uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Jianong Zhou

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.Sc. (Systems Science)

GRADE / DEGREE

Systems Science

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

3D Visualization of Multi-layered Graphs with Application to Communications

TITRE DE LA THÈSE / TITLE OF THESIS

Professor Dan Ionescu

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

C. D`Amours

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Professor Diana Inkpen

Professor Lucia Moura

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

3D Visualization of Multi-layered Graphs with Application to Communications

Jianong Zhou

A Thesis submitted to the Faculty of Graduate and Postdoctoral Studies in partial fulfillment of the requirement for the degree of Master of Science, System Science

System Science
School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario, Canada

Your file Votre référence
ISBN: 978-0-494-18490-5
Our file Notre référence
ISBN: 978-0-494-18490-5

Contents

List of Figures

Abstract

This thesis introduces two new algorithms for 3D graph drawing and network display. The first algorithm, the *Incremental Projection Algorithm*, is a new universal algorithm for displaying any graph of any vertex degree. The above algorithm can be implemented to display graph in 3D space without edge crossing. The number of edge bends produced by the algorithm does not excess two. The average time complexity of the *Incremental Projection Algorithm* is $\Omega(\sqrt{N}N)$, where N is the number of vertices in a graph. If there is no degree of vertices great than M, the time complexity of the *Incremental Projection Algorithm* is $O(N)$. The second algorithm is called *Depth-Height Buffer Algorithm*. The algorithm is designed for displaying special multi-layered networks. Actually, the algorithm is a method for hidden object elimination. It is useful for multi-layered communication networks visualization. The time complexity of the *Depth-Height Buffer Algorithm* is $O(N \log N)$. Two demonstration packages of the above algorithms are developed in order, to verify their correctness and functionality.

The thesis also discusses the methods and techniques for 2D graph drawing. In bi-layered crossing reduction aspect the thesis presents a new method called the *minimizing angle approach*, which may reduce the crossings among the edges with the time complexity of $O(|S|max(|N|,|S|))$.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Dr. Dan Ionescu for his guidance, advice, and encouragement throughout my research work. His significant thoughts and guidance will benefit me forever.

I would like to thank Dr. Dongli Zhang for his helpful discussions about my thesis.

Special thanks go to my wife Yan and my daughter Liwei for their understanding, encouragement, and support.

Chapter 1

Introduction

The information visualization technology has been an active research area since it emerged more than twenty years ago [1]. This research field includes 2D and 3D graph drawing and information representation with various algorithms and computer technologies. Graph drawing addresses the problem of constructing geometric representation of graphs, communication networks, and related combinatorial structures. It deals with many research aspects such as graph theory, topology, geometry, combinatorial and continuous optimization, algorithms, data structures, software engineering and computer graphics [1, 2, 3]. Though many graph drawing and network display algorithms and software have been developed, there is still a great demand for new algorithms and software for 3D graph drawing and network displaying.

1.1 2D and 3D Information Visualization

In the real world, people usually need to understand the inside information gathered from certain areas in the form of data, literal symbols, etc. It is obvious that the best way to intuitively understand

the information is to display related information in 2D or 3D space. In practice, there are high demands for displaying graph and network information in 2D or 3D space coming from various disciplines. Though there are many 2D graph drawing software in use, the tools to display graph and network information in 3D space are limited.

Take the communication network as an example. There is a demand that networks are to be displayed in 3D space to give a best view of the networks. The modern communication network is usually implemented with IP routers, ATM, SONET/SDH and DWDM switches. The planar network graphs that describe the complex multi-layered network structure have difficulties to express the information implied in the network. Though there are some existing software libraries for displaying network topology in 2D and 3D space, for example ILOG JView and OpenGL graphic libraries, they are not used straightforwardly for displaying multi-layered network systems. There is, therefore, a need to develop new algorithms to support the displaying of the communication network topology in three dimensional space. Some graph 3D drawing software implemented with these new algorithms can be developed to display the multi-layered networks in three dimensional space.

Since in 2D display, a single node stands for a combination of an IP router, an ATM, a SONET/SDH as well as DWDM switch, it is difficult to reveal their relationship and the differences between them. In a multi-layered 3D display, people can easily see the logic connections and physical connections between them and, in this way, can best understand the network structure. This justifies the need for developing graph 3D drawing algorithms for 3D visualization software.

1.2 Graph Drawing and Information Visualization Technologies

Graph drawing and information visualization is an interdisciplinary research field since many graph drawing and visualization applications deal with computer graphics, graph theory, scientific visualization and simulation modeling, information design, human-computer interaction, hypertext research, etc. To meet the needs of practical applications, many graph drawing software packages have been developed, such as ILOG JView, AutoCAD, OpenGL, etc. To best understand a system, some graphic objects and the relationships among them are usually depicted. For example, many types of complex systems can best be visualized abstractly as a set of nodes and interconnecting links, more commonly called a graph or a network. Examples of graphs include telecommunication network displays, workflow diagrams, business organizational charts, genealogical trees, system diagrams, etc. There are many software packages for displaying network topologies graphically. However, the complexity of a communication network with the Internet as a vivid example requires more elaborated tools to display. For instance, the ILOG JView [56] is the well known software that implemented many new approaches to meet these needs. It includes several layout algorithms such as Topological Mesh layout, Spring Embedder Layout, Uniform Length Edges Layout, Circular Layout, Hierarchical Layout, Link Layout, Tree Layout, Bus Layout and Grid Layout. ILOG JView is a convenient tool to display the 2D diagrams. CAIDA graph drawing package is another powerful tool . To intuitively understand a system, 3D feature software is highly demanded. The OpenGL, GLUT and AutoCAD are famous 3D software packages that produce 3D images [4]. However, for a multi-layered communication network system, the software mentioned above can not be used

straightforwardly to create a 3D display. The designer needs to build software capable to display the multi-layered networks with three dimensional feature and user-friendly interface.

On the other hand, few algorithms are known for the 3D drawing of a general graph with arbitrary vertex degree. It is therefore, necessary to develop new algorithms for a 3D network display to meet the demands. For this purpose, some graph related algorithms have to be designed.

1.3 Objectives of the Thesis

The main objective of the thesis is to develop new algorithms for graph and network 3D displaying by applying graph theory, computer graphics as well as software engineering results. The thesis also discusses the crossing reduction algorithms for the bi-layered graph in 2D space. To verify the correctness and the functionalities of the algorithms proposed in the thesis, software packages were designed and implemented.

The development of the packages will follow the pattern of Waterfall Model [5,6]. The lifecycle includes analyze, design, implement, and test. The first package will focus on the testing of a 3D display algorithm for general graph. The goal of the first algorithm is to draw a general graph with arbitrary vertex degree to 3D space without edge crossing. The second package is an application of a particular algorithm and the computer graphics to the multi-layered communication network display. The second package focuses on several aspects. At first, the architecture of the algorithm for multi-layered networks is described. Second, the dada structures to specify the objects are built. Third, the package is implemented by making use of the technique of the Object-Oriented design and programming. At the same times, many rendering methods of the computer graphics are implemented,

which include hidden line and surface elimination, coloring etc. On the whole, the thesis develops new algorithms for graph and network 3D drawing and verifies they are correct and working.

1.4 Thesis Overview

This thesis discusses the algorithms and techniques for 2D and 3D graph drawing, and network 3D visualization. At first, the thesis reviews the 2D and 3D graph drawing methods in use in recent years. In 2D graph drawing aspect, the bi-layered crossing reduction problem is addressed intensively. The thesis presents a new method called *minimizing angle approach*, with which one may reduce the crossings in a layered plane graph. Then the thesis discusses the algorithms and techniques for graph and network 3D display. The first algorithm presented is the *Incremental Projection Algorithm*, which is a general graph 3D drawing algorithm for any graph with arbitrary vertex degree. A package implementing this algorithm is developed. The second algorithm is the *Depth-Height Buffer Algorithm* designed for the multi-layered communication networks visualization. Based on the algorithm and computer graphics, a package for the *Depth-Height Buffer Algorithm* is implemented. The package is then used to display various multi-layered communication networks.

Chapter 1 is a brief introduction that addresses the questions of what are the 2D and 3D graph drawing, information visualization and the objective of the thesis.

Chapter 2 is a review and discussion of the various 2D graph drawing methods and 3D information visualization. Several 2D graph layouts and 3D graph visualization methods are presented.

Chapter 3 discusses in detail the *minimizing angle approach* for crossing reduction in bi-layered graphs. The basic concepts of the crossing reduction with the minimizing angle approach are explained at first. Then a comparison is made among the minimizing angle approach, the barycenter method, and the median method.

In Chapter 4 of thesis the architectures for the Incremental Projection Algorithm and the Depth-Height Buffer Algorithm are designed and the structure of the software packages for the two algorithms is also intruduced.

Chapter 5 discusses the graphic algorithms and computer graphics used in the 3D display algorithms and the packages, such as the projection methods, the geometric transformations, and the hidden objects elimination methods.

Chapter 6 presents results obtained by testing the Incremental Projection Algorithm and the Depth-Height Buffer Algorithm. The stability, complexity, and scalability of two algorithms are addressed as well.

Chapter 7 summarizes the thesis and draws conclusion. The future works are listed.

1.5 Thesis Contributions

This thesis discusses the 2D and 3D graph drawing methods. It focuses mainly on a 3D display algorithm for general graphs. Then a hidden object elimination method in 3D drawing for the special case of multi-layered network is discussed. A crossing reduction method for the hierarchical graph is addressed as well.

First, a new algorithm, the *Incremental Projection Algorithm*, is presented. The *Incremental Projection Algorithm* is a universal method for 3D drawing for a general graph. It can process any graph without the limitation of the number of the vertex degree. At most two bends per edge is produced in 3D drawing with the algorithm. Generally, no crossing is produced in a 3D drawing for a graph if the algorithm is employed. Furthermore, the time complexity of the algorithm is $\Omega(\sqrt{N}N)$, where N is the number of vertices in a graph. If there is no degree of vertices great than M, the time complexity of the *Incremental Projection Algorithm* is $O(N)$. A prototype for the *Incremental Projection Algorithm* is developed to verify that the *Incremental Projection Algorithm* works.

Second, the *Depth-height buffer algorithm* extends the well known *z-buffer* algorithm. It can be used for special case of multi-layered networks to eliminate the hidden facets and hidden lines efficiently. Since the *z-buffer* algorithm focuses just on the pixels on screen, it is very time- and memory-consuming; the *Depth-height buffer algorithm*, on the other hand, considers each geometric object as a unit. By sorting with y-coordinates and z-coordinates respectively for the objects with the same y-coordinate, the hidden facets and hidden line segments are eliminated efficiently. This ensures that the 3D drawing of a multi-layered network is displayed more clearly and naturally. The time complexity of the *Depth-height buffer algorithm* is just the same as the time complexity for sorting. A package for the algorithm is developed as well.

Third, the thesis presents a new method called *minimizing angle approach* for crossing reduction in layered 2D graph drawing. This new approach focuses on the minimization of a target function which is the sum of the angles between the edges and the vertical direction. Usually, *barycenter heuristics* and *median heuristics* are two methods to reduce the crossing. But they just give the trial

permutation for the nodes to reduce the crossings. Experiments show that the *minimizing angle approach* can reduce the sum of the angles between the edges and the vertical direction in bi-layer graphs. In this way, it may reduce the crossings in a bipartite graph.

Chapter 2

Literature Review

In graph drawing and information visualization research area, the 2D graph drawing theories and their applications have been well developed and formed a mature research discipline. But the 3D graph drawing and 3D information visualization is still a developing research area.

A large body of literatures exists about graph drawing and information visualization. This thesis first focuses on 2D graph drawing theories, then moves to 3D graph drawing and information visualization.

The 2D graph drawing theory includes graphs and their representation, graph planarity and embeddings, graph layout methods, and crossing counting and reduction methods. In addition, the crossing reduction is an important step to draw a graph satisfying certain aesthetic criteria. In 3D cases, the topics include graph 3D embedding algorithms and their complexity, interactive visualization of large graphs and networks as well as applications of information visualization. The following sections will examine some 2D and 3D graph drawing theories and summarize the current development of the theory and technology.

2.1 Graphs and Their Representation

The basic concepts pertaining to graphs are given in [1], as such terminologies developed in [1] will be used frequently in the following contexts.

2.1.1 Undirected Graphs

The notion of a Graph is the basic concept in graph theory and graph drawing. The definition of a graph can be found in any graph theory text book. A *graph* $G = (V, E)$ consists of a finite set V of *vertices* or *nodes* and a finite set E of *edges*, that is an unordered pair (u, v) of vertices. An edge (u, v) with $u = v$ is a *loop*. An edge which occurs more than once in E is a *multiple edge*. A *simple graph* has no loops and no multiple edges. Mostly, simple graphs are dealt with here. For an edge $e = (u, v)$, the vertices u and v are the end-vertices of e, and e is *incident* to u and v. An edge $(u, v) \in E$ *connects* the vertices u and v . Two vertices $u, v \in V$ are *adjacent* if $(u, v) \in E$. A *walk* of *length k* in a graph G is an alternating sequence of vertices and edges $v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k$, beginning and ending with the vertices v_0 and v_k, respectively, and $e_i = (v_{i-1}, v_i)$ for $i=1,2,\ldots,k$. A walk is called a *cycle* if all vertices are distinct except for $v_0 = v_k$ for $k \geq 2$. A graph G is *connected* if every pair of vertices is connected by a path, otherwise it is called *disconnected*. A graph is *bipartite* if and only if it does not contain a cycle of odd length.

2.1.2 Directed Graphs

A directed graph or digraph $G = (V, E)$ consists of a finite set $V = V(G)$ of vertices and a finite edge set $E \subseteq V \times V = \{(u, v) \mid u, v \in V\}$ of (directed) edges or arcs that are ordered pair of vertices. A

digraph $G = (V, E)$ is *simple* if it contains no loops or multi-arcs. A digraph G is *strongly connected* if each pair of vertices $u , v \in V$ is connected by a directed path from u to v. An *acyclic digraph* is a digraph with no directed cycle or loop. A *source* is a vertex with no incoming edges and a *sink* is a vertex with no outgoing edges. A *topological numbering* of G is an assignment of numbers *topnumber*(v) to the vertices v of G such that for every edge (u, v) of G the number assigned to v is greater than the one assigned to u. A topological sorting of G is a topological numbering of G such that every vertex is assigned a distinct integer between 1 and $|V|$, where $|V|$ is the cardinality of V. Obviously, G admits a topological numbering or sorting if and only if G is acyclic.

2.1.3 Representation of Graphs

There are several ways to represent an (undirected or directed) graph. Most people use the classical representation in graph drawing. A graph $G = (V, E)$ is generally visualized by a *drawing* in 2- or 3-dimensonal space with the vertices drawn as points or boxes of a pre-specified width and height, and the edge drawn as closed Jordan curves, connecting their incident vertices. Layout in which the coordinates of the vertex representation are restricted to integer value are called *grid layouts*.

2.1.4 Layered Graphs

Let $(L_1, L_2, \ldots, L_h)$ denote an ordered set of elements, called the *layers* of the graph. A *layered graph* $H = (G, \Lambda)$ consist of a (directed or undirected) graph $G = (V, E)$ and a function $\Lambda: V \rightarrow (L_1, L_2, \ldots, L_h)$ assigning each vertex $v \in V$ to exactly one layer L_i, $i \in \{1,2,\ldots,h\}$.

The directed graphs with layering structure are called *hierarchical graphs*. Hierarchical graphs appear in applications where *hierarchical* structure are involved. (PERT networks ad organization charts). Does every planar hierarchical graph admit a planar strait-line hierarchical drawing? Peter *et al.* [9] addressed the question.

Layered graphs are often represented "top-to-bottom" as follows: For $i=1,2,\ldots,h$, the vertex belonging to layer L_i is drawn on a horizontal line with y-coordinate y_i, satisfying the condition $y_1>y_2>\ldots>y_h$. A popular alternative is a "left-to-right" representation with vertical lines at x_i and $x_1<x_2<\ldots<x_h$.

In graph drawing, layered graphs occur in the context of directed acyclic graphs. For these graphs, a layering is generated based on a topological numbering, i.e. the function Λ satisfies $\Lambda(v)>\Lambda(u)$ for each edge $(u, v) \in E$. Usually, all the edges as curves monotonically decreasing in a vertical direction in a top-to-bottom representation and monotonically increasing in a horizontal direction in a left-to-right representation are drawn.

2.1.5 Clustered Graphs

Clustered graphs are graphs with recursive clustering structure over the vertices. A *clustering graph* $C = (G, T)$ consists of an undirected graph $G = (V, E)$ and a rooted tree T such that the leaves of T are exactly the vertices of G. Each vertex v of T represents a *cluster* $V(v)$ of the vertices of G that are the leaves of the sub-tree rooted at v. The root of T is called the *root cluster*. The tree T is called the *inclusion tree* of C because it describes an inclusion relation between clusters. The graph G is called the *underlying graph* of C. The tree $T(v)$ represents the sub-tree of T rooted at the vertex v,

and $G(v)$ denotes the sub-graph of G included by the cluster associated with vertex v. Define $C(v) = (G(v), T(v))$ to be the *sub-clustered graph* associated with vertex v. In a drawing of a *clustered graph* $C = (G, T)$, the graph G is drawn with points and curves as usual. For each vertex v of T, the cluster is drawn as a simple closed region R that contains the drawing of $V(G(v))$, such that the following three conditions hold:

(i) The regions for all sub-clusters of v are completely contained in the interior of R.

(ii) The regions for all other clusters are completely contained in the exterior of R.

(iii) If there is an edge e between two vertices of $V(v)$ then the drawing of e is completely contain in R.

2.1.6 Compound Graphs

Compound graphs have been introduced for representing graphs with both inclusion and adjacency relationships [28]. A Compound graph $C = (G, T)$ is defined as a graph $G = (V, E_G)$ and a rooted tree $T = (V, E_T)$ that share the same vertex set V. There is a one to one correspondence between the structure of the tree and the set of inclusions between the vertices, namely, a vertex u is in direct inclusion relation to v if and only if u is a child of v in the tree. If the end-vertices u and v of all edges $\{u, v\} \in E_G$ belong to different root-leaf paths in T, C is called a *simple compound graph*. In a simple compound graph, a pair of vertices (u, v) cannot be in a relation of adjacency and inclusion at the same time.

Edges connecting vertices of different tree levels are called *inter-level edges*. If a compound graph does not contain inter-level edges, it is called a *nested graph*. A further restriction allowing

only edges between the leaves of the tree leads to an alternative definition of clustered graphs. In a *drawing of a compound graph* $C = (G, T)$, the vertices of the graph G are drawn as closed regions so that a vertex u is included in the region representing the vertex parent(u) in T, and the edges in E_G are drawn as curves connecting the regions associated with its end-vertices.

2.1.7 Storage of Graphs and Digraphs

Common data structure for storing graphs and digraphs $G = (V,E)$ are adjacency matrices and adjacency lists. An *adjacency matrix* for an undirected graph is a $|V|$ by $|V|$ matrix $A(G) = (a_{uv})$, $u,v \in V$, where a_{uv} is the number of edges connecting the vertices u and v, i.e., $a_{uv} \in \{0, 1\}$ for simple graphs. For a digraph, a_{uv} is the number of edges leaving u and entering v, again, $a_{uv} \in \{0, 1\}$ for simple digraphs. Adjacency matrices are, due to their storage requirement of $|V|^2$, independently of $|E|$, unattractive for *sparse graphs* (i.e. graphs in which E contains only a small subset of all possible edges). In automatic graph drawing, sparse graphs are usually considered. In fact, $|E| \leq c|V|$ for some constant c. Therefore, a different data structure is usually more appropriate. Most common structures are *star-* and/or *adjacency-lists* that give (in-/out-)star(v) and/or (in-/out-)adj(v) for each $v \in V$ as linear, linked, or doubly linked lists, depending on the application. For digraphs, the in- and out-lists may be merged and equipped with an in-/out-flag. In addition to the adjacency lists, a list of the edges with their end-vertices is useful in most cases.

2.2 Graph Planarity and Embedding

Planar graph is an important class of graphs. Usually, the question of whether or not a given graph is a planar graph needs to be decided. Furthermore, given a graph, is there a planar embedding drawing and how can find a planar embedding drawing? A drawing of a graph G on the plane yields an *embedding* Π of G (i.e. a clockwise ordering of the incident edges for every vertex with respect to the drawing).

2.2.1 Planar Graphs

The article [1] gives the concepts of planar graphs and related entities in graph drawing manner. A graph $G = (V,E)$ is called *planar* if it can be drawn in the plane such that no two edges cross each other except at common endpoints. An intersection of two edges in a drawing other than at their endpoints is called a *crossing*. A *planar* or *combinatorial embedding* Π of a planar graph G is an embedding with respect to a planar drawing. A graph with a given fixed planar embedding Π is also called a *planar graph*. Given a drawing of a planar graph G, a *face* of G is a topologically connected region in the drawing bounded by the (Jordan curves corresponding to the) edges of G. A face of a planar graph is uniquely described by its boundary edges. The degree $deg(f)$ of a face f is defined as the number of its boundary edges, where each boundary edge with both sides on the boundary of f is counted twice. The faces of a planar graph are already described by the planar embedding. Two faces are *adjacent* if their boundaries share an edge. The one unbounded face of a planar graph is called the *outer face* or *exterior face*. All other faces are called *interior faces*. An equivalent definition of a planar embedding is an ordered list of the boundary edges for each face, clockwise for interior faces

and counter-clockwise for the exterior face.

A famous result of Euler for polytopes relates the number of vertices, edges, and faces in any planar embedding of a connected planar graph:

Euler's Formula : *Let Π be a planar embedding of a connected planar graph $G = (V,E)$ and let F be the set of faces in Π. Then $|V| - |E| + |F| = 2$.*

From Euler's formula, an upper bound on the number of edges of a planar graph with a given number of vertices is easily derived: *For any simple planar graph $G = (V,E)$ with at least 3 vertices we have $|E| \leq 3|V| - 6$.*The bound is attained for *triangulated* planar graphs (i.e. planar graphs in which every face is a triangle). In general, the number of different planar embeddings of a planar graph is exponential in $|V|$.

Given a planar embedding $\Pi(G)$ of a planar graph $G = (V,E)$ with face set F, the *dual graph* $G' = (V', E')$ is constructed as follows: $V' = F$ and E' contains an edge $\{f_i, f_j\}$ for each $e \in E$ such that e is on the boundary of both f_i and f_j. (f_i and f_j may be identical.) By definition, the degree $deg(f)$ of a face f of G agrees with the degree of f as a vertex of G'. The dual graph of a planar graph is in general a non-simple graph (i.e. it contains loops and multi-edges). Loops in G' correspond to bridges in G.

Planarity of a graph $G = (V,E)$ can be tested in $O(|V|)$ time by the algorithm of Hopcroft and Tarjan [21], or an approach of Lempel *et al.* [23] using the special data structure PQ-tree introduced by Booth and Lueker [8]. For a planar graph G, an embedding Π of G can be determined in linear time by the algorithms of Chiba *et al.* [11] or Mehlhorn and Mutzel [24].

2.2.2 Upward Planarity

Let G be an embedded digraph. A vertex v of G is called *bimodal* if the circular list of the edges incident to v can be partitioned into two (possibly empty) linear lists of edges, one consisting of the incoming edges and the other consisting of the outgoing edges. An embedding is called a *bimodal embedding* if every vertex is bimodal. A planar graph is said to be *bimodal* if it admits a planar bimodal embedding. A drawing of G such that all the edges are curves monotonically increasing in a given direction is known as an *upward drawing*. An *upward embedding* U is a representation of G that consists of the clockwise orderings of the incoming edges for every vertex with respect to an upward drawing. Necessary conditions for the existence of an upward planar drawing of an embedded graph G_{Π} are the acyclicity and the bimodality of G_{Π} itself [7]. However, these conditions are not sufficient. A polynomial time algorithm to test the existence of upward planar drawings of a planar embedded digraph is given in [7]. The problem is *NP*-complete in a variable embedding setting [19]. The quasi-upward drawing convention extends the upward drawing convention[33]. A *quasi-upward drawing* of a digraph in the left-to-right direction is such that the vertical line through each vertex v "locally" splits the incoming edges from the outgoing edges of v. The term *locally* is used to identify a sufficiently small connected region properly containing v. A *bend* of a quasi-upward drawing in the left-to-right direction is a point on an edge such that the vertical line through this point is tangential to the edge. Intuitively, a bend is a point in which an edge inverts its left-to-right direction. In [33] it is proven that a quasi-upward planar drawing of a digraph exists if and only if the digraph is planar bimodal, and a polynomial time algorithm for computing quasi-upward planar drawings with the minimum number of bends of an embedded bimodal digraph is described. A directed acyclic graph G

= (*V,E*) is called *upward planar* if it has an upward drawing without edge crossings. An *upward planar embedding* is an upward embedding with respect to an upward planar drawing. Upward planarity testing of directed acyclic graphs is *NP*-complete as has been shown by Garg and Tamassia [19]. Acyclic digraphs with a single source can be tested for upward planarity. Bertolazzi *et al.* [34] proven the result: *There is an O(|V |) time algorithm using SPQR-trees to test whether a single source acyclic digraph G = (V,E) is upward planar, and if so, it outputs an upward planar embedding.*

2.2.3 Cluster Planarity

In a drawing of a clustered graph, an edge *e* and a region *R* have an *edge-region crossing* if the drawing of *e* crosses the boundary of *R* more than once. A drawing of a clustered graph is *c-planar* if there are no edge crossings or edge-region crossings. A graph that admits a *c*-planar drawing is called *c-planar*. Notice that the planarity of the underlying graph does not imply the existence of a *c*-planar drawing of a clustered graph. A *c*-planar drawing of *C* induces a *c*-planar embedding. A *c-planar embedding* of *C* fixes the planar embedding of the underlying graph *G* and contains the circular ordering of all edges crossing the boundary of each non-trivial cluster region.

Unfortunately, no polynomial time algorithm is yet known for *c*-planarity testing. However, *c*-planarity can be tested for a subclass of clustered graphs. A clustered graph *C* = (*G, T*) is called *c-connected* if each cluster induces a connected sub-graph of *G*.

Feng *et al.* [15], Dahlhaus [13] shown that the *c-planarity* of a c-connected clustered graph C = (G, T) can be tested in linear time. The algorithm of [15] is based on the following theorem that gives a necessary and sufficient condition for *c*-planarity of *c*-connected graphs.

Feng *et al.* [15] also shown that a c-connected clustered graph $C = (G, T)$ is *c-planar* if and only if G is *planar,* and there exists a planar drawing of G such that for each vertex v of T, all vertices and edges of $G \backslash G(v)$ are in the outer face of the drawing of $G(v)$.

2.3 Graph Drawing Methods

In 2D graph drawing, there are several standard layouts such as *Tree Layout*, *Layered Layout*, *Orthogonal Layout*, *Force Directed Layout*, etc., which are introduced in order in the following context.

2.3.1 The Definition of Layout

Graph layout is used in graphical user interfaces of applications that need to display graph models. To layout a graph means to draw the graph so that an appropriate, readable representation is produced. Essentially, this involves determining the location of the nodes and the shape of the links. The ILOG JViews [56] implemented several common graph layouts. But what is meant by an "appropriate" drawing of a graph? In practical applications, it is often necessary for the graph drawing to respect certain quality criteria. These criteria may vary depending on the application field or on a given standard of representation. It is often difficult to speak of what a good layout consists. Each end user may have different, subjective criteria for qualifying a layout as "good". However, one common goal exists behind all the criteria and standards: the drawing must be easy to understand and provide easy navigation through the complex structure of the graph. The most common criteria are:

- Minimizing the **number** of link crossings;

- Minimizing the total **area** of the drawing;
- Minimizing the number of **bends** (in orthogonal drawings);
- Maximizing the smallest **angle** formed by consecutive incident links;
- Maximizing the display of **symmetries**.

There are many graph drawing layouts. These include tree layout, layered layout (Hierarchical layout), topological mesh layout, spring layout, uniform length layout, circular layout, link layout, bus layout and grid layout, etc. The tree layout and the layered layout are most common.

2.3.2 Tree Layout

In automatic graph drawing, the notion "Tree Layout" refers to drawing rooted trees. Before a general (undirected) tree can be processed, a root must be chosen and all edges must be directed away from the root. Forests are processed by drawing each connected component separately. Since for a tree $T = (V, E)$, $|E| = |V| - 1$, running times are given as functions of $|V|$. For a typical tree layout, see Figure 2.1.

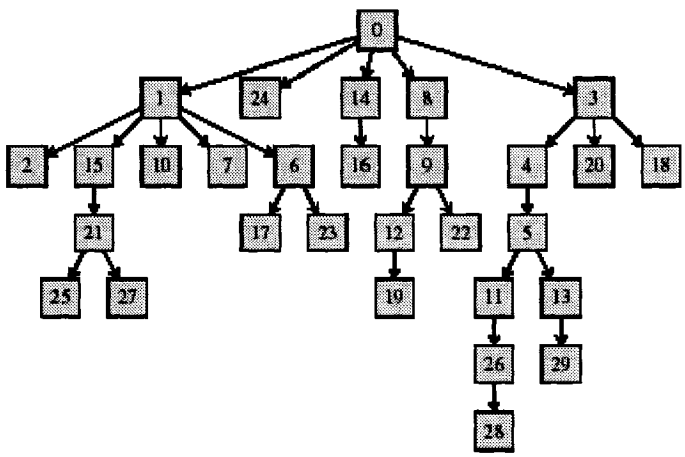


Figure 2.1 A typical tree layout (Jünger, M., Mutzel, P. [1])

Reingold and Tilford [26] gave an $O(|V|)$ algorithm to compute the tree layout for a graph whose grid layout satisfies the following aesthetic criteria:

1. All vertices $v \in V$ of the same depth are drawn on a straight horizontal line whose y-coordinate is $-depth(v)$.
2. A left child is placed to the left (smaller x-coordinate), a right child is placed to the right (bigger x-coordinate) of its parent. If a parent has two children, it is centered above its children.
3. A tree and its mirror image are drawn identically up to reflection.
4. Isomorphic sub-trees are drawn identically up to translation.

