabilities but was time consuming work.

Moreover, primarily, it was concerned that more C++ programming would be involved rather than Tcl/Tk. However, later it was found that Tcl/Tk is more powerful than it was expected, therefore more attention was paid to learn Tcl/Tk. Since this was the first time for me to use Tcl/Tk, the project was very challenging.

In order to complete the project, I had to learn several new things such as VTK library, VTK Graph library and while developing Tcl/Tk programming skills. Good time management was considered and/but it was very important to have a break. Talking to other people and visiting my supervisor refreshed my idea.

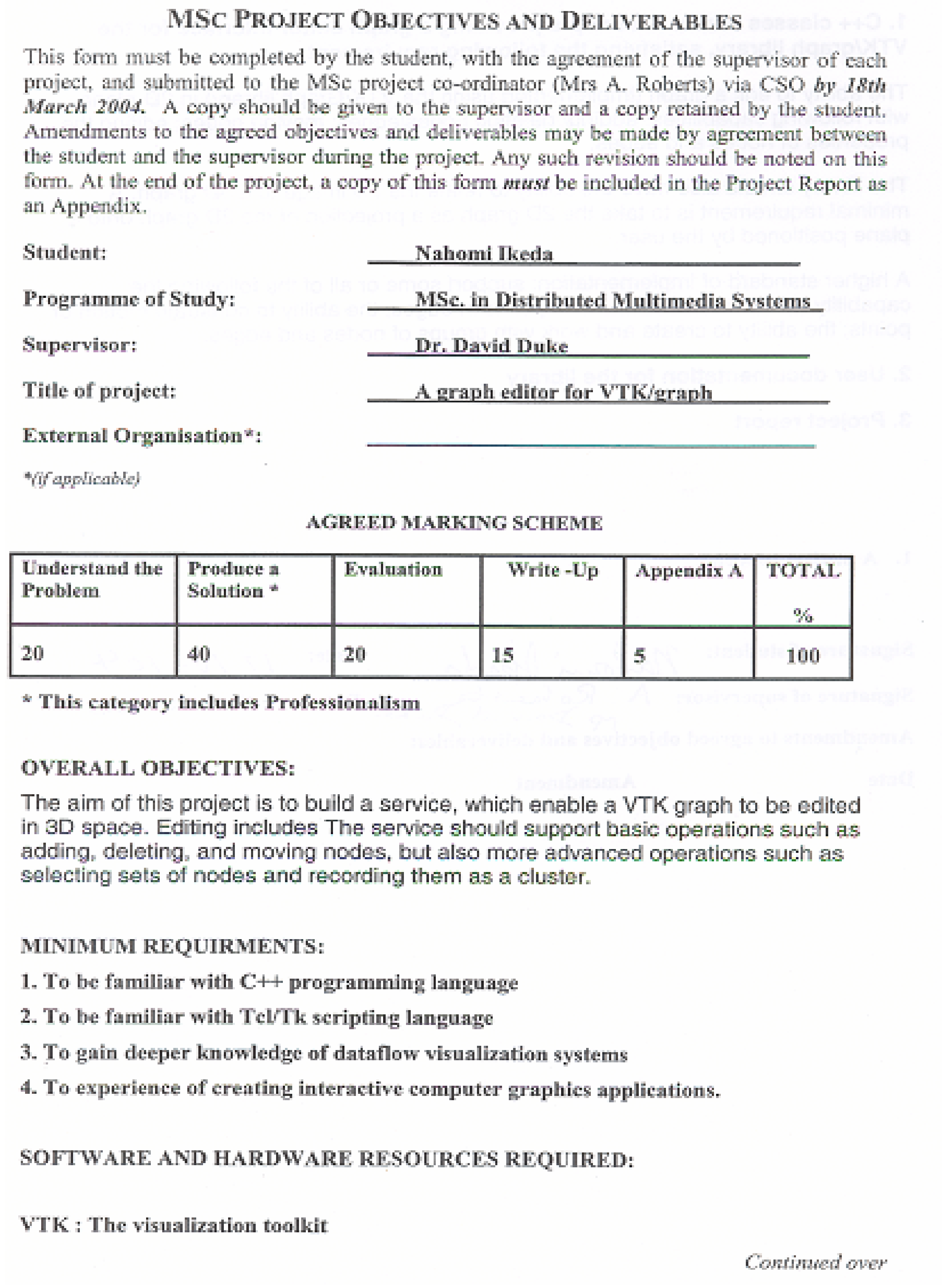
One of the important things I found through the implementation (programming) was that problems were not always in my own code. By this I mean, when the program does not work well and keeps returning an error, broad the eyes and think about various possible reasons. Sometimes, the reason of the error was just because of using different version of toolkit/software, or sometimes the library’s source code has a problem.

The participative evaluation was very valuable. I was impressed because the ideas gained from their opinions about the system were completely new to me. Their ideas were more practical and from a different view compared with what I have found through the background research.

The system is successful in terms of achieving functional requirements. However, there are still needs to improve from both non-functional and functional side. The system should be continuously improved.

Overall, it has been valuable experience to complete the system development from the begging to the end. I will make good use of the knowledge and skills I gained through this project.

Appendix B: Objectives and Deliverables Form



MSc PROJECT OBJECTIVES AND DELIVERABLES

This form must be completed by the student, with the agreement of the supervisor of each project, and submitted to the MSc project co-ordinator (Mrs A. Roberts) via CSO ***by 18th March 2004.*** A copy should be given to the supervisor and a copy retained by the student. Amendments to the agreed objectives and deliverables may be made by agreement between the student and the supervisor during the project. Any such revision should be noted on this form. At the end of the project, a copy of this form *must* be included in the Project Report as an Appendix.

Student:	**Nahomi Ikeda**
Programme of Study:	**MSc. in Distributed Multimedia Systems**
Supervisor:	**Dr. David Duke**
Title of project:	**A graph editor for VTK/graph**
External Organisation*:	

**(if applicable)*

AGREED MARKING SCHEME

Understand the Problem	Produce a Solution *	Evaluation	Write -Up	Appendix A	TOTAL %
20	40	20	15	5	100

*** This category includes Professionalism**

OVERALL OBJECTIVES:

The aim of this project is to build a service, which enable a VTK graph to be edited in 3D space. Editing includes The service should support basic operations such as adding, deleting, and moving nodes, but also more advanced operations such as selecting sets of nodes and recording them as a cluster.

MINIMUM REQUIRMENTS:

