AN OBJECT ORIENTED APPROACH TO 3D NETWORK VISUALIZATION

DONG LIN CHEN

A MAJOR REPORT

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER
CONCORDIA UNIVERSITY
MONTREAL, QUEBEC, CANADA

AUGUST 2002

ABSTRACT

An Object Oriented Approach to 3D Network Visualization

Dong Lin CHEN

A network diagram is a familiar graphic form that can represent may different kinds of information. To achieve an aesthetic layout presentation, in a 3-dimensional orthogonal drawing of a graph, vertices are mapped to grid points on a 3D rectangular integer lattice and edges are routed as segments along grid lines. In this report, we describe the design and implementation of an objected-oriented 3D network graphic layout program built upon OpenGL with simple and efficient layout algorithm. The system provides an easy-to-adapt graphic drawing framework for visualizing a network and for experimenting different layout algorithms in a pipelined fashion. Current implementation restricts vertices with maximum degree of six, and guarantees no crossings and optimized shortest path for edges.

Acknowledgments

Special thanks to **Professor Peter Grogono**, who gives me this opportunity to explore graph theory, object-oriented programming and OpenGL. I would like to express my gratitude for his supervision, valuable instructions and feedbacks.

I am grateful to **Xiao Wei She,** my wife, who supportively shared all the housework and looked after the kids, **Ao** and **Miao,** when I indulged myself in the project.

Thanks to my friends, **Ming Hao Xie, Wen Tian, Dong Yang and Tian Fang,** their friendship, humors and discussions made the study at Concordia University an enjoyable journey.

Table of Contents

Table of Figures

1 Introduction

1.1 Visualizing a Network

The explosive growth of computing has led to the desire to visualize huge amount information. A network is one of such example that could be applied to represent many different kinds of information. Network graphs may represent much of this information with the nodes corresponding to objects and the links to relationships among the objects. In other words, a network is an abstract representation of information that can be viewed as having nodes and links. A network is a labeled connected graph, either directed or non-directed. The most common technique for visualizing networks involves node and link diagrams [1]. Glyphs, graphical objects, represent the nodes positioned spatially with lines drawn between them encoding the link relationships. The conventional and intuitive way to represent a graph visually is to draw nodes as boxes and edges as line segments connecting the boxes [2]. There are many ways to draw a network diagram, such as poly-line drawing, straight-line drawing, orthogonal drawing and planar drawing.

Drawing graphs is an important problem that combines flavors of computational geometry and graph theory [3]. The usefulness of a graph depends on its capacity of conveying the meaning of a diagram effectively. Every approach to diagram layout representation requires some aesthetic criteria. One aesthetic principle is to minimize crossings between edges. Overlapping nodes and intersections between nodes and links must also not be allowed. Besides, diagram volume, aspect ratio, number of bends, and node-density distribution are often taken into account. Achieving such aesthetic can be

formulated as optimization goals for drawing algorithm [4]. There has been a lot of research on topic of graphic layout algorithms [5], both in 2D and 3D, to represent non-directed or directed graphs.

In this paper, we describe a 3-dimensional orthogonal grid drawing for a network of a maximum degree at most 6. That is, any node will have at most 6 links going in and out to other counterparts. The 3-dimensional orthogonal grid consists of grid points whose coordinates are defined as (x, y, z) with integer values, together with the axis-parallel grid lines determined by these grid points. A 3-dimensional orthogonal drawing of a graph G is an embedding of G into the 3-dimentional orthogonal grids with the vertices located at the grid points and each edge represented by a sequence of contiguous orthogonal segments of grid lines. An edge joins the two end points corresponding to vertices of u and v. An edge representing a network link is directed, that is, an arrow is drawn to indicate the link direction from one node to the other.

In a 3-dimensional orthogonal grid drawing, certain properties are observed. No pair of edge routes is permitted to intersect, except at common end vertex endpoints. Each grid point corresponding to a vertex can have at most 6 ports to be used for attaching edges, we name them as *front, back, left, right, top* and *bottom*. Other properties of drawing include volume, total edge length, total number of bends and aspect-ratio. A bounding box is used to describe the drawing, which is the minimum axis-parallel cubic that covers all the vertices and edges. We use the term volume to describe the exact number of grid points within a bounding box. Total edge length is the sum of lengths of the edge. A bend

is the point where two segments form the same edge meet. Aspect ratio is the longest to the shortest side of the smallest rectangle with horizontal and vertical sides covering the drawing. There are infinitely many drawings for a graph; however, we would like to take into account a variety of criteria, such as those mentioned above to achieve an optimized layout. Thus, many graph drawing problems can be formalized as multi-objective optimization problems, so that trade-offs are inherent in solving them.

1.2 Visualization Pipeline

Generally speaking, visualization comprises methods and techniques to generate image from some pre-processed information. Visualization often proceeds in several stages. That forms the so-called visualization pipeline [6].

To visualize a network, the following stages of processing are often observed:

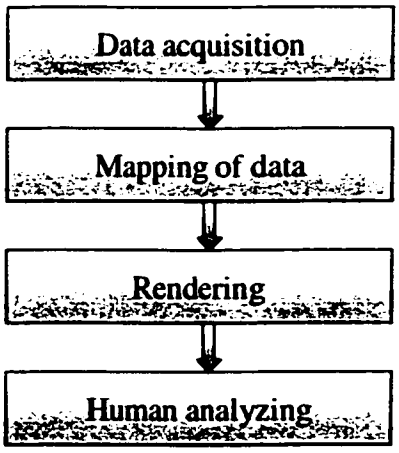

Figure 1: Visualization processing pipeline

At the very beginning is the data acquisition stage. This stage gathers the information from either pre-processed data or a design blueprint of a system. Any information that can be abstracted as a network can be represented in a format by later processing for visualization. After that, a mapping of the network information onto the attributes of a visual representation is performed. Attributes of this visual representation can be geometry, color, transparency, and material properties. We abstract the mapped data into geometric objects and assign coordinates to the vertices and edges. The rendering is projecting the visual representation onto an image planes for user analyzing. After analyzing an image the user may change various parameters in the different stages of the visualization pipeline to further increase the insight into the properties of a given diagram.

A pipelined visualization approach enables rapid and easy adaptation to different source data and rendering libraries. It isolates different processing stages and separates graph drawing algorithm from how a graphic is rendered. Changes in any components can be made without affects others. In this report, we will focus on implementation of a network visualization system that reads input of a network description and generates a 3D graphic layout, as well as outputs coordinates of vertices and edges. We try to formulate the system in a pipelined fashion.

1.3 Contribution of this Report

This report and its implemented program may be useful for future research on drawing algorithms and graphic layout representation. The implementation tries to achieve a generic graphic layout framework in a pipelined fashion. Its objected-oriented approach with a model-view-controller architecture makes the program be easily extended to accommodate different input data formats, layout algorithms and rendering libraries. In addition, this report introduces a simple and efficient heuristic algorithm for designing the network layout representation.

1.4 Organization of the Report

The rest of the report is organized as follows: Section 2 overviews the problem statement and background literatures related to graph drawing techniques. Section 3 focuses on the analysis of the graph drawing strategy. Section 4 is the design of the object-oriented network visualization framework. Implementation details and solutions to specific problems are described in Section 5, and the results of layout are presented in Section 6. Finally, a conclusion and future work remark is given in Section 7.

2 Background

2.1 Graph Drawing Glossary

First of all, it may be helpful to refer some terminology on graphs pertinent to graph drawing that has been established in this community. Here we only list the terms related to this report and detailed references can be found in [7, 8].

degree-k graph: graph with maximum degree d <= k

digraph: directed graph, i.e., graph with directed edges drawn as arrows

acyclic digraph: without directed cycles

transitive edge: edge (u, v) of a digraph is transitive if there is a directed path from u to v not containing edge (u, v)

reduced digraph: without transitive edges

source: vertex of a digraph without incoming edges

sink: vertex of a digraph without outgoing edges

connected graph: any two vertices are joined by a path

biconnected graph: any two vertices are joined by two vertex-disjoint paths

triconnected graph: any two vertices are joined by three vertex-disjoint paths

straight-line drawing: each edge is a straight line segment

orthogonal drawing: each edge is a chain of horizontal and vertical segments

bend: in a polyline drawing, point where two segment parts of the same edge meet

crossing: point where two edges intersect

grid drawing: polyline drawing such that vertices crossings and bends have integer coordinates

planar drawing: no two edges cross

planar di-graph: admits a planar drawing

2.2 Problem Statement and Requirements

A "network" consists of components connected by links. The description of a network is depicted as in the following format:

241 I1=245 I2=246 O1=242 O2=245

242 I1=245 I2=241 O1=246

245 I1=241 O1=242 O2=241

246 I1=242 O1=241

This means that component 241 has two inputs, from components 245 and 246, and two outputs, to components 242 and 245, similarly for the others.