While the Reingold-Tilford algorithm is very efficient and delivers aesthetically pleasing drawings, the width of the drawing may be arbitrarily far from the minimum width subject to the five aesthetic criteria, more precisely, there is a family of binary trees $T = (V, E)$ that can be drawn on a grid of width 2, yet the Reingold-Tilford algorithm delivers a drawing of width $(|V|+2)/3$. It has been shown by Supowit and Reingold [29] that achieving minimum grid width is NP-hard, and, even worse, that, unless $P = NP$, there is no polynomial time algorithm for achieving a width that is smaller than 25/24 times the minimum width. On the other hand, if continuous coordinates (rather than integral grid coordinates) are allowed, Supowit and Reingold [29] have shown that minimum width drawings can be found with the help of a linear programming technique in polynomial time.

2.3.3 Layered Layout (Hierarchical Layout)

Figure 2.1 shows a special case of layered layout. In the figure, all vertices of a rooted tree $T = (V,E)$ were drawn on parallel horizontal lines (i.e. assigned to parallel horizontal layers), and all directed tree edges $(u, v) \in E$ were drawn as straight lines between two consecutive layers such that the y-coordinate of u was one grid unit bigger than the y-coordinate of v. It was Sugiyama *et al.* [27] worked out the idea in an automatic graph drawing called *Sugiyama-style layout*. Conceptually, this drawing style easily extends to general acyclic digraphs $G = (V,E)$ by stipulating that, again, all vertices are drawn on parallel horizontal lines such that for each directed edge $(u, v) \in E$ the y-coordinate of u is bigger than the y-coordinate of v.

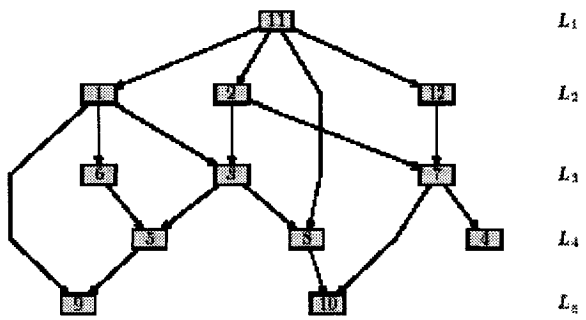


Figure 2.2 A typical Sugiyama-style layout (Jünger, M., Mutzel, P. [1]).

Sugiyama-style layout is popular because any directed or undirected graph can be converted into an acyclic digraph either by reversing directed edges in the case of a digraph or assigning appropriate directions to the edges in the case of an undirected graph. Sugiyama-style layout for an acyclic digraph has three steps which will be discussed below. First, the layered layout for acyclic digraphs is described and then the conversion of any graph into an acyclic digraph is discussed. Sugiyama-style layout for an acyclic digraph $G = (V,E)$ decomposes into three phases:

1) **Layer Assignment:** Compute a layering of the graph via a topological numbering. I.e. assign all vertices to disjoint nonempty subsets $L_1, L_2, \ldots, L_h$ of V called *layers* such that for each edge (u, v) the following holds: If the end-vertex u is assigned to layer L_i and the end-vertex v is assigned to layer L_j then $j > i$. If $j > i+1$ (i.e. the edge traverses intermediate layers), the edge is called a *long* edge and it is replaced by a directed path from u to v with $j- i-1$ artificial intermediate vertices for each traversed layer.

2) **Crossing Minimization:** For each layer L, determine permutations of the vertices in L with the goal of obtaining few crossings under the following assumptions: (1) the layers are parallel horizontal lines; (2) each vertex is drawn on the line corresponding to its layer with x-coordinate compatible with its position in the layer permutation; (3) all edges are drawn as straight line segments between consecutive layers.

3) **Coordinate Assignment:** Turn the topological layout of (S2) into a geometric layout by assigning to each $v \in V$ the y-coordinate $-i$ if $v \in L_i$ and an x-coordinate compatible with the vertex permutation of L_i. Finally, suppress the artificial vertices in the drawing.

Layer Assignment. In phase (i), the nodes should be assigned to proper layers and unnecessary artificial vertices should be avoided, because they induce long edge drawings and have a negative influence on the running times of the later phases. Call h the *height* and call $\max_{1 \le i \le h} |L_i|$ the *width* of the layer assignment. When postulating a compact final drawing as an aesthetic requirement, the tradeoff in keeping both height and width small or reasonably related must be dealt with in order to obtain some desired aspect ratio. The minimization of the number of artificial vertices has been successfully addressed by Gansner *et al.* [18]. If y_v denotes the vertical coordinate of vertex $v \in V$,

then the problem can be formulated as the integer linear programming problem:

$$\text{minimize} \quad \sum_{(u,v)\in E} (y_u - y_v)$$

$$\text{subject to} \quad y_u - y_v \geq 1 \text{ for all } (u, v) \in E$$

$$y_v \geq 0 \text{ and integral for all } v \in V.$$

Integer linear programming can be solved with network flow techniques according to [12]. It is quite simple to compute a layer assignment with minimum height by observing that the length of a longest directed path in G is a lower bound on the height of the layer assignment, and that an easy modification of a topological sorting algorithm, called *longest path layering*, can be used to find a layer assignment with this height. The method uses a first-in-first-out queue Q of vertices initialized with all vertices $v \in V$ with indeg(v) = 0. Starting with $i = 1$, each vertex $v \in Q$ is removed from Q, assigned to L_i, and all edges in outstar(v) are deleted. When a deletion operation leads to indeg(w) = 0 for a vertex $w \in$ outadj(v), then w is inserted as a new vertex at the end of Q. Finally, i is increased by one as soon as all old vertices have been processed and then the new vertices in Q become old vertices. This procedure takes $O(|V| + |E|)$ time and space. Its drawback is that control over the width of the layer assignment is impossible.

Unfortunately, it is *NP*-hard to minimize the height for a given width $w \geq 3$. This can be proven via a simple transformation from a well-known *NP*-hard multiprocessor scheduling problem in which $|V|$ unit-time jobs with $|E|$ precedence constraints between pairs of jobs must be processed on w parallel machines so as to minimize the completion time. This relation suggests the application of a very popular polynomial time approximation algorithm by Coffman and Graham [63] for this

multiprocessor scheduling problem to the layer assignment problem. The article states that if h is the height attained by the algorithm and h_{min} is the minimum possible height given width w, then $h \leq (2 - 2/w)h_{min}$ (i.e. the algorithm computes the optimum solution for $w = 2$ and has a decent performance guarantee for larger w).

Crossing Minimization. As mentioned before, the crossing minimization problem is *NP*-hard even when restricted to 2-layer instances. Nevertheless, Healy and Kuusik [20] proposed a method based on the *branch-and-cut* framework that produces an optimum solution in exponential running time. However, such methods have not yet reached the maturity and practical efficiency necessary to be used in software systems for automatic graph drawing. Instead, most systems apply a *layer by layer sweep* as follows: Starting from some initial permutation of the vertices on each layer, such heuristics consider pairs of layers $(L_{fixed}, L_{free}) = (L_1, L_2), (L_2, L_3), \ldots, (L_{h-1}, L_h), (L_h, L_{h+1}), \ldots, (L_2, L_1), (L_1, L_2), \ldots$ and try to determine a permutation of the vertices in L_{free} that induces a small bi-layer cross count for the sub-graph induced by the two layers, while keeping L_{fixed} temporarily fixed. These down and up sweeps continue until no improvement is achieved. Thus the problem is reduced to the *2-layer crossing minimization problem* in which a vertex permutation on one layer is fixed and the other is computed so as to induce the minimum number of pairwise interior edge crossings among the edges connecting vertices of the two layers. Eades and Wormald [14] have shown that the 2-layer crossing minimization problem with one fixed layer is *NP*-hard as well; nevertheless, in this case, the optimum can be found efficiently in practice for instances with up to about 60 vertices on the free layer, though with a rather complicated branch-and-cut algorithm by Jünger and Mutzel [22]. Experimental studies in the same article have shown that certain efficient

heuristics perform very well in practice.

There are two simple and efficient heuristics. One is the *barycenter heuristic* of Sugiyama *et al.* [27] and the other is the *median heuristic* of Eades and Wormald [14]. According to [14,27], in the 2-layer crossing minimization problem with one fixed layer a bipartite graph $G = (V,E)$ with bipartition $V = N \cup S$ such that all edges $(u, v) \in E$ have $u \in N$ (the "northern layer") and $v \in S$ (the "southern layer") are dealt with. Given a permutation $n_1, n_2, ..., n_p$ of all $n_i \in N, i \in \{1, 2, \ldots, p\}$ the wish is to find a permutation $s_1, s_2, ..., s_q$ of all $s_j \in S, j \in \{1, 2, \ldots, q\}$ that induces a small number of interior edge crossings when the edges are drawn as straight line segments connecting the positions of their end-vertices which are placed on two parallel lines according to the permutations. For ease of exposition, assume without loss of generality that there are no isolated vertices. For each vertex $s \in S$, let

$$barycenter(s) = \frac{1}{in\deg(s)} \sum_{n_i \in in\deg(s)} i$$

$$median(s) = med\{i \mid n_i \in \text{inadj}(s)\}$$

where $med(M)$ denotes the *median*, i.e., the element in position $\lfloor |X|/2 \rfloor$ when the finite multi-set $M \subseteq \mathbf{N}$ is sorted in ascending order. The $barycenter(s)$ is the barycenter of the node s and the $median(s)$ means the median pertaining to the node s.

The medians and barycenters can be computed for all $s \in S$ in $O(|E|)$ time and space. The two heuristics return S sorted by barycenters or medians, respectively, as the southern vertex permutation. The time and space complexity is as follows: the sorting step takes $O(|S| \log |S|)$ time in the barycenter heuristic and $O(|N|)$ time in the median heuristic, so that the total running time is $O(|E|+|S|$

log $|S|$) for the former and $O(|E|)$ for the latter, with $O(|E|)$ space for both. The barycenter heuristic and the median heuristic are very efficient.

In order to decide whether the layer by layer sweep should be continued or terminate, the number of crossings needs to be counted once crossing minimization heuristics have been performed,. Since all crossings occur between consecutive layer pairs, the problem reduces to the *2-layer cross counting problem*: Given permutations π_N of N and π_S of S, determine the number of pairwise interior edge crossings in the above setting. Of course, it is easy to determine whether or not two given edges in a 2-layer graph with given permutations π_N and π_S cross by simple comparisons of the relative orderings of their end vertices on L_N and L_S. This leads to an obvious algorithm with running time $O(|E|^2)$. This algorithm can even output the crossings rather than only count them, and since the number of crossings is $\Theta(|E|^2)$ in the worst case, there can be no asymptotically better algorithm. However, a list of all crossings is not needed, but only their total number. The best known approaches to the 2-layer cross counting problem, both in theory and in practice, are by Waddle and Malhotra [32] and by Barth *et al.* [1] and run in $O(|E| \log |E|)$ and $O(|E| \log(\min\{|N|, |S|\}))$ time, respectively. The former is a sweep line algorithm and the latter uses a reduction of the cross counting problem to the counting of the inversions of a certain sequence. Practice shows that a combination of the median heuristic for crossing minimization and cross counting with any of the $O(|E| \log |E|)$ algorithms can be performed for very large graphs very quickly.

A new approach for crossing reduction called *Minimizing Angle Method* with $O(|N|(\min\{|N|, |S|\}))$ time complexity will be discussed in the next chapter.

Coordinate Assignment. After the topology of the layout has been fixed by the previous two phases (i) and (ii), the purpose of phase (iii) is the assignment of coordinates to the vertices. According to [1], since each artificial vertex introduced in phase (i) gives rise to a possible edge bend in the final layout, a careless implementation of phase (iii) usually leads to the so-called "spaghetti effect" as shown in Figure 2.3(a).

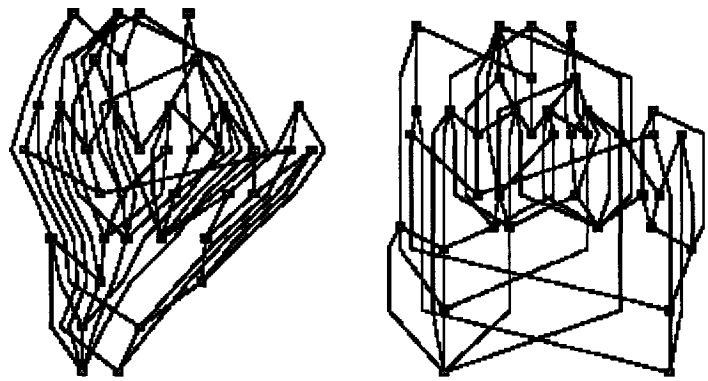


Figure 2.3 (a) The spaghetti effect and (b) a remedy (Jünger, M., Mutzel, P. [1]).

The goal is to avoid this effect by "straightening" the long edges that traverse layers in the layout so as to obtain results as shown in Figure 2.3 (b). Gansner paper [18] on layered layout treats this problem in much detail. In addition to a layer by layer sweep heuristic similar in spirit to the one described in the previous subsection for crossing reduction, Jünger-Mutzel [1] give an integer linear programming formulation as follows:

For each $e \in E$, let $\Omega(e)$ be a priority for edge e to be drawn as a vertical line segment. We wish to assign integer x-coordinates within a grid drawing in which the vertices are separated by at least one grid unit. If x_v is the x-coordinate to be assigned to $v \in V$, then the spaghetti avoidance problem can be formulated as the integer non-linear program

$$\text{minimize} \sum_{(u,v)\in E} \Omega((u,v)) \mid x_v - x_u \mid$$

subject to $x_j - x_i \geq 1$ for all pairs i and j of vertices within a layer permutation, where i is immediately followed by j $x_v \geq 0$ and integral for all $v \in V$

that can be solved efficiently in polynomial time using network flow techniques.

The authors recommend choosing

$$\Omega((u, v)) = \begin{cases} 1 & \text{if both } u \text{ and } v \text{ are non-artificial,} \\ 2 & \text{if exactly one of } u \text{ and } v \text{ is artificial,} \\ 8 & \text{if both } u \text{ and } v \text{ are artificial.} \end{cases}$$

Ideally, a long layer traversing edge has at most two bends, one at its first and one at its last artificial vertex, and is drawn vertically in between. Unfortunately the optimum of the above optimization problem cannot guarantee this. With certain additional requirements on the outcome of the crossing minimization phase (ii), two fast heuristics overcome this problem by trying to obtain near optimum solutions to the above optimization problem under the additional restriction that all long edges indeed have at most two bends and the internal parts are drawn vertically. The first (Buchheim *et al.* [10]) runs in $O(|E| \log 2 |E|)$ time; the second (Brandes and Kopf [35]) runs in $O(|E|)$ time. Both produce visually pleasing results very efficiently.

2.3.4 Planarization

The majority of graphs are non-planar. Planarization means drawing a non-planar graph as a planar graph with artificial vertices. The basic idea of the planarization method was

introduced by Tamassia *et al.* [31]. There are different ways of *planarizing* a given non-planar graph. The most widely used method in graph drawing is to construct an embedding of G with a small number of crossings and then to substitute each crossing and its involved pair of edges with an artificial vertex. The resulting planar graph G' is called a *planarized* graph from G.

After the *planarization phase*, the resulting planar graph G' is drawn using a planar drawing algorithm, and then the artificial vertices are re-substituted by crossings.

The *crossing minimization problem* searches for a drawing with the minimum number of crossings. Unfortunately, this problem is *NP*-hard [17]. A classical approach for generating a drawing with a small number of crossings is to compute a planar sub-graph P of G by temporarily removing edges. Then the removed edges are re-inserted while trying to keep the number of crossings small. In practice, this method usually leads to drawings with few crossings. The two steps are shown below:

Step 1. Edge Removal Consider the *maximum planar subgraph problem*, which is the problem of removing the minimum number of edges of a non-planar graph so that the resulting graph P is planar. This problem is *NP*-hard. Jünger and Mutzel suggest a branch-and-cut algorithm which is able to solve small problem instances. But this algorithm has exponential running time in the worst case and heuristics are often used for solving this problem in practice.

A sub-graph $P = (V,E')$ of a graph $G = (V,E)$ is called a *maximal planar Sub-graph* of G if there is no edge $e \in E \setminus E'$ so that the graph $(V,E' \cup e)$ is planar. A simple algorithm for computing a maximal planar sub-graph of a given graph G is to start with the empty graph and successively add the edges of G if their addition results in a planar graph. Edges destroying the planarity are discarded.

The incremental algorithm can be implemented in time $O(|V||E|)$.

Step 2. Edge Re-Insertion For a planar sub-graph P of $G = (V,E)$, the task is to re-insert the removed edges so that the number of edge crossings is small. More formally, the problem is to create a drawing with the minimum number of crossings in which the planar sub-graph is drawn crossing-free. The re-insertion algorithm is to choose a planar embedding of P and then successively to insert the edges of F. After each re-insertion step, the crossings are substituted by artificial vertices, thus providing a planar graph after each step. A typical drawing using planarization in combination with a planar orthogonal layout method is shown in Figure 2.4.

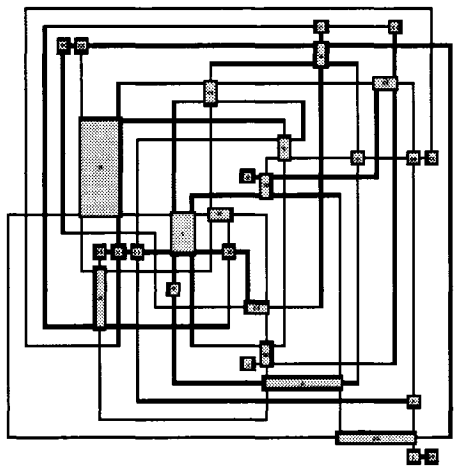


Figure 2.4 A typical drawing in planarization style (Jünger, M., Mutzel, P. [1]).

2.3.5 Orthogonal Layout

A popular style in graph drawing is orthogonal drawing. In an *orthogonal drawing*, each edge is represented as a chain of horizontal and vertical segments. A point where a horizontal and a vertical segment of an edge meet is called a *bend.*

Batini *et al.* [36] proposed a popular orthogonal drawing method called the

topology-shape-metrics approach. The topology-shape-metrics method focuses on the aesthetic criteria crossings, bends, and area of the drawing and tries to keep these numbers small while fixing the topology, the shape, and the edge lengths, in this order. It deals with the topology, shape, and geometry of the drawing separately, allowing for each aesthetic criterion to be addressed in the corresponding step, avoiding the complexity of a global optimization. The following algorithm gives an overview of the topology-shape-metrics method.

Algorithm : *The topology-shape-metrics method*

Input : Graph $G = (V,E)$;

Output: Orthogonal drawing of $G = (V,E)$

Planarization. If G is planar, then a planar embedding for G is computed. If G is not planar, a set of artificial vertices is added to replace crossings.

Orthogonalization. During this step, an orthogonal representation H of G is computed within the previously computed embedding.

Compaction. In this step a final geometry for H is determined. Namely, coordinates are assigned to vertices and bends of H.

The topology is fixed using the planarization method based on planar sub-graphs. Then, a planar orthogonal drawing method is used for planar graphs. The orthogonal drawing method will be discussed below:

Orthogonalization. Tamassia [30] gave a *region preserving* algorithm to compute a planar orthogonal grid embedding with the minimum number of bends in polynomial time by transforming it into a minimum-cost flow problem in a network. Initially, he considered only *4-planar* graphs (i.e.

planar graphs G with maxdeg(G) = 4). Given a 4-planar graph G with planar embedding Π, the network is based on the dual graph given by Π and contains $O(|V|+|F|)$ vertices and $O(|V| + |E|)$ edges for a planar embedded graph with $|F|$ faces.

Each feasible flow in the network corresponds to a possible shape of G. In particular, the minimum cost flow leads to the orthogonal shape with the lowest number of bends since each unit of cost is associated with a bend in the drawing. A simple example is shown below:

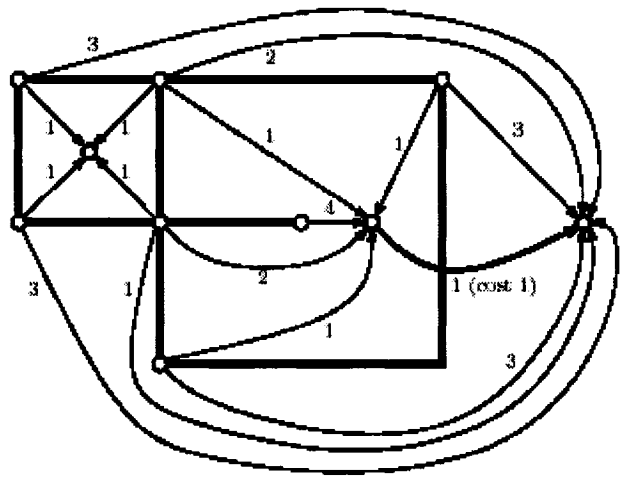


Figure 2.5 Minimal cost flow in network N and a resulting grid embedding (Jünger, M., Mutzel, P. [1]).

Compaction. After the orthogonalization phase, the description is dimensionless, and coordinates need to be assigned to the vertices and bends. The problem of computing the edge lengths of an orthogonal representation minimizing the area or the total edge length of the drawing is called the *compaction problem*. This problem is *NP*-hard [25].

Tamassia [66] has suggested the first approach in the context of graph drawing and provided a linear time algorithm based on rectilinear dissection of the orthogonal representation.

Klau and Mutzel [54] have presented a branch-and-cut algorithm which is able to solve the

compaction problem with respect to edge length minimization to provable optimality. The approach is based on a new combinatorial formulation of the problem. Besides the possibility of providing an integer linear programming formulation for the problem, the new approach also provides new classes of orthogonal representations that can be solved in polynomial time.

2.3.6 High Degree Orthogonal Drawings

Since the bend minimization algorithm by Tamassia only works for graphs with a vertex degree bounded by four as shown before, in order to generate orthogonal drawings for graphs of arbitrary vertex degree, different drawing conventions have been introduced in the literature. Fößmeier & Kaufmann [16] proposed the basic *Kandinsky drawing* convention.

A *basic Kandinsky drawing* is an orthogonal drawing such that:

1) Segments representing edges may not cross, with the exception that two segments that are incident on the same vertex may overlap.
2) All the polygons representing the faces have an area strictly greater than zero.

Basic Kandinsky drawings are usually visualized by representing vertices as boxes with equal size and representing two overlapping segments as two very near segments. Kandinsky, in [36], presented an algorithm that computes a basic Kandinsky drawing of an embedded planar graph with the minimum number of bends. As an example of the drawing see Figure 2.6 below:

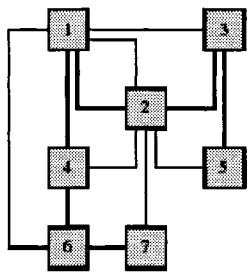


Figure 2.6 A basic Kandinsky drawing (Jünger, M., Mutzel, P. [1])

2.3.7 Force Directed Layout

Fruchterman *et al.* presented force-directed algorithms in [37]. The algorithm was designed to use a physical analogy to draw graphs. One can view a graph as a system of bodies with forces acting between the bodies. The basic idea of force-directed methods is to associate the vertices of a graph with physical entities and the edges with interactions between their end-vertex entities. Imagine that the vertices are charged particles with mutual repulsion and that the edges are springs attached at their end-vertices.

The algorithm seeks a configuration of the bodies with local minimal energy. If the local minimal energy state is attained in three-dimensional space, then assign to the vertices the Cartesian coordinates of their corresponding vertex particles in this state and draw the edges connecting the positions of their end-vertices as straight lines. This captures the general idea of force-directed methods for three-dimensional graph drawing. A typical force-directed layout in two dimensions is as follows:

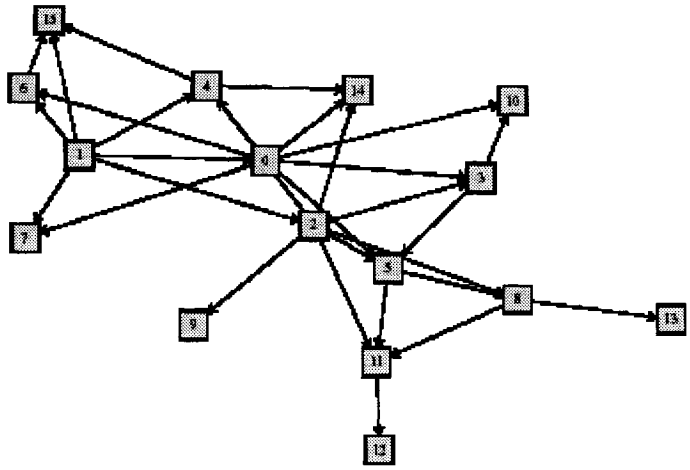
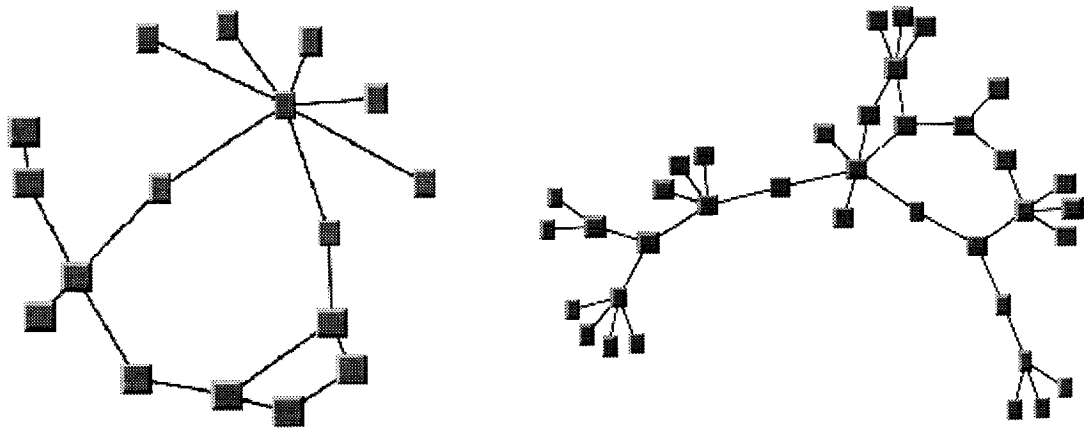


Figure 2.7 A typical force-directed layout (Jünger, M., Mutzel, P. [1])

There are two other special kinds of force-directed layout. These are the spring embedder layout and the uniform length edges layout shown below:



(a) Spring Embedder Layout (b) Uniform Length Edges Layout

Figure 2.8 Two other special kinds of force-directed layout (ILOG [56])

The spring embedder algorithm was first presented by Eades [38] and is a heuristic approach to graph drawing based on a mechanical system in which a graph's edges are replaced by springs and vertices are replaced by rings. From the initial configuration of ring positions, the system oscillates

until it stabilizes at a minimum-energy configuration. In the *Uniform Length Edges Layout* style, a uniform size will be assigned to all nodes. All edges will be routed orthogonally.

Circular Layout is a layout algorithm that draws interconnected ring and star topologies. *Circular Layouter* produces layouts that emphasize group and tree structures within a network. It partitions nodes into groups by analyzing the connectivity structure of the network. The detected groups are laid out on separate circles. The circles themselves are arranged in a radial tree layout fashion. *Circular Layouter* delegates these two major layout tasks to other more specialized layouters called Single Cycle Layout and Balloon Layout, respectively. Each group of nodes will be arranged on a separate circle. Available options are:

BCC Compact : Each group will represent a so-called biconnected component of the graph. A biconnected component consists of nodes that are reachable by two edge-disjoint paths. Nodes that belong to more than one biconnected component will be assigned exclusively to one group.

BCC Isolated: Groups will be formed as with "BCC Compact" with the difference that all nodes belonging to more than one biconnected component will be assigned an isolated group.

Single Cycle: All nodes will be arranged on a single circle.

The following is an example of the *Circular Layout*:

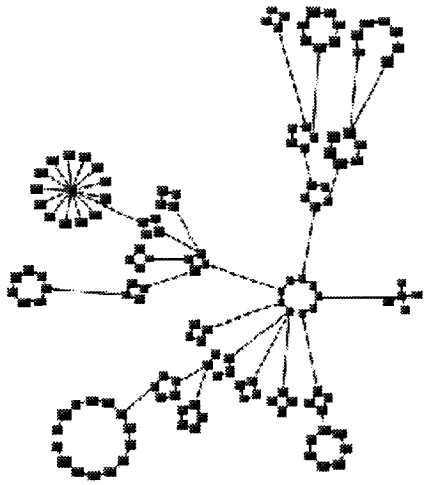


Figure 2.9 Circular Layout (Ring/Star) (ILOG [56])

2.4 3D Graph Visualization

There has been a recent trend in Graph Drawing to visualize graphs in 3-dimensional space. Although the number of applications that require such a representation for graphs is still limited, there is no doubt that 3D Graph Drawing will find many applications in the future. 3D graph visualization has many aspects, including 3D visibility representation and information 3D visualization.

2.4.1 3D Visibility Representation

3D visibility representation deals with abstract graphs and their display in 3D space. A number of software systems that produce straight line 3D drawings of graphs have been introduced. Bruss *et al.* in [39] presented a 3D graph drawing method based on the spring-embedder paradigm. Actually, they just extended the 2D Spring-embedder algorithm to 3D space.

I. Cruz *et al.* [40] used the *Simulated Annealing* method to produce straight-line 3D drawings of graphs. The idea here is that there is a predefined cost associated with the current 3D drawing of

the graph, and the system moves to drawings of lower costs until no further improvement is possible.

Orthogonal drawing in 3-dimensional space is an area that has received attention recently. A 3D orthogonal drawing typically has the following properties:

1) The vertices are points with integer coordinates in 3D space;

2) Each edge is a polyline sequence of consecutive straight line segments; each one of these line segments is parallel either to the x-axis, y-axis, or z-axis;

3) The meeting point of two consecutive straight line segments of the same edge is a bend and has integer coordinates;

4) Line segments coming from routes of two different edges are not allowed to overlap.

Papakostas *et al.* [41] presented an algorithm for producing 3D orthogonal drawings of graphs of maximum degree six and an algorithm that produces 3D orthogonal drawings of graphs of arbitrary degree. The example of the 3D orthogonal drawing of K_7 by using of the Papakostas algorithm is shown below.

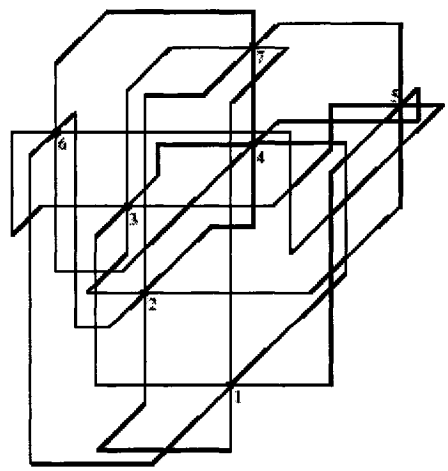


Figure 2.10 3D orthogonal drawing of K_7 (Papakostas [41])

Harel *et al.* [42] presented another 3D graph drawing method based on the *principal component analysis* tool. In the method, the graph is embedded into a high-dimensional space then projected into 2-dimensional space. It is said that the method is very fast that it can display a graph with 10^5 nodes in few seconds.