1. To be familiar with C++ programming language

2. To be familiar with Tcl/Tk scripting language

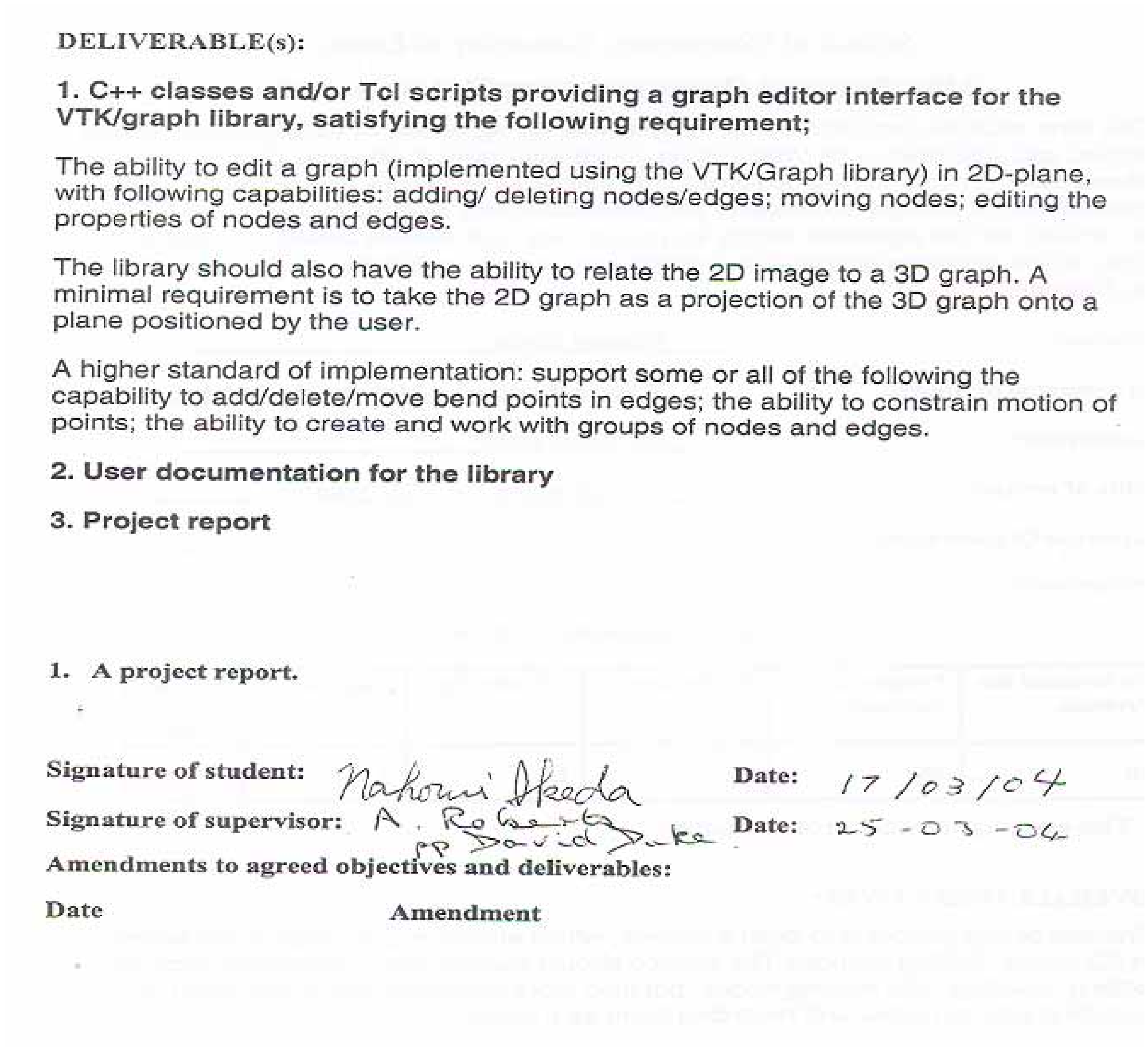
3. To gain deeper knowledge of dataflow visualization systems

4. To experience of creating interactive computer graphics applications.

SOFTWARE AND HARDWARE RESOURCES REQUIRED:

VTK : The visualization toolkit

Continued over



DELIVERABLE(s):

1. C++ classes and/or Tcl scripts providing a graph editor interface for the VTK/graph library, satisfying the following requirement;

The ability to edit a graph (implemented using the VTK/Graph library) in 2D-plane, with following capabilities: adding/ deleting nodes/edges; moving nodes; editing the properties of nodes and edges.

The library should also have the ability to relate the 2D image to a 3D graph. A minimal requirement is to take the 2D graph as a projection of the 3D graph onto a plane positioned by the user.

A higher standard of implementation: support some or all of the following the capability to add/delete/move bend points in edges; the ability to constrain motion of points; the ability to create and work with groups of nodes and edges.

2. User documentation for the library

3. Project report

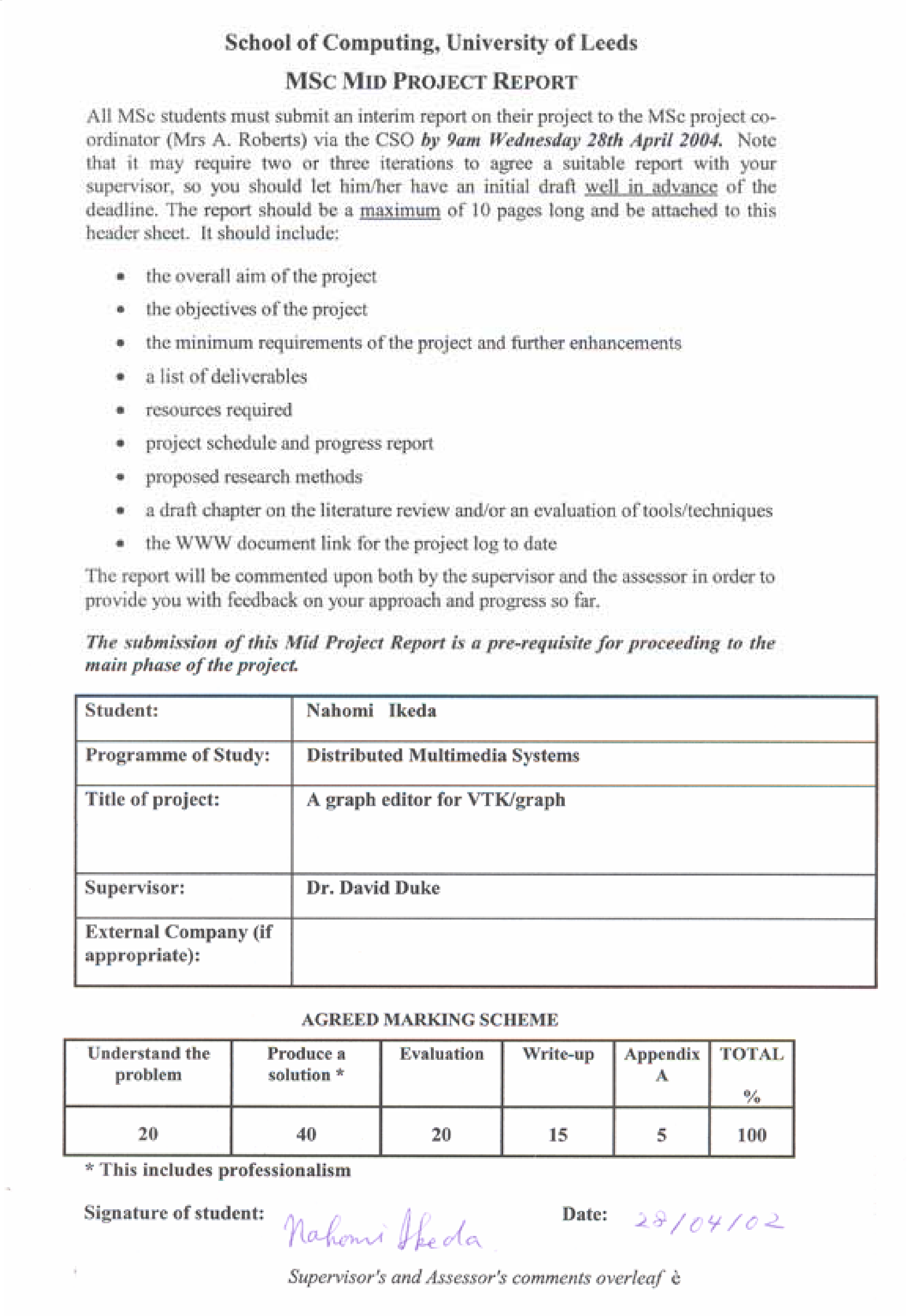
1. A project report.

Signature of student:	Nahomi Ikeda	**Date:**	17/03/04
Signature of supervisor:	A. Roberts pp David Duke	**Date:**	25-03-04

Amendments to agreed objectives and deliverables:

Date	**Amendment**

Appendix C: Mid Project Report



School of Computing, University of Leeds

MSc Mid Project Report

All MSc students must submit an interim report on their project to the MSc project co-ordinator (Mrs A. Roberts) via the CSO ***by 9am Wednesday 28th April 2004.*** Note that it may require two or three iterations to agree a suitable report with your supervisor, so you should let him/her have an initial draft well in advance of the deadline. The report should be a maximum of 10 pages long and be attached to this header sheet. It should include:

- the overall aim of the project
- the objectives of the project
- the minimum requirements of the project and further enhancements
- a list of deliverables
- resources required
- project schedule and progress report
- proposed research methods
- a draft chapter on the literature review and/or an evaluation of tools/techniques
- the WWW document link for the project log to date

The report will be commented upon both by the supervisor and the assessor in order to provide you with feedback on your approach and progress so far.

The submission of this Mid Project Report is a pre-requisite for proceeding to the main phase of the project.