The problem is to convert the above output of a network description into a graphical representation and display on screen. A graphic layout designer maps network nodes and links into a graph in which component is a vertex and the inputs and outputs correspond

to edges. The problem that a layout designer has to solve is easy to state but hard to implement.

Here is a formal description of the problem statement [9]:

> Given a graph $G = (V, E)$, assign a coordinate to each vertex in V in such a way that a visual representation of the graph that uses these coordinates will be easy to read and aesthetically pleasing.

Given a network representation of $G(V, E)$, it has the following properties: it is a digraph, which means each link is directed and will be drawn with arrows. It may not be a connected graph since nodes could stand alone without any connections. (This may be true in real situation when a server being setup but not connected yet, however, we try to abstract from real world network)

A graph is described simply by the set of its vertices and the set of its edges, no particular geometry is associated with it. The graph layout designer embeds the abstract graph in some geometric space, i.e., it assigns a position to each vertex and a curve to each edge. The goal is to find a geometrical configuration of the vertices and edges such that the layout is easy to read and aesthetically pleasing.

This task can be characterized by a list of user requirements for visualization. A summary of the user requirements for the graph layout designer includes:

- Readability of the graph

  The information contained in the graph should be easy to read, that is, it should effectively display the information of interest to the user. This implies that some drawing criteria [4][5][7] should be followed simultaneously when assigning the coordinates. The following lists some of them:

  - minimize the number of edge crossings;
  - draw edges as straight as possible;
  - vertices should be evenly distributed;
  - the majority directed edges should be drawn pointing in the same direction;
  - in polyline drawings, minimize bends in the edges;
  - minimize the area of the area drawing;
  - maximize display of symmetries;

- Easy visual analysis

  Design the system to provide interactions for users to navigate the network representation by using rotation, shifting and zooming. And also, other graphic technique such as depth cuing, perspective viewing, texture and surface characteristics can be introduced to enhance virtual reality.

- Produce text output of coordinates for graphic representation

The system should produce an intermediate text log that records the coordinates of vertices and edges. The formatted data is required to feed into other rendering engines.

- Easy to adopt different graph drawing algorithms

  Different layout algorithms can be easily adopted into the system without major re-implementation of the framework. This allows the users to quickly and easily realize the effects of such algorithm.

- Reusable components

  To achieve the reusability, the designed system should be considered as a generic framework and visualization is processed in a pipelined fashion. The visualization process should have well-formed input and output formats at each stage.

2.3 Graph Theory and Drawing Techniques

Research on graph drawing has been especially active in the last decades. Literatures on the subject are hundreds [5][7]. Most of the previous work focuses on the graph drawing algorithms and its computing complexity, layout aesthetics, drawing constraints. Graph drawing involves tradeoffs between those drawing properties. The decision mainly depends on the domain requirements and the speed of computer in computing and graphic rendering.

2.3.1 Complexity Issues and Constraints

To achieve the above-mentioned aesthetic criteria, there is a computation complexity associating to the optimization. In general, the optimization problems associated with these aesthetics mentioned in previous section are NP-hard [4]. (A problem is NP-hard if an algorithm for solving it can be translated into one for solving any other NP-problem (nondeterministic polynomial time) problem. NP-hard therefore means "at least as hard as any NP-problem," although it might, in fact, is harder [10]). Research shows that, minimizing the number of crossings are NP-hard, minimizing the total number of bends is also NP-hard [4]. Besides time complexity limitations, the above aesthetics are also "competitive" in that the optimality of one often prevents the optimality of others [5]. That is, in most cases, you cannot simultaneously optimize two aesthetic criteria [4]. Because of such difficulties, general approaches to graph drawing are usually heuristic.

There are also factors that beyond the aesthetic criteria, these factors place constraints on graph drawings. Some readability aspects require knowledge about the semantics of the specific graph (e.g., place "most important" vertex in the middle). Constraints are provided as additional input to a graph drawing algorithm. Constraints, one the one hand, can help to optimize our final graph representation, but it may also make the optimization more complicated [4].

2.3.2 Algorithmic Approach vs. Declarative Approach

Algorithmic approach produces a layout of the graph generated according to a pre-specified set of aesthetic criteria. Aesthetic criteria embodied in an algorithm as optimization goals, e.g. minimization of crossings, minimization of area. The Advantages of algorithm approach is its computational efficiency. However, it has a drawback of that, user-defined constraints are not naturally supported [4]. Extensions are explored to attach a limited constraint-satisfaction capability within the algorithmic approach

Declarative approach uses a set of user-defined constraints and produces layout generated by the solution of a system of constraints. In other words, this is a rule-based approach. Advantages of this approach are its expressive power in defining the rules of how to generate the graph. However, some natural aesthetics (e.g., planarity) need complicated constraints to be expressed, and general constraint-solving systems are computationally inefficient [4]. It lacks a powerful language for the specification of constraints (currently done with a detailed enumeration of facts, or with a set notation)

2.3.3 Orthogonal Grid Drawings

Many papers on algorithms and bounds for aesthetic criteria of three dimensional orthogonal graph drawings have been published. The *Slices* algorithm proposed by Biedle shows a linear time algorithm that draws a graph in $O(n^2)$ volume with at most 14 bends per edge [11]. The drawing is obtained by placing all the vertices on a certain horizontal plane and by assigning a further horizontal plan to every edge, i.e., "one slice per edge".

An $O(n^{2/3})$ time algorithm *Komogorov* is introduced by Eades, Stirk and Whitesides [12] based on augmenting the graph to a Eulerian graph and on applying a variation of an algorithm by Komogorov and Barzdin. The algorithm produces drawings having $O(n^{3/2})$ volume and at most 16 bends per edge. The *Compact* algorithm proposed by Eades, Symvonis, and Whitesizes [13] requires $O(n^{3/2})$ time and volume and introduces at most 7 bends per edge. In the same paper, they introduced a *three-bends* algorithm whose complexity is linear with a volume of $27n^3$ and at most 3 bends. This algorithm is based on augmenting the graph to a 6-regurlar graph and on a coloring technique.

Papakostas and Tollis present a linear time algorithm *Interactive* [14] that requires at most $4.63n^3$ volume and at most 3 bends per edge. Its key feature is incremental and it can be extended to draw graphs with vertices of arbitrary degree. The arbitrary vertices are represented by solid 3-D boxes whose surfaces is proportional to their degree. The produced drawing has two bends per edge and guarantees no crossings and can be used under interactive setting.

Woods [15] presents an algorithm for 3-dimensional orthogonal graph drawing based on the movement of vertices from an initial layout along the main diagonal of a cube. The result shows at most 4 bends per edge for a maximum of 6 degrees. The volume is at most of $2.37n^3$ and the total number of bends is always less than $7\frac{m}{3}$. (m is the number of edges). He also extends the 3-BENDS algorithm to produce 3-bend drawings with n^3

+ $O(n^3)$ volume, which is the best known upper bound for the volume of 3-D orthogonal graph drawings with at most three bends per edge [16]. Woods also in [17] presents the DLM (Diagonal Layout and Movement) algorithm for 3-D orthogonal grid drawing that combines the layout- and routing-based approaches and can produce general position drawings with an average of at most 2 and 2/7 bends per edge.

The *Dynamic Staircase* algorithm presented in [18] introduces a volume bounded by $O(n^2)$ for 3D orthogonal drawing of degree 6. This algorithm supports full dynamic insertion and deletion of vertices and edges with O(n) time. In the same paper, it also introduces a *Dynamic Spiral* algorithm, which draws in a bounding box of $O(\sqrt{n}) \times O(\sqrt{n}) \times O(n)$ at the expenses of allowing 7 bends per edge.

An ad-hoc approach to 3-dimensional orthogonal graph drawing is presented in [19] as *Reduce-Forks*. This algorithm starts with degenerate drawing with all vertices on one point and repeatedly inserts planes splitting the drawing apart until all crossings are removed.

Although the algorithms presented in above-mentioned papers are interesting and have been explored deeply. They are generally good but complicated in analysis and implementation. As we mentioned early, most of the aesthetic criteria optimization is a trade-off between different metrics, and the problem is generally NP-Hard. In most cases, we do not necessary require such limitations on bounds and bends of edges, therefore, in

later section; we will discuss a simple and efficient heuristic approach used in this report for generating a 3D graphic network layout.