2.4.2 Information 3D Visualization

The field of *Computer-based Information Visualization* draws on ideas from several intellectual traditions: computer science, psychology, semiotics, graphic design, cartography, and art. The standard argument for visualization is that exploiting visual processing can help people explore or explain data. Those geographical systems that show a telecommunications network topology or traffic information, medical statistical information system, hierarchical task network system and the weather forecast system are simple examples of the information visualization [43,44].

There are several information visualization software packages. Cox and Patterson used the NCSA Digital Information System to depict the 3D NSFNET T1 backbone image. This image is a visualization study of inbound traffic measured in billions of bytes on the NSFNET T1 backbone for September 1991. The traffic volume range is depicted from purple (zero bytes) to white (100 billion bytes). It represents data collected by Merit Network, Inc. The image is shown below:

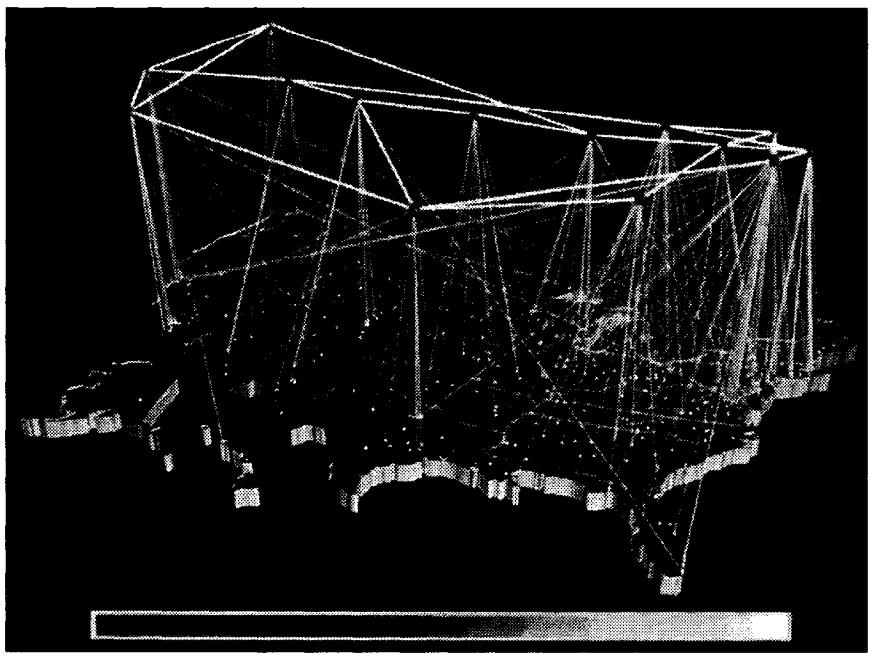
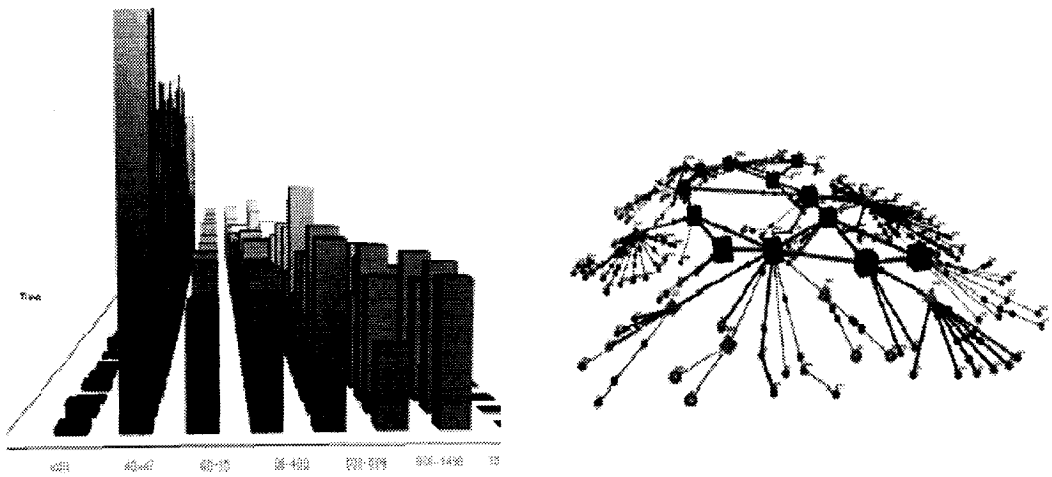


Figure2.11 Visualization Study of the NSFNET (D. Cox & R. Patterson [45])

They use visualization tools to produce a graphical rendering of the data that allows a user to explore the data "by eye."

Brown *et al.* [46] used the graph software *Cichlid* developed by NLANR and have done some network performance visualization work. *Cichlid* is a visualization tool that provides high- quality 3-D, animated visualizations of a wide range of network analysis related data sets. *Cichlid* allows the viewer to explore and interact with the data sets in real time. It was designed with remote data generation and machine independence in mind; data is transmitted via TCP from any number of sources (data servers) to the visualization engine (the client), which displays them concurrently. With *Cichlid*, one can draw *Bar Charts and Vertex-Edge Graph*. The examples are shown below:



(a) Packet Length Distribution Over Time (b) Vertex-Edge Graph Example

Figure 2.12 (a)Bar Charts and (b)Vertex-Edge Graph (Brown J.A. [46])

Eleftherios E. *et al.* [47] introduced the software SWIFT-3D, an integrated data visualization and exploration system created at AT&T Labs for large scale network analysis. SWIFT-3D integrates a collection of interactive tools that includes pixel-oriented 2D maps, interactive 3D maps, statistical displays, network topology diagrams and an interactive drill-down query interface. Figure 2.13 is an example of the interface and the picture about the virtual private network activity:

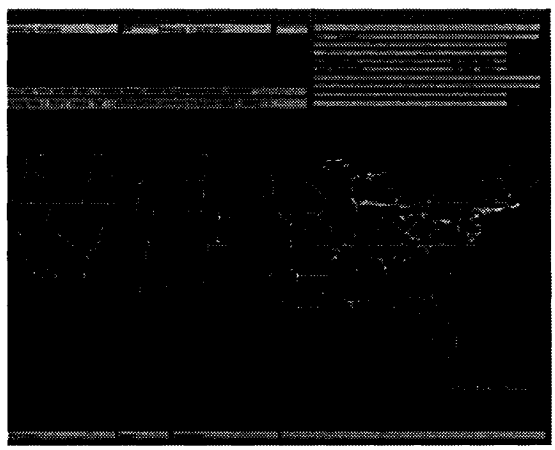


Figure 2.13 Virtual private network activity (Eleftherios E. [47])

Figure 2.14 is another example of an information visualization system. The National Laboratory for Applied Network Research (NLANR) created a 3D, interactive, immersive visualization of the very-high-performance Backbone Network Service (vBNS) and Abilene networks

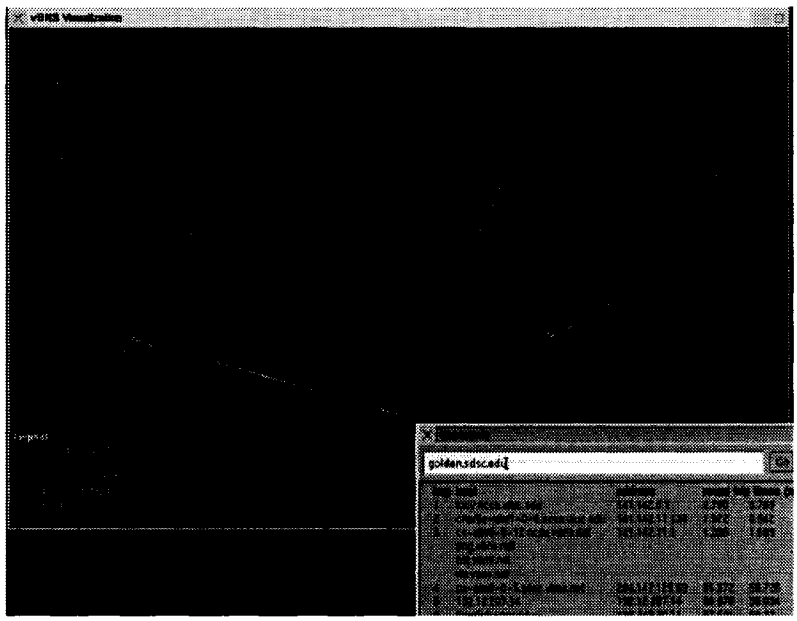


Figure 2.14 The visualization of the vBNS (NLANR)[B]

In this system, a visual trace route is developed, which plots a network path against a geographic map. An initial version was shown at the Boston Chautauqua and at Distributed Computing Workshops. This is a sample snapshot of the route between NCSA in Illinois and SDSC in California.

Chapter 3

Crossing Reduction Methods

In graph drawing, many applications such as telecommunications network display, business organizational charts, workflow diagrams and genealogical trees need to draw the graphs satisfying certain common criteria. Minimizing the number of link crossings is the first criterion. The well-known crossing reduction methods are sorting crossing reduction, barycenter heuristics, median heuristics [52] and integer programming, all of which will be described below. The minimizing angle method proposed by the author will be discussed later.

3.1 Introduction to Crossing Reduction

Given a graph $G=(V,E)$, in automatic graph drawing, especially in layered layout, usually there are three phases to draw a graph satisfying certain criteria :

Step1 Layer Assignment: Compute a layering of the graph via a topological numbering. That means assign all vertices to disjoint nonempty subsets $L_1,L_2,...,L_h$ of V called layers such that for

each edge $e=(u,v)$ if the end-vertex u is assigned to layer L_i and the end-vertex v is assigned to layer L_j then $j>i$. If $j>i+1$, add some artificial intermediate vertices for each traversed layer.

Step2 Crossing Minimization: For each layer L_i, determine the permutations of the vertices in L_i such that there are smallest crossings among edges connecting layer L_{i-1} and L_i.

Step3 Coordinate Assignment: Assign each vertex $v \in V$ the y-coordinate i if $v \in L_i$ and an x-coordinate compatible with the vertex permutation of L_i. Sometimes suppress the artificial vertices in the drawing.

Sugiyama *et al.* [50] used the three steps above to produce the so called *Sugiyama-Style layout.* For counting the crossing, Wilhelm *et al.* [48] gave an efficient crossing counting method.

The key step in layered graph drawing is step 2. Here crossing minimization is mainly focused on. Li *et al* [51] and Garey *et al* [55] recognized that the crossing minimization problem is *NP*-hard even when restricted to two-layer instance. Furthermore the two-layer crossing minimization problem with the fixed layer is *NP*-hard, as well [54], which means that there exists no "best algorithm" for the problem. Nevertheless, the optimum can be found efficiently in practice with a rather complicated branch-and-cut algorithm [49]. Certain efficient heuristics perform very well in practice.

To minimize crossing the *layer-by-layer sweep* [27] method is adopted usually. This method can be described as follows: first, a vetex ordering of layer L_1 is chosen. Then, for i=2,3,...,h, the vertex ordering of layer L_{i-1} is held fixed while reordering the vertices in layer L_i. So, the whole thing is reduced to the two-layer crossing problem: Given a fixed vertex ordering of layer L_{i-1}, choose a vertex ordering of layer L_i to minimize the number of edge crossing.

In the two-layer crossing minimization problem with one fixed layer, a bipartite graph $G=(V,E)$ is a graph with bipartition $V=N \cup S$ such that all edges $(u,v) \in E$, the node $u \in N$ (the northern layer) and the $v \in S$ (the southern layer). Given a permutation $(n_1, n_2, ..., n_p)$ of all $n_i \in N$, $i=1,2,...,p$, the goal is to find a permutation $(s_1, s_2, ..., s_q)$ of all $s_j \in S$, such that the permutation induces a small number of interior edge crossings when the edges are drawn as straight lines connecting the positions of their end-vertices placed on two parallel line according to the permutations.

Example 3.1 The graph below is an example of a bipartite graph $G=(N,S,E)$, where $N=\{n_1, n_2, n_3, n_4, n_5\}$, $S=\{s_1, s_2, s_3, s_4, s_5, s_6\}$ and $E=\{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}\}$.

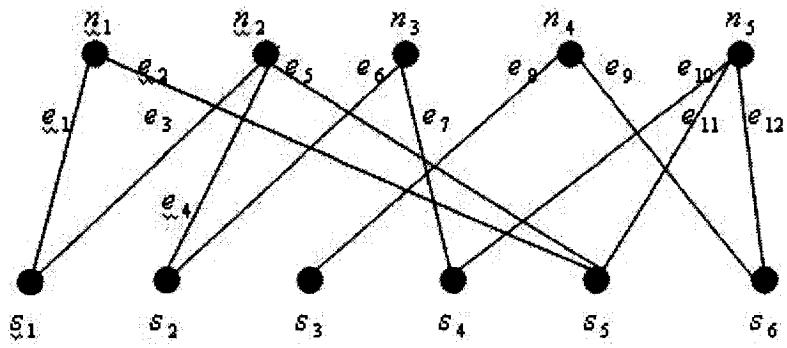


Figure 3.1 A bi-layer diagraph

As mentioned before, the two-layer crossing problem is also a *NP*-hard problem.

Nevertheless, there are many heuristic methods that can be used to solve the problem efficiently. The following sections discuss the *sorting crossing reduction methods*, the *barycenter* and *median methods*, and the *integer programming method*. Finally, the *minimizing angle method* will be discussed in detail.

3.2 The Sorting Crossing Reduction Method

The well known sorting crossing reduction method was the quick-sort algorithm. First, choose a pivot vertex $p \in L_2$, and place each vertex $u \neq p \in L_2$ to the left of p if $c_{up} < c_{pu}$ and to the right of p otherwise, where c_{up} is the crossing number formed by the edges incident with u making with edges incident with p. The algorithm is then applied recursively to the sets of vertices to the left and right of p.

Algorithm ***Sorting Crossing Reduction***

Input: two-layered digraph $G=(L_1, L_2, E)$, an order x_1 for L_1

Output: vertex order x_2 for L_2

if L_2 is not empty **then**

(a) Choose a pivot vertex $p \in L_2$.

(b) $V_l = \phi$; $V_r = \phi$

(c) **for** each vertex $u \neq p \in L_2$ **do**

if $c_{up} < c_{pu}$ **then** place u in V_l **else** place u in V_r

(d) Recursively apply the algorithm to the digraph included by V_l and V_r, and output the concatenation of the output of these two applications.

The time complexity of the algorithm is $O(|L_2|^2)$. Several methods similar to the algorithm above are described in [57].

3.3 The Barycenter and Median Methods

The most common and simple method employed for the two-layer crossing problem are variations of the barycenter method. Roughly speaking, the x-coordinate of each vertex $u \in L_2$ is chosen as the barycenter (average) of the x-coordinates of its neighbors.

Let $inadj(u)=\{v \in L_1 \mid (v, u) \in E\}$ and $indeg(u)=|inadj(u)|$. For each vertex $u \in L_2$, let

$$b(u) = \text{barycenter}(u) = \frac{1}{in\deg(u)} \sum_{v \in in\deg(u)} x_1(v). \qquad (3.3.1)$$

The $b(u)$ is called the barycenter of the vertex u. If two vertices have the same barycenter, then they are separated arbitrarily by a small amount. The barycenter method can be implemented in linear time. The number of crossings of output by the barycenter method is denoted by $bary(G)$.

The median method is similar to the barycenter method. Roughly speaking, the x-coordinate of each vertex $u \in L_2$ is chosen to be a median of the x-coordinates of its neighbors as follows:

$$m(u) = \text{median}(u) = \text{med}\{ x_1(v) \mid v \in inadj(u)\}. \qquad (3.3.2)$$

where med$\{X\}$ denote the element in position $\lfloor |X|/2 \rfloor$ when the finite set $X \subseteq N$ is sorted in ascending order.

The $m(u)$ is called the median of the vertex u. The vertices are re-arranged by sorting their median coordinates. The following criterion is used to break the ties: if $m(u_i) = m(u_j)$ and one vertex has odd degree and the other even, then the odd degree vertex is placed on the left of the even degree vertex. If the parity of the degree of u_i and u_j is the same, then the order is chosen arbitrarily. Also, the median method can be implemented in linear time. The number of crossings output by the barycenter method is denoted by $med(G)$.

According to the definitions above, the barycenters and medians of all the southern nodes can be easily computed as follows:

$b(s_1)$ =(1/2)(1+2) =1.5, $b(s_2)$ =(1/2)(2+3) =2.5, $b(s_3)$ =(1/1)(4) =4

$b(s_4)$ =(1/2)(3+5) =4, $b(s_5)$ =(1/3)(1+2+5) =2.7, $b(s_6)$ =(1/2)(4+5) =4.5

and

$m(s_1)$ = median(1,2) =1, $m(s_2)$ = median(2,3) =2, $m(s_3)$ = median(4) =4

$m(s_4)$ = median(3,5) =3, $m(s_5)$ = median(1,2,5) =2, $m(s_6)$ = median(4,5) =4

After sorting the barycenters as (1.5, 2.5, 2.7, 4, 4, 4.5), the sorted vertices (s_1, s_2, s_5, s_3, s_4, s_6) by the barycenter heuristics, which corresponds to the following bipartite graph, is obtained (Figure 3.2):

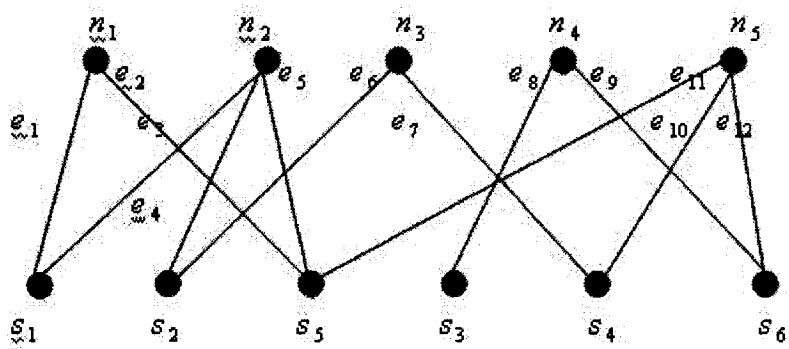


Figure 3.2 A bipartite graph produced by barycenter heuristics

From the figure above, it is shown that the permutation of the southern layer includes just 9 crossings.

The median heuristics gives the permutation of the southern vertices as (s_1, s_5, s_2, s_4, s_3, s_6) which corresponds to the bipartite graph below (Figure 3.3):

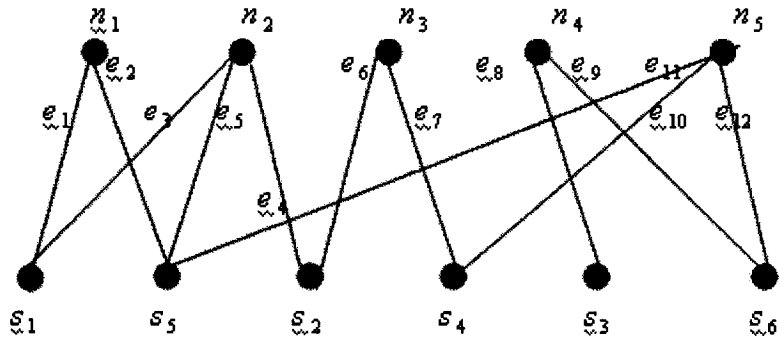


Figure 3.3 A bipartite graph produced by median heuristics

The number of crossings in the above figure is 8 .

Actually, the number of crossings with the minimizing angle algorithm is also 8, which corresponds to Figure 3.4 below:

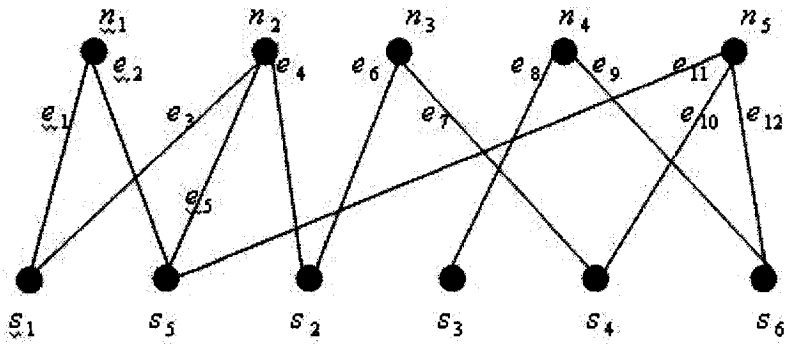


Figure 3.4 A bipartite graph produced by the minimizing angle algorithm

3.4 Integer Programming Method for Crossing Reduction

An integer programming approach may be used to solve the two-layer crossing problem. For a two-layer digraph $G=(L_1, L_2, E)$, define a binary vector $x \in \{0, 1\}^{\binom{|L_2|}{2}}$, with entries x_{uv} for $u < v$. The value $x_{uv} = 1$ if u is to the left of v and $x_{uv} = 0$ otherwise. Then the crossing number *cross*(G, x_1, x_2) can be reduced to

$$cross(G, x_1, x_2) = \sum_{u<v\in L_2} (c_{uv}x_{uv} + c_{vu}(1-x_{uv}))$$

$$= \sum_{u<v\in L_2} (c_{uv} - c_{vu})x_{uv} + \sum_{u<v\in L_2} c_{vu}\ .$$

Noting that $\sum_{u<v\in L_2} c_{vu}$ is a constant, the crossing problem can be re-stated as follows:

Minimizing $\quad z = \sum_{u<v\in L_2} (c_{uv} - c_{vu})x_{uv}$

Subject to: 1) $0 \leq x_{uv} + x_{vu} - x_{uw} \leq 1$ for all triples $u < v < w$ of distinct vertices in L_2.

2) $x_{uv} \in \{0,1\}$ for all pairs $u < v$ of distinct vertices in L_2.

Integer programming seems simple at first sight. In general, solving it requires sophisticated techniques. A *branch and cut approach* [58] can be used to obtain an optimal solution for digraph of limited size. If using standard linear programming, the first set of constraints is large ($O(|L_2|^3)$). Therefore, a cutting plane approach is used. The algorithm starts using only the above inequalities then the elements of the set are iteratively used to "cut" the solution space. If this finds a solution, then the procedure stops, otherwise, a variable x_{uv} is chosen, and two problems are spawned, one with $x_{uv} = 0$, the other with $x_{uv} = 1$. These problems are solved recursively.

3.5 Minimizing Angle Approach

This is the method for the reduction of crossings proposed in this thesis. The following observation leads to the minimizing angle algorithm.

Intuitively, the wider an edge spans, the more it crosses other edges. For an edge $e_i \in E$, let α_i denote the angle formed by the edge e_i and a vertical line as shown below, where $0 \leq \alpha_i < 90°$:

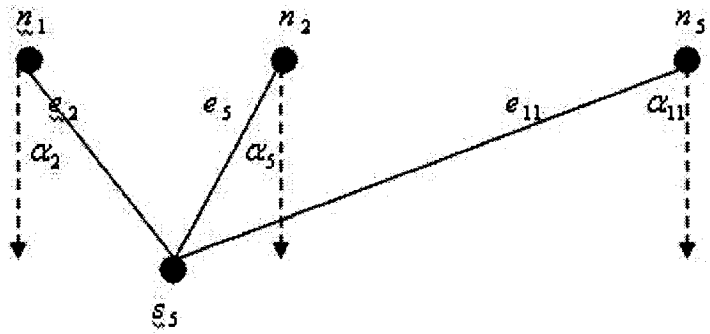


Figure 3.5 The angle definition for minimizing angle algorithm

In the figure above, the three edges e_2, e_5 and e_{11} are adjacent to s_5 and the corresponding angles are α_2, α_5 and α_{11}. The sum of the angles for the node s_5 is $\alpha_2+\alpha_5+\alpha_{11}$. Obviously, for each southern node s_j, if the sum of the angles related to the edges that are adjacent to the node s_j is kept as small as possible, then the crossings will be reduced efficiently.

To describe the approach precisely, some notations and definitions are needed.

In a given bi-layer graph $G=(N,S,E)$ with fixed northern layer permutation $N=(n_1, n_2, ..., n_p)$, notice that the number of the crossings formed by the edges between the northern layer nodes and the southern layer nodes depends on the relative position of the southern layer nodes on a horizontal line or the permutation $S=(s_1, s_2, ..., s_q)$ not on the absolute position on the line. So, all the northern

layer nodes on a horizontal line y=1 are arranged with the first node on (1,1), the second node on (2,1), ..., the last node on (p,1) as shown below:

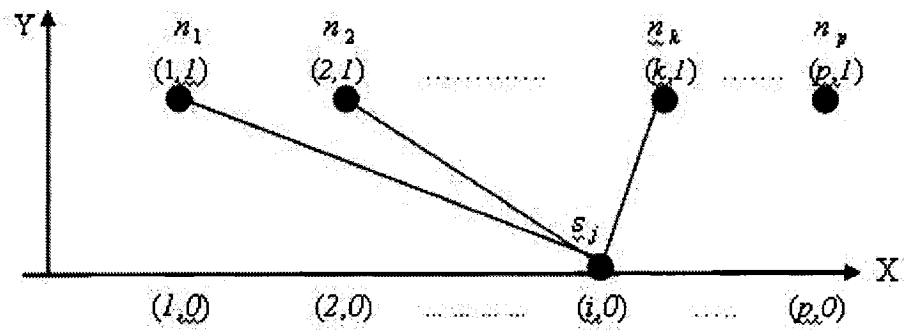


Figure 3.6 Layer nodes arrangement

The angles formed by the edges and the vertical line can be easily expressed. For example, if an edge determined by a northern node n_k and a southern node s_j with x-coordinate (x, 0), then the angle formed by the edge $e_{kj} = (n_k, s_j)$ and the vertical direction can be calculated by the function $\alpha_k(x) = arctan(|x\text{-}k|)$. Obviously, $\alpha_k(x) \geq 0$.

Define

$$A(s_j, x) = \sum_{n_k \in inadj(s_j)} \alpha_k(x), \qquad (3.5.1)$$

where the $\alpha_k(x)$ is the angle formed by the edge $e_{kj} = (n_k, s_j)$ and the vertical direction as shown in Figure 3.6, where the number x is s_j's x-coordinate (x, 0). For the function $A(s_j, x)$, the following consequence isdrawn:

Lemma 3.5.1 The sum of the angles $A(s_j, x)$ can attain its minimum somewhere on the x-axis at a finite x-coordinate.

Proof: Obviously, the sum of the angles $A(s_j,x)$ is just the function of the position of the southern node s_j on the x-axis. If the node s_j is moved towards $-\infty$ or $+\infty$, the value of the function $A(s_j,x)$ will be increased to $+\infty$. So, the function $A(s_j,x)$ can definitely attain its minimum somewhere on the x-axis at a finite x-coordinate $x_{\min}$.

Although there may be several minima for the function $A(s_j,x)$, there exists at least one and that is enough. To think that if all minimum points on the x-axis for all southern nodes (maybe some of them overlapped) are obtained, they can be sorted to get a permutation of the southern nodes, which will definitely reduce the crossings among the edges. But how can find the positions on the x-axis for the minima? Generally, this can be done by nonlinear programming. For each southern node s_j, the minimal angle position can be determined by the following nonlinear integer programming:

$$\begin{cases} \text{minimize } A(s_j,x) = \sum_{n_k \in inadj(s_j)} \alpha_k(x) \\ \text{subject to: } x \text{ is integer on } x\text{-axis} \end{cases}$$

where $\alpha_k(x) = arctan|x-k|$.

Actually, the nonlinear programming above can be solved by the branch-and-bound algorithm. To determine the minimizing angle position for a southern node, first a trial position can be gotten by computing of the barycenter of the node s_j:

$$b(s_j) = \text{barycenter}(s_j) = \frac{1}{in\deg(s_j)} \sum_{n_k \in in\deg(s_j)} x(n_k) \qquad (3.5.2)$$

where $x(n_k)$ is the x-coordinate of the node n_k. Let $b_j = int(b(s_j))$. Then calculate three values $A(s_j,b_j)$, $A(s_j,b_j\text{-}1)$ and $A(s_j,b_j\text{+}1)$. If $A(s_j,b_j\text{-}1) < A(s_j,b_j) < A(s_j,b_j\text{+}1)$ then the b_j is one of the minimal positions. If $A(s_j,b_j) < A(s_j,b_j\text{-}1)$, then move s_j one unit left to $x_j = b_j\text{-}1$.

Otherwise move s_j one unit right to $x_j = b_j + 1$. Continue the process and, by lemma 3.5.1, in finite step, a position can be reached such that $A(s_j, x_j - 1) < A(s_j, x_j) < A(s_j, x_j + 1)$. Here the x-coordinate x_j is just the position for the node s_j. Now the algorithm can be summarized as follows:

Algorithm *mini_angle*(N, s_j)

Input: $N = \{n_1, n_2, ..., n_p\}$, s_j, where the permutation of N is fixed.

Output: A x-coordinate x_j for s_j such that the function $A(s_j, x_j)$ attains one of its minimums.

1. Compute the barycenter b_j of the node s_j by the formula (3.5.2).
2. Compute the value of the function $A(s_j, b_j)$, $A(s_j, b_j - 1)$ and $A(s_j, b_j + 1)$ respectively according to the formula (3.5.1).
3. *If* $A(s_j, b_j - 1) < A(s_j, b_j) < A(s_j, b_j + 1)$ *then* return b_j.

   Else if $A(s_j, b_j) < A(s_j, b_j - 1)$ *Then* $b_j \Leftarrow b_j - 1$ *Goto* step 2.

   Else if $A(s_j, b_j + 1) < A(s_j, b_j)$ *Then* $b_j \Leftarrow b_j + 1$ *Goto* step 2.

To show the correctness of the algorithm *mini_angle*(N, s_j), the following lemma is needed.

Lemma 3.5.2 Let $l = max\{|N|, |S|\}$. Then the number of steps that will be moved to attain the minimal position for s_j does not exceed $max\{b_j, l - b_j\}$.