Student:	**Nahomi Ikeda**
Programme of Study:	**Distributed Multimedia Systems**
Title of project:	**A graph editor for VTK/graph**
Supervisor:	**Dr. David Duke**
External Company (if appropriate):	

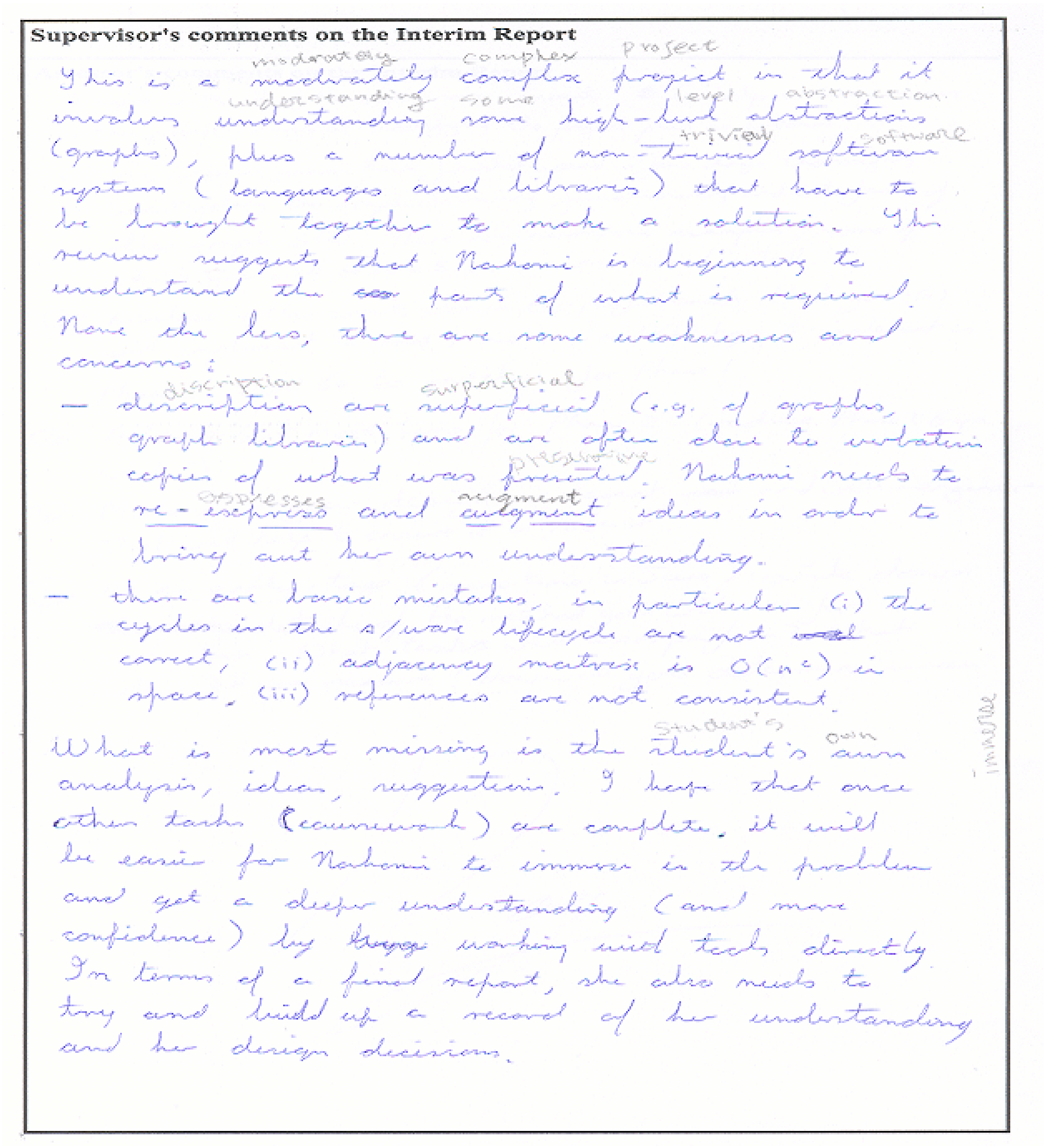
AGREED MARKING SCHEME

Understand the problem	Produce a solution *	Evaluation	Write-up	Appendix A	TOTAL %
20	40	20	15	5	100

*** This includes professionalism**

Signature of student: Nahomi Ikeda **Date:** 28/04/02

Supervisor's and Assessor's comments overleaf è

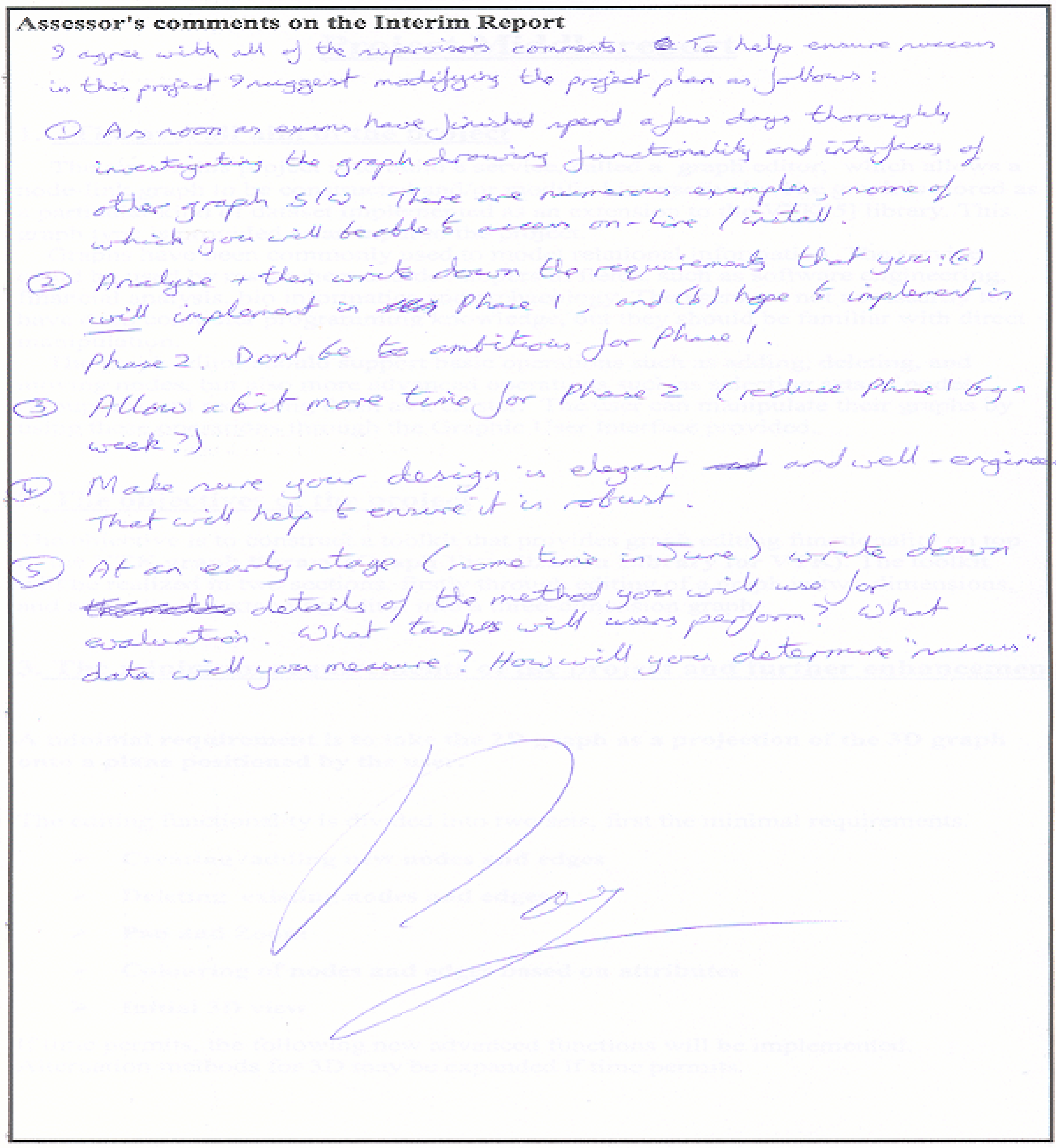


Supervisor's comments on the Interim Report

This is a moderately complex project in that it involves understanding some high-level abstractions (graphs), plus a number of non-trivial software systems (languages and libraries) that have to be brought together to make a solution. This review suggests that Nahomi is beginning to understand the ~~sco~~ parts of what is required. None the less, there are some weaknesses and concerns:

- descriptions are superficial (e.g. of graphs, graph libraries) and are often close to verbatim copies of what was presented. Nahomi needs to re-express and augment ideas in order to bring out her own understanding.
- there are basic mistakes, in particular (i) the cycles in the s/ware lifecycle are not ~~well~~ correct, (ii) adjacency matrix is $O(n^2)$ in space, (iii) references are not consistent.