2.4 From 2D to 3D

Almost all of the early work on visualization has involved 2D presentations. Moving network visualization from 2-dimensional to 3-dimensional has the obvious advantage of obtaining an extra degree of freedom. However, it is beneficial to determine whether the expense and complexity of using three dimensions could significantly improve visualization effect and help understanding of a network diagram.

While there has been much work in 2D displays, the number of different strategies for visualization has been quite limited. Most displays are simple graphs containing boxes and arcs. However, in 3D spaces, the representation of visualized objects will be much richer. In recent years, 3D graph drawing and visualization has drawn significant attention from graph research community.

Recent advances in hardware and software technology for computer graphics open the possibility of displaying three dimensional visualizations on a variety of low cost workstations and a handful of researchers have begun to explore the possibilities of displaying graphs using this new technology. Previous research on 3D graph drawing has focused on the development of visualization systems; however, much work needs to be done on the theoretical foundations of 3D graph drawing [7].

Most of the solutions devised in early days for utilizing 3D for data visualization involve extending what is normally a 2D representation into a 3D one. This is desirable since it maintains a 2D philosophy and presentation, allowing the viewer to see all the data at once while also allowing the additional dimension to be used for a variety of purposes. There are a variety of techniques that can be used here. Several of these techniques do a 2D layout and then extend the graph into the third dimension using some property of the data. Other solutions to moving from 2D to 3D space attempt to actually use the full capabilities of three dimensions without attempting to preserve a full 2D view from some perspectives.

2.5 Rendering Engine and OpenGL

Combined advances in hardware and software technology for computer graphics produces high cost-effective rendering facilities. Displaying three-dimensional visualization on a variety of low cost workstations is feasible. Once we have mapped the vertices and edges of a graph, we come to how to render the geometric objects on a computer screen. These geometric objects are simply cubics, pipes and cylinders.

Choosing a rendering package is based on the following factors: availability, platform independent and performance. There are a couple of rendering libraries available, for example, OpenGL, QuickDraw 3D, Direct 3D, etc. Some of them are free packages

whereas others are proprieties of respective corporations. OpenGL turns out to be the candidate suitable for our research purposes.

OpenGL is a cross-platform standard for 3D rendering and 3D hardware acceleration. There are software runtime libraries available shipped with all Windows, MacOS, Linux and Unix systems. It delivers fast and complete 3D hardware acceleration [20]. Our experience of using it in this project turns out to be satisfactory.

2.6 OpenGL Features and its Use for Visualization

We choose OpenGL for its platform independent interface and its availability on different platforms. Moreover, we are also interested in OpenGL's interactive 3D features that are essential to the usability of a network visualization system. Bear in mind, a 3D network visualization system should provide easy navigation, satisfactory 3D effect, and on-line interactions with users.

2.6.1 Geometric Primitives

OpenGL is a software interface to graphics hardware. The interface provides about 150 distinct commands to specify the objects and operations needed to produce interactive three-dimensional applications [21] [22]. All modules are built from primitive parts, and OpenGL has functions for drawing primitive objects, including points, lines, and polygons. Such commands might allow us to specify relatively complicated shapes such

as automobiles, parts of the body, airplanes, or molecules. There are also a few functions for building familiar objects, such as spheres, cones, cylinders, toruses and teapots, and certain sophisticated libraries built upon OpenGL provide these features. The primary use in this project is to use OpenGL to build graphic representation of vertices and edges. This is an easy task by using GLU and GLUT libraries. We use the familiar object cubic for denoting a vertex, and a pipe attached with a cylinder as arrow to denote an edge.

2.6.2 Callbacks and Interactions

OpenGL also provides callback functions to handle user events. This enables users to have interactions with the rendering system. If you want to handle an event such as keystroke or mouse movement, you may register your functions with OpenGL callbacks. With this feature, we can implement the visualization system to accept user inputs to dynamically add/delete network nodes, reorganizing layout, and navigating the layout insights.

2.6.3 Viewing 3D Model and Navigation

OpenGL provides rich transformation functions for viewing a 3D model in different aspects. The effect of a transformation is to move an object, or a set of objects, with respect to its coordinate system. These features allow the most advanced visual effects when viewing a 3D model. The viewing transformation is used to position the eye (camera) position and adjust the aim and the direction of viewing. The modeling transformation is used to position and orient the model. You can rotate, translate, or scale

the model – or perform some combination of these operations. Note that, the viewing transformation and the modeling transformation are complementary, the effect of applying viewing and modeling transformation should be judged simultaneously [22]. The projection transformation is to determine what the field of view or view volume is and therefore what objects are inside it and to some extent how they look. The project transformation also determines how the objects are projected onto the screen, that is, perspective or orthographic. In addition, the viewport transformation determines the size and location of the available screen area into which the scene is mapped.

All the different kinds of transformation, or their combination, are essential to build a visualization layout with easy navigating capacities. Layout of network graphs, especially large graphs with hundreds nodes and links, will be of very limited use or no use at all if there are no navigating abilities.

Many other OpenGL features may also help on building a more realistic and high usability graphic layout system. For example, we can improve the illusion of depth by simulating the lighting of a scene. We can also apply texture images onto geometric objects to make them look more realistic. We will discuss details of how those features can be used in this project in later section.

3 A Strategy for Constructing 3D Orthogonal Drawings

We choose to use a 3D orthogonal grid drawing to represent our network layout based on the requirement of that, there should be no edge crossings. A 3D orthogonal grid drawing can produce aesthetically pleasing layout. Network is a directed graph, and its nodes can be represented at grid points in 3D spaces and the links are chains of segments parallel to the axes. An arrow is drawn at the ends of a link to represent the direction of network connection.

Strategies can be applied to the layout designer for assigning coordinates to vertices and edges according to the desired aesthetic criteria. A general strategy would construct the drawing in a sequence of steps. At each phase, we apply certain optimization criteria. The focuses of optimization include vertex scattering, direction distribution, edge routing and crossing removal.

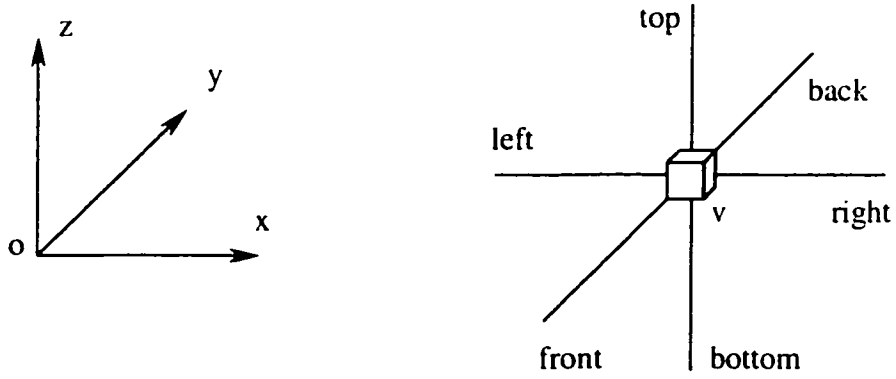

Figure 2: (a) 3D coordinates (b) possible free ports connecting vertex v

Each vertex of a 3D orthogonal drawing has six possible ports around it from incident edges that may connect to the vertex; namely, *top, bottom, left, right, front* and *back*. The two directions parallel to z-axis are *top* (extending towards the positive part of z-axis) and *bottom* (extending towards the negative part of z-axis), which enters v perpendicular to the xy-plane. *Front* (extending towards the negative part of y-axis) and *back* (extending towards the positive part of y-axis) directions are parallel to the y-axis and enter v perpendicular to the xz-plane. The remaining two directions are parallel to the x-axis and are called *left* (extending towards negative part of x-axis) and *right* (extending towards positive part of x-axis), they enter v perpendicularly to the yz-plane. The six contiguous grid points of vertex v is called its *neighbors*, which could be potential points for connected vertex or edge joints. If the neighboring grid point is taken by either a vertex or an edge joint (two grid lines merge to form a straight line is still considered as a joint), this neighbor is considered *blocked* and thus cannot be allocated for vertex point or edge joint. The opposite case will consider as *free* port, and consequently allocation of either vertex or edge joint is allowed.

As shown in Figure 3, vertex v is placed on a grid point at the center, it neighbors *left* and *right* are blocked since they are allocated to vertices w and u, and also its *bottom* neighbor is also blocked since there is an edge joint on the bottom grid points. However, its *top*, *front* and *back* neighbors are *free* since there are neither vertices nor edge joints at these neighbors. To route an edge connecting nodes at grid points A and D, we cannot go

via the center point which is taken by *v*. Instead, we should seek a free port, for example, B, and route along C and D. The total number of bends of routing this edge will be 1.