Proof: Since the first nodes n_1 and s_1 are put on the same vertical line with x-coordinate 1 and b_j is the barycenter of the node s_j, then the positions at which the s_j can be put on the x-axis to attain its minimal angle position will be between 1 to b_j if s_j is moved to the left of the b_j or

between b_j to l-b_j if s_j is moved to the right of the b_j. So the maximum of the steps is not greater than the value max{ b_j, l-b_j }.

The correctness of the Algorithm *mini_angle*(N, s_j):

Since the number of the times that the node s_j is moved to the left or right of the barycenter b_j does not exceed max{ b_j, l-b_j }, then the iteration in Step 3 in the algorithm will be terminated definitely in finite steps. Therefore, in finite steps, the minimal value position on x-axis for the node s_j can be located.

In fact, if the trial position for the node s_j is obtained by computing its barycenter first, one of its minimal angle positions can quickly be found. Take the bipartite graph in Figure 3.1 as an example. To find the minimal angle position for the node s_5, the barycenter for s_5 is computed and b_5 = int(2.6) = 3. Unfortunately, the position x=3 is not the minimal angle position, because $A(s_5,2)$ = *arctan*(1)+*arctan*(4) $\approx$ 2.326, which is less than $A(s_5,3)$ = *arctan*(2)+*arctan*(3) $\approx$ 2.357. Furthermore, since $A(s_5,1)$ = arctan(1)+arctan(5) $\approx$ 2.373, which is larger than $A(s_5,2)$, $A(s_5,2)$ is the minimum for s_5 and the minimal angle position is x=2.

It is the case that some nodes have minimal angle positions greater than one. For instance, in the bipartite graph in Figure 3.1, the node s_1 has two minimal angle positions x=1 and x=2 and the node s_4 has minimal angle positions x=3 and x=5. Either of them is eligible and can be used if it is needed.

The time complexity of the algorithm *mini_angle*(N, s_j):

It is obvious that lines 1 and 2 takes constant time and line 3 will loop at most $max\{b_j, l\text{-}b_j\}$ times. That is $O(max(|N|,|S|))$. To be convenient, the definition of the min-angle coordinate for a node is given below.

Definition 3.5.1 For a southern node s, the minimal angle position for the node s is called the *min-angle coordinate* of the node s.

The Algorithm *mini_angle*(N, s_j) can give the minimal angle position for s_j (i=1,2,...,|S|). In the case that different nodes may have the same min-angle coordinate, a way to break the tie must be found and a criterion to adjust them needs to be set. Since the number of crossings among the edges depends on the relative rather than the absolute position, related nodes can be translated along the specific direction without changing the relative position of the nodes. For example, if there are k nodes which have the same min-angle coordinate x_i, the node with the largest *in degree* to the position $x=x_i$ is set first, then second largest node to $x=x_i+1$ and so on. Finally, the least is set to the position $x=x_i+k$. Furthermore, the node with min-angle coordinate x_j (where originally $x_j > x_i$) is translated to the position $x=x_j+k+1$. After finishing all the adjusting work, a permutation of the southern nodes, with which the bipartite graph may have less crossings, can be obtained finally. Denote the minimal angle position for s_j by m_j (i=1,2,...,|S|) and denote $A(s_j, m_j)$ the minimal angle value for s_j.

The minimal angle algorithm is given here:

Algorithm *mini_angle*(N, S)

Input: A bipartite graph $G=(N,S,E)$, where $N=\{ n_1, n_2, ..., n_p \}$,

$S=\{ s_1, s_2, ..., s_q \}$ and the permutation of N is fixed.

Output: A permutation for S such that the number of crossings is reduced.

1. *Initialize* a permutation $P = \{1,2,... \mid S \mid\}$.
2. *For j=1 to* $|S|$
3. *Do* m_j =*mini_angle*(N, s_j)
4. *Update* the permutation $P = \{m_1, m_2, ..., m_j, ..., j, ..., \mid S \mid\}$.
5. *End do*
6. *Return* the permutation P.

The correctness of the Algorithm *mini_angle*(N,S):

Since for each node s_j the procedure *mini_angle*(N, s_j) can return its local min-angle coordinate properly and the position m_j for s_j can be recorded into the permutation P. Then the permutation P will be updated one at a time. After the loop from 1 to $\mid S \mid$, the permutation P gives an order of the adjusted south nodes from left to right. This permutation of the southern nodes stands for a local minimum of the sum of the angles, which may cause less crossings in a bipartite graph $G=(N,S,E)$.

The time complexity of the algorithm:

Since line 3 will take $O(max(|N|,|S|))$ time, the overall time will be $O(|S|max(|N|,|S|))$.

Minimizing angle algorithm is actually a heuristic method that can be used to reduce the crossings in a bipartite graph. If a bipartite graph is given, one can employ the algorithm

***mini_angle*(N, s_j)** to each southern node to find a local minimal angle position for the node. Then, the number of crossings is counted. If the number of the crossings is reduced, store the permutation obtained. Otherwise keep the position of the node and skip to next node to do the same job. After loop from the first node to the last one, the angles have been reduced definitely. Since the total of the angles are reduced, the number of crossings may be reduced as well.

Definition 3.5.2 The minimizing angle approach is called *minimizing angle heuristics* to reduce the edge crossings for a bipartite graph G.

The following figure shows the process view of the *minimizing angle heuristics*:

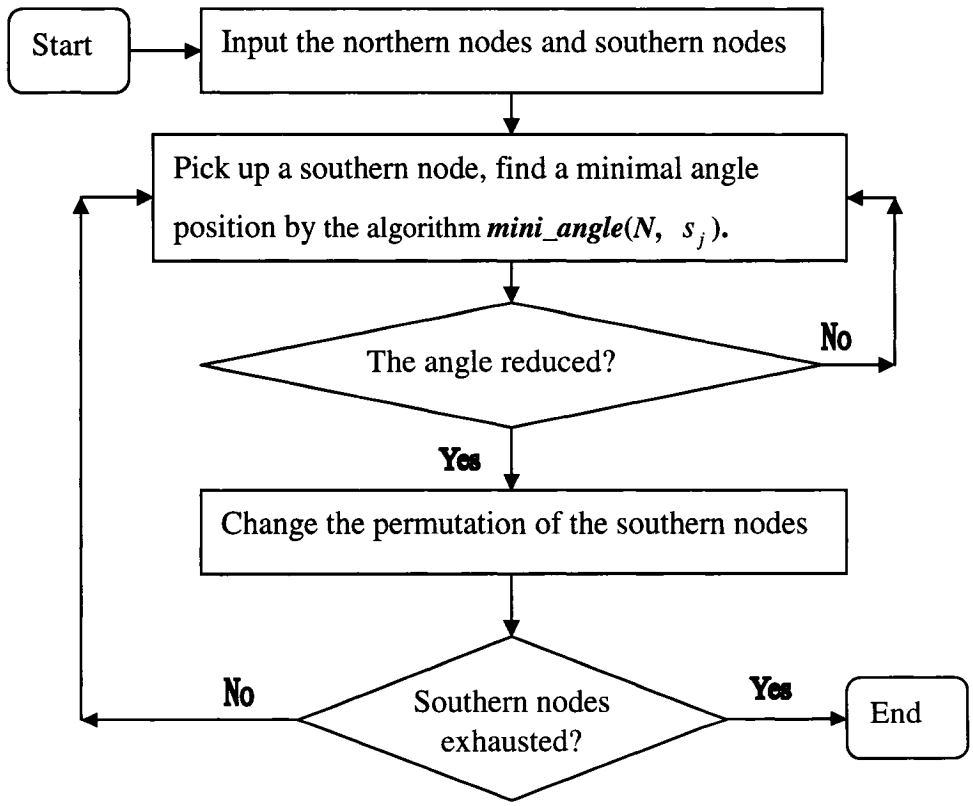


Figure 3.7 The process view of the minimizing angle heuristics

Now the results obtained above can be summarized to the following theorem:

Theorem 3.5.1 Given a bipartite graph $G=(N,S,E)$, the minimizing angle approach can produce a permutation of the south nodes, which may reduce the crossings among the edges with the time complexity of $O(|S|max(|N|,|S|))$.

Minimizing angle heuristics can be used to reduce the crossings. Now the *minimizing angle heuristics* is used to solve the crossing reduction problem in Figure 3.1. By calculating the sum of the tangent values for every $s_j \in S$, the calculating result is as follows:

mini_angle(N, s_1)={1,2}, *minimizeA*(s_1,1) = *minimizeA*(s_1,2)=0.785;

mini_angle(N, s_2)={2,3}, *minimizeA*(s_2,2) = *minimizeA*(s_2,3)=0.785;

mini_angle(N, s_3)={4}, *minimizeA*(s_3, 4) = *0;*

mini_angle(N, s_4)={3,5}, *minimizeA*(s_4,3) = *minimizeA*(s_4,5)=1.107;

mini_angle(N, s_5)={2}, *minimizeA*(s_5, 2) = 2.029*;*

mini_angle(N, s_6)={5}. *minimizeA*(s_6,5) =0.785.

According to the value given above, s_1 is set to position number one, s_2 is set to position number three, s_3 is set to position number four, s_4 is set to position number 5, s_5 is set to position number 2, and s_6 finally, is set to position number 6. Therefore the corresponding graph is shown in Figure 3.4. The number of crossings is 8, which is the same as the number in the median method.

Example 3.2 Another example is given here, in which just the edges for s_5 are adjusted from Figure3.8:

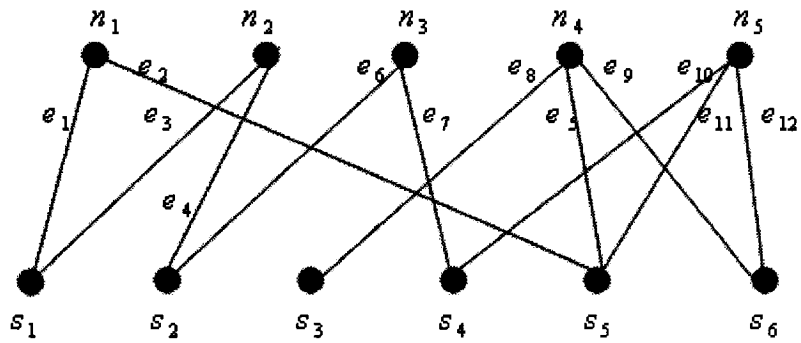


Figure 3.8 A bipartite diagraph with crossing 10

There are 10 crossings in the graph.

At first, calculate its barycenters and medians, respectively:

$b(s_1)$ =(1/2)(1+2) =1.5, $b(s_2)$ =(1/2)(2+3) =2.5, $b(s_3)$ =(1/1)(4) =4,

$b(s_4)$ =(1/2)(3+5) =4, $b(s_5)$ =(1/3)(1+4+5) =3.3, $b(s_6)$ =(1/2)(4+5) =4.5;

$m(s_1)$ = median(1,2) =1, $m(s_2)$ = median(2,3) =2, $m(s_3)$ = median(4) =4,

$m(s_4)$ = median(3,5) =3, $m(s_5)$ = median(1,4,5) =4, $m(s_6)$ = median(4,5) =4;

After sorting the barycenters to get (1.5, 2.5, 3.3, 4, 4, 4.5), the permutation of the vertices is $(s_1, s_2, s_5, s_3, s_4, s_6)$, which is correspond to the following bipartite graph (Figure 3.9):

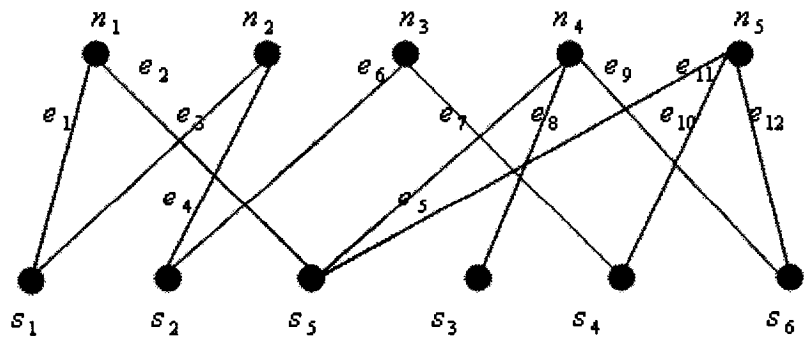


Figure 3.9 Bipartite graph produced by barycenter heuristics

In the figure above, there are 9 crossings.

By using the medians, another permutation of the vertices is obtained $(s_1, s_2, s_4, s_5, s_3, s_6)$, which corresponds to the following bipartite graph (Figure 3.10):

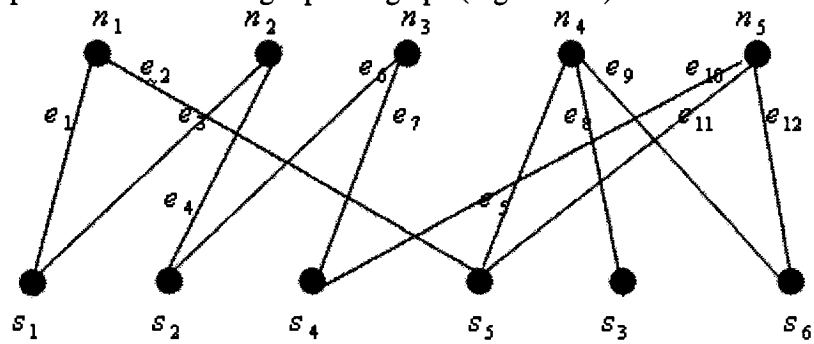


Figure 3.10 Bipartite graph produced by median heuristics

The number of the crossings in this case is not reduced (10 crossings).

By using *minimizing angle heuristics*, the results are listed below:

mini_angle(N, s_1)={1,2}, *minimizeA*(s_1,1) = *minimizeA*(s_1,2)=0.785;

mini_angle(N, s_2)={2,3}, *minimizeA*(s_2,2) = *minimizeA*(s_2,3)=0.785;

mini_angle(N, s_3)={4}, *minimizeA*(s_3, 4) = 0;

mini_angle(N, s_4)={3,5}, *minimizeA*(s_4,3) = *minimizeA*(s_4,5)=1.107;

mini_angle(N, s_5)={4,5}, *minimizeA*(s_5, 2) = 2.111;

mini_angle(N, s_6)={5}. *minimizeA*(s_6,5) =0.785.

The permutation is determined by the information provided above and can be listed as $(s_1, s_2, s_5, s_3, s_4, s_6)$. The corresponding figure is the same as the Figure 3.9.

Example 3.3 A bipartite diagraph with crossing 35:

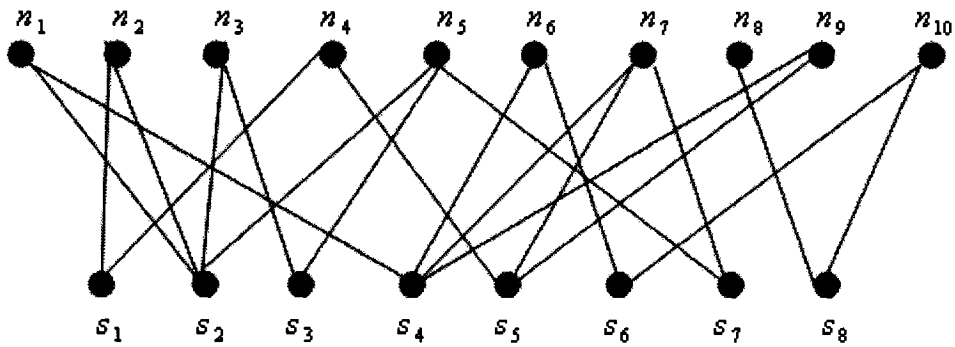


Figure 3.11 A bipartite diagraph with crossing 35

Its barycenters are calculated as follows:

$b(s_1)$ =(1/2)(2+4) =3, $b(s_2)$ =(1/4)(1+2+3+5) =2.4, $b(s_3)$ =(1/2)(3+5) =4,

$b(s_4)$ =(1/4)(1+6+7+9) =5.7, $b(s_5)$ =(1/3)(4+7+9) =6.67, $b(s_6)$ =(1/2)(6+10) =8;

$b(s_7)$ =(1/2)(5+7) =6, $b(s_8)$ =(1/2)(8+10) =9.

By sorting the barycenter values, the permutation of the vertices is obtained as (s_2, s_1, s_3, s_4, s_7, s_5 ,s_6, s_8), which corresponding to the following bipartite graph (Figure 3.12):

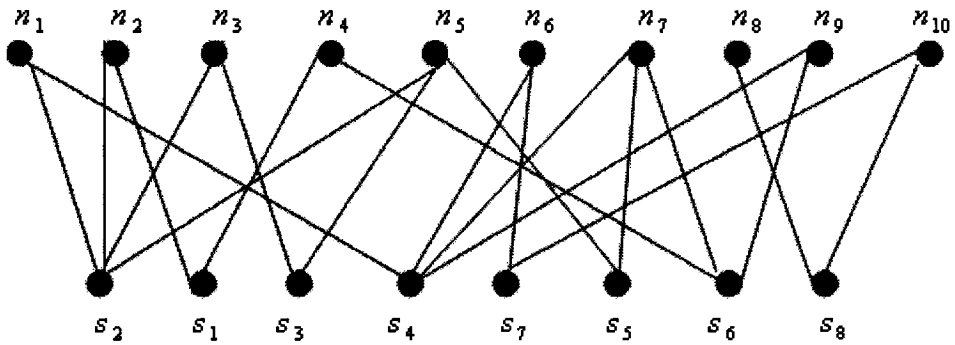


Figure 3.12 Bipartite graph produced by barycenter heuristics

Then calculate the medians:

$m(s_1)$ = median(2,4) =2, $m(s_2)$ = median(1,2,3,5) =2, $m(s_3)$ = median(3,5) =3,

$m(s_4)$ = median(1,6,7,9) =6, $m(s_5)$ = median(4,7,9) =7, $m(s_6)$ = median(6,10) =6;

$m(s_7)$ = median(5,7) =5, $m(s_8)$ = median(8,10) =8,

Sort the medians to get a permutation of the vertices as $(s_2, s_1, s_3, s_7\ s_4, s_6\ s_5, s_8)$ what does corresponds to the following bipartite graph (Figure 3.13):

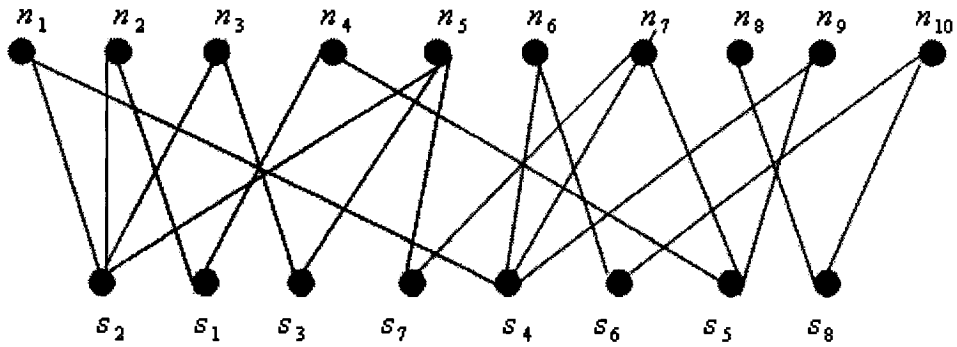


Figure 3.13 Bipartite graph produced by median heuristics

The figure above shows that the number of the crossings is 33.

By using the *minimizing angle heuristics*, the calculating results are listed below:

mini_angle(*N*, s_1)={2,4}, *minimizeA*(s_1,2) = *minimizeA*(s_1,4)=1.017;

mini_angle(*N*, s_2)={2}, *minimizeA*(s_2,2) =2,819;

mini_angle(*N*, s_3)={3,5}, *minimizeA*(s_3, 3) = *minimizeA*(s_3, 5)=1.017;

mini_angle(*N*, s_4)={6}, *minimizeA*(s_4,6) =3.4;

mini_angle(*N*, s_5)={7}, *minimizeA*(s_5, 7) = 2.356;

mini_angle(*N*, s_6)={6}. *minimizeA*(s_6,6) =1.325;

mini_angle(*N*, s_7)={5,7}. *minimizeA*(s_7,5) = *minimizeA*(s_7,7)=1.017;

mini_angle(*N*, s_8)={8}. *minimizeA*(s_8,8) =1.017.

In this case, though, s_1 and s_2 can take position 2, since s_2 has more edges; therefore, put s_2 at position one and s_1 at position two. Then put s_3 at position three, and s_7 at position four. Because s_4 has more edges than s_6, s_4 goes before s_6. Then put s_5 and s_8 at position seven

and eight, respectively. Finally, the permutation is listed as (s_2, s_1, s_3, s_7 s_4, s_6 s_5, s_8). The corresponding figure is the same as Figure 3.13. It reduced the crossing number to 33 as well.

As mentioned before, two of the popular methods for crossing reduction are barycenter heuristics and median heuristics. The advantage of the two methods is that they are simple, efficient and easy to be implemented. But sometimes they do not give the better result. *Minimizing angle heuristics* can produce the better result at cost of little more time.

Minimizing angle heuristics is an alternative method that can be used to reduce crossings. It minimizes the sum of angles to get a permutation for the southern nodes in a bipartite graph. With the minimizing angle heuristics, the number of the crossings among the edges of the bipartite graph may be reduced. The crossing reducing can be achieved further by employing other methods, such as exchanging a node with its neighbors. Still much work needs to be done to implement the method for crossing reduction.

Chapter 4

Architecture of Algorithms for 3D graphs and multi-layered networks

Graph drawing addresses the problem of automatically generating geometric representations of abstract graphs or networks. There has been a recent trend in graph drawing to visualize graphs in three dimensional space. Although the number of applications that require such a representation for graphs is still limited, there is no doubt that 3D graph drawing will find many applications in the future. 3D graph drawing and network display need the support of graph and network 3D display algorithms. Few such algorithms can be found in the literature. This thesis proposes two graph drawing algorithms. The first one is a general 3D graph drawing algorithm, *the Incremental Projection Algorithm.* The other is a 3D display algorithm for special multi-layered networks, which is called the *Depth-Height Buffer algorithm.* These two algorithms are presented in detail in this chapter.

4.1 Introduction to the Incremental Projection Algorithm

Usually a general graph is given for a specific application problem. It is a theoretical and practical problem of how to display the graph in a 3D space such that it satisfies certain aesthetic criteria. There are many planar graph drawing algorithms as shown before, but little is known about the theory of 3D graph drawing. One of the methods for 3D graph drawing is the well known Orthogonal Graph Drawing, which was intensively studied by A. Papakostas and Tollis. The 3D Orthogonal Graph Drawing algorithms were given in [64]. In that paper, the first algorithm is restricted to graph of maximum degree six and allows for at most three bends per edge. Though the second algorithm can deal with arbitrary degree of a graph, it seems more complicated to represent vertices with solid 3D boxes whose surface is proportional to their degree. To quickly display a mess graph with arbitrary degree in 3D, a new algorithm called *Incremental Projection Algorithm* is developed. With the *Incremental Projection algorithm*, any mess graph can be displayed in 3D space such that there is no edge crossing or overlap. Furthermore the maximum number of bends is only two. The *Incremental Projection algorithm* is explained in detail in the following sections.

4.2 Preliminaries for the Incremental Projection Algorithm

To display a graph in 3D space, a system of coordinates is needed. Typically, the orthogonal right hand 3D system is used, which is based on three axes x, y, z, so that each one of them is perpendicular to the other two axes. The xz-plane is defined by the x-axis and z-axis, the yz-plane is defined by the y-axis and z-axis, and the xy-plane is defined by the x-axis and

y-axis. Each one of these planes is called a *base* plane. The following geometric concepts and results are common knowledge.

A general plane π in the coordinate system is defined by a one-degree equation with three variables x, y, z as follows

$$\pi: \quad Ax + By + Cz + D = 0.$$

where A, B, C are coefficients and D is constant. The vector (A, B, C) is called the normal vector of the plane π. A point P(x_0, y_0, z_0) is on the plane π if and only if $Ax_0 + By_0 + Cz_0 + D = 0$.

If two planes $\pi_1: \ A_1x + B_1y + C_1z + D_1 = 0$ and $\pi_2: \ A_2x + B_2y + C_2z + D_2 = 0$ are parallel, then the normal vector (A_1, B_1, C_1) must be proportional to the normal vectors (A_2, B_2, C_2), vice versa.

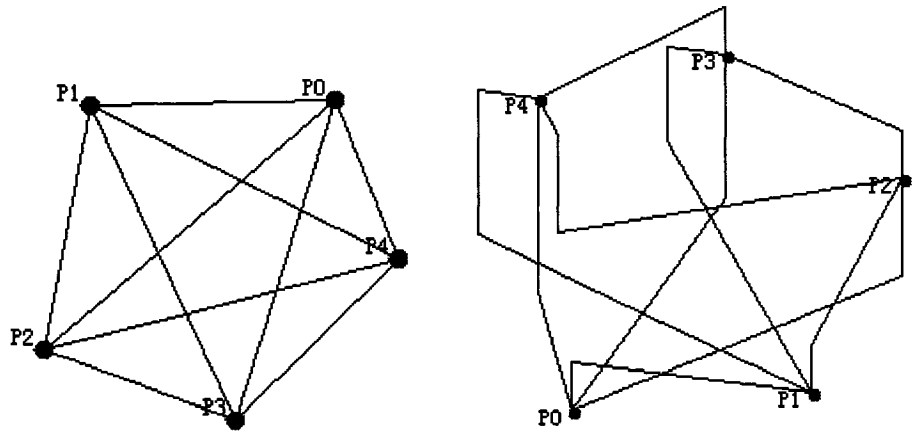
When the coefficient A vanishes, the plane $By + Cz + D = 0$ is perpendicular to yz-plane or parallel to the x-axis. Similarly, if the coefficient B vanishes, the plane $Ax + Cz + D = 0$ is perpendicular to the xz-plane or parallel to the y-axis. If the coefficient C vanishes, the plane $Ax + By + D = 0$ is perpendicular to the xy-plane or parallel to the z-axis. If the coefficients A and B vanish simultaneously, the plane $Cz + D = 0$ is parallel to the xy-plane and so on.

Since the *Incremental Projection* method will insert each vertex of a graph step by step into the current graph drawing, the first work to be done is to determine what directions should be used to connect the adjacent points. Obviously, every point on the plane $z = z_0$ (constant) to which it belongs has many possible directions around the point. The *top* direction is towards the positive part of the z-axis and the *bottom* direction points to the negative. The *right* direction and *left*

direction are parallel to the x-axis extending towards the positive or negative part respectively. The *front* direction and the *back* direction are parallel to the y-axis extending towards the positive or negative part respectively. If a direction is obtained by rotating the positive direction of the x-axis counterclockwise for an angle θ, the direction is called θ-direction. Usually, if there are n directions on the horizontal plane around a vertex, the n directions need to be arranged. In this case, the full angle 2π is divided by n to get the incremental angle $\theta = 2\pi / n$. Then the first direction corresponds to the angle $\theta * 0$ and the second direction corresponds to the angle $\theta * 1$. The k^{th} direction around a point on the horizontal plane corresponds to the angle $\theta * (k\text{-}1)$ (k=1,2,..., n). Suppose a vertex p is considered and there are k vertices p_i that are adjacent to the vertex p. The number k is the *local degree* of the vertex p. The free direction around p that is picked for connecting p_i is called p_i's connector. The number of the left connectors is called free degree of the vertex p.

Further work involves finding a route connecting two adjacent vertices p_j and p_i. This is called routing edge (p_j , p_i). The *Incremental Projection Algorithm* employs three basic routings:

Without loss of generality, suppose that the vertex p_j is above the vertex p_i and the vertex p_j , p_i is on the plane $z = z_j$ and the plane $z = z_i$, respectively. Take the non-planar graph K_5 and its 3D representation with the algorithm as an example:



(a) (b)

Figure 4.1 2D graph K_5 (a) and its 3D *Increment Projection* Drawing (b)

Route1: If the vertex p_i has free *top* direction, then project p_i to the plane $z=z_j$ orthogonally to get the foot q_1. Now p_j and q_1 are on the same plane $z=z_j$. The route from the vertex p_j to the vertex p_i is formed by two line segments (p_j, q_1) and (q_1, p_i). The routing is denoted by *Route1*(p_j, p_i) which has just one bend. For example, the *Route1*(p_1, p_0) in Figure4.1 (b) is the route connecting p_1 to p_0.

Route2: If the vertex p_i has no free *top* direction and the vertex p_j has free *bottom* direction, then project the vertex p_j to the plane $z=z_i$ orthogonally to get the foot q_1. The route from p_j to the vertex p_i is formed by connecting the vertex p_j to the foot q_1 and connecting the foot q_1 to the vertex p_i. The routing is denoted by *Route2*(p_j, p_i). In *Route2*, there is just one bend as well. An example of *Route2* can be found in Figure 4.1(b) where the route connecting p_2 to p_0 forms a *Route2 type* path.

Route3: If the vertex p_i has no free *top* direction and the vertex p_j has no free *bottom* direction, the route is formed by three line segments as follows:

First, choose one of the connectors of the vertex p_j, say v_l, then orthogonally project p_j along direction v_l to the plane π_l with normal vector v_l and fixed distance r_0 from p_j. If denote the foot as q_1, the first line segment is obtained by connecting p_j to q_1. Note that the plane π_l is perpendicular to both planes $z=z_i$ and $z=z_j$.

Second, orthogonally project q_1 to the intersection line formed by the plane π_l and the plane $z=z_i$ to get the foot q_2. Then the second line segment is formed by connecting the foot q_1 to foot q_2.

Thirdly, since the vertex p_i and the foot q_2 are both on the plane $z=z_j$, the third line segment is formed by connecting the foot q_2 to the vertex p_i. Then the Route3 is formed by three line segments (p_j, q_1), (q_1, q_2) and (q_2 q_1).The routing is denoted by *Route3*(p_j, p_i). In *Route3*, there are just two bends. In Figure 4.1(b), the route connecting p_3 to p_1 is an example of *Route3*.

Before the *Incremental Projection algorithm* is presented, the interactive graph drawing terminologies are briefly repeated here. The *current drawing* is the drawing before the insertion of a new vertex; the number of vertices of the current drawing that are going to be connected with a vertex p through new edges is p 's *local degree*. Those vertices are called adjacent vertices of p. A plane is called the *topmost plane* if it is parallel to the xy-plane and located one unit above the point of the current drawing with the highest z-coordinate.