What is most missing is the student's own analysis, ideas, suggestions. I hope that once other tasks (coursework) are complete, it will be easier for Nahomi to immerse in the problem and get a deeper understanding (and more confidence) by ~~[illegible]~~ working with tools directly. In terms of a final report, she also needs to try and build up a record of her understanding and her design decisions.



Assessor's comments on the Interim Report

I agree with all of the supervisor's comments. To help ensure success in this project I suggest modifying the project plan as follows:

① As soon as exams have finished spend a few days thoroughly investigating the graph drawing functionality and interfaces of other graph S/W. There are numerous examples, some of which you will be able to access on-line / install.

② Analyse + then write down the requirements that you: (a) will implement in ~~the~~ Phase 1, and (b) hope to implement in Phase 2. Don't be too ambitious for Phase 1.

③ Allow a bit more time for Phase 2 (reduce Phase 1 by a week?)

④ Make sure your design is elegant ~~and~~ and well-engineered. That will help to ensure it is robust.

⑤ At an early stage (some time in June) write down ~~the meth~~ details of the method you will use for evaluation. What tasks will users perform? What data will you measure? How will you determine "success"

[illegible]

Appendix D: User Manual for the VTK Graph editor

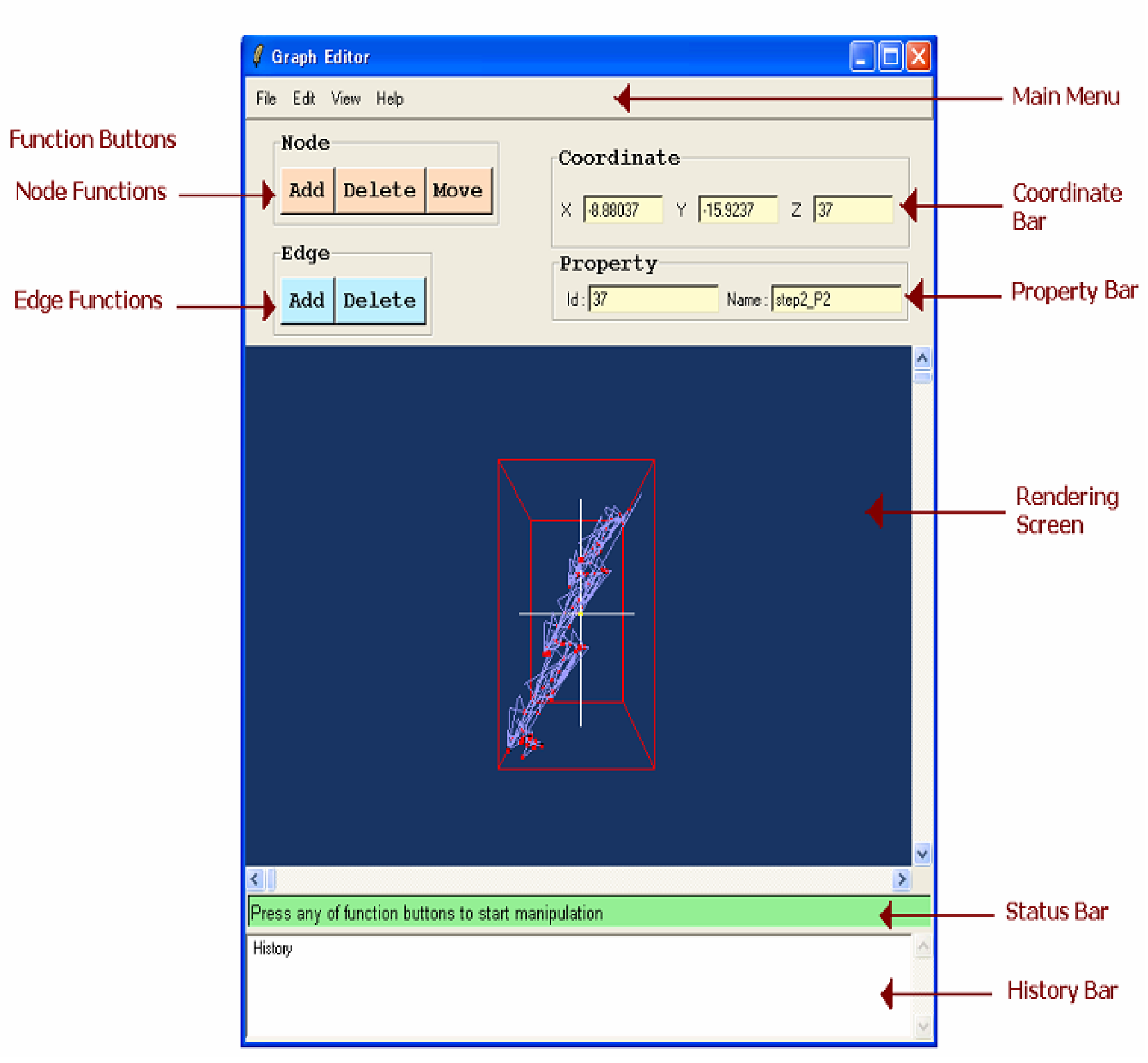
User Manual

1. Introduction

The graph editor for VTK/Graph library is three-dimensional graph visualization and editing system.

2. The Application Window

When the application is started, you can see the window being showed below. The window contains Menu bar, Function Buttons, Coordinate Bar, Property Bar, Status Bar and History Bar.



Press any of function buttons to start manipulation

Figure 2.0 Application Window

2.1 Menu Bar

The Menu bar is composed of four sub-menu, File, Edit, View and Help.

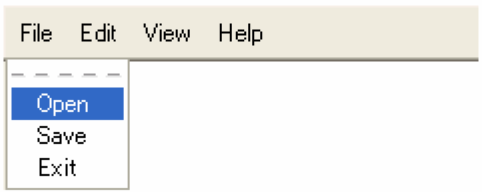


Figure 2.1 ; Menu Bar

➢ **File**

Open --- This enables to load a new graph in GML format.
Save --- Save the file modified or Save to the new file.
Exit --- Exit the application

➢ **Edit**

Clean --- Make the screen empty. Remove all images from the Rendering Screen.
Undo --- Operate undo functions.

➢ **View**

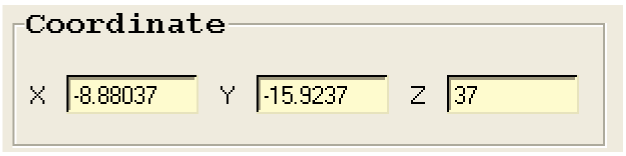
Front --- This enables to come back to initial view point.

➢ **Help**

Help document is showed in a different window. Help document specifies how to interact to the rendering window and describes the ways to implement functions.

2.2 Coordinate Bar

This shows the coordinate of a node that a user is dealing with. Especially, when a user is moving a node, these specify the current position of the node so that a user can see which direction the node is being moved.



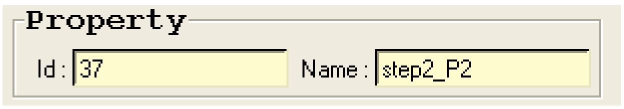
Coordinate

X -8.88037 Y -15.9237 Z 37

Figure 2.2 ; Coordinate Bar

2.3 Property Bar

When a node is picked, it shows its property; node ID and label. This is useful when a user want to modify particular node.



Property

Id : 37 Name : step2_P2

Figure 2.3 ; Property Bar

2.4 Function Buttons

These buttons are used to operate functions to manipulate/create a graph. Pink buttons and blue buttons are related to nodes and edges respectively. When a button is pressed, the function mode is available and the button is appeared sunken. The mode can be changed by either pressing another function button or pressing itself. When the mode of the function is disable, the button is appeared arise. Figure 2.4 shows Delete Node mode. The instruction of carrying out the operations are described the following section.