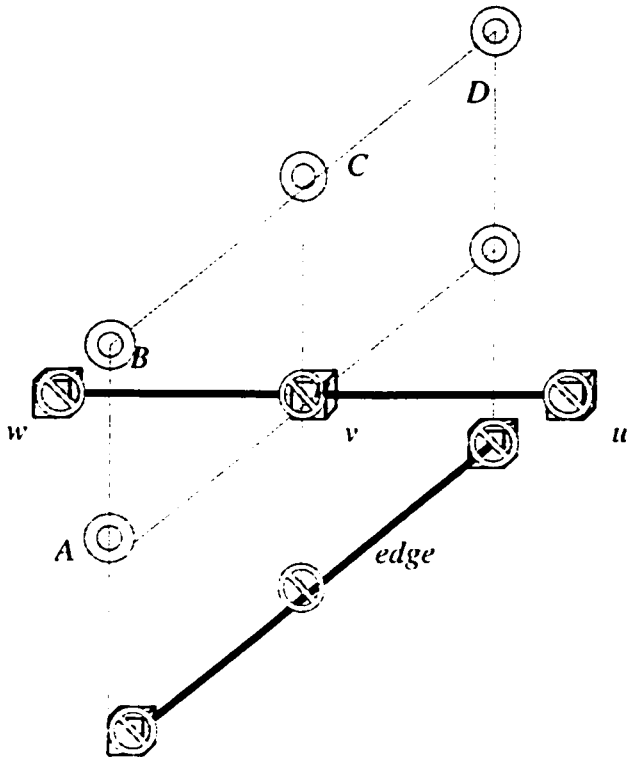

Figure 3: Illustration of free *and* blocked *neighbors*

All the grid points have a relative *distance* to others. The distance of *v* to *u* (or *u* to *v*) is simply the sum of the absolute differences of the 3 axes. That is,

$$distance(u,v) = distance(v,u) = sum(abs(vx - ux) + abs(vy - uy) + abs(vz - uz))$$

(vx, vy, vz) and *(ux, uy, uz)* denotes the coordinates of grid point of *v* and *u*.

In the following sections, we present our strategy of how a vertex is placed in a graph and how its incident edges are routed. The decision on where to place a vertex and how its

incident edges will be routed depends entirely on the free ports around the adjacent vertices and the criteria for optimization. The layout must be performed heuristically since there is, in general, no optimal layout.

3.1.1 Placing Vertices and Controlling Distribution Balance

Our approach to choose which vertex to be placed first and where it will be placed based on the intuition of that, vertices have higher degrees should be placed first; along with all its incident edges and adjacent vertices, a kind of *breadth-first* strategy. This will localize the edge routings in a small area as much as possible and reduces the chances of blocking free ports by surrounding vertices and edge joints.

We initialize all the grid points as *free* and sort them according to the *distance* to the origin of the coordinate. The next available grid point will be the one that is free and has shortest distance to the origin of the coordinate. Since the grid points are sorted according to their distance to the origin, the choosing of next available grid points will control where the vertex be placed and also the vertex distribution. As we can see, vertices distribution will be balanced along all the six dimensions since they have equal distance to the origin.

Assume we start from an empty graph; we choose the first vertex v that has the highest rank of degrees and place it at the center of the coordinate. Then we look at all incident edges of v, i.e, $e(v, u)$. For each vertex u of $e(v, u)$, we place it at next available grid

points, we also rout this edge *e(v, u)* immediately according to the shortest path optimization criteria. Then we pick up the next vertex with highest degree, if it is not placed, we place it at the next available grid points, and place all its adjacent vertices and incident edges similarly. We continue such process until all the vertices were placed.

3.1.2 Routing Edge and Finding Shortest Path

An edge *e(v, u)* is routed depends on the placed grid points of *v* and *u* and the free ports along the way that potentially connect these two vertices. We will discuss the algorithms used to find a shortest path between two vertices.

3.1.2.1 Dijkstra's Shortest Path Algorithm

Dijkstra's Algorithm, introduced in 1959, provides one of the most efficient algorithms for solving the shortest-path problem. In a weighted graph, it is frequently desired to find the shortest path between two vertices. The weights attached to the edges can be used to represent quantities such as distances, costs or times. In general, the distance along a path is the sum of the weights of that path. The minimum distance from vertex *u* to *v* is the minimum of the distance of any path from vertex *u* to *v*.

The basic idea of Dijkstra's algorithm is, let we define a triple of <V, P, L> as <Vertex, Path, Length>:

1. initialize all vertices with <V, null, INFINITY>
2. push start vertex triple S into to priority queue PQ

3. while (PQ.size() < total vertices) do
    - pop the vertex V from PQ
    - get vertex V's neighbors
    - foreach vertex U that has edge to V
        - if weight(V, U) + length (V) < length (U)
            1. set U with triple <U, V+U, weight(V, U) + length (V)>
            2. if no such U exist in PQ, add triple U into it, else update it in PQ if the length is less

This algorithm will calculate the shortest path starting from vertex S to all vertices.

3.1.2.2 A Shortest Path Algorithm*

A* is probably a better choice for path finding since it can be significantly faster than Dijkstra's algorithm. It was developed in 1968 to combine heuristic methods (which use information about the problem to be solved to make decisions) and formal methods (which don't use problem-specific information, but can be formally analyzed). Unlike most graph searching algorithms, A* utilizes a heuristic function that estimates how close its current position is to the goal. By using the heuristic, it can guide its search to look in the best direction first.

A* is like other graph-searching algorithms in that it can potentially search a huge area of the graph. It is also like step-taking algorithms in that it starts out going straight for the

goal. A* can "backtrack" if going straight for the goal doesn't take you there. It does this as it's going by keeping track of possible path that might lead to a good path.

The heuristic function tells A* an estimate of the cost from the current position to the goal. To find the best path, A* uses both the heuristic and the cost of the best path from the start to position goal. The sum of these two values is called the estimated cost. It's important to choose a good heuristic function. A bad heuristic can really slow down A* or make it produce bad paths.

We can take advantage of the nature of a grid orthogonal graph. We know that, to connect one vertex to the destination, the relative coordinate shows how close they are. As we search along the path on each grid points for potential routing candidates, we can estimate how far the current position to goal is by calculating the distance from current position to the goal. Thus we define our heuristic function as:

heuristic (p) = distance (p, g)

where p stands for the current position and g stands for the goal

Considering that, we do not need to find the shortest path to all vertices in graph, we only need the shortest path to the destination vertex. Therefore, our algorithm is slightly revised. Finally, the algorithm to find the shortest path between two vertices can be depicted as the following:

1. initialize all vertices with <V, null, INFINITY>

2. push start vertex triple S into to priority queue PQ

3. while (PQ is not empty) do

    a. pop the vertex V from PQ

    b. get vertex V's neighbours

    c. foreach vertex U that has edge to V

        i. if U is destination, exit (found shortest path)

        ii. if cost(V, U) + length (V) + heuristic < length (U)

            1. set U with triple <U, V+U, weight(V, U) + length (V)>

            2. add triple U into PQ

As we can see from the revised algorithm, we do not update the triples (delete the old one and add the new one) when there is a better path. Instead, we just add the triple with the new value (smaller length) into the priority queue. This is a trade-off between performance and space.

3.1.3 Advantage, Disadvantage and Enhancement

The advantage of this approach is its runtime efficiency and simplicity in implementation. Unlike other algorithms, this approach only needs one run of computation to allocate coordinates of vertices and edges. It also satisfies the aesthetic criteria in terms of drawing volume and minimized bends. The produced graph maintains quite satisfactory layout with vertices well distributed. It produces a perfectly optimized K6 graph with at most 2 bends of a minimum drawing volume.

Due to the time constraints and the nature of this report, however, the proposed approach suffers some limitation on the degrees of a vertex. In the case of a K7 graph, where every degree is used for an incident edge, there may have some edges that cannot be successfully routed due to the intervention of vertices and edges that may block inner free ports to connect to an outer vertex. This behavior can be well understood since the algorithm is doing only one run of best-effort optimization.

Enhancement can be added upon the current implementation. One heuristic approach is to, instead of arbitrary getting the next available grid point to place a new vertex, we might place the vertex in random positions and then try to move them: a move is considered "good" (and performed) if it reduces the number of edge crossings (and of course, eliminating the failure to route an edge), otherwise it is considered "bad" [9]. However, this approach will involve substantial computation for backtracking and finding the best optimization. Moreover, due to the heuristic nature of this approach, an optimal layout with no failures on routing still cannot be guaranteed in clustered edges.