4.3 The process oriented view of the Incremental Projection Algorithm

The process oriented view of the *Incremental Projection Algorithm* is shown below:

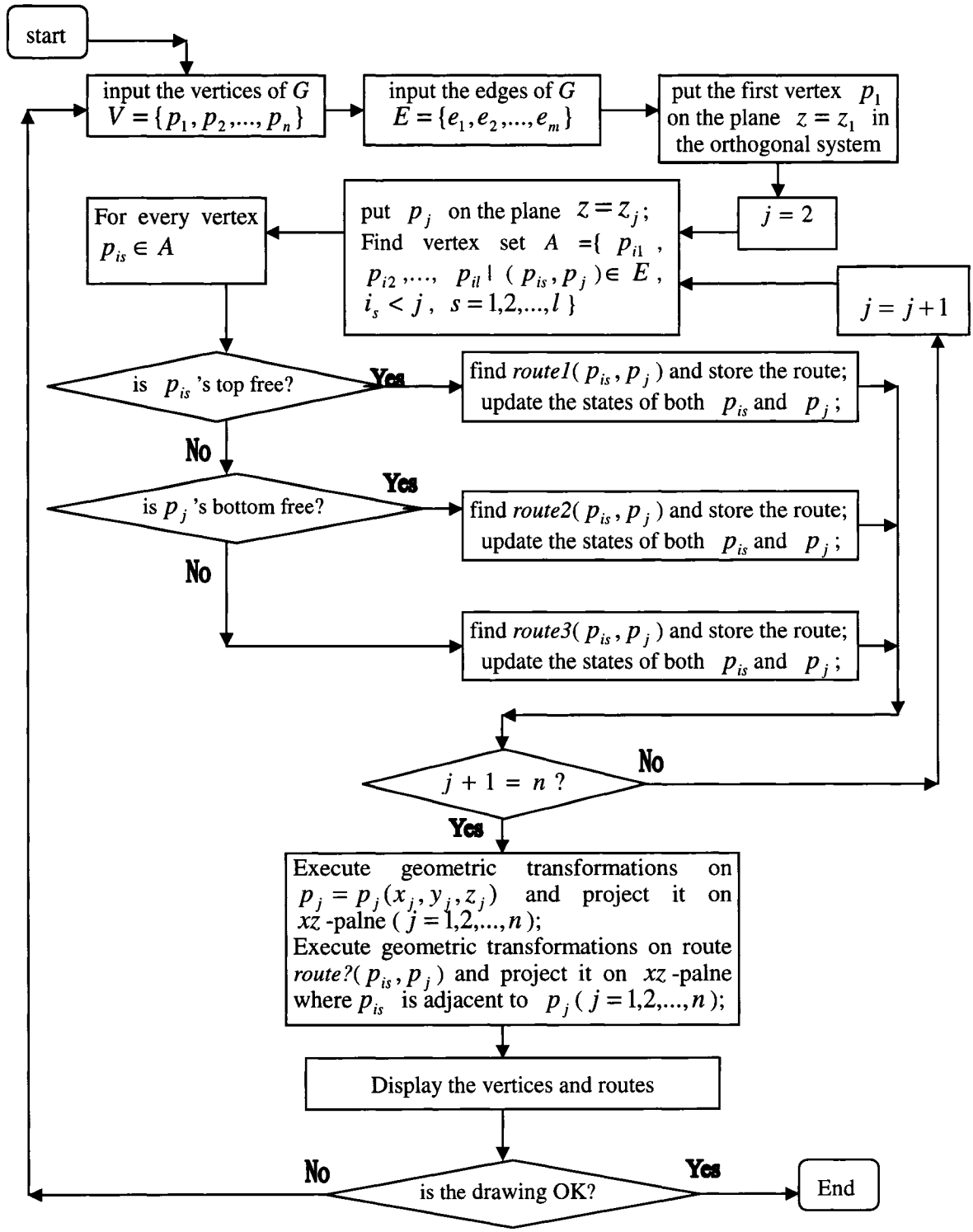


Figure 4.2 The process oriented view of the *Incremental Projection Algorithm*

4.4 The Incremental projection Algorithm

In this section, the *Incremental Projection algorithm* for producing 3D drawing of graphs with any degree is presented. It has the following properties:

1) It can draw any graph without limitation on the vertex degree;

2) There is no edge crossing and overlap;

3) At most 2 bends of the edges;

4) It keeps the relative positions of the original nodes on the *xy*-plane;

5) One can draw the graph incrementally (i.e. put nodes and links one by one on a current drawing);

6) It has linear time complexity.

The *Incremental Projection Algorithm* is given below:

Algorithm : Incremenal Projection

Input: A graph $G=(V, E)$ where V is the set of vertices and E is the set of edges of the graph G.

Output: A 3D display of the graph G with no edge crossings.

1. Number the nodes of the graph G, $V=\{ p_1, p_2,..., p_n \}$ where $p_i = p_i(x_i, y_i)$, $(i=1,2,...,n)$. Let d_i and d_i^f denote the *local degree* and the *free degree* of the vertex p_i $(i=1,2,...,n)$ respectively. In the beginning, let $d_i^f = d_i$ $(i=1,2,...,n)$.

2. Put the first vertex p_1 on the plane $z = z_1$ in three dimensional space such that $p_1 = p_1(x_1, y_1, z_1)$.

3. For $j=2$ to n then put the vertex p_j on the plane $z = z_j$ such that $z_j - z_{j-1} = c$

(*constant*). Set $p_j = p_j(x_j, y_j, z_j)$.

For $i<j$, if p_j is adjacent to p_i then

1) If $d_j^f = d_j$:

(A) If the top direction of the point p_i is free, then project the p_i onto the plane $z = z_j$ to get a point $q=(x_i, y_i, z_j)$. Store the links (p_j, q) and (q, p_i). Let $d_j^f = d_j^f$ -*1*. (Route1)

(B) If the top direction of the point p_i is not free, then project the p_j onto the plane $z = z_i$ to get a point $q=(x_j, y_j, z_i)$. Store the links (p_j, q) and (q, p_i). Let $d_j^f = d_j^f$ -*1*. (Route2)

2) If $d_j^f < d_j$:

(A) If the top direction of the point p_i is free, then project the p_i onto the plane $z = z_j$ to get a point $q=(x_i, y_i, z_j)$. Store the links (p_j, q) and (q, p_i). Let $d_j^f = d_j^f$ -*1*. (Route1)

(B) If the top direction of the point p_i is not free and the bottom direction of the point p_j is free, then project the p_j onto the plane $z = z_i$ to get a point $q=(x_j, y_j, z_i)$. Store the links (p_j, q) and (q, p_i). Let $d_j^f = d_j^f$ -*1*. (Route2)

(C) If the top direction of the point p_i and the bottom direction of the point p_j are not free, then build d_j -1 connectors centered at p_j such that the consecutive two directions of the connecters form fixed angle $\theta = 2\pi/(d_j - 1)$. Then pick up one of the direction on the plane $z = z_j$, say v_k, and project the vertex p_j onto the plane with

normal vector paralleling the direction v_k and with fixed distance r_0 to get the point $q_1=(x_j+r_0\cos(\theta), y_j+r_0\sin(\theta), z_j)$. Furthermore, project the point q_1 onto the plane $z=z_i$ to get another point $q_2=(x_j+r_0\cos(\theta), y_j+r_0\sin(\theta), z_i)$. Store the three edges (p_j, q_1), (q_1, q_2) and (q_2, p_i).(Route3)

4. Project all vertices and all the edges produced from step 3 above to a projection plane to get the graph drawing.

4.5 The correctness of the Incremental Projection Algorithm

At first, it needs to be shown that the *Incremental Projection algorithm* gives a 3D drawing of the given graph *G*. Suppose *G*=(*V*, *E*), where $V=\{p_1, p_2,\ldots, p_n\}$ and $E=\{(p_i, p_j) \mid i, j \in \{1,2,\ldots,n\}\}$, the set of edges of the graph *G*. Since the algorithm places the vertices into 3D space one by one from first vertex p_1 to the last vertex p_n, the vertices in *V* are all exhausted.

As to the edges in *E*, it needs to be shown that every edge in *E* is mapped to a route in 3D space and connects just two vertices as it is in the original graph *G*. If G just has two vertices and one edge, obviously the *Incremental Projection Algorithm* can produce its 3D drawing. Now assume that the current drawing of the graph *G* placed k vertices already. The set of the vertices *V* can be portioned into two subsets $V_1=\{p_1, p_2,\ldots, p_k\}$ and $V_2=\{p_{k+1}, p_2,\ldots, p_n\}$. Furthermore assume that for any edge (p_i, p_j) in *E*, if p_i and p_j both belong to V_1, the route connecting the vertex p_i to p_j is in the current drawing. Take one edge from the set of the edges *E*, say (p_i, p_j) and $i<j$, then one of the following three cases must happen:

(1) p_i and p_j both belong to V_1;

(2) p_i belongs to V_1 and p_j belongs to V_2;

(3) p_i and p_j both belong to V_2.

If (1) is the case, it is assumed that the route connecting the vertex p_i to p_j exists.

If (2) is true, then the index $k < j$. It is obvious that after applying the *Incremental Projection Algorithm* $j-k$ iterations, the vertex p_j must belong to the subset V_1, which implies that the edge (p_i, p_j) is definitely mapped to 3D space after $j-k$ more iterations.

If the case (3) happens, then $k < i$. After $i-k$ more iterations, the *Incremental Projection Algorithm* can guarantee that the vertex p_i belongs to subset V_1. It converts the case (3) to the case (2). Therefore the *Incremental Projection Algorithm* really produces a 3D drawing for the graph *G*.

Secondly, it needs to be verified that, for any pair of routes, there is no crossing in 3D space. To this end, pick a pair of routes arbitrarily from the *Incremental Projection* 3D drawing, for instance, *route*(p_i, p_j) and *route*(p_k, p_l). One case is that the vertices p_i, p_j, p_k, p_l are distinct. Notice that in the construction of the routes, every route can be separated into two or three line segments. One or two line segments of a route are on the horizontal planes and one line segment is on a plane that is perpendicular to xy-plane. Therefore, the horizontal segments of *route*(p_i, p_j) and *route*(p_k, p_l) belong to different parallel planes. These segments never cross. On the other hand, the two vertical line segments, one belonging to *route*(p_i, p_j) and other one belonging to *route*(p_k, p_l), are parallel. They never meet. Another case is that two ends of the routes are identical. Without loss of generality, suppose $p_j = p_l$. In this case, if two route are

type *route1*, the line segments on the plane $z=z_j$ only have common end p_j. The other two line segments are vertical line segment passing different vertices. They never meet.

If one route is type *route1* and the other is type *route2*, the two vertical line segments are different and the other two horizontal line segments belong to two parallel planes respectively. So the two routes never cross either.

If one route is type *route1* and the other is type *route3*, the line segments on the plane $z=z_j$ meets only at the end p_j. The other horizontal segment is on the plane $z=z_i$. The horizontal segments in the routes never cross. The vertical line segments are parallel and pass through different vertices. Therefore the vertical line segments never meet.

If one route is type *route2* and the other is type *route3*, the three horizontal line segments belong to three different parallel planes respectively. Two vertical line segments pass different points. The result is the same.

If both routes are type *route3*, the two horizontal line segments on the plane $z=z_j$ only meet at the end p_j. The other two horizontal line segments are on two different parallel planes respectively. The two vertical line segments pass different points. The result is the same.

The case that both routes are type *route2* is prevented by the *Incremental Projection Algorithm*, in which case the overlap of segments never occurs.

A fact repeated here is that the maximum number of bends with the *Incremental Projection Algorithm* is two, not three. This satisfies the aesthetic criteria of minimizing bends.

Finally, it needs to be shown that the *Incremental Projection Algorithm* can be used to draw a graph with any vertex degree in 3D space. Suppose a vertex p in G has degree d. As shown

before, usually the direction at the vertex p can be portioned into the *top* direction, the *bottom* direction and d-2 horizontal directions. For the d-2 horizontal directions, the incremental angle θ can be set to $\theta = 2\pi/(d\text{-}2)$. In this way, the d-2 horizontal directions are arranged evenly. Every direction can be a connector that links an adjacent vertex of the vertex p. Since the number d is a finite integer, the incremental angle θ is not zero. So, the connectors around the vertex p never overlap. The routes of type *route3* are available at any time. The *Incremental Projection Algorithm* can manipulate any vertices and any edges until the graph G is displayed in 3D space.

The following theorem summaries the above results:

Theorem 4.1 Given an n-vertex connected graph G of any degree, the *Incremental Projection Algorithm* produces a 3D drawing of G, so that each edge has at most two bends and no two edges cross.

Figure 4.3 is an example of the 3D drawing for graph K_8 with the *Incremental Projection Algorithm*:

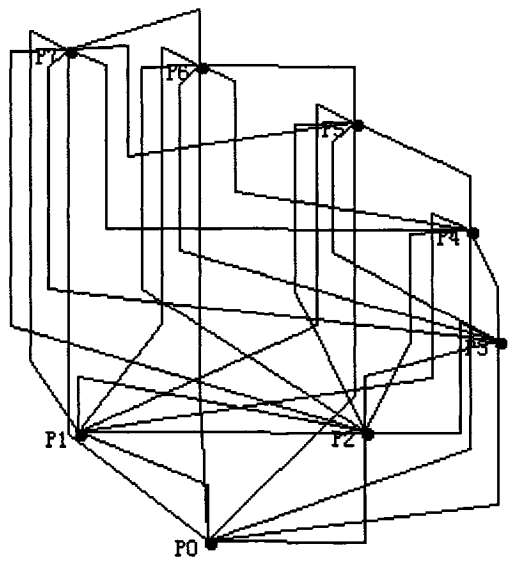


Figure 4.3 3D drawing of graph K_8 by the *Incremental Projection Algorithm*

4.6 Architecture of the Incremental Projection Package

In order to show how the *Incremental Projection Algorithm* works, a package is needed. The architecture of the package consists of the following modules:

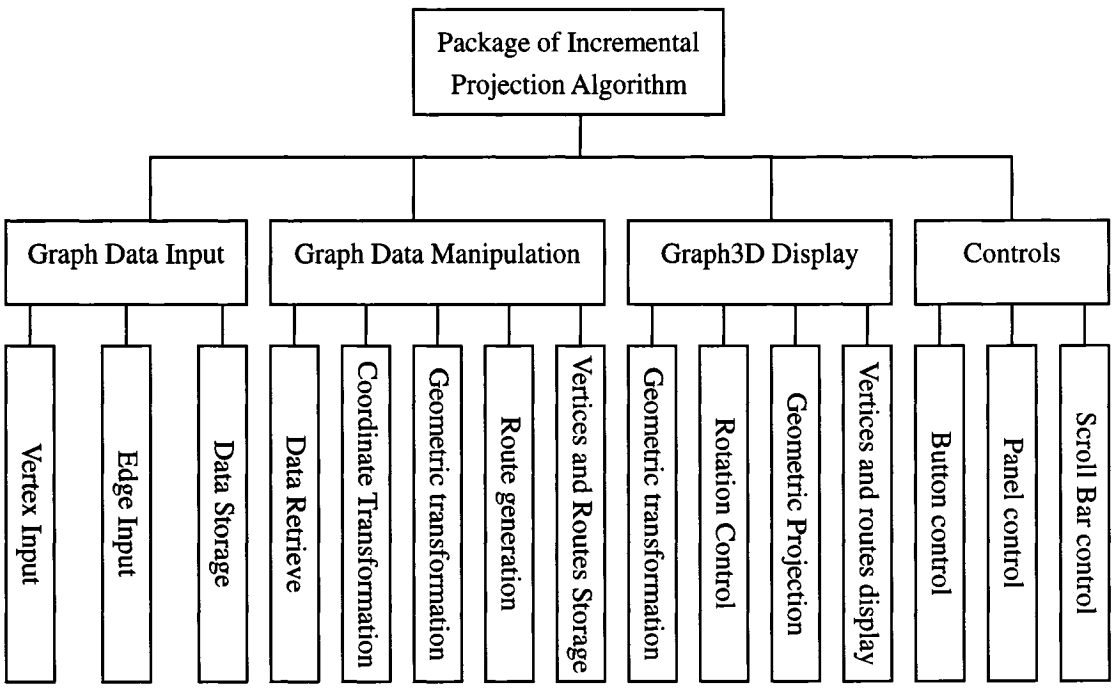


Figure 4.4 The architecture of the Package for the algorithm

The hierarchical module diagram above shows the components of the Package and their interrelations. The structure and functions of the modules are explained as follows:

4.6.1. Graph Data Input Module

This module provides data of a graph to the other modules. The easy way to describe a graph is to input the vertices and edges of the graph by using a mouse. When the mouse is clicked on screen, a vertex is displayed and its coordinate is stored. When the mouse is dragged from one vertex to another vertex, an edge (link) is drawn and stored. In this way, the positions of the

vertices and edges linking vertices of a graph can be determined conveniently. A data structure *ArrayList* is used to store the information of a graph, such as the position coordinates, the links of the adjacent vertices, the degree and the free degree and the tags that indicate free top or bottom direction of a vertex. The *ArrayList* can support many ways of data manipulation for example, inserting, deleting and locating an element of the *ArrayList*. The *ArrayList* can be used to store edges as well.

4.6.2. Graph Data Manipulation module

This module plays an important role in the Package. In the module, the *Incremental Projection Algorithm* is implemented. One of the functions of the module is to retrieve data from the *ArrayList*. After the data is obtained, it is then manipulated by the *Incremental Projection Algorithm* to determine the position coordinates of the routes. These position coordinates are stored into another *ArrayList*. The data manipulation module usually transfers the device coordinates (left hand system and downward) to world coordinates (right hand, upward and the origin centered) when the vertex and edge information is stored. All geometric transformations are done in the world coordinate system. Before the graph is displayed, the position data of the vertices and the routes need to be converted to device coordinates.

4.6.3. Graph 3D Display Module

This module realizes the 3D display of a graph. After the preparation work is done by previous modules, this module makes use of the provided data to display the image of the 3D drawing of a graph. To get the better 3D view, the graph usually needs to be transformed, such as

to do rotations for some angles along the z-axis or the x-axis. There is a sub-module to control the rotations. The rotations can be performed along the z-axis (horizontal rotation) or x-axis (vertical rotation) respectively. Finally the graph is projected to a coordinate plane, for instance xz-plane, to get the image of the 3D display of the graph.

4.6.4. Interface Control Module

The interface form of the package consists of buttons, panels, and scroll bars. They are used to shift the operations for the objects on screen.

(1) The "DrawPoint" button is a switch for input vertices on screen with the use of a mouse. If it is clocked, the switch is on.

(2) The "DrawEdge" button is another switch for edges input by mouse. If it is clicked, the switch is on and edges can be drawn by dragging the mouse.

(3) The "Draw3D " button is the third switch for triggering the 3D drawing display. If it is clicked, a graph drawing will be shown on the panel.

(4) The "Return2D" button is the fourth switch for return to the previous 2D picture.

(5) The "Clear" button is for clearing the screen and releasing the memory occupied by objects.

(6) The "Exit" button closes the form.

(7) The "H-Rotation" scroll bar is for rotating the 3D drawing of a graph horizontally. The rotation changes from $0°$ to $360°$.

(8) The "V-Rotation" scroll bar is for rotating the 3D drawing of a graph vertically. The

rotation changes from 0° to 360° as well.

(9) The panel1 is the first panel to be used to input the vertices and edges of a graph and display the 2D graph.

(10) The panel2 is the second panel to be used to display the 3D drawing the graph inputted. When panel1 is visible panel2 is invisible and vice versa.

4.7 The Flow Chart of the Package for the Incremental Projection

The flow chart of the package describes the process of data input, data manipulation and drawing output.

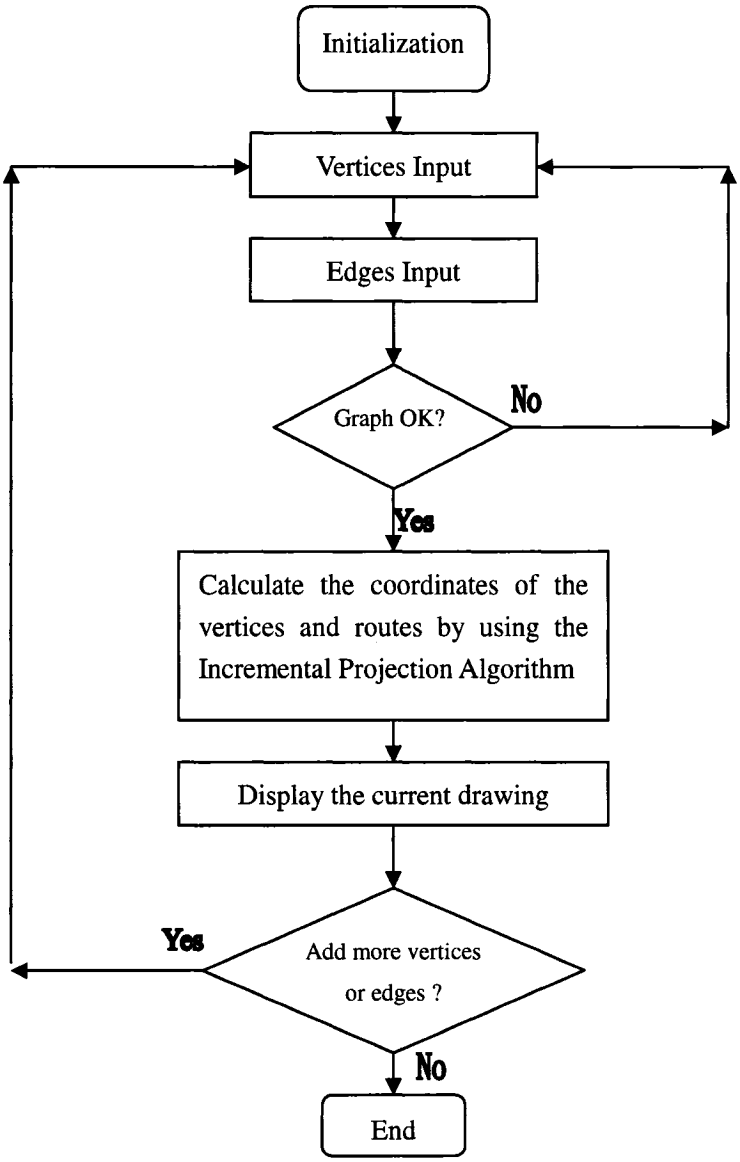


Figure 4.5 Flow Chart of the Package for the *Incremental Projection Algorithm*

The package for the Incremental Projection Algorithm has been implemented. The testing for the algorithm will be addressed in Chapter 6.

4.8 Depth-Height Buffer Algorithm

The *Depth-Height Buffer Algorithm* is a new method developed for 3D network visualization. It's actually an efficient algorithm for hidden objects elimination in special layered networks, in which the 3D objects are grouped by same shape and different group are located on different layer.

One of the difficult tasks in 3D network visualization is the hidden objects elimination. Usually, the complicated hidden line eliminating and surface eliminating techniques are used to cut out the hidden parts in a 3D drawing of a group of geometric objects. The process is very time consuming for general 3D object display. But if the 3D objects have same shape, or the 3D objects are grouped by same shape and different group are located on different layer, the hidden object elimination work can be done efficiently by employing the algorithm called *Depth-Height Buffer Algorithm.*

The *Depth-Height Buffer Algorithm* (*zy-buffer* algorithm) is the extension of the z-buffer algorithm. The *Depth-Height Buffer Algorithm* deals with objects instead of pixels. Take the left hand orthogonal coordinate system as the world coordinate system and the *xy*-plane as the projection plane. Suppose there are N objects that need to be displayed on the screen. If the objects are displayed randomly on the screen, some objects with smaller z value would be blocked by some other objects with larger z value, so that the result picture would be very messy. To solve the problem, the *Depth-Height Buffer Algorithm* is developed.

The outline of the algorithm is shown here. At first, take a characteristic point on each object. Usually, take a point with the smallest z value or the centre point on an object as the characteristic

point. These points form a set $\Sigma = \{ p_i(x, y, z) | \ p_i(x, y, z)$ is the point with the smallest z value of the i^{th} object, $i = 1$ to $N\}$. Then sort the points in the set according to their y-coordinates to get an ordered point set $\Sigma_1 = \{ p_k(x, y, z) | \ p_i(x_i, y_i, z_i) \ \pi \ p_j(x_j, y_j, z_j)$ if $y_i \leq y_j \}$. It seems that we should display the objects according to the order of the points in Σ_1. But there exist objects that they have characteristic points with the same y-coordinate value. In this case, we need to sort the z-coordinates of the points that have same y-coordinate value. After we sort the z-coordinates of the points that have the same y-coordinate value, then display those objects one by one from the objects with smaller y-coordinate value and larger z-coordinate value to the objects with larger y-coordinate value and smaller z-coordinate value. The resulting picture will be more real. So we need to re-order the points in Σ_1.

4.8.1 The process view of the *Depth-Height Buffer Algorithm*

The following diagram shows the process view of the *Depth-Height Buffer Algorithm*

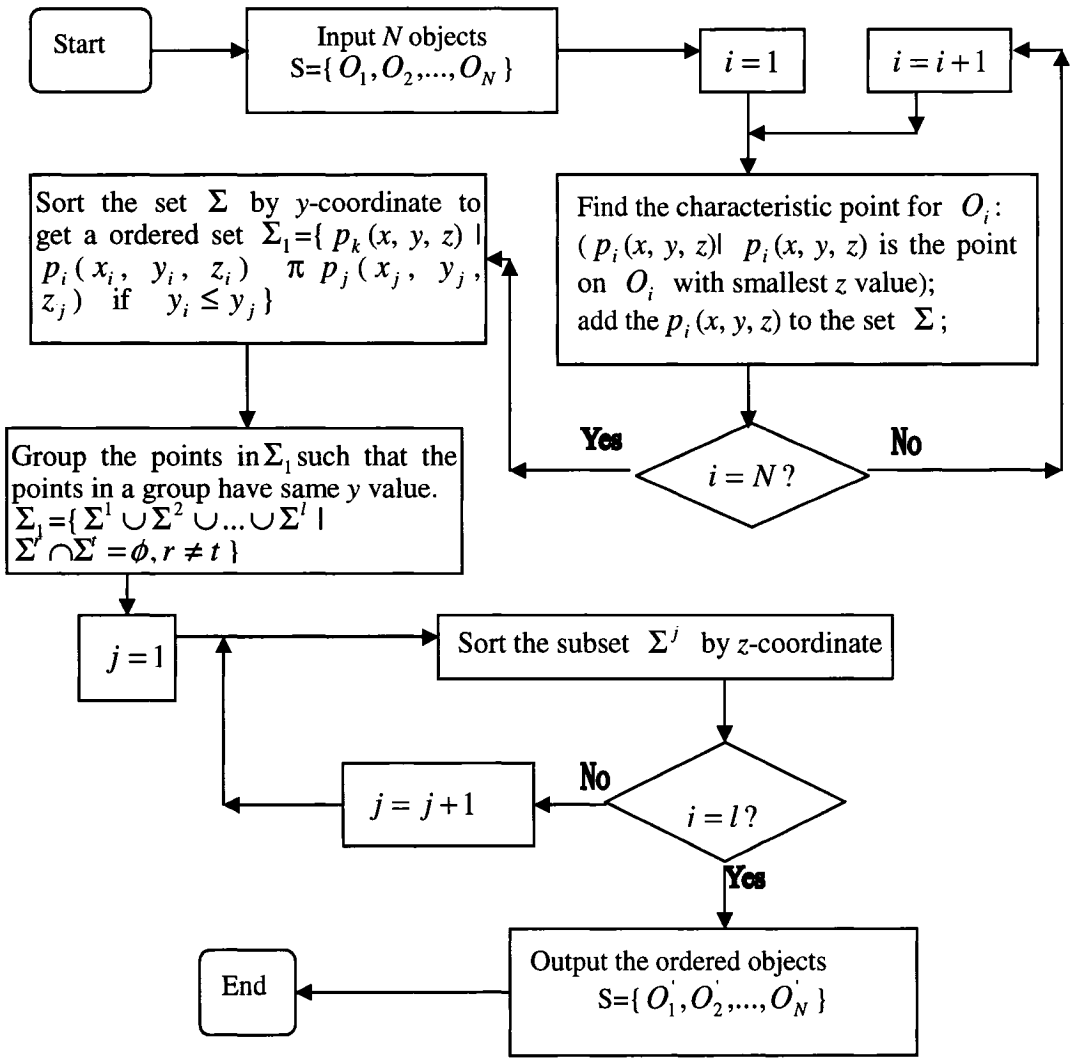


Figure 4.6 The process view of the *Depth-Height Buffer Algorithm*

4.8.2 The *Depth-Height Buffer Algorithm*

The *Depth-Height Buffer Algorithm* is summarized below:

Depth-Height Buffer Algorithm

***Input*:** A network with N objects that can be grouped according to their shape, such that the objects in same group will be displayed on same plane.

***Output*:** N ordered objects, such that, if display the objects according to the order obtained, the hidden surfaces among those objects will be eliminated.

Step1. Number the objects in the network and construct a matrix ***CM*** (Characteristic Matrix) to record the number, the smallest z coordinate and the largest y coordinate for each object.

Step 2. Re-order the objects by sorting the y-coordinates of the objects and update the matrix ***CM*** so that the object with the smallest y-coordinate on the top of the matrix ***CM***, which could be displayed first.

Step 3. Check the matrix ***CM*** to find those objects with same y-coordinate. Group the objects by their y-coordinates. Suppose there are $\boldsymbol{k}$ groups divided. Denote their Characteristic Matrix by CM_1, CM_2,..., CM_k. For each sub-matrix CM_i, re-order the objects by sorting their z-coordinates so that the object with the largest z-coordinate is set on top of the sub-matrix CM_i (i=1,2,...,$\boldsymbol{k}$). The objects with the smallest z-coordinate will be displayed first.

Step 4. Put all sub-matrices together according to their y-coordinates in ascending order to form a new matrix CM'. Then the new Characteristic Matrix CM'

corresponds to the ordered objects. The objects in the network can be displayed according to the order in the matrix CM' to get a 3D drawing without hidden surfaces.

4.8.3 The correctness of the *Depth-Height Buffer Algorithm*

First, notice that the coordinate system adopted is left hand orthogonal system in which the xy-plane is the projection plane (right hand system, the x-axis points to the right and the y-axis is point to the top) and the z-axis points to inside of the screen. The larger a point's z-coordinate value is, the farther the point is away from the screen. Therefore if an object has a larger z-coordinate value, it is more likely to be blocked by other objects with smaller z-coordinate values. On the other hand, since the y-axis is point to the top, an object with a smaller y-coordinate value is more likely to be blocked by other objects with larger y-coordinate values. The algorithm deals with grouped objects. The objects in the same group have same geometric shape and are located at the same layer of a network. Obviously, the objects on the lower layer of a network will be blocked by the objects on the higher layer. Therefore, the algorithm displays the objects from lower layer objects to higher layer objects in a network. Furthermore, the objects on the same layer are displayed according to their descending z-coordinates value. In this way, the images of the objects with smaller z-coordinates value will not be blocked by those with larger z-coordinates value. So, the hidden surfaces and lines will be eliminated naturally. But the *Depth-Height Buffer Algorithm* is just suitable for this kind of multi-layer networks.