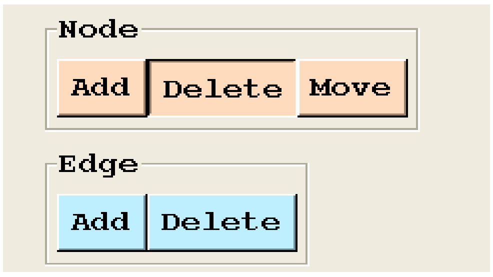


Figure 2.4 ; Function Bar

2.5 Rendering Screen

This shows a graph. A user can change the view by using the functions such as rotating, zooming and panning. These functions are implemented by keyboard and mouse control based on VTK(Visualization toolkit) standard.

- Keypress “ j” / Keypress “ t “ --- taggle between joystick and torackball styles.
- Keypress “ c “ / Kepress “ o “ --- taggle between camera and object modes.
- Keypress “ p" --- perform a pick operation.
- Button1 (mouse left) --- Rotate the camera or objects (if object mode)
- Button2 (shift + mouse middle) --- Pan the camera or translate the objects (if object mode)
- Button2 (shift + mouse right) --- Zoom in/up the camera or scale the objects(if object mode).

2.6 Status Bar

The message of the status bar specifies current situation and also leads a user to carry out the function properly**.**

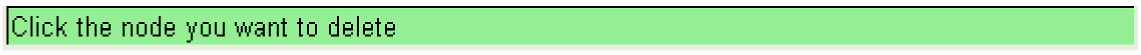

Click the node you want to delete

Figure 2.5 ; Status Bar

2.7 History Bar

This shows the history of functions implemented with node IDs.

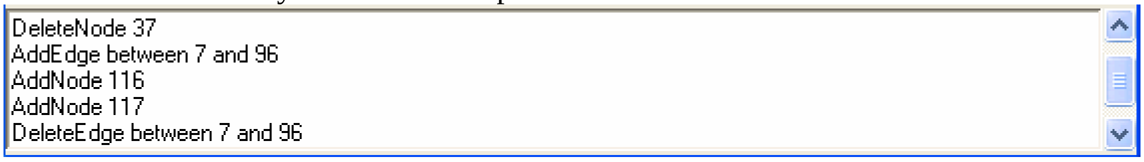

DeleteNode 37
AddEdge between 7 and 96
AddNode 116
AddNode 117
DeleteEdge between 7 and 96

Figure 2.6 ; History bar

3. Functions implementation

In order to get a position in 3-dimensional space, the graph editor uses special tool called PointWidget. When the application is started, the widget function is disabled and appears as a white line(Figure 3.1 **A**). Once the widget is clicked, it appears as a green line(Figure3.1 **B**) and the user can control/move the widget. When a node is picked, the widget is automatically set to the centre of the node. The colour of the node picked changes to yellow to distinguish from others (Figure 3.1 **C**).

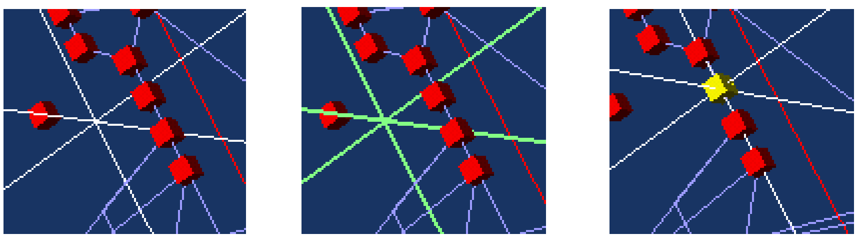


Figure 3.1 PointWidget

3.1 Picking up a node

Press “ P ” key on the node you want to choose.

3.2 Add Node

1.Move the centre of PointWidget to where you want to position node
2.Press "AddNode" button

3.3 Delete Node

1.Press "DeleteNode" button
2.Pick up the node you want to delete

3.4 Move Node

1.Pick up the node you want to move
2.Drag node
When you move the node, you can see the current coordinate in Coordinate Bar.

3.5 Add Edge

1.Press "AddEdge" button
2.Pick up the first node to be source Node
3.Pick up the second node to be destination Node

3.6 Delete Edge

1.Press "DeleteEdge" button
2.Pick up the first node to be source Node
3.Pick up the second node to be destination Node