Another enhancement may adopt the *Dynamic Staircase* and *Dynamic Spiral* model as we discussed early to reserve some free ports when placing the vertices [18]. The vertices are embedded in an orthogonal staircase and spiral manner. The edge routings are analyzed and described in a list of rules to formulate how the edge could be routed. This

approach requires detailed analysis of the routing cases and it is not a simple task in implementation.

Interested readers are also referred to literatures in the Appendices for other drawing techniques, such as Battista's split&push approach [19] and the incremental approach proposed by Papakostas [14].

4 Design

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder [23]. One of the main design goals of this project is to achieve reusability, which allows the system serve as a framework for future enhancements of particular needs in 3D graph drawings.

4.1 System Architecture

With such goals in mind, the model-view-controller (MVC) [23] architecture is a perfect candidate for this project's architecture. The MVC can help decoupling components and allows that changes to one can affect any number of others without requiring the changed object to know the details of others. In our design, we adopt the MVC architecture to minimize the coupling between drawing a graph and modeling a graph representation. The view object is the component responsible for displaying a graph, the model object is the component for modeling a graph layout, and the controller is responsible for taking user interactions for navigating a graph, which has very limited use in current implementation.

4.2 Components

The system is conceived according to an object-oriented methodology. It consists of three main components that interchange information and services to each other, as shown in the following figure. These components are responsible for parsing input, modeling graphic

representations and rendering the graph. The parser component takes a pre-defined format of network description file and parses it into a data structure. The modeler component applies certain layout algorithm upon a network and assigns coordinates for vertices and edges. The viewer component takes graphic layout representation data and renders a graph on screen by using a graphic package. We will discuss the design decisions on how the components could share and exchange data later.

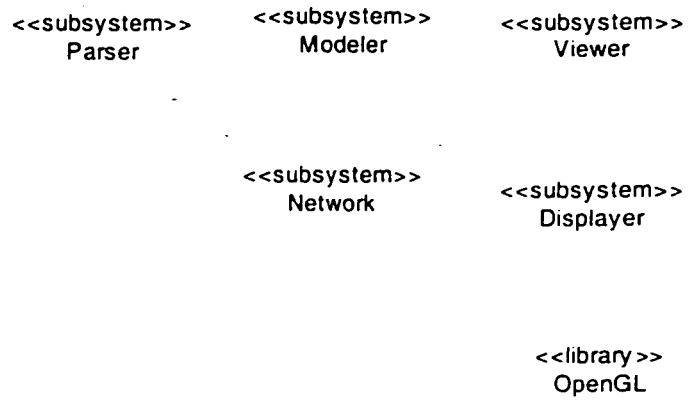

Figure 4: System architecture and components

4.3 Design Rationale

4.3.1 Hierarchical Data Structure for Network Representation

It turned out to be a tough decision on how should the three components share data. Let's discuss the possible approaches and choose the one best suits our needs.

The first one is to share one data structure across the three components. This data structure will contain information about a network (e.g., node ID, connections) as well as

the vertex and edge coordinates. The parser will firstly read in the descriptions from a file and load the information into the data structure. Then the data structure is passed to modeler to set coordinates and then the viewer uses those coordinates to render a graph.

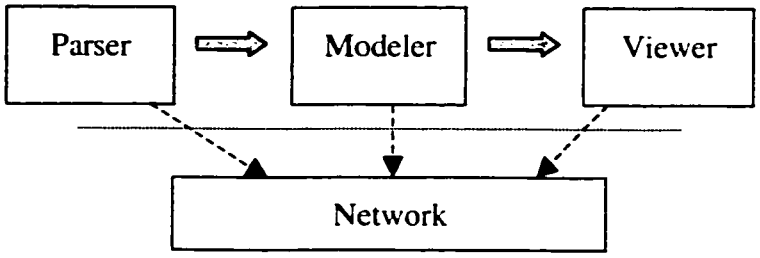

Figure 5: Case 1 of data structure for network representation

A second approach would use two different data structures. The first structure is used to load the information by parser, and the second data structure is used for modeler to set the coordinates and then the viewer uses the coordinates of graph (vertices and edges) for displaying. This approach obviously follows the visualization pipeline processes.

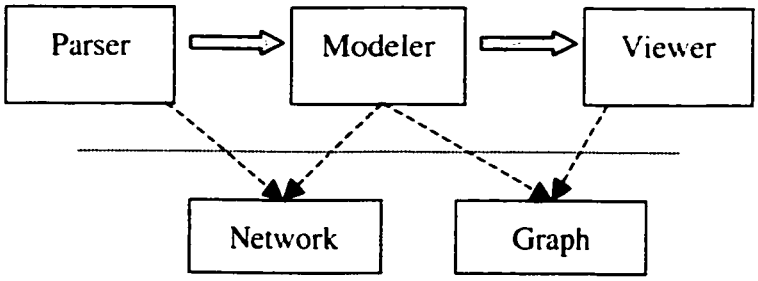

Figure 6: Case 2 of data structure for network representation

Both of the above approaches are not satisfactory. The first approach has a tight coupling between three components. Since the parser only uses the information regarding to network description, the rendering component only concerns vertex and edge coordinates

(possibly also a tag for the network connection and nodes), the coupling resulting from the shared data structure makes it hard to develop the components in parallel. And also, the change of the data structure at any stage will affect the other components. The second approach does not retain network information (e.g., network node and connection tag) that we may need to display them when rendering a graph.

We introduce a hierarchy of information that could be used by parser, modeler and viewer respectively. We consider a network consists of nodes and connections, which is the description of a network and are mainly used by the parser to retrieve data. Node and connection can be abstracted as vertex and edge in a graph term; those two types of information are used by modeler to assign layout coordinates. Vertex and edge are also used by viewer to render a graph. We come up with a dada structure as depicted in the following:

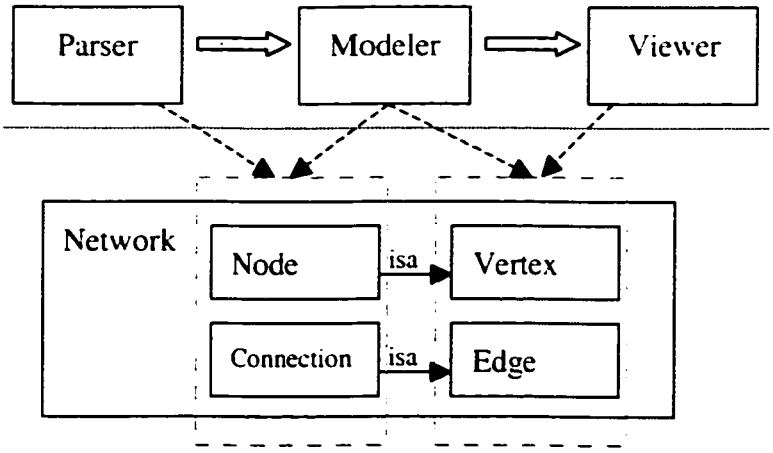

Figure 6: Case 3 of data structure for network representation

The above strategy has the obvious advantage of enabling parallel development of parser and viewer, and the change of one structure does not affect the other components that are not using it. It also conceptually separates the processing phases into data acquisition, data processing, and rendering in a pipelined fashion.

4.3.2 Abstracted Viewer Interface

It is desirable to separate the interface of displaying a graph from the details of how a graph is displayed. This abstraction allows us to easily extend the system to render a graph in different forms, or using different rendering packages. For example, we may display the layout representation in a textual histogram, render a 3D graph in 2D spaces, or switch the rendering package from OpenGL to QuickDraw etc. Thus, we introduce a base class for defining the interfaces as Viewer, and a concrete class GraphicViewer for viewing the graph.

4.3.3 Wrapping OpenGL API

The rendering functionalities of a graph are built upon lower-level 3D rendering library of OpenGL. The OpenGL APIs are basically C functions, and its use has to follow certain procedures. Thus, developers are forced to understand those details and are exposed to another different technical domain. We would like to hide the OpenGL details from developers and encapsulate its APIs in C++ classes. The Displayer class is introduced for that purpose.

4.4 Classes and Object Model

4.4.1 Parser

There is only one class in the parser component, which is the Parser class. This class is responsible for parsing a network description file and loading it into the system. The network description file serves as a communicating portal to other systems, it has well defined syntax of what a network description file format is.