4.9 The Architecture of the Package for the Depth-Height Buffer Algorithm

A package is developed to show the functionality of the *Depth-Height Buffer Algorithm*. The architecture of the package for *Depth-Height Buffer Algorithm* is given below:

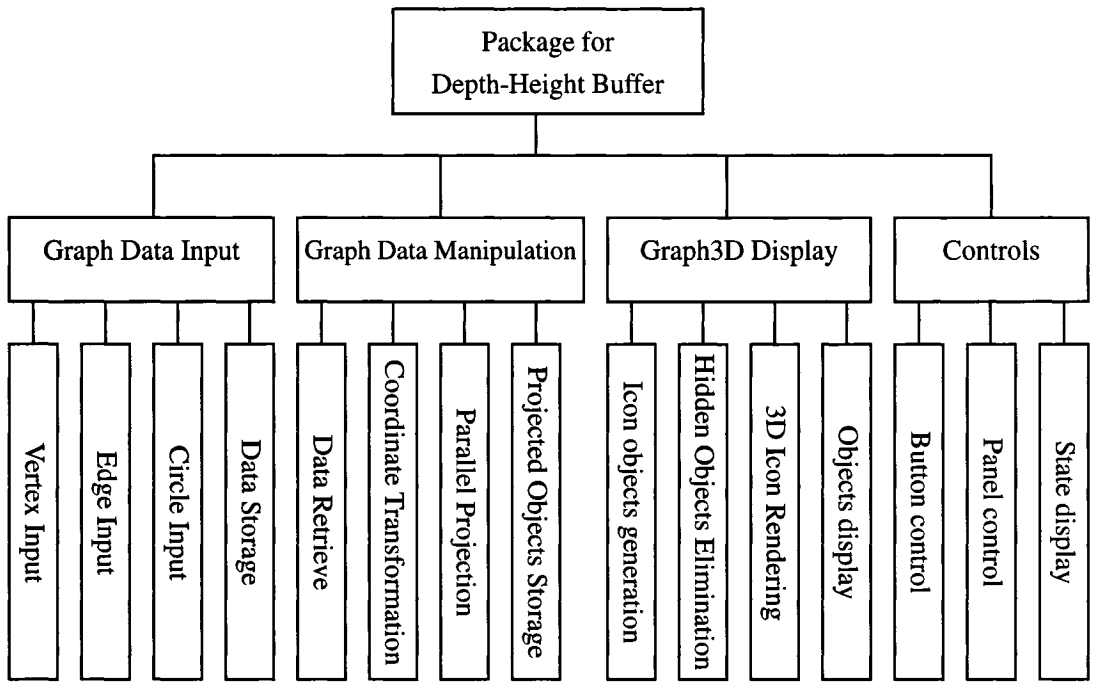


Figure 4.7 The architecture of the package for *Depth-Height Buffer Algorithm*

The main components and their interrelations of the package are expressed in the hierarchical module diagram above. The functionalities of the modules are defined as follows:

4.9.1 Graph Data Input Module

The topology of a graph, such as vertices and edges, is inputted in the module. By clicking the mouse on screen, the vertices of the graph are displayed and the position data are stored in vectors. The edges are drawn by dragging the mouse from one vertex to another. The edge data are stored in another vector. In the module, circles can be drawn as well. The circles are drawn by clicking three vertices consecutively on the screen.

4.9.2 Graph Data Manipulation module

This module can retrieve data from the vectors to get the position data of the vertices and edges as well as the circles. Then those objects are transformed by rotating the objects for an amount of angle around the x-axis. Then they are projected orthogonally to the xy-plane. The new positions of the projected objects are then stored to be displayed later.

4.9.3 Graph 3D Display Module

Since this package uses a special multi-layered network as an example which includes IP router icon, ATM icon, SONET icon, and DWDM switch icon, those 3D icons are generated in the module. Then these 3D icons are rendered by applying the Phong model [61] to calculate the value of the specular reflection. After these 3D icons are prepared, their positions are calculated according to the layer to which they belong. Furthermore, the *Depth-Height Buffer Algorithm* is applied to these objects to eliminate the hidden surfaces and display these objects.

4.9.4 Interface Control Module

The interface of the package consists of buttons, panels, and a state box. The buttons are used to shift the operations for the objects on screen. The panels are used to input the graph information and display the 3D drawing.

(1) The "Draw node" button is a switch for input vertices on screen with the use of a mouse. If it is clocked, the switch is on.

(2) The "Draw edge" button is another switch for edges input by mouse. If it is clicked,

the switch is on and edges can be drawn by dragging the mouse.

(3) The "Draw circle" button is the third switch for circle input. If it is clicked, the switch is on. Then the circles can be drawn by clicking three nodes successively.

(4) The "Draw3D " button is the fourth switch for triggering the 3D drawing display. If it is clicked once, the 3D drawing of the network will be shown on the panel. If it is clicked again, the 2D graph will return.

(5) The "Clear" button is for clearing the screen and releasing the memory occupied by objects.

(6) The "Exit" button is for closing the form.

4.9.5 The Design of the Package

Following software engineering principles, this package is developed according to the waterfall model. In the system design stage, the UML method is used to outline the system and adopt the Object-Oriented programming technology to develop the package.

Here is the diagram for the design structure:

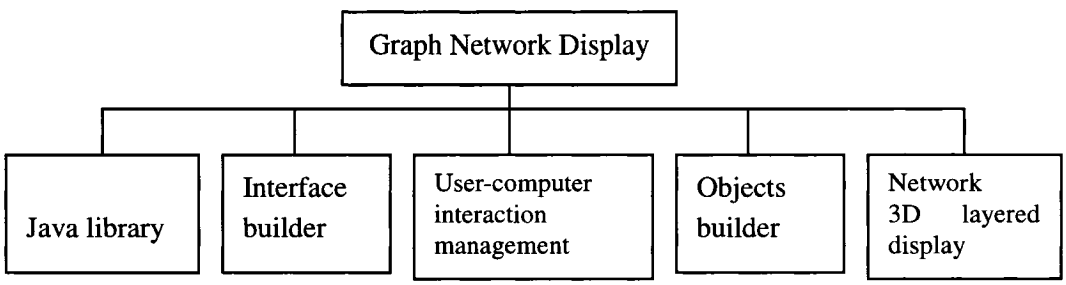


Figure 4.8 The design structure of the package

To display the components in 3D space, several computer graphics techniques are employed.

For example, the objects need to be projected the 3D objects into 2D screen. In the package, the parallel projection is employed. To display the layered information on one screen, the graphic transformation and clipping algorithms are used. To increase the 3D effect, the Phong models are also employed to render the objects.

The package is designed and implemented according to the Object-Oriented Programming approach. The first step to use Object-Oriented Programming is the description of the objects in the system. Three kinds of objects points, line segments, and circles are used in the package. To display a network with a 3D feature, we also need more objects, for instance a combined object for ATM, another combined object for SONET/SDH, etc.

A point has its x and y coordinates on the screen coordinate system. To display the points clearly, filled ovals are used. A circle (oval) has a centre, width, height and color. A link is a line segment. To describe a link, its end point, line thickness, line pattern and color are specified.

One of the combined objects called Atm is a cube with four arrows on its front facet. The size of the edge, the color, the angle of projection and the arrows are determined.

Another combined object is the IP router. The name of the object is Ipr. A solid cylinder is used to denote it. On the top of the cylinder there are also four arrows. The centre of the base circle, the height of the cylinder, the color, and the angle all need to be specified.

For Sonet icon, a short cylinder with a hexagon hat is designed. Finally a pyramid shape object is designed for the DWDM Icon.

The system flowchart can be drawn below:

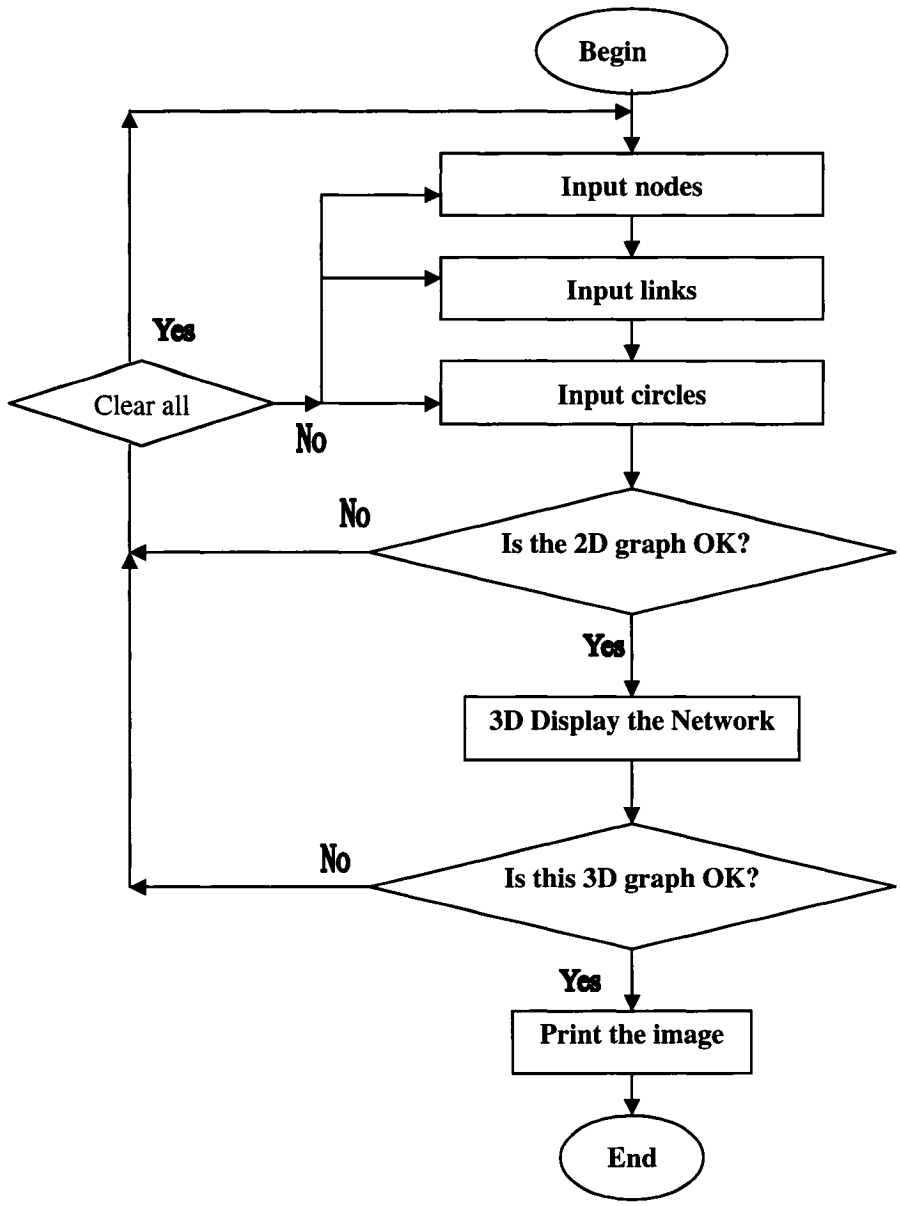


Figure 4.9 The flowchart of the software

The class diagram addresses the static structural aspects of a problem. Here is the class diagram for the package as follows:

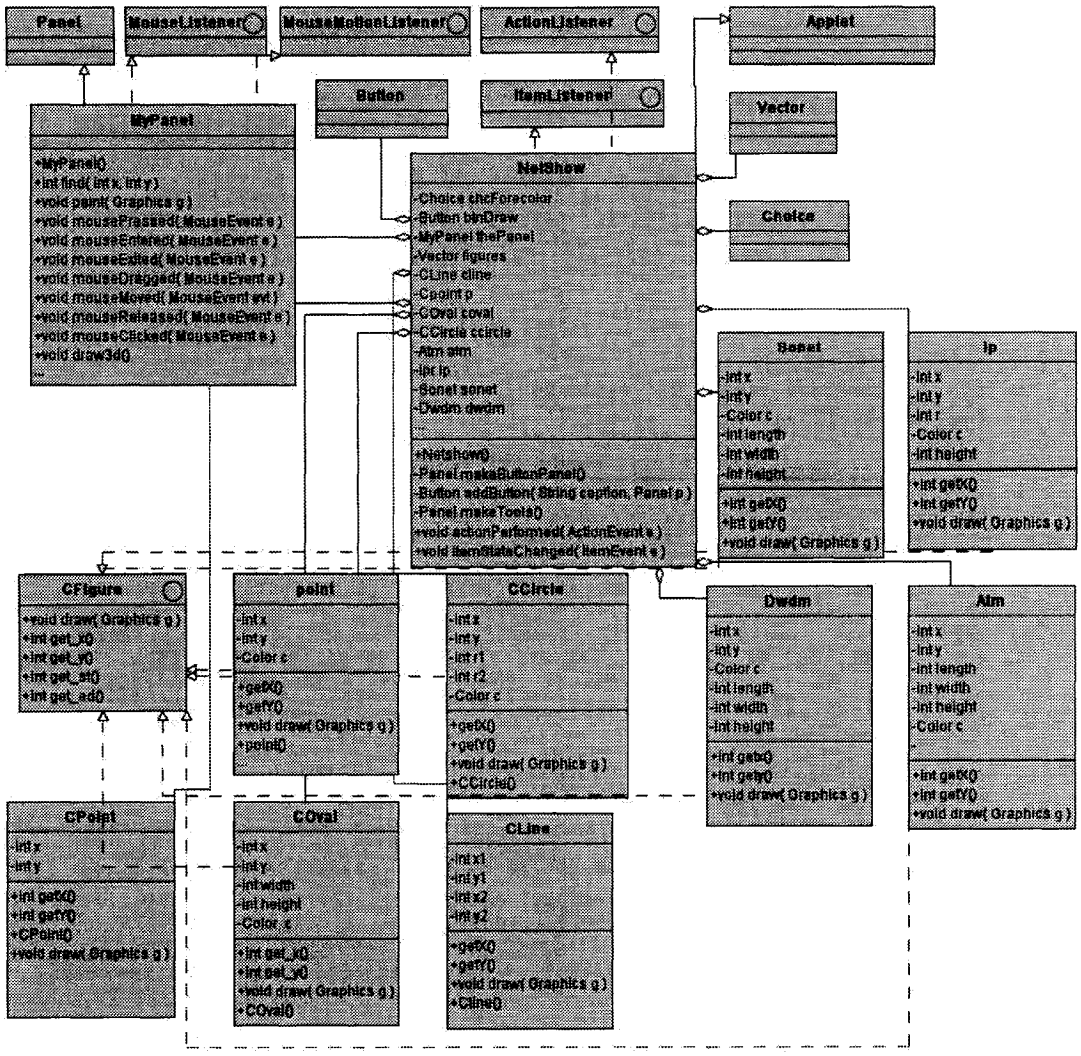


Figure 4.10 The class diagram for the package of *zy*-buffer algorithm

The class **NetShow** is derived from several classes, such as Java Applet, ItemListener, ActionListener. It has several components, for example, the MyPanel, the CFigure, Vector, Choice, Sonet, Ip, Atm, Dwdm, point, Ccircle, Cline, Cpoint, Buttion, etc. The function of the class **NetShow** is to bind various classes and their objects together and to perform some jobs. The class CFigure derived many sub-classes. The Sonet, Ip, Atm, Dwdm, point, Ccircle, Cline, Cpoint all are sub-class of the class CFigure. Class CFigure is an abstract class. Its sub-classes are models for various geometric objects. The class MyPanel is derived from class Panel,

MouseListener and MouseMotionListener, which forms the interface of the software and listen the actions from the mouse.

4.9.6 The display methods in the package for the multi-layered networks

Before display the 3D image of a multi-layered network, some transformations and projection are needed. There are many methods to produce 3D effect images [59,60]. To keep the sizes of the objects, the parallel projection method is adopted. Notice that the device coordinate system is a left-handed system with the origin at the northwest corner (up-left corner) and the positive direction points to the south. For any drawings, before displaying them on the screen, translation is needed to convert the world coordinates to corresponding device coordinate. Suppose a point P has 2D coordinates (x, y) in a world coordinate system centered at (x_0, y_0) and has device coordinates (x', y'). Then the translation equation for two coordinate systems is as follows:

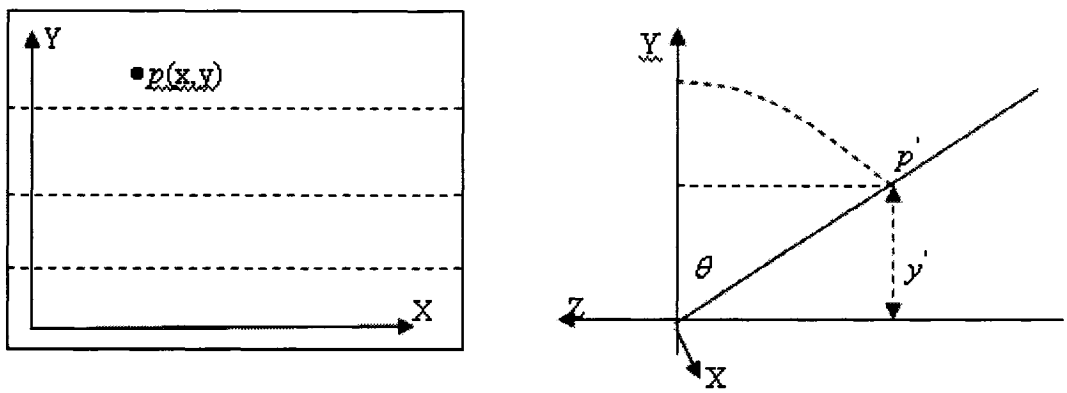
$$x' = x + x_0, \qquad y' = y_0 - y.$$

To get the displayed image, we adopt the parallel projection method, especially, the projection perpendicular to the screen. First we will do the rotation transformation about the x axis for curtain degree θ. Then project the objects orthographically to the x-y plane (the screen) to get the image coordinates for bottom objects. For the second layer, we just need to translate objects upward for a distance, say h_0, to get the coordinates for the nodes of SONET/SDH. And we left the bottom objects above for 2* h_0 to get the coordinates for the ATM nodes on the third tier. To get the top tier image, the bottom objects are translated along y axis for 3* h_0 above. This

projection process can be illustrated as follows:

In viewing coordinate system (right-handed), suppose a point $p(x,y,0)$ is on the screen as shown in Figure 4.10 (a) below. If the screen is rotated about the x axis clockwise, the point p will be transfered to $p'(x', y', z')$ where $x' = x$ and the height y' of the image p' is $y' = y\cos\theta$, referring to the figure 4.10 (b) below ($z' = -y\sin\theta$). Actually, the rotation transformation and the projection to x-y screen are shown below:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} == \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \quad z = 0.$$



(a) (b)

Figure 4.11 (a) A point before transformation. (b) The result image transformed

If the screen is divided horizontally into four equal belts, four images corresponding to the four network tiers can be put on the belts respectively. The coordinate difference between the nodes belonging to different tier which have the same projection on the bottom is just the y coordinates that are the multiple of the height h_0.

Since the communications network can be divided into four layers (IP, ATM, SONET/SDH

and DWDM), the network is displayed in four tiers. The first tier is on the bottom which is for DWDM network. The second tier is for SONET/SDH shown by the circles. The next tier is for ATM and links between them. The top tier is used to display the IP nodes and links.

Chapter 5

Graphic Algorithms and Computer Graphics

The packages for graph and network 3D display algorithms need the support of the technologies in graphics algorithm and computer graphics, which include geometric transformations, coordinate transformations, projection methods, hidden objects elimination, and object shading models. This chapter introduces those methods used in the 3D display algorithms and the packages.

5.1 Projective Approaches

Any 3D effect displays are 2D images on plane (or screen). At first, the objects need to be described by a three dimensional word coordinates. Secondly, the objects are transformed to certain position and angle in world coordinate system. Then certain projection is applied to the objects to map the objects onto the plane or screen in 2D plane coordinate system (Viewing coordinate system). Finally, the coordinates of the objects are transformed to a device coordinate system which can be displayed by a CRT or by a drawing machine. The following diagram shows

the process:

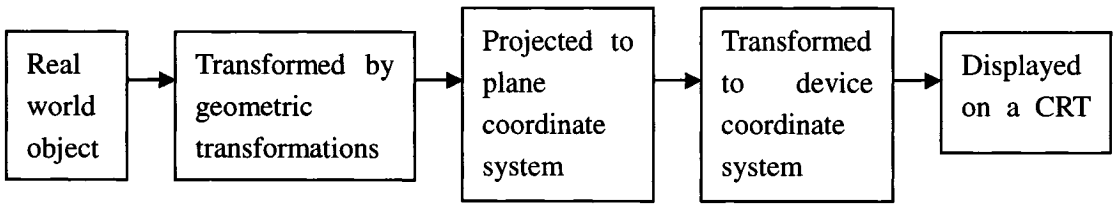


Figure 5.1 The process of computer graph display

In the real world, the objects are described by a right hand orthogonal system as follows:

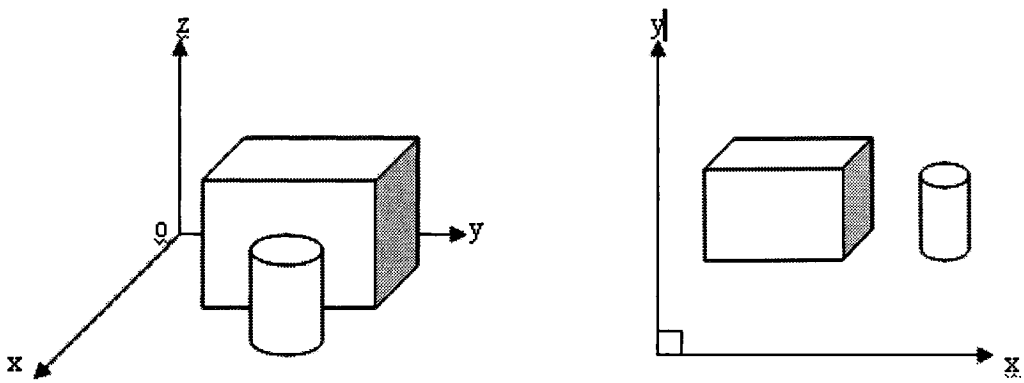


Figure 5.2 3D and 2D coordinate systems

Usually, the basic objects are points, lines, and ellipses (circles). These can be used to create more complex objects according to certain combinational rules.

A point in plane and in 3D space can be defined by a pair of integers (x, y) or by a triple (x, y, z) respectively. A link (line segment) can be described by two points $P_1(x_1, y_1, z_1)$ and $P_2(x_2, y_2, z_2)$. An ellipse (circle) can be define by a centre point C(x, y) and two radii r_1, r_2 (if $r_1 = r_2$, it is a circle). A three-dimensional object can be described by a combination of the basic objects. For example, a cube can be represented by eight points and twelve links. A cylinder can be drawn by two ellipse and two line segments.

An object is outlined by the characteristic points and the topology of the objects

(relationships between the points). Usually a matrix is used to hold the coordinates of the points as follows:

$$P=\begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ \cdots & \cdots & \cdots \\ x_n & y_n & z_n \end{pmatrix}$$

The topology of an object is also described by a matrix called the conjunction matrix C = $(c_{ij})_{nn}$, where c_{ij} =1 if p_i and p_j is connected by a link, otherwise c_{ij} =0.

A surface in an object can be denoted by points on it or by links by which it is formed.

For example, a cube with side $2a$ is described by following figure and matrices:

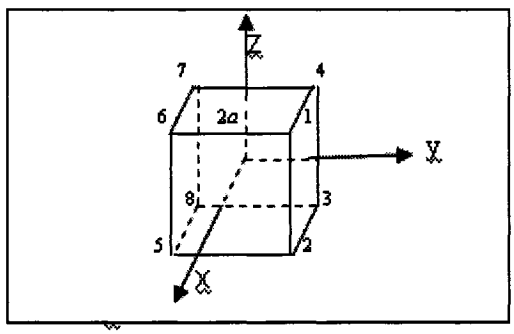


Figure 5.3 A cube in a Cartesian coordinate system

$$P=\begin{pmatrix} a & a & a \\ a & a & -a \\ -a & a & -a \\ -a & a & a \\ a & -a & -a \\ a & -a & a \\ -a & -a & a \\ -a & -a & -a \end{pmatrix},\quad C=\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix},\quad F=\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 8 & 7 & 4 \\ 5 & 8 & 7 & 6 \\ 2 & 1 & 6 & 5 \\ 1 & 4 & 7 & 6 \\ 2 & 5 & 8 & 3 \end{pmatrix}.$$

How can a 3D object be drawn on a 2D plane? The answer is that a 3D object can be drawn on a 2D plane by projections.

There are two basic methods for projecting three-dimensional objects onto a two-dimensional viewing surface. All points of the object can be projected to the surface along parallel lines, or the points can be projected along lines that converge to a position called the center of projection. The two methods, called parallel projection and perspective projection, respectively, are illustrated in the following figure :

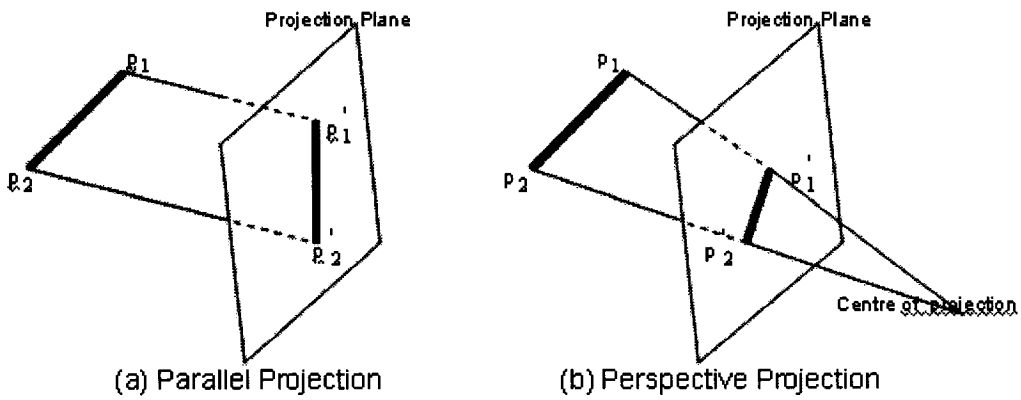


Figure 5.4 Parallel and perspective projections

In both cases, the intersection of a projection line with the viewing surface determines the coordinates of the projected point on this projection plane. For the present, the view projection plane is z=0 plane of a left-handed coordinate system, as shown below:

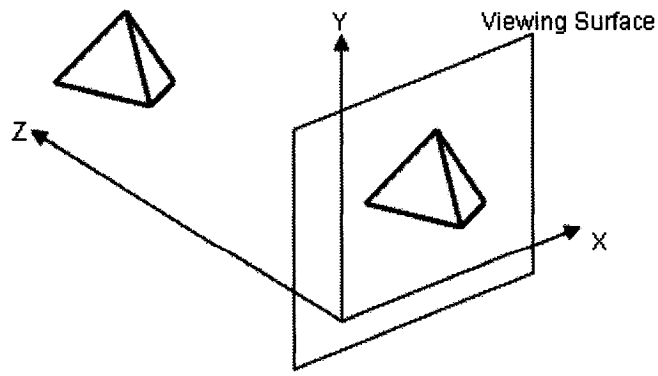


Figure 5.5 Left-handed coordinate system and the Viewing Surface

5.1.1 Parallel Projections

Views formed with parallel projection can be characterized according to the angle that the direction of projection makes with the projection plane. When the direction of projection is perpendicular to the projection plane, an orthographic projection is obtained. A projection that is not perpendicular to the plane is called an oblique projection. Orthographic projections are most often used to produce the front, side, and top views of an object. Orthographic projections can also be formed to produce images showing more than one face of an object. Such view is called axonometric orthographic projection.

The most commonly used axonometric projection is the isometric projection. An isometric projection is obtained by aligning the projection plane so that it intersects each coordinate axis in which the object is defined at the same distance from the origin. Transformation equations for performing an orthographic parallel projection are straightforward. For any point (x, y, z), the projection point (x_p, y_p, z_p) on the viewing surface is obtained as $x_p = x$, $y_p = y$, $z_p = 0$.

An oblique projection is obtained by projecting points along parallel lines that are not perpendicular to the projection plane. The oblique projection lines makes an angle α with the line on the projection plane that joins (x_p, y_p) and $(x, y,)$, where the point $(x, y,)$ is the orthographic projection for the space point (x, y, z) onto the projection plane. This line of length L is at an angle ϕ with the horizontal direction in the projection plane. The transformation equations can be expressed in terms of x, y, L and ϕ as follows:

$$x_p = x + L\cos\phi,$$

$$y_p = y + L\sin\phi.$$

A projection direction can be defined by selecting values for the angles α and ϕ. Common choices for angle ϕ are 30° and 45°, which display a combination view of the front, side and top of an object. Obviously $tan(\alpha)$=z/L. If let L_1 denotes the length of the projection line from (x_p, y_p) to (x, y,) when z=1, then $L=zL_1$ and the transformation equations can be re-expressed in terms of x, y, L_1 and ϕ as follows:

$$x_p = x + z(L_1 \cos\phi),$$

$$y_p = y + z(L_1 \sin\phi).$$

Therefore an orthographic projection is obtained when $L_1 = 0$ (when $\alpha = 90^\circ$).

Two commonly used angles in oblique projection are those for which $tan(\alpha)$=1 and $tan(\alpha)$=2. For the first case, $\alpha = 45^\circ$ and the views obtained are called cavalier projections. All lines perpendicular to the projection plane are projected with no change in length. When the projection angle is chosen such that $tan(\alpha)$=2, the resulting view is called a cabinet projection. This projection angle of approximately 63.4° causes lines perpendicular to the viewing surface to be projected at one-half their length. Cabinet projections appear more realistic than cavalier projections because of this reduction in the length of perpendiculars.

In this package, the cabinet projection will be employed to produce the 2D images of objects.