Appendix E: Project Code

```
package require vtk
package require vtkinteraction

if { [catch {load libvtkGraphsTCL}] != 0 && \
    [catch {load vtkGraphsTCL}] != 0} {
    puts "Could not load Graph package."
    exit 1
}

#Read a file

vtkGMLReader reader
    reader SetFileName "E:/VTK/Examples/Tutorial/Graph/fsm.gml "

vtkSpanningDAG sd
vtkConeLayout c1
    c1 CompressionOn

#Define the layout filter for graph

vtkSpanLayout layout
    layout SetInput [reader GetOutput]
    layout SetSpanningDAG sd
    layout SetTreeLayout c1
    layout SetSpacing 1.0

set graph [layout GetOutput]

#Set the colour (Graph strashler Metric is used)

vtkGraphStrahlerMetric strahler
    strahler SetInput $graph
    strahler NormalizeOn
    strahler Update

set gsource [$graph GetSource]
$graph SetSource ""

# Map the edges of the graph to geometry (Edge Part)
vtkEdgeGeometry geometry
```

```
   geometry SetInput [strahler GetOutput]

vtkAssignAttribute colourEdge
   colourEdge SetInput [geometry GetOutput]
   colourEdge Assign "strahler" SCALARS POINT_DATA

vtkLookupTable lut
   lut SetNumberOfColors 64
   lut SetHueRange .6367  .01953
   lut SetValueRange .7773 .84375
   lut SetSaturationRange .457 .457
   lut Build

vtkPolyDataMapper mapper
   mapper SetInput [geometry GetOutput]
   mapper SetLookupTable lut
   mapper SetColorModeToMapScalars
   mapper SetScalarRange 0.0 1.0

vtkActor actor
   actor SetMapper mapper
   [actor GetProperty] SetColor 0.6 0.6 1.0

#GLYPHING PIPELINE ---------------------------------

vtkCubeSource cube
   cube SetXLength 1.0
   cube SetYLength 1.0
   cube SetZLength 1.0

# Map the graph to geometry ( Node Part)
vtkNodeGeometry nodes
   nodes SetInput $graph



# Selection feedback-------------------------------------------
# Create an array to store the colour inforamtion
# The default value is 0 == red.
# All nodes are represented red cubes.

global colour graph

vtkIntArray colour
   colour SetNumberOfTuples [$graph GetNumberOfNodes]
```

```
    colour SetName "colour"
    for {set i 0 } {$i < [$graph GetNumberOfNodes]} {incr i } {
        colour InsertValue $i 0
    }
    [$graph GetNodeData] AddArray colour

vtkAssignAttribute colourNode
    colourNode SetInput [nodes GetOutput]
    colourNode Assign "colour" SCALARS POINT_DATA

#------------------------------------------------------------------



# To map graphical representation to nodes, this case "cube"
# The colour is determied by Scalar

vtkGlyph3D glyphs
    glyphs SetSource [cube GetOutput]
    glyphs SetInput [colourNode GetOutput]
    glyphs SetColorModeToColorByScalar
    glyphs SetScaleFactor 0.5
    glyphs GeneratePointIdsOn
    glyphs SetScaleModeToDataScalingOff

# To map scalar values into rgba
# (red-green-blue-alpha transparency) color specification

vtkLookupTable soTable
    soTable SetNumberOfColors 2
    soTable SetTableRange 0 1
# Colours are Red Green Blue Alpha
    soTable SetTableValue 0  1 0 0 1
    soTable SetTableValue 1  1 1 0 1
    soTable Build



vtkPolyDataMapper glyphMapper
    glyphMapper SetInput [glyphs GetOutput]
    glyphMapper SetColorModeToMapScalars
    glyphMapper SetLookupTable soTable
    glyphMapper SetScalarModeToUsePointFieldData
    glyphMapper SelectColorArray colour
    glyphMapper SetScalarRange 0 1
```

```
# The Glyph actor determines the color of the
# Glyphs, and is set non-pickable.

vtkActor glyphActor
   glyphActor SetMapper glyphMapper
   glyphActor PickableOff

# Pickable setting---------------------------------------------------------

# The plane widget is used probe the dataset.

global x y z

#Set Pointwidget. Its centre and bound is dependent on node (geometry) data
vtkPointWidget pointWidget
   pointWidget SetInput [nodes GetOutput]
   pointWidget AllOff
   pointWidget PlaceWidget

vtkPolyData point
   pointWidget GetPolyData point

#--------------------------------------------------------------------

#Window and interaction setting --------------------------------------

vtkRenderer ren
   ren AddActor actor
   ren AddActor glyphActor
   ren SetBackground 0.1 0.2 0.4

# Picker will apply only the nodes but not the edges

   actor PickableOff
   glyphActor PickableOn

vtkRenderWindow renWin
  renWin AddRenderer ren
  renWin SetSize 800 600

# Picker function

vtkPointPicker picker
   picker AddObserver EndPickEvent HandlePick
```

```
    picker SetTolerance 0.1

vtkGenericRenderWindowInteractor iren
    iren AddObserver UserEvent {wm deiconify  .vtkInteract}
    iren SetRenderWindow renWin

    pointWidget SetInteractor iren
    pointWidget EnabledOn

# For undo action, record the history of functions implemented.
# The list to store parent node(Source node)
# The list to store child node (Target node)

set hist [list]
set parentList [list]
set childList [list]



#-------------------------------------------------------------------
# Handle pick Function
# This is invoked when "P" Key is pressed on a Node.
# It activate the node and make a graph available to modify
# x, y, z coordinate of the centre of the pointWidget ( Default xp, yp,zp)
# The node Id picked one step before. Used to selection feedback.



set x xp
set y yp
set z zp
set oldSelection " "

proc HandlePick {} {
    global graph colour carray
    global nodeName nodeId x y z
    global addEdgeFlag delEdgeFlag delFlag
    global srcNode desNode
    global menustatus
    global memory oldSelection

# pid is a point in the rendering window
# picked by user. This is a point on a glyph
# check if the point is picked

    set pid [picker GetPointId]
```

```
   if { $pid >= 0 } {

      set pids [[[glyphs GetOutput] GetPointData] GetArray "InputPointIds"]

      if { $pid < [$pids GetNumberOfTuples] } {

         set ipid [$pids GetValue $pid]
         set memory 0

# Check the point Id if it is within the bound of the array
# Lookup the array to get the id of the point.
# ipid is the index of the node in the graph
# Find nodeID and set "nodeId "

         if { $ipid >= 0 } {
            if { $ipid < [$graph GetNumberOfNodes] } {
               set nodeId [$graph GetNodeId $ipid]
               set labels [[$graph GetNodeData] GetArray label]
               set nodeName [[$graph GetStrings] GetString [$labels GetValue $ipid]]
               set carray [[$graph GetNodeData] GetArray colour]
               if {$oldSelection != " " && [$graph HasNode $oldSelection]} {
                  $carray SetValue [$graph GetNodeIndex $oldSelection] 0
               }
               $carray SetValue [$graph GetNodeIndex $nodeId] 1
               set oldSelection $nodeId

# Callback of "DeleteNode"
               DeleteNode

#Callback of "Edge" function which convey both Adding and deleting an edge.
               Edge

# Tell the glyphs is modified ( colour)
# Render the pipeline so that the selected node will be shown as yellow.

               glyphs Modified
               renWin Render

               set x [$graph GetX $nodeId]
               set y [$graph GetY $nodeId]
               set z [$graph GetZ $nodeId]
               pointWidget SetPosition $x $y $z
```

```
                } else {
                    set nodeId " "
                    set nodeName "unknown object"
                }
            }
        } else {
            set focusName "not a glyph"
        }
    }
}



# Move Node function ----------------------------------------------
# The callback proc invoked when the "Move" button is sunken.
# It checks that a valid node has been selected (picked)
# where Move function is active. If so, store the coordinate of a new position
# and set the node position into it.
# No input arguments. No output.



proc MovePoint {} {
    global graph hist memory
    global x y z oldId
    global nodeId moveFlag
    global histBar
    pointWidget GetPolyData point

    set arr [point GetPoint 0]
    set x [lindex $arr 0]
    set y [lindex $arr 1]
    set z [lindex $arr 2]

 if { $nodeId != " "  && $moveFlag == 1 } {

        if { $memory == 0 } {
            set x [$graph GetX $nodeId]
            set y [$graph GetY $nodeId]
            set z [$graph GetZ $nodeId]
            lappend hist "MoveNode $nodeId $x $y $z"

          $histBar.text insert end "MoveNode $nodeId \n"
            set memory 1
```

```
         } else {

            set oldId $nodeId
            $graph Print
            $graph SetPosition $nodeId $x $y $z
            $graph DataHasBeenGenerated

#Callback updateGraph
            UpdateGraph
        }
    }
}

# Move Function.---------------------------------------------------
# The callback proc invoked when the "Move" button is pressed.
# The default value is 0 where move mode is not active. Once it is pressed,
# the mode becomes active. When it is pressed again, it becomes unactive.
# No input arguments. No output.

set moveFlag 0
set memory 0

proc Move {} {
    global moveFlag moveNode_button
    global menustatus nodeId graph oldId
    if {$moveFlag == 1} {
        set moveFlag 0
        $moveNode_button configure -relief raised
        set memory 0



     } else {
        Reset
        set moveFlag 1
        $moveNode_button configure -relief sunken
        set menustatus "Select the node you want to move"
        set oldId " "
     }
}

#Add Node Function.----------------------------------------------
# The callback proc invoked when the "AddNode" button is pressed.
# Get the position picked by pointWidget and store the position into an array.
# Create a new node to the position.
# CreateNode function return nodeId.Also new node has to have a label.
# No input arguments. No output.


set n 1
proc AddNode {} {
     global graph hist h
     global n addFlag carray