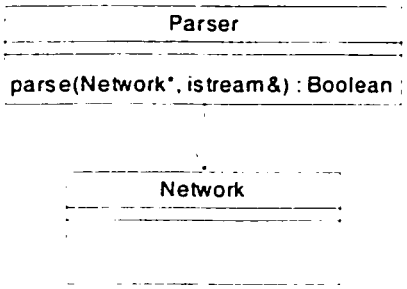

Figure 7: Object model of parser component

4.4.2 Network

The network component consists of class Network, class Node, class Connection, as well as class Vertex and class Edge. Class Network is the core of the system, which serves as a composite of retaining network information as well as chaining the rendering operation of its components. Class Node and Connection match to the real world entities of a network node and connection, as such, they maintain information associating to a network's hosts and wires. Class Vertex and Edge are the abstraction of a network node and connection in a graph term that maintain the coordinates of vertices and edges.

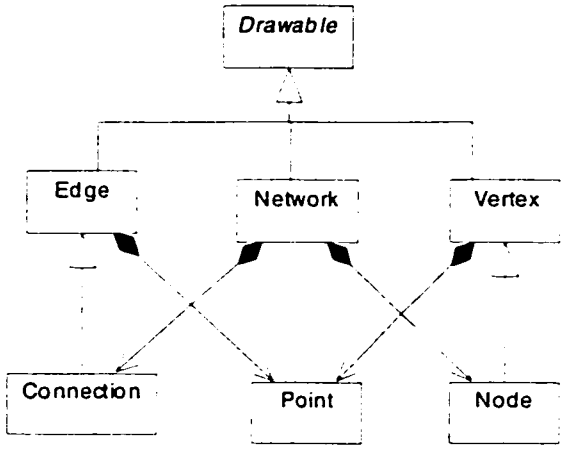

Figure 8: Object model of network component

The hierarchical design of Node versus Vertex, Connection versus Edge demonstrates how we model the network and represent their characteristics in object-oriented methodology. Conceptually, a Node is a Vertex in graph abstraction; similarly a Connection is an Edge. The hierarchical design minimizes the dependencies between different components and allows us to develop then independently.

Both Network and Edge and Vertex share the characteristics of a drawable object. That means, by inheriting the interface of base class Drawable, we can draw a Network by chaining the operation to draw its vertices and edges. This polymorphism makes the drawing generic, regardless what the real objects are drawn.

4.4.3 Modeler

The modeler component is essentially a layout designer, which is responsible for assigning coordinates for vertices and edges according to a specific drawing algorithm. It takes the graph relationships of vertices and edges as input and assigns the coordinates in either 2D or 3D spaces based on the chosen drawing method and algorithm. There are two classes in this component, class Modeler and class Mesh. Modeler applies certain graphic layout algorithm on a predefined 3D space, which is a 3D mesh. Mesh is a utility used by Modeler to manage 3D grid coordinates.

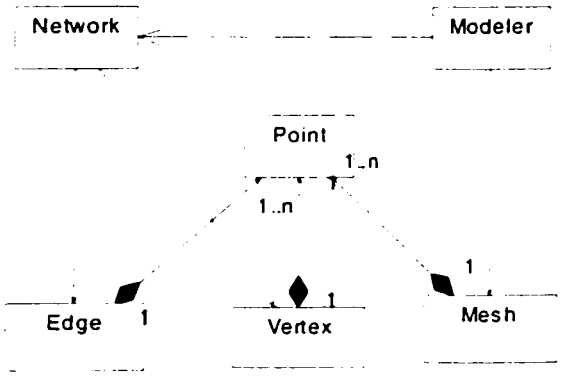

Figure 9: Object model of modeler component

4.4.4 Viewer

The viewer component is responsible for displaying a produced graph visually on a computer screen. This component consists three classes, a base class Viewer; GraphicViewer, which is a specification of Viewer; and a Displayer. Class Displayer wraps OpenGL APIs and hides the rendering library from clients. It provides uniformed methods for displaying a graph. Base class Viewer provides interfaces for drawing (or displaying) the layout representation, whereas GraphicDisplayer is a subclass of Viewer

that implements methods of drawing 3D geometric objects. The abstract interface defined by Viewer provides flexibility to represent a graph layout in other forms, for example, in 2D or textually.

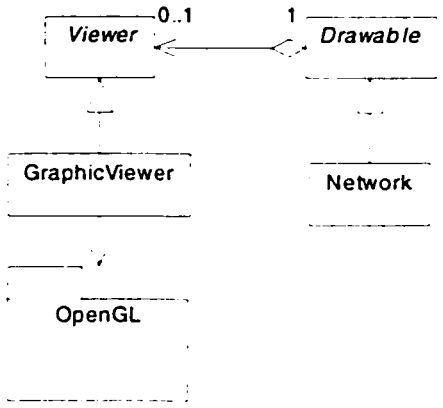

Figure 10: Object model of modeler viewer

4.5 Behavior Model

There are no complicated behaviors of the system. The sequence of call is simply as depicted in the following:

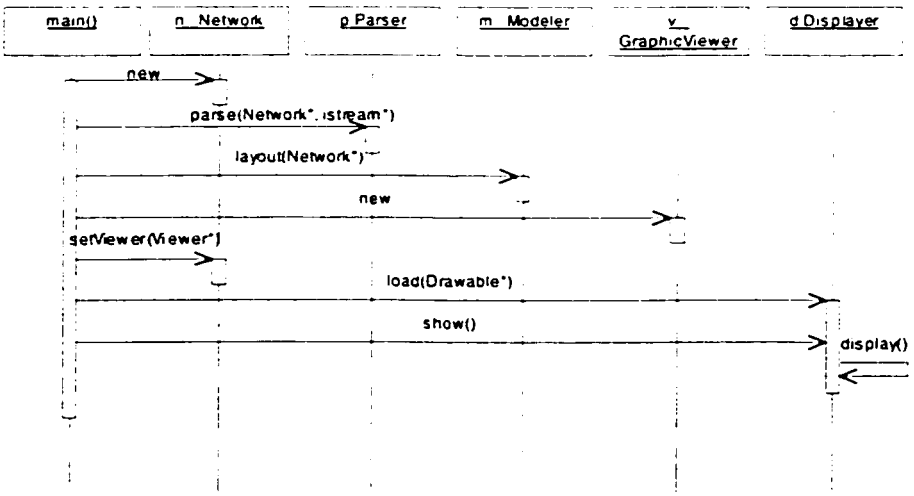

Figure 10:Sequence diagram

From the above sequence diagram, we can see the sequences of object interactions. The main function will create a Network object; this object is passed to the Parser object for parsing (call parse()). After the parsing is done, the Modeler object is initialized and layout message is passed to it with a Network object. This call will setup all the coordinates of the graphic layout. A GraphicViewer is then created and setViewer is passed to the Network to point to the Viewer object for graphic display. The Network object is then loaded to the Displayer and the show() message is passed to it. After that, the Displayer is in the main loop of display().

5 Implementation Details and Problems Solved

5.1 Parsing: Grammar and Pesudo-code

Parser provides solely one method, which takes an input file and loads the information into a network. The parser creates a Node for each entry of the input file and creates a Connection for each item in the in- and out- connection list. We delegate checking of duplicated nodes and connections to the Network since the parser does not retain information of a network as a whole, whereas a Network does.

The parsing follows the following grammar:

network:	entry*, eof
entry:	node, (connection)*, endl
node:	id
connection:	conn_direct, id, eq_sign, node
conn_direct:	'I'|'O'
id	number
eq_sign:	'='

The pseudo-code for reading in the network information is:

```
For each line of entry,
Read in the Node ID,
Create Node,
        While (more connections)
                Read in the connection properties
                Create new Connection
                If (is in-directionCconnection)
                        Add indirection connection to Node
                Else if (is out-directionCconnection)
                        Add out direction connection Node
                Add connection into Network (network check dups)
                If (add Connection return false)
                        Delete Connection  // duplicate
Add Node into network (network check dups)
If (add Node returns false)
        Delete Node // duplicate
```

5.2 Node and Connection Relationship

There are many ways to model the relationship of Node and Connection [24]. We can model it as many-to-many association between nodes and connections are treated as a linked attributes. We can also do it as many-to-many association between connections and nodes are treated as linked attributes. Another way is to treat both nodes and connections as objects and have associations to nodes for each end of the connection.

We need a binary traversal of association for efficiency when modeling vertices and edges. That is, we should be able to easily get the two end nodes from a connection, and also we can easily query all the in- and out connections of a node. However, this will introduce a cyclic dependency between Node and Connection if we implement the association by embedding a list of pointer to Connections (in-connection list and out-connection list) in class Node and embedding a pointer to Node as two ends (from-end and to-end) in class Connection. The problem can be solved by introducing a primitive type, e.g. NodeId as int, shared by both of Node and Connection. In this way, class Node contains a list of NodeId for its in- and out- connections and Connection will have two NodeIds as for the two ends of nodes.