5.5.2 Perspective Projections

To display the objects according to some perspective rules (such as: the farther an object is,

the smaller the image looks) the perspective projection model can be employed. To obtain a perspective projection of a 3-dimensional object, the points are projected along the projection lines that meet at the centre of the projection. The centre of projection is on the negative z axis at a distance d behind the projection plane. Any position can be selected for the centre of projection, but choosing a position along the z axis simplifies the calculations in the transformation equations.

Transformation equations for a perspective projection from the parametric equations describing the projection line from point P to the centre of projection can be obtained as shown below:

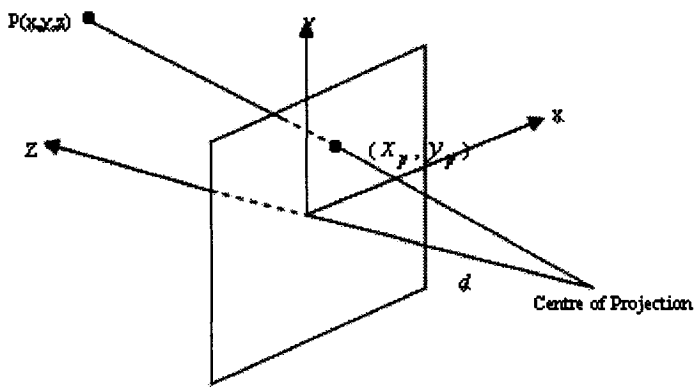


Figure 5.6 Perspective projection principle

The parametric form for this projection line is

$$x' = x - xu \quad , \quad y' = y - yu \, , \quad z' = z - (z + d)u.$$

Parameter u takes values from 0 to 1, and coordinates (x', y', z') represent any position along the projection line. When $u = 0$, yield point P at coordinates (x, y, z). At the other end of the line $u = 1$, the coordinate for the centre of projection $(0,0,-d)$, is obtained. To obtain the coordinates on the projection plane, $z' = 0$ is set and the parameter u is solved:

$$u = \frac{z}{z+d}.$$

This value for parameter u produces the intersection of projection line with the projection plane at $(x_p, y_p, 0)$. Finally, the following perspective transformation equations are obtained:

$$x_p = x(\frac{d}{z+d}) = x(\frac{1}{z/d+1}),$$

$$y_p = y(\frac{d}{z+d}) = y(\frac{1}{z/d+1}),$$

$$z_p = 0.$$

When a three-dimensional object is projected onto a plane using perspective transformation equations, any set of parallel lines in the object that are not parallel to the plane are projected into converging lines. Parallel lines that are parallel to the plane project as parallel lines. The point at which a set of projected parallel lines appears to converge is called the vanishing point. Each set of projected parallel lines will have a separate vanishing point.

5.2 Geometric Transformation

To obtain a better projection image of an object, suitable geometric transformations need to be applied to that object. For example, if a cube is set in a three-dimensional coordinate system such that the faces are parallel the coordinate axes, then the images of the orthographic projection of an object will show no 3D effect. In this case, the cube needs to be rotated along the x and/or y axes before applying the orthographic projection on the object.

There are several elementary 3D transformations that is summarized here:

1) Translation transformations

Usually, one wants to translate a picture into a different position on a graphics display. Let $P(x, y, z)$ be a point in 3D space and $P'(x', y', z')$ be the transformed point of P. The transformation equations in 3D space can be expressed as

$$x' = x + T_x, \quad y' = y + T_y, \quad z' = z + T_z.$$

T_x, T_y and T_z are translation constants. In matrix form, the translation can be re-written as follows:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

In the package, some translations are needed to lift the basic network pictures.

2) Scaling transformations

Scaling changes the size of a picture and involves three scale factors: S_x, S_y and S_z. The transformation equations are:

$$x' = S_x x, \quad y' = S_y y, \quad z' = S_z z.$$

The matrix form of the Scaling transformation can be expressed as :

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

The scaling in this fashion is most accurately called scaling about the origin. If a scale factor is negative, then there is also a reflection about a coordinate axis.

3) Rotation transformations

Rotations in three dimensions are common in graphics because an object or a camera is rotated in order to obtain different views. There is a much greater variety of rotation in three than in two dimensional space, since an axis must be specified, about which the rotation occurs, rather than a single point. A complex rotation can usually be decomposed into a combination of several basic rotations called elementary rotations about a coordinate axis.

The simplest rotation is a rotation about one of the coordinate axes. The basic rotations are listed as follows, in which the angle α represents the rotation angle in a rotation transformation of points about a coordinate.

- A x-rotation

$$x' = x, \quad y' = y\cos(\alpha) - z\sin(\alpha), \quad z' = y\sin(\alpha) + z\cos(\alpha),$$

In matrix form, the x-rotation is as follows:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

- A y-rotation

$$x' = x\cos(\alpha) + z\sin(\alpha), \quad y' = y, \quad z' = -x\sin(\alpha) + z\cos(\alpha),$$

In matrix form:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

- A z-rotation

$$x' = x\cos(\alpha) - y\sin(\alpha), \quad y' = x\sin(\alpha) + y\cos(\alpha), z' = z,$$

In matrix form:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

It can be proven that any general rotation transformation can be expressed as a combination of the basic rotations and any geometric transformation can be decomposed into the combination of several basic transformations (including translation, scaling and rotations).

5.3 Changing Coordinate System

There is another way to think about transformations. Changing coordinate system is a more natural approach when modeling a scene. Instead of viewing a transformation as producing a different point in a fixed coordinate system, think of it as producing a new coordinate system in which to represent points.

Suppose that a 3D coordinate frame #1 is drawn as shown in Figure5.7 below with origin O and axes $\boldsymbol{i}$, $\boldsymbol{j}$ and $\boldsymbol{k}$. Suppose a transformation $\boldsymbol{T}$ represented by the matrix M is specified. So $\boldsymbol{T}$ transforms frame #1 into coordinate frame #2, with new origin $O' = \boldsymbol{T}(O)$ and new axes $\boldsymbol{i}' = \boldsymbol{T}(\boldsymbol{i})$, $\boldsymbol{j}' = \boldsymbol{T}(\boldsymbol{j})$ and $\boldsymbol{k}' = \boldsymbol{T}(\boldsymbol{k})$.

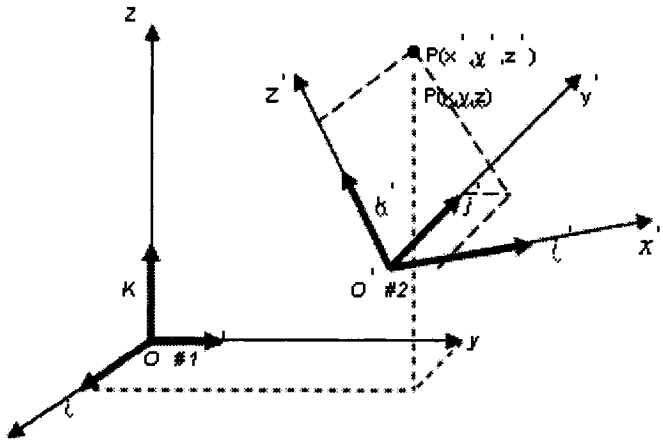


Figure 5.7 The coordinate system change

Suppose P is a point with representation $(x', y', z')^{tr}$ in the new system #2. What are the values of $(x, y, z)^{tr}$ in the representation of P in the original system #1? The answer is found just by multiplying $(x', y', z')^{tr}$ by the transformation matrix M:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \mathrm{M} \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

Changing of coordinates is a powerful tool to obtain a suitable viewing coordinate system that can best view objects in 3D space. With coordinate transformation methods the viewing transformation in the following section can be easily found.

5.4 Viewing Transformation

Generating a view of an object in three dimensions is similar to photographing the object. A photographer can walk around and take its picture from any angle, at various distance, and with

varying camera orientations. Whatever appears in the viewfinder is projected onto the flat film surface.

A user specifies a particular view for a scene by defining a view plane. The view plane is a surface upon which the view of an object is projected. Think of it as the film in a camera that has been positioned and oriented for a desired shot. The view plane is established by defining a view coordinate system as shown below:

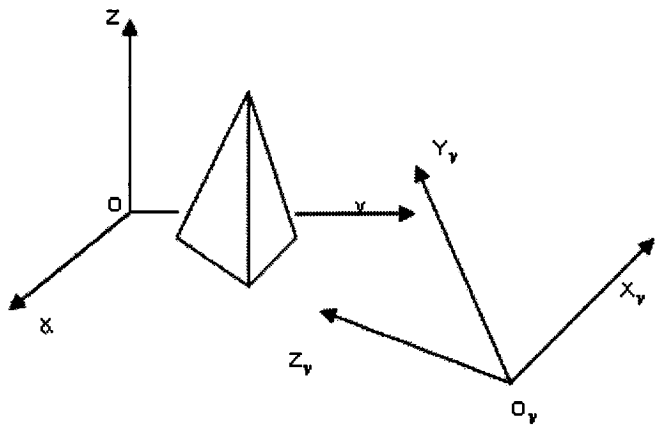


Figure 5.8 The viewing plane X_v, Y_v

To establish the viewing coordinates, a world coordinate position is picked to serve as the view point. This will be the origin of the view coordinate system. The rotation of the view plane is defined by specifying the view plane normal vector ***N***. This vector establishes the direction for the positive *z* axis of the viewing coordinate system. A vertical vector ***V*** is used to define the direction for the positive *y* axis. The view plane normal vector ***N*** can be established by specifying a coordinate position relative to the world coordinate origin. This defines the direction of the normal vector as the line from the origin to the specified coordinate position. Vector ***V*** can be specified in a similar manner. Sometimes a third vector ***U*** is used to indicate the x direction of

the viewing coordinate system. The ***NVU*** system should be consistent with the standard orientation of the x and y axes on a display device. Usually a left-handed system is used.

To generate a view from the user-specified point, positions defined relative to the world coordinate origin must be redefined relative to the viewing coordinate origin. That is, the coordinates from the world coordinate system to the viewing coordinate system is transformed. This transformation is accomplished with the sequence of translations and rotations that map the viewing system axes onto the world coordinate axes. The matrix representing this sequence of transformations can be obtained by concatenating the following transformation matrices:

1) Reflect relative to the xy plane, reversing the sign of each z coordinate. This changes the left-handed viewing coordinate system to a right-handed system.

2) Translate the view reference point to the origin of the world coordinate system.

3) Rotate about the word coordinate x axis to bring the viewing coordinate z axis into the xz plane of the world coordinate system.

4) Rotate about the world coordinate y axis until the z axis of both systems are aligned.

5) Rotate about the world coordinate z axis to align the viewing and world y-axis.

The viewing transformation can be used to produce the camera view effect to view an object.

5.5 Hidden Lines and Hidden Surface Elimination

While in 3D display, some links (edges), curves and surfaces of objects may be blocked by other objects or surfaces. If those hidden objects do not be eliminated, confusion with the mess graph will occur. When only the outline of an object is to be displayed, hidden-line algorithms are

used to remove the edges of objects that are obscured by surfaces closer to the view plane.

In the software package , the objects are cubes and cylinders. Links, curves and surfaces need to be hidden so that they look more concrete. For example, if the hidden objects are not eliminated, a cylinder and two cubes will appear as shown below:

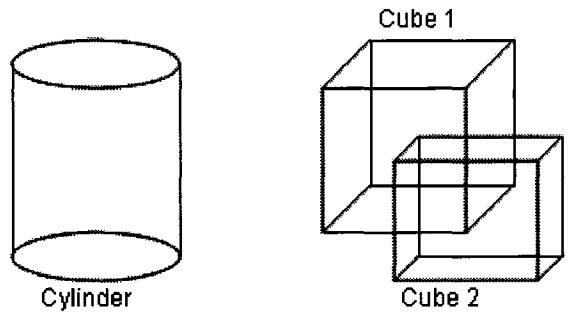


Figure 5.9 A cylinder and two cubes without eliminating the hidden objects

The figures above will lead to confusion. It is not clear which faces are close to us and which cube is blocked by another cube.

To eliminate the invisible surfaces and the hidden lines, first the invisible surfaces of the objects are eliminated. Then the hidden lines are eliminated.

If a viewing point is chosen, the visible faces are determined by calculating the inner products of the normal vectors and the viewing vectors. In Figure5.10, the visible surfaces can be determined according to the signs of the inner products:

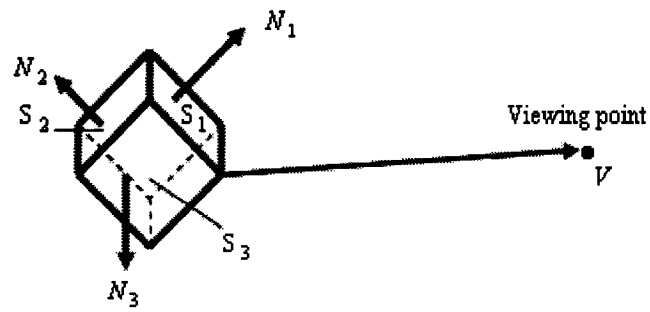


Figure 5.10 Normal vectors and viewing vector

Since the inner product $N_1 \bullet V \geq 0$, then the surface S_1 is visible from the viewing point V. So does S_3 for $N_3 \bullet V \geq 0$. But S_2 is not visible from the viewing point V because the inner product $N_2 \bullet V < 0$. Other surfaces are treated accordingly. In this way, the invisible surfaces are eliminated.

Now consider the elimination of the hidden lines. If there are surfaces formed by polygons, every line segments must be checked to see whether it is blocked by polygons in 3D space. If such is the case, the line segment is clipped with respect to the polygons as shown below (Figure 5.11):

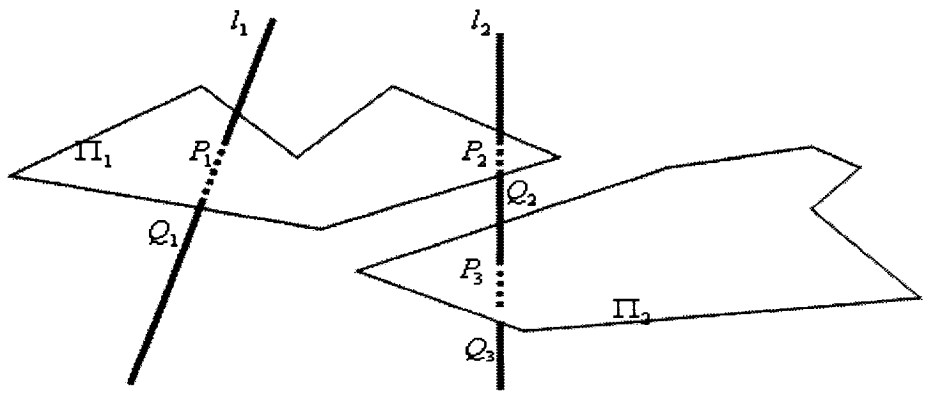


Figure 5.11 The clipping of the hidden lines

In the above figure, there are two polygons: Π_1, Π_2, and two line segments: l_1, l_2. The line segment l_1 intersects polygon Π_1 at P_1 and l_2 intersects polygons Π_1, Π_2 at P_2, P_3

respectively. The positions of the points P_1, P_2, P_3 and Q_1, Q_2, Q_3 are determined, where the point Q_1 is the intersection between the line segment l_1 and the boundary of the polygon Π_1 and two points Q_2, Q_3 are the intersections between the line segment l_2 and the boundary of the polygons Π_1 and Π_2 respectively.

The direct approach to eliminate the hidden lines is to compare each line to each surface of the polygon. The main job of the algorithm is to clip out the parts of the line segments hidden by the surfaces. This job is not easy because there are many situations needed to be dealt with. Usually there are three steps to do the job:

1) For each line segment, calculate all real intersections between the line segment and each polygon.

2) For each line segment, calculate all the start intersections between the line segment and each boundary of the polygons.

3) For each line segment, determine which part of the line segment is visible or hidden.

Fortunately, in the software the polygons are just four squares (belts). So the intersections according to the projected coordinates can be easily calculated. If the situation that the block objects are those belts is considered, then the vertical line segments are cut into several pieces. One part consists of the visible line segments and the other part is the set of the hidden-line segments. The visible line segments are displayed and the hidden ones are ignored or the hidden line segments are drawn with the dash lines.

5.6 Shading Model for the Objects

To display the objects more realistically, shading needs to be applied to the objects. A shading model attempts to imitate how light that emanates from light sources would interact with objects in a scene. As such, different points on surfaces of the objects are drawn with different gray level or brightness of color.

A shading model also dictates how light is scattered or reflected from a surface. There are two types of light sources illuminating the objects in a scene: point light sources and ambient light. These light sources shine on the various surfaces of the objects, and the incident light interacts with the surfaces in three different ways:

1) Some is absorbed by the surface and is converted to heat.

2) Some is reflected from the surface.

3) Some is transmitted into the interior of the objects.

If all of the incident light is absorbed, the object appears black. If all of the light is transmitted, the object is visible only through the effects of refraction. Assume that there are two types of reflection of incident light:

1) Diffuse scattering occurs when some of the incident light penetrates the surface slightly and is re-radiated uniformly in all directions. Scatted light interacts strongly with the surface. Color is usually affected by the nature of the material.

2) Specular reflections are more mirrorlike and are highly directional: incident light does not penetrate the object, but instead is reflected directly from its outer surface. This gives rise to highlights and makes the surface look shiny.

To calculate the diffuse and specular components of light, three vectors are needed.

1) The normal vector ***m*** to the surface at a point ***P***.

2) The vector ***v*** from ***P*** to the viewer's eye.

3) The vector ***s*** from ***P*** to the light source.

To calculate the diffuse component, suppose that light falls from a point source onto one side of a facet. A fraction of the light is re-radiated diffusely in all directions from that side. Some fraction of the re-radiated part reach the eye, with an intensity denoted by I_d . Obviously, I_d depends on the directions of the vectors ***m***, ***v***, and ***s***. By Labert's law, the intensity I_d is promotional to the inner product $s \bullet m$. Therefore, for the intensity of the diffuse component, the expression below is adopted:

$$I_d = I_s \rho_d \frac{s \bullet m}{|s||m|}.$$

Where I_s is the intensity of the light source and ρ_d is the diffuse reflection coefficient. If the facet is aimed away from the eye, this dot product is negative, the I_d is evaluated zero. So a more practice formula is as follows:

$$I_d = I_s \rho_d \max\left(0, \frac{s \bullet m}{|s||m|}\right)$$

Another part of light source is Specular reflections. Since real objects do not scatter light uniformly in all directions, a specular component is added to the shading model. Specular reflection causes highlights, which can add significantly to the realism of a picture when objects are shiny. A model called the Phong model [61] is used to calculate the value of Specular reflection. The Phong model is described as follows, with reference to Figure5.12.

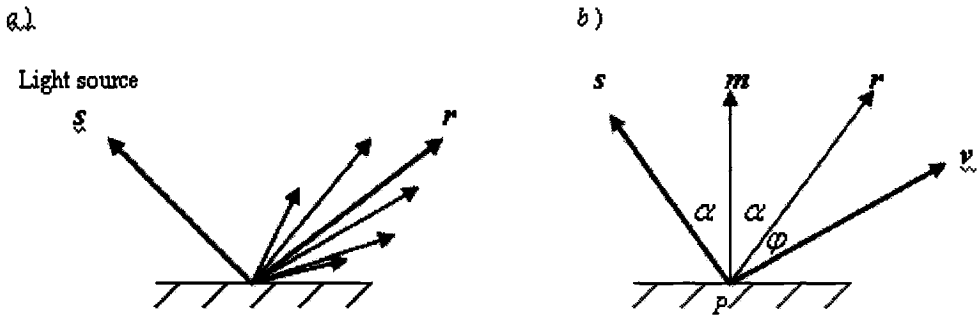


Figure 5.12 The Phong model

Usually, light from a source impinges on a surface and is reflected in different directions as shown in Figure5.12(*a*). The amount of light reflected is greatest in the direction of perfect mirror reflection, ***r***, where the angle of incidence equals the angle of reflection. This is the direction in which all light would travel if the surface were a perfect mirror. At other nearby angles, the amount of reflected light diminishes rapidly. The distribution of the reflected light forms a "beam pattern". Figure5.12(*b*) shows that for a shiny surface that is not a true mirror, the amount of light reflected falls off as the angle φ between ***r*** and ***v*** (the viewing vector) increases. The actual amount of falloff is a complicated function of the angle φ, but in the Phong model it is said to vary as some power of the cosine of φ ($\cos^f \varphi$), in which f is chosen experimentally. Using the fact that $\cos(\varphi)$ is equivalent to the dot product of ***r*** and ***v*** (after they are normalized), the contribution I_{sp} due to specular reflection can be modeled by the formula below:

$$I_{sp} = I_s \rho_s \left(\frac{r \bullet v}{|r||v|} \right)^f ,$$

where the term ρ_s is the specular reflection coefficient, and the mirror-reflection vector can be calculated by the following expression:

$$\boldsymbol{r} = -\boldsymbol{s} + 2\frac{(\boldsymbol{s}\bullet\boldsymbol{m})}{|\boldsymbol{m}|^2}\ \boldsymbol{m}.$$

A faster algorithm for calculating the amount of specular reflection I_{sp} was proposed by Blinn[62]. Instead of finding the reflection vector $\boldsymbol{r}$, the method finds a vector halfway between $\boldsymbol{s}$ and $\boldsymbol{v}$, that is $\boldsymbol{h}= \boldsymbol{s} + \boldsymbol{v}$. The normal to the surface were oriented along the vector $\boldsymbol{h}$, the viewer would see the brightest specular highlight. Let β denote the angle formed by the normal vector $\boldsymbol{m}$ and the vector $\boldsymbol{h}$, β can be used to measure the falloff of specular intensity that the viewer sees. The angle is not the same as the angle φ, but this difference can be compensated by using a different value of the exponent f. It is common practice to base the specular term on $\cos(\beta)$, using the dot product of $\boldsymbol{h}$ and $\boldsymbol{m}$:

$$I_{sp} = I_s \rho_s \max\left(0, \left(\frac{\boldsymbol{h}}{|\boldsymbol{h}|}\bullet\frac{\boldsymbol{m}}{|\boldsymbol{m}|}\right)^f\right)$$

Since with only diffuse and specular reflections, any parts of a surface that are shadowed from the point source receive no light and so are drawn black. But this is not our everyday experience. The scenes observed around us always seem to be bathed in some soft non-directional light. This light arrives by multiple reflections from various objects in the surroundings and from light sources that populate the environment. This amount of light needs to be calculated, but it would be computationally very expensive to model it. The diffuse and specular components of reflected light are found by simplifying the rules by which physical light reflects from physical surfaces. But the shadows of the objects will be seen to be unrealistically deep and harsh. To soften these shadows, ambient light can be added to the model as the third component. Since ambient light is not situated at any particular place, it spreads in all direction uniformly. This source is assigned an

intensity I_a. Each face in the model is assigned a value for its ambient reflection coefficient ρ_a, and the term $I_a \rho_a$ is simply added to whatever diffuse and spacular light that which is reaching the eye from each point *P* on that face.

Now three light contributions (diffuse, specular, and ambient) are combined together to form the total amount of light I that reaches the eye from point P:

$$I = I_a \rho_a + I_d \rho_d \times lambert + I_{sp} \rho_s \times phong^f,$$

where the values

$$lambert = \max\left(0, \frac{s \bullet m}{|s||m|}\right) \text{ and } phong = \max\left(0, \frac{h}{|h|} \bullet \frac{m}{|m|}\right)$$

I depends on the various intensities of the source and reflection coefficients of the object, as well as on the relative positions of point *P*, the eye, and the point light sources.

It is straightforward to extend the shading model under consideration to the case of colored light reflecting from colored surfaces since light of any color can be constructed by adding certain amount of red, green and blue light. When dealing with colored sources and surfaces, each color component is calculated and is simply added individually to form the final color of reflected light. So, to calculate the red, green and blue components of reflected light, the formula above is applied three times as follows:

$$I_r = I_{ar} \rho_{ar} + I_{dr} \rho_{dr} \times lambert + I_{spr} \rho_{sr} \times phong^f,$$

$$I_g = I_{ag} \rho_{ag} + I_{dg} \rho_{dg} \times lambert + I_{spg} \rho_{sg} \times phong^f,$$

$$I_b = I_{ab} \rho_{ab} + I_{db} \rho_{db} \times lambert + I_{spb} \rho_{sb} \times phong^f.$$

Where the coefficients *lambert* and *phong* are as before. The corresponding coefficients are as follows:

ambient reflection coefficients: ρ_{ar} , ρ_{ag} and ρ_{ab}

diffuse flection coefficients: ρ_{dr} , ρ_{dg} and ρ_{db}

specular reflection coefficients: ρ_{sr} , ρ_{sg} and ρ_{sb}

For example, for golden material the coefficients are as follows:

$$\rho_{ar}=0.2427, \rho_{ag}=0.1995, \quad \rho_{ab}=0.0745;$$

$$\rho_{dr}=0.7516, \rho_{dg}=0.6065, \quad \rho_{db}=0.2265;$$

$$\rho_{sr}=0.6283, \rho_{sg}=0.5558, \quad \rho_{sb}0.3660;$$

In the software package, all those shading formulas will be used to calculate the light intensities of the objects.

5.7 Z-Buffer Algorithm for Hidden Surface Elimination

A commonly used image-space approach to eliminating hidden surfaces is the depth-buffer method, also called z-buffer method. Basically, this algorithm tests the visibility of surfaces one point at a time. For each pixel position (x, y) on the view plane, the surface with the smallest z coordinate at that position is visible. Figure 5.13 shows three surfaces at varying depth with respect to position (x, y) in a left-handed view system. Surface S_1 has this position. So its intensity value at (x, y) is served.

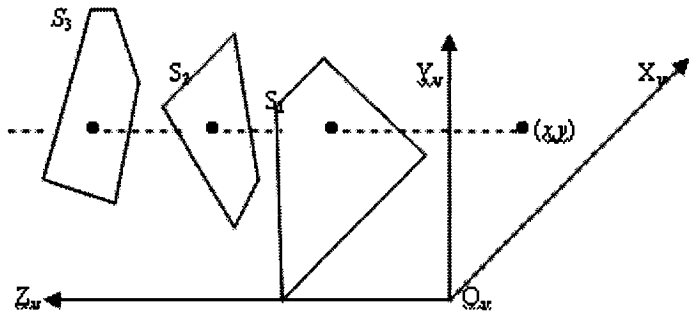


Figure 5.13 z-buffer algorithm principle

· Two buffer areas are required for implementation of this method. A depth buffer is used to store z values for each (x, y) position as surfaces are compared, and the refresh buffer stores the intensity (color) values for each position.

The steps of the Z-Buffer algorithm can be summarized as follows:

Set $d(x, y)$ = *large_number* and *refresh*(x, y) = *background color*
for (each face F of the objects)
  for (each pixel (x, y) covering F)
  {*depth* = depth of F at (x, y);
    if (*depth*< $d(x, y)$)
      { $d(x, y)$ = *depth;* *refresh*(x, y) = intensity or color of F at (x, y);
        set the pixel intensity or color at (x, y) to *refresh*(x, y);
      }
  }

The algorithm is simple but it needs a large quantity of memory and vast amounts of time.

5.8 The 2D and 3D objects building for the *zy*-buffer package

In this software package, the geometric objects that need to be dealt with are points, links (line segments), ellipses (circles), facets of the 3D objects, cubes and cylinders. All of them need to be constructed ahead before they are used.

1) Points

A point on a plane is specified by a pair of ordered integers (x, y) in a right-handed Descartes coordinate system, where the x is the first coordinate and y is the second. Notice that on the CRT the device coordinate system is a left-handed system. On the CRT a pixel stands for a point. But a pixel is too small to be seen. So a filled oval with a small radius are used to represent a point on screen. In class "point", the constructor will generate the filled oval by using the Java statement: "g.fillOval(x, y, 5, 4); ". The methods get_x() and get_y() return the x and y coordinate of the point respectively.

2) Links (line segments)

Many links are needed to stand for cables, cords and optic fibers. A link is determined by two points (x_1, y_1) and (x_2, y_2). Other attributes are width, color, dashed pattern or solid pattern.

3) Ovals (circles)

An oval on a plane is determined by a centre (x, y) and a radius r. The Oval class specified the attributes of an oval such as the centre, radius and color. To describe the optic fiber rings the dashed ellipses are displayed. The methods get_x() and get_y() return the x and y coordinate of the centre respectively.

4) Cylinders

The software needs a type of picture to stand for the IP switch icon. A cylinder is employed to this end. To construct a cylinder icon, two ellipses and two vertical line segments are drawn as follows:

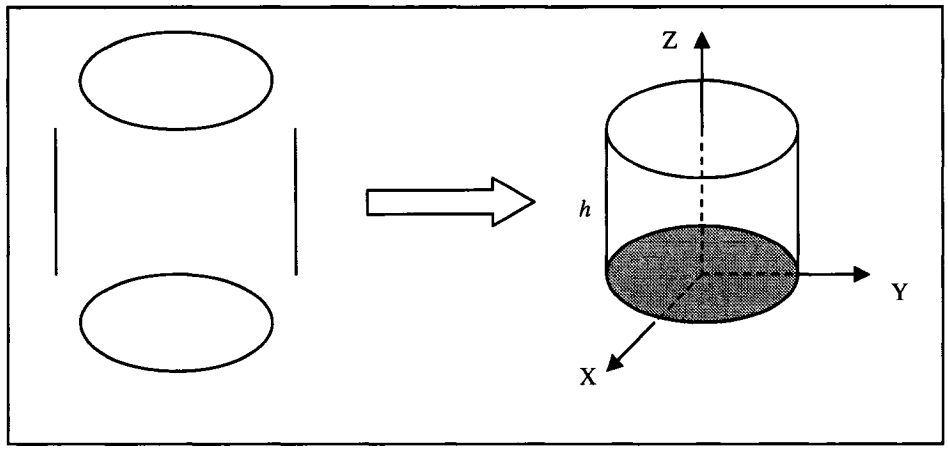


Figure 5.14 Cylinder building

In 3D coordinate system, the ellipse on the bottom of the cylinder is centered at (0,0,0) with radius r and the ellipse on the top is centered at (0,0,h), where h is the height of the cylinder.

For convenience, the polar system is adopted to describe the equation of the ellipse. Let θ be the angle between the vector point to the ellipse and the x axis. Then the equation of the ellipse (circle) on the bottom can be expressed as

$$x = r\cos(\theta),\ y = r\sin(\theta),\ z = 0.$$

The equation of the ellipse (circle) on the top is

$$x = r\cos(\theta),\ y = r\sin(\theta),\ z = h.$$

With the equations, the objects can be easily projected to the screen and other objects related to them are calculated.