     global menustatus histBar

     Reset

     set menustatus "Move the pointwidege and press"
     set arr [point GetPoint 0]

     set x [lindex $arr 0]
     set y [lindex $arr 1]
     set z [lindex $arr 2]
     set nodeId [$graph CreateNode]
     $graph SetPosition $nodeId $x $y $z
     set name [[$graph GetStrings] AddString "<new node$n>"]
     incr n
     set larray [[$graph GetNodeData] GetArray label]
     $larray InsertValue [$graph GetNodeIndex $nodeId] $name
     set carray [[$graph GetNodeData] GetArray colour]
     $carray InsertValue [$graph GetNodeIndex $nodeId] 0

     lappend hist "AddNode $nodeId "
     puts "store addfunction "

   $histBar.text insert end "AddNode $nodeId \n"

     UpdateGraph
     set addFlag 0

}

global plist clist

#DeleteNode-----------------------------------------------------------------
proc DeleteNode {} {
    global graph nodeId hist h
    global sourceList targetList
    global menustatus histBar
```

```
    global delFlag

   if { $delFlag == 1 } {
         set menustatus "choose node to delete by pressing P "

       if { [$graph HasNode $nodeId]} {
         set x [$graph GetX $nodeId]
         set y [$graph GetY $nodeId]
         set z [$graph GetZ $nodeId]

         set parent [$graph GetParents $nodeId]
         set plist [list]
         while {[$parent HasNext]} {
             set src [$parent GetNext]
             lappend plist $src
        }
         lappend sourceList $plist


         set child [$graph GetChildren $nodeId]
         set clist [list]
         while {[$child HasNext]} {
             set des [$child GetNext]
             lappend clist $des
         }
         lappend targetList $clist

         $graph DeleteNode $nodeId
         lappend hist "DeleteNode $nodeId $x $y $z"

         puts "store deletefunction "

         $histBar.text insert end "DeleteNode $nodeId \n"

         UpdateGraph
       }
   }
}



#-------------------------------------------------------------------------------
set delFlag 0
```

```
proc Delete {} {
    global delFlag delNode_button moveNode_button moveFlag
    global menustatus

    if {$delFlag == 1} {
        set delFlag 0
        $delNode_button configure -relief raised
     } else {

        Reset

        set delFlag 1
        $delNode_button configure -relief sunken
        set menustatus "Select the node you want to delete"
     }
}



# Edge add&delete Function -------------------------------------------

proc Edge {} {

    global graph hist h
    global nodeName nodeId x y z
    global addEdgeFlag delEdgeFlag delFlag
    global srcNode desNode
    global menustatus histBar

    if { $addEdgeFlag == 1 || $delEdgeFlag == 1 } {
        if { $srcNode == "none" } {
            set srcNode $nodeId
            set menustatus "Source Node is $srcNode. Now, choose target Node"
         } else {
          if { $nodeId != $srcNode } {
                set desNode $nodeId
                if { $addEdgeFlag == 1} {
                    $graph CreateEdge $srcNode $desNode
                    set menustatus " The edge between $srcNode and $desNode was created. If you want to add another edge,
choose source Node"
                    lappend hist "AddEdge $srcNode $desNode"

            $histBar.text insert end "AddEdge between $srcNode and $desNode \n"
                } else {
                    if { [$graph HasEdge $srcNode $desNode]} {
                        $graph DeleteEdge $srcNode $desNode
                        set menustatus "The edge between $srcNode & $desNode has been deleted "

                        lappend hist "DeleteEdge $srcNode $desNode "

                         $histBar.text insert end "DeleteEdge between $srcNode and $desNode \n"

                    } elseif { [$graph HasEdge $desNode $srcNode]} {
                        $graph DeleteEdge $desNode $srcNode
                        set menustatus "The edge between $srcNode & $desNode has been deleted "
                        lappend hist "DeleteEdge $desNode $srcNode "
                        $histBar.text insert end "DeleteEdge between $desNode and $srcNode \n"
                    } else {
                        set menustatus "It does not have any edges with $srcNode. Try again"
                    }
                }
                UpdateGraph
                set srcNode "none"
            }
        }
    }
}

#Set add/deleteEdgeFlag On and Off -------------------------------------------------------------------------------

set addEdgeFlag 0
set delEdgeFlag 0

proc EdgeFlag { e } {
    global addEdgeFlag addEdge_button nodeId
    global delEdgeFlag delEdge_button
    global srcNode nodeId menustatus

    if { $e == "add"  } {
        if {$addEdgeFlag == 1 } {
            set addEdgeFlag 0
            $addEdge_button configure -relief raised
        } else {
            Reset
            set addEdgeFlag 1
            $addEdge_button configure -relief sunken
        }
    } elseif { $e == "delete"  } {
        if {$delEdgeFlag == 1 } {
            set delEdgeFlag 0
            $delEdge_button configure -relief raised
        } else {
            Reset
            set delEdgeFlag 1
            $delEdge_button configure -relief sunken
        }
    }
    set srcNode "none"
    set menustatus "Choose source Node"
}



#------------------------------------------------------------------

# Refresh the view when the data is modified.

proc UpdateGraph {} {
    global graph
    $graph Modified
    renWin Render
}

# Reset button -------------------------------------------------------

proc Reset {} {
   global moveFlag addEdgeFlag delEdgeFlag delFlag
   global moveNode_button addEdge_button delEdge_button delNode_button
   global menustatus

        set moveFlag 0
        $moveNode_button configure -relief raised
        set addEdgeFlag 0
        $addEdge_button configure -relief raised
        set delEdgeFlag 0
        $delEdge_button configure -relief raised
        set delFlag 0
        $delNode_button configure -relief raised

 }

#------------------------------------------------------------------------
```

```
proc Undo {} {
   global hist
   global sourceList targetList
   global graph histBar

   set index [expr [llength $hist]-1]
   set last [lindex $hist $index]
   scan $last "%s" nm

   if { $nm == "AddNode" } {
      scan $last "%s %d" nm id
      $graph DeleteNode $id

      $histBar.text insert end "Undo:AddNode $id \n"


   } elseif { $nm == "DeleteNode" } {
      scan $last "%s %d %f %f %f" nm id x y z
      $graph CreateNode $id
      $graph SetPosition $id $x $y $z
      set name [[$graph GetStrings] AddString "$id"]
      set larray [[$graph GetNodeData] GetArray label]
      $larray InsertValue [$graph GetNodeIndex $id] $name
      set plist [lindex $sourceList end]

      foreach i $plist {
         $graph CreateEdge $i $id
      }

      set clist [lindex $targetList end]

      foreach j $clist {
         $graph CreateEdge $id $j
      }

      set sourceList [lreplace $sourceList end end]
      set targetList [lreplace $targetList end end ]

      $histBar.text insert end "Undo:DeleteNode $id \n"


   } elseif { $nm == "AddEdge" } {
      scan $last "%s %d %d" nm sid did
```

```
        $graph DeleteEdge $sid $did

        $histBar.text insert end "Undo:AddEdge between $sid and $did \n"

    } elseif { $nm == "DeleteEdge" } {
        scan $last "%s %d %d" nm sid did
        $graph CreateEdge $sid $did

        $histBar.text insert end "Undo:DeleteEdge between $sid and $did \n"

    } elseif { $nm == "MoveNode" } {
        scan $last "%s %d %f %f %f" nm id x y z
        $graph SetPosition $id $x $y $z

            $histBar.text insert end "Undo:MoveNode $id \n"

        Reset
    }

    UpdateGraph

    set hist [lreplace $hist end end]

}


proc OpenFile {} {
    global graph gsource
    global renWin

    set types {
        {{GML file}        {.gml} }
        {{All Files }      *      }    }

    set filename [tk_getOpenFile -filetypes $types]
    if {$filename != " " } {
        if { [string match *.gml $filename] } {
            $graph SetSource $gsource
            ren RemoveAllProps
            ren AddActor actor
            ren AddActor glyphActor
            reader SetFileName $filename
            reader Update
            set graph [reader GetOutput]
            set gsource [$graph GetSource]
            strahler SetInput $graph
            nodes SetInput $graph
            $graph SetSource ""
            renWin Render

            wm title . $filename

        } else {
            puts " Can't read this file"
            return
        }
    }
}

proc SaveFile {} {
    global graph

    set types {
        {{GML file}        {.gml} }
        {{All Files }       *      }
    }
    set filename [tk_getSaveFile -filetypes $types]
    if { $filename != "" } {
        if { [string match *.gml $filename] } {
            vtkGMLWriter writer
            writer SetFileName $filename
            writer SetInput $graph
            writer Write
            writer Delete
        } else {
            puts "Can't write this file"
            return
        }
    }
}


vtkRendererSource rsource
  rsource SetInput ren
  rsource WholeWindowOn


##########################################################################################
```

```