5.3 Using STL priority_queue

In our implementation of finding the shortest path, according to the algorithm, we need to store the triple <Vertex, Path, Length> in a priority queue. The priority is based on the length, i.e., the cost from the starting vertex to where the current position is. The triple having the shortest length will have the highest priority and it will be guaranteed as the top element.

Standard C++ library (STL) provides a priority_queue [25][26], which is an adaptor that provides a restricted subset of container functionality: it provides insertion of elements, and inspection and removal of the top element. It is guaranteed that the top element is the

largest element in the priority_queueue, where the function object `Compare` is used for comparisons. Priority_queue does not allow iteration through its elements.

The restriction on iterating through its element can be overcome by slightly revising the algorithm. Instead of maintaining the processed queue (CLOSED) and the pending queue (OPEN), we could just jump out the processing when our goal is met.

We need to define a function object, or a functor, to be passed to the priority_queue for comparison. Bear in mind that, the standard behavior of priority queue is the largest value being sorted as top element, we need to redefine this behavior as the smallest value of length to be sorted as the top element. The defined comparison functor looks like the following:

```
struct Compare  {
      bool operator()(const Vertex& x, const Vertex& y) const {
            // we need the opposite to get the least length
            return x->length() > y->length();
      }
};
```

5.4 Drawing Text in OpenGL

It is desirable to draw a text string to denote the network node names. There are three approaches to rendering fonts in OpenGL: raster fonts, geometric fonts, and texture mapped fonts. Each method has its own advantages and disadvantages [27].

Raster fonts: Use glBitmap or glDrawPixels to draw a rectangular bunch of pixels onto the screen. The disadvantage of using bitmap fonts is that they are not possible to rotate and scale. However, there is one significant advantage to raster fonts on software-only OpenGL implementations, they are likely to be faster than the other approaches.

Geometric Fonts: Draw the characters of the font using geometric primitives - lines, triangles, whatever. The disadvantages of this are, it is bad for performance, and it is difficult in designing fonts. The advantage is that geometric fonts can be scaled, rotated, twisted etc as normal geometric objects.

Texture-Mapped Fonts: Typically, the entire font is stored in one or two large texture maps and each letter is drawn as a single quadrilateral. The advantage of this approach is its generality. Texture fonts can be rotated and scaled. It's easy to convert other kinds of fonts into texture maps. It is probably an order of magnitude faster than either raster or geometric fonts.

Many resources can be found on how to draw a text in OpenGL [27], and thanks to their hard work and sharing the knowledge and sources. It would be a good idea to use existing library instead of reinventing the wheel. In this project, we adopt Brad Fish's glFont [28]. For detailed information on how it works and how to use it, please refer to the source files glfint2.h and glfont2.cpp.

6 Experimental Results

One important feature of the drawing strategy described in this report is its simplicity. We have implemented the system in C++ using GNU gcc 2.95 in a Cygwin (version 2.249.2.2) environment on Windows. It uses C++ STL containers and algorithms extensively. The rendering package uses OpenGL libraries bundled within the above-mentioned Cygwin.

The implemented program can read network descriptions either from a file or from standard input. The format of the input file is attached in the Appendices. The program will display the generated 3D layout on screen, and it also produces a log file recording the generated coordinates of vertices and edges. The format of the log file is also attached in the Appendices.

User can explore the graphic network displayed on screen from different angels and distances. By pressing the left mouse key or right mouse key, the displaying will be in animation mode to rotate the graph in left-hand side or right-hand side direction. The left and right arrow keys adjust displaying angels similarly. Pressing the up arrow key will zoom out the graph and let user look at it from a far distance for an overall view. The down arrow key will drag the graph close to the user and let the user have clear insights.

Our trial tests show that the implemented program runs efficiently. Although, due to time constraints, we have not conducted comparisons to other implementations or algorithms,

we can say that, due to the algorithm's simplicity and efficiency (only one run of processing is needed), it should give quite satisfactory performance.

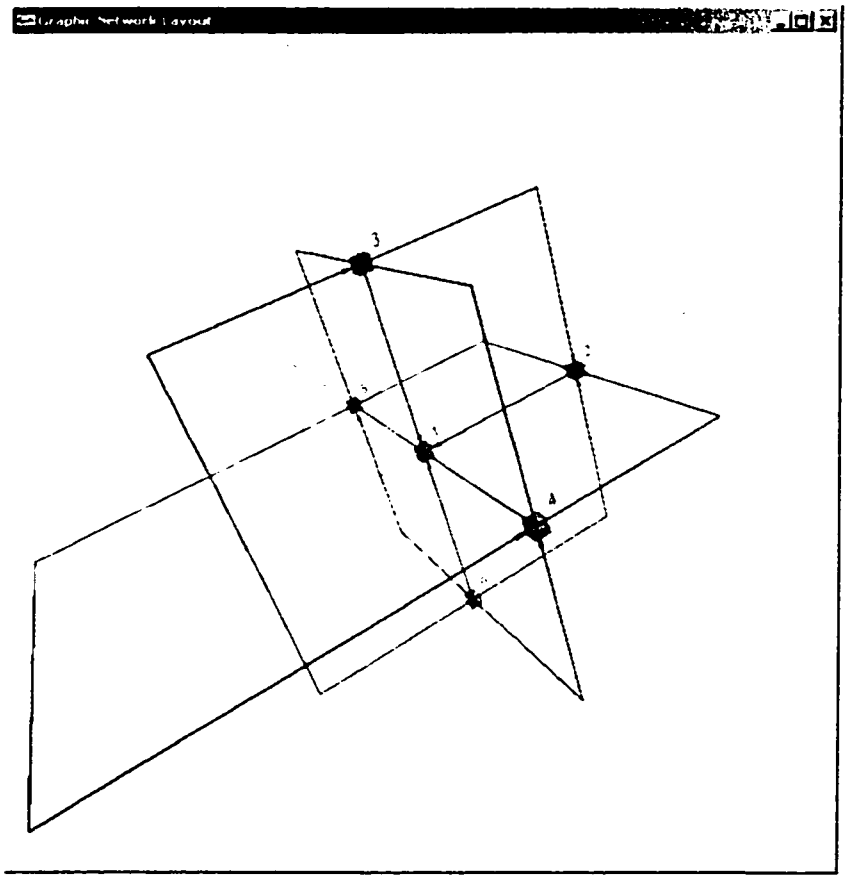

Figure 11: Generated K6 graph

We consider the number of bends, the average edge length and the volume of the graph as the readability of the graph. Extensive tests have not been available due to time constraints. From the layout generated for a K6 graph, we obtained as good measurements as other algorithms (Reduce-Forks) [19] in terms of the number of bends (achieved a maximum of 2 bends) and drawing volume.

No algorithm can be considered "the best". The tradeoff between efficiency and readability in this project tends to favor efficiency. As mentioned in early sections, the algorithm may fail to route edges with no free ports (that may be blocked by others) to connect to. This is understandable for a heuristic approach and its best-effort nature of the algorithm. The generated K7 graph layout shows the defect of this algorithm with failed edges been indicated as "failed".

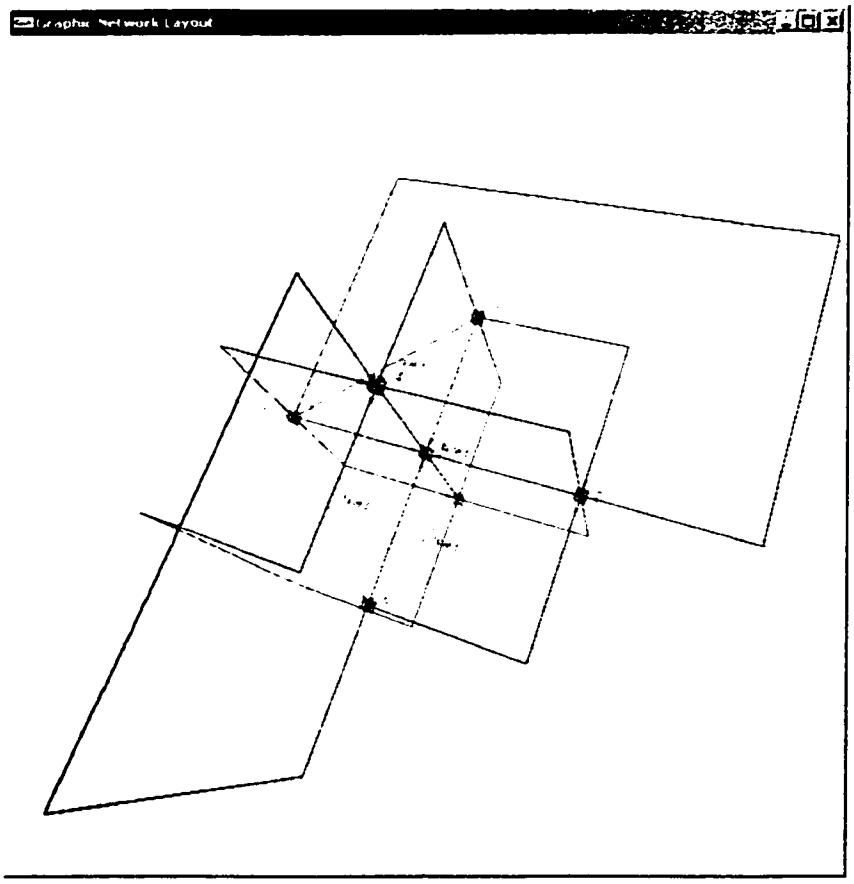

Figure 12: Generated K7 graph

7 Conclusions and Future Work

In this report, we presented the design and implementation of an objected-oriented network 3D graphic layout system built upon OpenGL. We also introduced a simple and efficient layout strategy for 3D orthogonal grid drawing of graphs of maximum degree of 6. Our trial tests show satisfactory efficiency and layout aesthetics.

The implementation provides an easy-to-adapt graphic drawing framework for visualizing a network, as well as for experimenting different layout algorithms in a pipelined fashion. The generated layout coordinates may be used by other rendering entities. Enhancement to the framework can be made to introduce components for parsing the generated coordinates and feed it directly into other rendering engines. Beautifying the drawing geometric objects by using OpenGL features and enhancing user interactions can be beneficial.

Improving the heuristic in placing vertices of the proposed strategy and providing backtracking when a bad placing decision is made is a desirable enhancement. And also, experimenting a strategy to reserve some free ports to ultimately eliminate failed routings is also valuable. More tests and comparisons to other algorithms in terms of efficiency and readability are an area for more work. Determining the trade-offs in applying heuristics is also an interesting problem.

References

[1] Jacques Bertin, Graphics and Graphic Information Processing, Walter de Gruter &Co., Berlin, 1981

[2] Stephen G. Eick, "Aspects of Network Visulization", Computer Graphics and Applications, Vol. 16, No. 2, pages. 69-72, March 1996.

[3] R. F. Cohen, G. Di Battista, R. Tamassia, and I. G. Tollis, "A Framework for Dynamic Graph Drawing", Technical Report No. CS-92-34, August, 1992

[4] Isabel F. Cruz, Roberto Tamassia, Graph Drawing Tutorial

[5] G. Di Battista, P. Eades, R. Tamassia and I.G. Tollis, "Algorithms for Drawing Graphs: An Annotated Bibliography", Jun, 1994

[6] Lukas Mroz and Helwig Loffelmann and Eduard Groller, Selected Trends in Scientific Visualization, Institute of Computer Graphics, Vienna University of Technology, Austria

[7] Roberto Tamassia, Advances in the Theory and Practice of Graph Drawing, Department of Computer Science, Brown University, 1996

[8] Roberto Tamassia, GRAPH DRAWING, 1997

[9] Peter Grogono, Layout Designer Requirements, Concordia University, Sept. 2001

[10] Eric W. Weisstein, NP-Hard Problems - A Wolfram Web Resource, http://mathworld.wolfram.com/NP-HardProblem.html

[11] T. C. Biedl. Heuristics for 3d-orthogonal graph drawings. In Proc. 4th Twente Workshop on Graphs and Combinatorial Optimization, pages 41-44, 1995

[12] P. Eades, C. Stirk, and S. Whitesides. The Techniques of Komolgorov and Bardzin for Three Dimentional Orthogonal Graph Drawings. Inform. Process. Lett., 1996

[13] P. Eades, A. Symvonis, and S.Whitesides. Two Algorithms for Three Dimensional Orthogonal Graph Drawing. In S. North, editor, Graph Drawing (Proc.GD'96), volume 1190 of lecture notes, pages 139-154, Springer-Verlag, 1997

[14] Achilleas Papakostas and Ioannis G. Tollis, Algorithms for Incremental Orthogonal Graph Drawing in Three Dimensions, Journal of Graph Algorithms and Applications, vol. 3, no. 4, pp.81-115, 1999

[15] David R. Wood, An Algorithm for Three-Dimensional Orthogonal Graph Drawing. In S. H. Whitesides, editor, Graph Drawing (Proc GD '98), volume 1547, 1998

[16] David R. Wood, Minimizing the Number of Bends and Volume in Three-Dimensional Orthogonal Graph Drawings with a Diagonal Vertex layout, Technical Report CS-AAG-2001-03, The University of Sydney, 2001

[17] David R. Wood, The DLM Algorithm for Three-Dimensional Graph Drawing in the General Position Model, Technical Report CS-AAG-2001-04, The University of Sydney, 2001

[18] M. Closson, S. Gartshore, J. Jonansen, S. K. Wismath, Fully Dynamic 3-Dimensional Orthogonal Graph Drawing, Journal of Graph Algorithms and Applications, vol. 5, no. 2, pp 1-34, 2000

[19] Giuseppe Di Battista, Murizio Patrignani, A Split&Puch Approach to 3D Orthogonal Drawing, Journal of Graph Algorithms and Appilications, vol.4, no. 3, pp. 105-122, 2000

[20] OpenGL Overview, http://www.opengl.org/developers/about/overview.html

[21] Mason Woo, Jackie Neider, Tom Davis, OpenGL Programming Guide, 2nd Edition, ISBN 0-201-46138-2, 1996

[22] Peter Grogono, Getting Started with OpenGL, Course Notes for COMP 471 and COMP 676, Department of Computer Science, Concordia University, 1998

[23] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns, Addison Wesley, ISBN 0-201-63361-2, 1994.

[24] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, Object-Oriented Modeling and Design, Prentice Hall, ISBN 0-13-629841-9, 1991

[25] Bjarne Stroustrup, The C++ Programming Language, Special Edition, Addison Wesley, 2000, ISBN 0-201-70073-5.

[26] Standard Template Library Programmer's Guide, http://www.sgi.com/tech/stl/

[27] Gerard Lanois, Survey of OpenGL Font Technology, http://www.opengl.org/developers/code/features/fontsurvey/. 2001

[28] Brad Fish's glFont Version 2.0, http://students.cs.byu.edu/~bfish/glfont.php

Appendices

A Example of inputs of network description (K6)

```
1 I1=2 I2=3 I3=4 I4=5 I5=6
2 O1=1 I2=3 I3=4 I4=5 I5=6
3 O1=1 O2=2 I3=4 I4=5 I5=6
4 O1=1 O2=2 O3=3 I4=5 I5=6
5 O1=1 O2=2 O3=3 O4=4 I5=6
6 O1=1 O2=2 O3=3 O4=4 O5=5
```

B Example of generated layout coordinates (K6)

```
1 P(0,0,0)
2 P(64,0,0)
3 P(0,64,0)
4 P(0,0,64)
5 P(0,0,-64)
6 P(0,-64,0)
2->1 P(64,0,0)->P(0,0,0)
3->1 P(0,64,0)->P(0,0,0)
4->1 P(0,0,64)->P(0,0,0)
5->1 P(0,0,-64)->P(0,0,0)
6->1 P(0,-64,0)->P(0,0,0)
3->2 P(0,64,0)->P(64,64,0)->P(64,0,0)
4->2 P(0,0,64)->P(64,0,64)->P(64,0,0)
5->2 P(0,0,-64)->P(64,0,-64)->P(64,0,0)
6->2 P(0,-64,0)->P(64,-64,0)->P(64,0,0)
4->3 P(0,0,64)->P(0,64,64)->P(0,64,0)
5->3 P(0,0,-64)->P(0,64,-64)->P(0,64,0)
6->3 P(0,-64,0)->P(-64,-64,0)->P(-64,0,0)->P(-64,64,0)->P(0,64,0)
5->4 P(0,0,-64)->P(-64,0,-64)->P(-128,0,-64)->P(-128,0,0)->P(-
128,0,64)->P(-64,0,64)->P(0,0,64)
6->4 P(0,-64,0)->P(0,-64,64)->P(0,0,64)
6->5 P(0,-64,0)->P(0,-64,-64)->P(0,0,-64)
```