# Create the GUI                                         | ___________________________________________|
# Constructed of 6 widgets.                              |                .top.menuBar                 |
#                                                        |----------------------------|--------------------------------|
# 1:menuBar--- .top.mbar                                 |        .top.f.f1           |  .top.f.f2.coord               |
# 2:Function Buttons--- .top.f.f1                        |    ( Function button)      |--------------------------------|
# 3:Showing xyz coordinate--- .top.f.f2.coord            |                            |  .top.f.f2.property            |
# 4:Showing nodeId & nodeName---.top.f.f2.property       |----------------------------|---------------------------------|
# 5:vtkRenderWidget--- .top.win                          |                                                              |
# 6:statusBar--- .top.statusBar                          |                      .top.window                             |
# 7:histryBar--- .top.histBar                            |                                                              |
#                                                        |                  ( VTK renderWindow)                         |
#                                                        |--------------------------------------------------------------|
#                                                        |                     .top.statusBar                           |
#                                                        |--------------------------------------------------------------|
#                                                        |                     .top.histBar                             |
#                                                        |______________________________________________________________|
#


wm withdraw .



toplevel .top  -visual best
wm title .top "Graph Editor"
wm protocol .top WM_DELETE_WINDOW ::vtk::cb_exit

set nodeId " "
set nodeName " "
set pid 0
set menustatus "Press any of function buttons to start manipulation "


# Menu bar (mbar) -------------------------------------------------------------
# File --- OpenFile, SaveFile, Exit
# Edit ---
# View
# Option
# Help


frame .top.mbar -relief raised -bd 2
```

```
foreach m {File Edit View Help} {
    set lower [string tolower $m]
    menubutton .top.mbar.$lower -text $m \
            -menu .top.mbar.$lower.menu

    pack .top.mbar.$lower -side left
}

menu .top.mbar.file.menu
   .top.mbar.file.menu add command -label Open -command OpenFile
   .top.mbar.file.menu add command -label Save -command SaveFile
#-state disabled
   .top.mbar.file.menu add command -label Exit -command exit

menu .top.mbar.edit.menu
   .top.mbar.edit.menu add command -label Clean -command Clean -state disabled
   .top.mbar.edit.menu add command -label "Undo/Redo" -command Undo

menu .top.mbar.view.menu
   .top.mbar.view.menu add radiobutton -label Front -variable view -value Front \
         -command FrontView



proc FrontView {} {
    ren ResetCamera
    renWin Render

}






menu .top.mbar.help.menu
   .top.mbar.help.menu add command -label "Help" -command Help



proc Help {} {
   toplevel .top.help  -visual best
   wm title .top.help "Help"
   wm protocol .top.help WM_DELETE_WINDOW ::vtk::cb_exit

   text .top.help.text -relief sunken -bd 2 -yscrollcommand ".top.help.scroll set" -setgrid 1 \
      -height 30 -undo 1 -autosep 1
    scrollbar .top.help.scroll -command ".top.help.text yview"
    pack .top.help.scroll -side right -fill y
    pack .top.help.text -expand yes -fill both

    .top.help.text tag configure big -font {Courier 14 bold}
    .top.help.text tag configure bold -font {Courier 12 bold}

.top.help.text insert 0.0 {Basic functions are following.} big
.top.help.text insert end {

Press "j"/"t" --- Toggle between joystick and trackball styles.
Press "p" key --- Picking
Button1 ( mouse left) --- Rotate
Button2 ( shift + mouse middle) --- Pan
Button2 ( shift + mouse right) --- Zoom in/up

}
.top.help.text insert end {Add Node} bold
.top.help.text insert end {
1.Move the centre of pointWidget to where you want to position node
2.Press "AddNode" button

}

.top.help.text insert end {Delete Node} bold
.top.help.text insert end {
1.Press "DeleteNode" button
2.Choose the node you want to delete

}

.top.help.text insert end {Add Edge} bold
.top.help.text insert end {
1.Press "AddEdge" button
2.Choose the first node to be source Node
3.Choose the second node to be destination Node

}
.top.help.text insert end {Delete Edge} bold
.top.help.text insert end {
1.Press "DeleteEdge" button
2.Choose the first node to be source Node
3.Choose the second node to be destination Node
}

.top.help.text insert end {Move} bold
.top.help.text insert end {
1.Press "P" key to choose the Node you want to move
2.Drag node }
.top.help.text mark set insert 0.0

}



#if { [llength $hist] < 0 } {
#    .top.mbar.edit.menu entryconfigure 1 -state disabled
#    .top.mbar.edit.menu entryconfigure 1 -state disabled
#if { [llength $hist] > 0 } {
#    .top.mbar.file.menu  -state active
#    .top.mbar.edit.menu  -state active
#} elseif { [llength $hist] == 0 } {
#    .top.mbar.edit.menu  -state disabled
#
#}



########################################################################



# Function button (f1) ----------------------------------------------------------
set w .top.f

set a "add"
set d "delete"

frame $w

frame $w.f1

labelframe $w.f1.node -pady 4 -text "Node" -font {Courier 12 bold} -padx 4
labelframe $w.f1.edge -pady 4 -text "Edge" -font {Courier 12 bold} -padx 4

pack $w.f1.node $w.f1.edge -side top -expand yes -pady 4 -padx 4 -anchor w

set addNode_button [button $w.f1.node.add -text "Add" -font {Courier 12 bold} -bg PeachPuff1 -command AddNode]
set delNode_button [button $w.f1.node.del  -text "Delete" -font {Courier 12 bold} -bg PeachPuff1 -command Delete]
set addEdge_button [button $w.f1.edge.add -text "Add"  -font {Courier 12 bold} -bg LightBlue1 -command " EdgeFlag
$a " ]
set delEdge_button [button $w.f1.edge.del -text "Delete"  -font {Courier 12 bold} -bg LightBlue1 -command " EdgeFlag
$d " ]
set moveNode_button [button $w.f1.node.moven -text "Move" -font {Courier 12 bold}  -bg PeachPuff1 -command Move
]



pack $addNode_button $delNode_button $moveNode_button \
    -side left -pady 2 -anchor w -fill x
pack $addEdge_button $delEdge_button -side left -pady 2 -anchor w -fill x



# xyz coordinate (.top.f.f2.coord)---------------------------------------------------------

frame $w.f2
global x y z

labelframe $w.f2.coord -pady 4 -text "Coordinate" -font {Courier 12 bold} -padx 4

foreach l {X Y Z} {
   set lower [string tolower $l]
   label $w.f2.coord.$lower-label -text $l

   label $w.f2.coord.$lower-Position -textvariable $lower \
        -bg LemonChiffon \
        -relief sunken \
        -width 10 \
        -anchor w
}

grid $w.f2.coord.x-label     -row 0 -column 0 -sticky w -pady 3m
grid $w.f2.coord.x-Position  -row 0 -column 1 -sticky w -padx 2m -pady 3m
grid $w.f2.coord.y-label     -row 0 -column 2 -sticky w -pady 3m
grid $w.f2.coord.y-Position  -row 0 -column 3 -sticky w -padx 2m -pady 3m
grid $w.f2.coord.z-label     -row 0 -column 4 -sticky w -pady 3m
grid $w.f2.coord.z-Position  -row 0 -column 5 -sticky w -padx 2m -pady 3m



# Showing nodeId and nodeName (.top.f.f2.label) ---------------------------------------------

labelframe $w.f2.property -pady 4 -text "Property" -font {Courier 12 bold} -padx 4

foreach m { Id Name } {
    set lower [string tolower $m]
    label $w.f2.property.node$m-label -text " $m :"
    label $w.f2.property.node$m-text  -textvariable node$m \
          -bg LemonChiffon \
          -relief sunken \
          -width 17 \
          -anchor w

}

grid $w.f2.property.nodeId-label -row 0 -column 0 -sticky w
grid $w.f2.property.nodeName-label -row 0 -column 2 -sticky w
grid $w.f2.property.nodeId-text    -row 0 -column 1 -sticky w
grid $w.f2.property.nodeName-text  -row 0 -column 3 -sticky w


pack $w.f2.coord $w.f2.property -side top

pack $w.f1 $w.f2 -side left -expand yes -pady 2 -padx .5c -anchor w



# Render Window (window) ---------------------------------------------------------

frame .top.win

vtkTkRenderWidget .top.win.window -width 600 -height 400 -rw renWin

scrollbar .top.win.xscr -orient vertical
scrollbar .top.win.yscr -orient horizontal

grid .top.win.window .top.win.xscr -sticky nsew
grid .top.win.yscr  -sticky nsew

# StatusBar (statusBar) ------------------------------------------------------------------
# This shows current status and next process.
# It is located in the bottom of screen.

frame .top.statusBar
label .top.statusBar.label -textvariable menustatus -relief sunken -bd 1 \
            -bg Lightgreen -font "Helvetica 10" -anchor w
pack .top.statusBar.label -side left -padx 2 -expand yes -fill both

# History Bar ------------------------------------------------------------------------------
```

```
global histBar

set histBar .top.histBar

frame $histBar
text $histBar.text -relief sunken -bd 2 -yscrollcommand "$histBar.scroll set" -setgrid 1 \
    -height 5 -undo 1 -autosep 1

scrollbar $histBar.scroll -command "$histBar.text yview"

pack $histBar.scroll -side right -fill y
pack $histBar.text -expand yes -fill both

$histBar.text insert 0.0 " History \n"



##########################################################################################

pack .top.mbar -side top -fill x
pack $w -side top
#pack .top.f3  -side top
pack .top.win -side top -expand t -fill both
pack .top.histBar .top.statusBar -side bottom -fill x -pady 2



::vtk::bind_tk_render_widget .top.win.window
bind .top.win.window <KeyPress> { };



#[renWin GetInteractor] StartPickCallback "OnLeftButtonDown "
#[renWin GetInteractor] EndPickCallback "OnLeftButtonUp"

[renWin GetInteractor] SetPicker picker
#pointWidget SetInteractor [renWin GetInteractor]
pointWidget AddObserver InteractionEvent MovePoint

renWin Render
tkwait window .
```