To project the cylinder onto a screen, the cylinder is partitioned to facets to approximate it. This is done as follows:

Suppose the cylinder is partitioned into 40 pieces. The increment $\Delta\theta$ is set to $2\pi/40 =$

3.14159/20. Then the ellipse is divided accordingly into 40 pieces:

$$E = \{ E_i = \{(x_i, y_i), (x_{i+1}, y_{i+1})\} \mid x_i = r\cos(\Delta\theta \times i), y_i = r\sin(\Delta\theta \times i),\ i = 0,1,2,...,39 \}.$$

Now construct 40 facets:

$$F = \{ F_i \mid F_i = \text{rect}\{(x_i, y_i, 0), (x_{i+1}, y_{i+1}, 0), (x_{i+1}, y_{i+1}, h), (x_i, y_i, h)\},\ i = 0,1,2,...,39 \}.$$

After those 40 rectangles are projected onto the screen, the approximated image of the cylinder is obtained. This separation will be used to calculate the intensity of light reflected by the cylinder in a shading process.

To get the shading effect for the cylinder, the intensity of light for each rectangle is calculated. In the software, assume the viewing vector $\boldsymbol{v}$ = {100, 3000,-20} and the light source vector $\boldsymbol{s}$ = {1000, 1000, 400} in world coordinate system. And let

$$I_a = 0.1,\ \rho_a = 0.7,\ I_d = 0.5,\ I_{sp} = 0.5,\ \rho_d = 0.9,\ \rho_{sp} = 0.6.$$

For each facet F_i, according to the shading model, the intensity of light reflected by facet F_i are calculated ($i = 0,1,2,...,39$). To this end, the normal vector m_i for each facet F_i and the half-way vector $\boldsymbol{h}$ must be found. Obviously, $\boldsymbol{h} = \boldsymbol{s} + \boldsymbol{v}$. For calculating the vector m_i, let vector $u_i = (x_{i+1} - x_i, y_{i+1} - y_i, 0)$ and vector $v_i = (x_{i+1} - x_i, y_{i+1} - y_i, h)$. Then the normal vector m_i is parallel to the outer product vector $u_i \times v_i$. Finally, the intensity of the light reflected by the facet F_i is obtained:

$$I_i = I_a \rho_a + I_d \rho_d (lambert)_i + I_{sp} \rho_s (phong)_i^f,\ (i = 1,2,...,39)$$

where the values

$$(lambert)_i = \max\left(0, \frac{s \bullet m_i}{|s||m_i|}\right) \text{ and } (phong)_i = \max\left(0, \frac{h}{|h|} \bullet \frac{m_i}{|m_i|}\right),\ f = 9.$$

For the top of the cylinder, four arrows are drawn on it. They can be treated as four polygons. The design for the top cap is shown below:

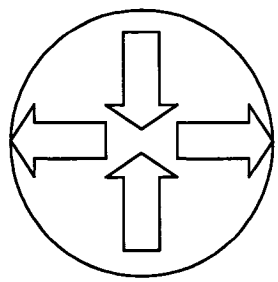
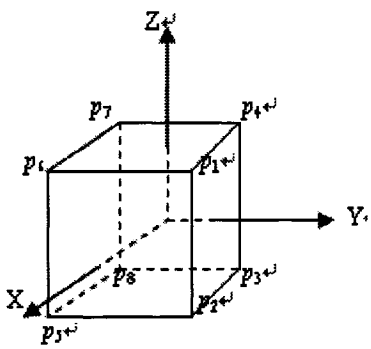


Figure 5.15 The cap of the cylinder

This figure stands for the IP routers.

5) Cubes

To display the icon for ATM, a cube to stand for the facility is constructed. Suppose the length of the edge on a cube is $2a$. The coordinates of the eight vertices are listed in Figure 5.16:



$$
\begin{aligned}
p_1 &: (a, \ a, \ a) \\
p_2 &: (a, \ a, \ -a) \\
p_3 &: (-a, \ a, \ -a) \\
p_4 &: (-a, \ a, \ a) \\
p_5 &: (a, \ -a, \ -a) \\
p_6 &: (a, \ -a, \ a) \\
p_7 &: (-a, \ -a, \ a) \\
p_8 &: (-a, \ -a, \ -a)
\end{aligned}
$$

Figure 5.16 A cube in a Descartes coordinate system with origin at the centre

The six facets are listed below:

F_1=square$\{p_1, \ p_2, p_3, \ p_4\}$, F_2=square$\{p_1, \ p_6, p_5, \ p_2\}$, F_3=square$\{p_1, \ p_4, p_7, \ p_6\}$,

F_4=square$\{p_4, \ p_3, p_8, \ p_7\}$, F_5=square$\{p_6, \ p_7, p_8, \ p_5\}$, F_6=square$\{p_2, \ p_5, p_8, \ p_3\}$.

Then the cube can be projected to the screen.

To get the shading effect for the cube, the intensity of light for all facets is calculated. Again, assume that the viewing vector $\boldsymbol{v}$ = {100, 3000,-20} and the light source vector $\boldsymbol{s}$ = {1000, 1000, 400} in world coordinate system. And let

$$I_a=0.1,\ \rho_a=0.7,\ I_d=0.5,\ I_{sp}=0.5,\ \rho_d=0.9,\ \rho_{sp}=0.6.\ \boldsymbol{h}=\boldsymbol{s}+\boldsymbol{v}.$$

For each facet F_i, the intensity of light reflected by the facet F_i $(i=1,2,..,6)$ can be calculated. First the normal vector m_i for each facet F_i must be obtained. For calculating the vector m_i, let

$$u_1=p_2-p_1,\quad v_1=p_4-p_1;\quad u_2=p_6-p_1,\quad v_2=p_2-p_1\ ;$$

$$u_3=p_4-p_1,\quad v_3=p_6-p_1;\quad u_4=p_3-p_4,\quad v_4=p_7-p_4;$$

$$u_5=p_7-p_6,\quad v_5=p_5-p_6\ ;\quad u_6=p_5-p_2,\quad v_6=p_3-p_2.$$

Then the normal vector $m_i=u_i\times v_i\,(i=1,2,..,6)$. And the halfway vector $\boldsymbol{h}=\boldsymbol{s}+\boldsymbol{v}$. Finally the intensity of the light reflected by the facet F_i is obtained:

$$I_i \;=\; I_a\rho_a \;+\; I_d\rho_d(lambert)_i \;+\; I_{sp}\rho_s(phong)_i^{\,f},\ (i=1,2,...,6).$$

where the values $(lambert)_i$, $(phong)_i$ $(i=1,2,...,6)$ are as before and $f=6$.

For the front facets of the cube, an X like figure is drawn on it which stands for the exchange of the information with an ATM. A polygon is drawn on the front facet and the design for the icon is shown below:

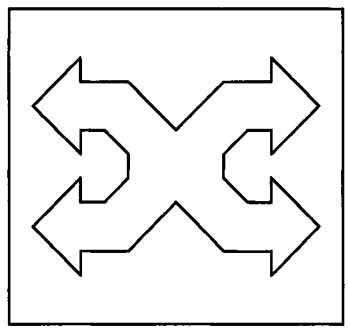


Figure 5.17 The design for the front facet of the cube icon

6) Dishes

A dish like icon is designed to stand for the SONET/SDH node. The dish consists of a thin cylinder and a cap with a "Stelliform" hexagon pattern. The design for the thin cylinder is the same as the cylinder in 4). On the top of the thin cylinder, a "Stelliform" hexagon is designed as follows:

The hexagon is composed of 12 vertices. The vertices are divided into two groups. The equations for the group 1 and group2 are listed below:

Group1: $x_i = r_1 \cos(\Delta\theta \times i), y_i = r_1 \sin(\Delta\theta \times i), \quad (i = 0,2,4,6,8,10)$;

Group2: $x_j = r_2 \cos(\Delta\theta \times j), y_j = r_2 \sin(\Delta\theta \times j), \ (j = 1,3,5,7,9,11)$,

where $\Delta\theta = \frac{2\pi}{12} = \frac{\pi}{6}$, r_1 (r_2) is the radius of the outer (inner) circle and $r_1 > r_2$.

The appearance of the cap is given below:

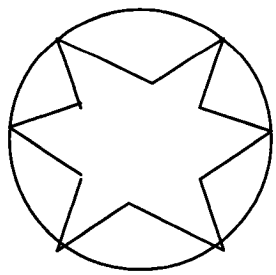
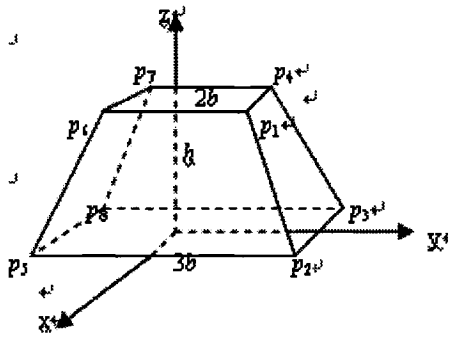


Figure 5.18 The cap of the dish

7) Pyramid

The pyramid icon stands for the DWDM node. It is constructed as follows:

Suppose the length of edges on top and on bottom square is $2b$ and $3b$ respectively, and the height of the pyramid is h. Then the coordinates of the eight vertices are listed below:



$$
\begin{aligned}
p_1 &: (b, \quad b, \quad h)\\
p_2 &: (3b/2, \quad 3b/2, \quad 0)\\
p_3 &: (-3b/2, \quad 3b/2, \quad 0)\\
p_4 &: (-b, \quad b, \quad h)\\
p_5 &: (3b/2, \quad -3b/2, \quad 0)\\
p_6 &: (b, \quad -b, \quad h)\\
p_7 &: (-b, \quad -b, \quad h)\\
p_8 &: (-3b/2, \quad -3b/2, \quad 0)
\end{aligned}
$$

Figure 5.19 The design for a pyramid icon

The six facets of the pyramid are listed below:

F_1=square$\{p_1, \ p_2, p_3, \ p_4\}$, F_2=square$\{p_1, \ p_6, p_5, \ p_2\}$, F_3=square$\{p_1, \ p_4, p_7, \ p_6\}$,

F_4=square$\{p_4, \ p_3, p_8, \ p_7\}$, F_5=square$\{p_6, \ p_7, p_8, \ p_5\}$, F_6=square$\{p_2, \ p_5, p_8, \ p_3\}$.

Then the pyramid can be projected to the screen.

The shading effect can be obtained by the process that is same as in cube building. Again, let v = {100, 3000,-20} and s = {1000, 1000, 400}. And let

$$I_a=0.1, \ \rho_a=0.7, \ I_d=0.5, \ I_{sp}=0.5, \ \rho_d=0.9, \ \rho_{sp}=0.6. \ \boldsymbol{h} = s + v.$$

To calculate the intensity of light reflected by the facet F_i $(i = 1,2,..,6)$, the normal vector m_i for each facet F_i is calculated. Let

$$u_1 = p_2 - p_1, \quad v_1 = p_4 - p_1; \quad u_2 = p_6 - p_1, \quad v_2 = p_2 - p_1 \ ;$$

$$u_3 = p_4 - p_1, \quad v_3 = p_6 - p_1; \quad u_4 = p_3 - p_4, \quad v_4 = p_7 - p_4;$$

$$u_5 = p_7 - p_6, \quad v_5 = p_5 - p_6 \ ; \ u_6 = p_5 - p_2, \quad v_6 = p_3 - p_2 .$$

Then the normal vector $m_i = u_i \times v_i \, (i = 1,2,..,6)$, and the halfway vector $\boldsymbol{h} = \boldsymbol{s} + \boldsymbol{v}$. Finally the intensity of the light reflected by the facet F_i is obtained:

$$I_i \ = \ I_a \rho_a \ + \ I_d \rho_d (lambert)_i \ + \ I_{sp} \rho_s (phong)_i^{\ f}, (i = 1,2,...,6).$$

where the values $(lambert)_i$, $(phong)_i$ $(i = 1,2,...,6)$ are as before and $f = 6$.

Now all of the objects are constructed. The next chapter concerns how to display the objects on a CRT screen.

Chapter 6

Experimental Results

The *Incremental Projection Algorithm* and the *Depth-Height Buffer Algorithm* are two new algorithms for graphical or network 3D display presented in this thesis. This chapter tests the *Incremental Projection Algorithm* and the *Depth-Height Buffer Algorithm* respectively. Then the stability, scalability and the complexity of each algorithm are discussed.

6.1 Testing for the Incremental Projection Algorithm

Theoretically, the *Incremental Projection Algorithm* is a general 3D display algorithm for any graph. It is suitable for any graph with any vertex degree. Given a mess graph, it can be displayed in 3D space without any edge crossing and overlap by employing of the *Incremental Projection Algorithm.* To show the *Incremental Projection Algorithm* is correct and functional, several test methods are used.

6.1.1 Structure testing (Logic testing or white box testing)

Structure testing focuses on the correctness of the algorithm. There are four basic aspects of the structure testing: statement coverage, branch coverage, condition/branch coverage, and path coverage.

(1) **Statement Coverage Testing:** This requires that every statement in an algorithm has been executed at least once.

Check the statement of the algorithm. It is obvious that the first and second statements are initial steps. Statement one numbers the nodes of a graph, sets the vertex degree and free degree. Statement two puts the first vertex of the graph on the first plane. Those two statements are executed at the beginning of the algorithm. In statement three, if the graph has at least two vertices, the algorithm takes the second vertex as the current vertex to be manipulated. If there are more than two vertices, the loop will go on until the vertices are exhausted. Therefore, statement three is executed. The branch statements will be checked later. Statement four is for graph drawing. Of course, it is executed.

(2) **Branch Coverage Testing**: The Branch Coverage Testing focuses on testing branches in an algorithm.

There are two main branches in statement three. When a vertex is an insertion vertex, there are exactly two possible cases about its *local degree* and its *free degree*: they are equal or not equal.

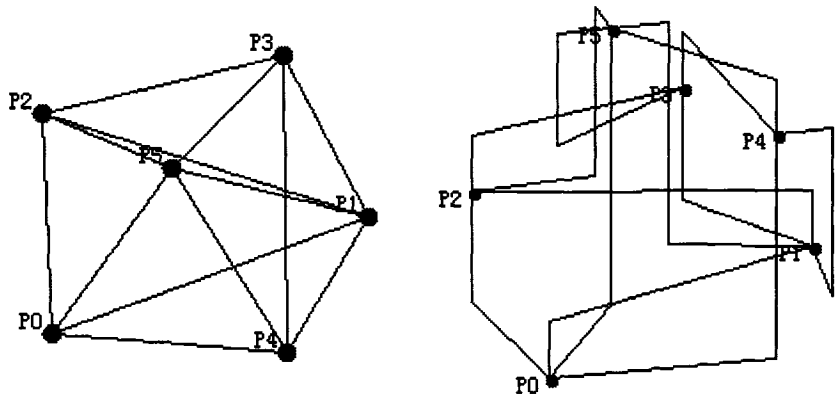
For the first case (the *local degree* is equal to the *free degree*), the insertion vertex has free *bottom* connector. Then the *route1* and *route2* exist.

For the case that the *free degree* is less than the *local degree*, the insertion vertex has no free *bottom* connector (it is used before). In this case, *route3* exists. If the insertion vertex has free *bottom* connector, *route1* and *route2* exist as well.

(3) **Condition/Branch Coverage Testing:** This focuses on testing whether every branch is invoked at least once and all possible combinations of conditions in decisions are exercised.

In statements of the *Incremental Projection Algorithm*, the algorithm exhausts all possible cases that two adjacent vertices can happen. Firstly, the insertion vertex has just two cases about its *local degree* and *free degree*. One case is that its *local degree* is equal to its *free degree*. Another case is that its *local degree* is not equal to its *free degree*. There are just two sub-cases in case one about the free *top* direction of the first vertex. The *top* direction of the first vertex may be free or may not be free. If the *top* direction of the first vertex is free, *route1* is invoked. If the *top* direction of the first vertex is not free, *route2* is invoked because in case one the bottom direction of the insertion vertex is always free.

As to the case that the *local degree* is not equal to its *free degree* of the insertion vertex, there are three sub-cases that may happen. The first sub-case is when the *top* direction of the first vertex is free, whether or not the *bottom* direction of the insertion vertex is free, *route1* is invoked. The second case is when the *top* direction of the first vertex is not free and the *bottom* direction of the insertion vertex is free. In this case, *route2* is constructed. The third sub-case corresponds to the situation that both the *top* direction of the first vertex and the *bottom* direction of the insertion vertex are not free. Then *route3* is built. Figure 6.1 shows the cases discussed.



(a) (b)

Figure 6.1 Example for Condition/Branch Coverage Testing

Figure 6.1(a) is a non-planar graph and Figure 6.1(b) is one of the 3D displays with the *Incremental Projection Algorithm.* The graph *G*'s ordered set of vertices is $S=\{ p_0, p_1, p_2, p_3, p_4, p_5, p_6 \}$. After the vertex p_1 is inserted, the local degree of the vertex p_1 is just one which is also its *free degree.* Since the *top* direction of the vertex p_0 is free, *route1*(p_1, p_0) is drawn. When p_2 is inserted, the *top* direction of the vertex p_0 is occupied by *route1*(p_1, p_0). Since the *bottom* direction of the vertex p_2 is free, *route2*(p_2, p_0) is invoked. For the pair p_2 and p_1, because the *top* direction of the vertex p_1 is free, *route2*(p_2, p_1) is generated. After p_4 is inserted, since the bottom direction of p_4 is occupied by *route2*(p_4, p_0) and the top direction of p_1 is not free, then *route2*(p_4, p_1) is invoked.

(4) **Path Coverage Testing**: Path coverage testing considers all possible local paths in an algorithm and leads to test cases aimed at exercising each path. Actually, from the example above, all paths in the algorithm are exercised.

6.1.2 Functional testing (Black box testing)

Functional testing focuses on inputs, outputs, and the principal functions of the algorithm module. Actually a package that the specific algorithm is implemented is tested. Some test cases are given below, which exercises all functions of the algorithm.

Case 1: *Input*: A non-plane graph K_7.

Output: A 3D drawing for K_7 without edge crossing.

The graph K_7 is non-planar. Its 3D displays with the *Incremental Projection Algorithm* are shown below, where sub-figures (b), (c) and (d) are different representations corresponding to different angles. Some edges seem to be crossing each other in one figure. But this is not the case in different views. For example, in Figure 6.2(b) *route2*(p_0, p_6) and *route1*(p_2, p_3) seems to cross. However, in Figure 6.2(d) they obviously do not. In Figure 6.2(b) *route1*(p_3, p_4) and *route2*(p_0, p_6) seems to cross. But Figure 6.2(c) shows that this is not the case.

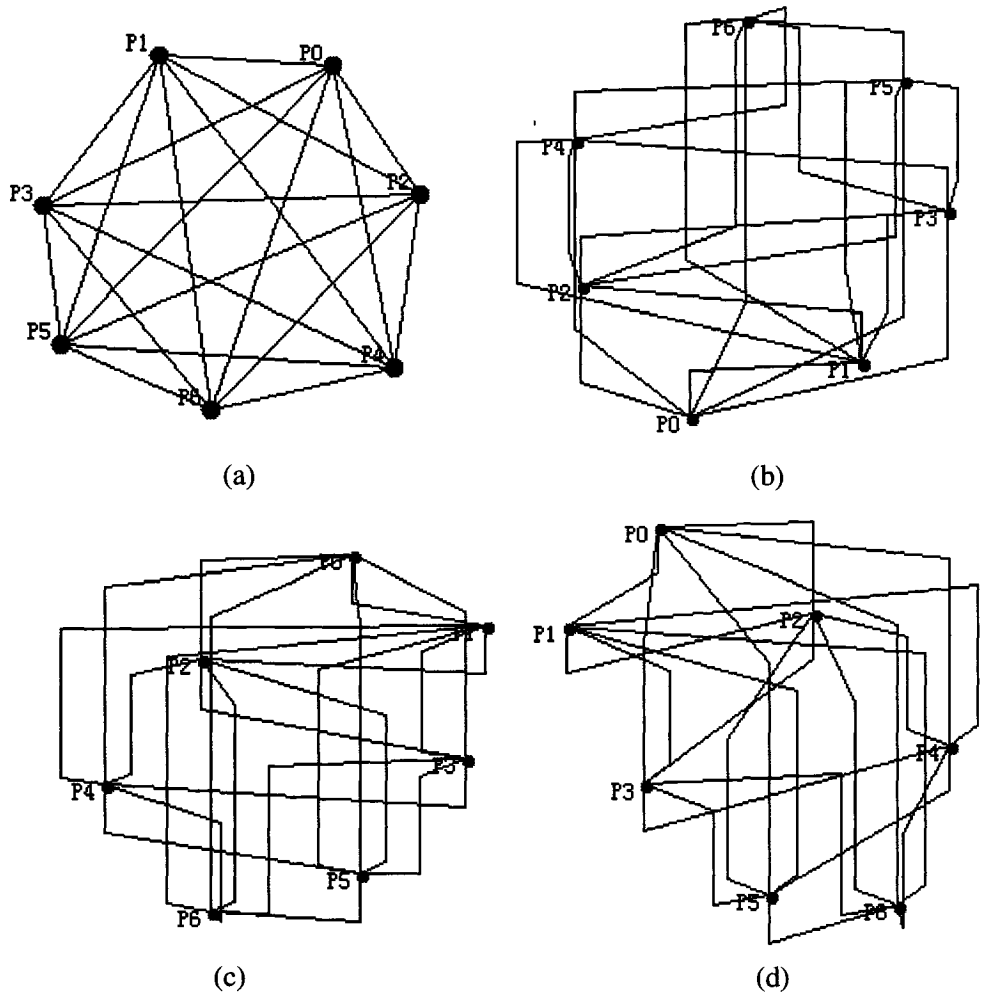


Figure 6.2 3D displays for K_7

Figure 6.2(a) is a 2D drawing of K_7. Figure 6.2(b) is the 3D display for K_7 with horizontal rotation $20°$ and vertical rotation $30°$. Figure (c) displays with horizontal rotation $200°$ and vertical rotation. Figure (d) displays with horizontal rotation $90°$ and vertical rotation $160°$.

Case2 : *Input*: A graph G in which some vertices have degree seven.

Output: A 3D drawing of the graph G without edge crossing and has at most two bends per edge.

As mentioned before [65], if a graph with degree not more than six, it has an orthogonal

drawing such that each line segment of the edge is parallel to one of the three coordinate axes and it has at most three bends per edge. Figure 6.3 is a graph that has vertices with degree more than six. The 3D drawing of the graph produced by employing the *Incremental Projection Algorithm* is

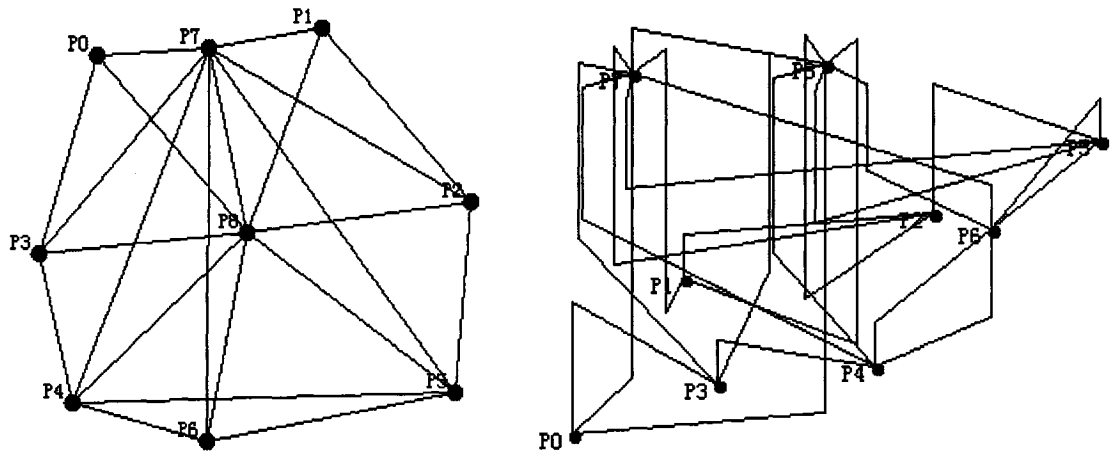


Figure 6.3 A graph *G* with some degree more than seven vertices and its 3D drawing

This graph, see Figure 6.3(a), has two vertices (P_7 and P_8) whose degree is eight. In Figure 6.3(b), the vertex P_7 is the placement of the original vertex P_7 in Figure 6.3(a). It has degree eight and connects its adjacent vertices normally according to the *Incremental Projection Algorithm*. So does the vertex P_8. Furthermore, there is no crossing between any pair of edges, which can be checked by rotating it by use the package. The bends of each edge in Figure 6.3(b) is not more than two.

Case 3: *Input*: A non-planar and non-connected graph *G* with three connected components.

Output: A 3D drawing of the graph *G* with no edge crossing and keeping the components unchanged.

Given a non-connected graph *G* that has three connected components. What will happen if it is displayed in 3D by using of the *Incremental Projection Algorithm*? Does the 3D drawing of the

graph keep the connected components? The non-connected graph G is shown below:

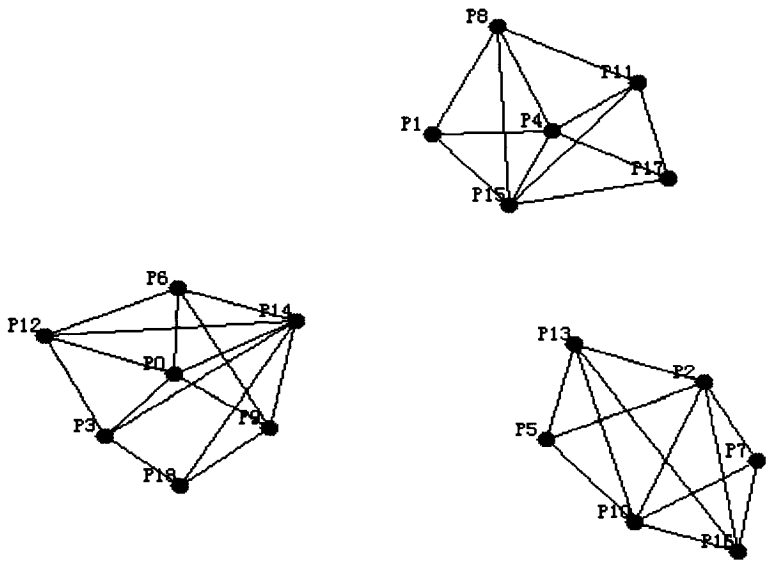


Figure 6.4 A non-planar graph with three connected components

The graph is processed by the *Incremental Projection Algorithm* implemented in the package. Its 3D drawing is displayed as follows:

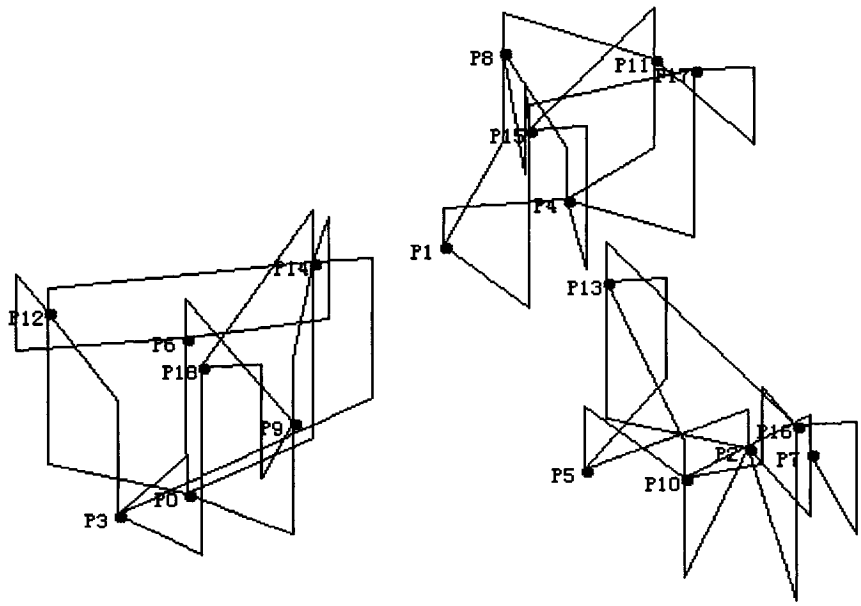


Figure 6.5 The 3D drawing of a non-connected graph with three connected components

It is obvious that the *Incremental Projection Algorithm* for the 3D drawing of the non-connected graph G keeps the topology of the original graph.

6.2 Stability of the Incremental Projection Algorithm

The stability of an algorithm means that if the input is fixed, then the output result is also fixed. If the input is fixed and the outputs are variable, the algorithm is not stable. It can be verified that the *Incremental Projection Algorithm* for 3D display of a graph or network is stable. The assertion is supported theoretically and practically.

Firstly, the theoretic proof of the stability for the *Incremental Projection Algorithm* is given here. At the beginning of the algorithm, a vertex set of a graph is given. Then the vertices in the set are numbered to get a permutation of the vertices $V=\{ p_1, p_2,..., p_n \}$. The remaining steps of the algorithm are based on the permutation of the vertices. Statement 2 of the algorithm is definite without divergence. In statement 3, the algorithm works on the vertices pairwise. For a fixed vertex p_j ($j = 2,3,...,n$), find all its adjacent vertices whose index is less than the number j. Those vertices have been permuted already in an array list while the edges were drawn according to the edge input. Therefore, the permutation of the adjacent vertices is fixed. Then the algorithm builds the routes in turn according to the permutation. This guarantees that the routes are constructed in a unique order. Furthermore, suppose the vertex p_i is one of the adjacent vertices of the vertex p_j, since that the p_i's *top* direction or p_j's bottom direction is free or not is definite, the choice of the route from *rouet1*, *route2* and *route3* is also definite. As a result, the routes are produced in a definite order, which guarantees that the 3D drawing of the graph is definite. Therefore, the *Incremental Projection Algorithm* is stable theoretically.

Second, the *Incremental Projection Algorithm* is stable in practice. In fact, for any graph, if the permutation of the vertices is fixed, then the 3D drawing of the graph is unchanged however

many times the algorithm is repeated. The change of the drawing just depends on the permutation of the vertices of a graph. Therefore, the *Incremental Projection Algorithm* is stable practically.

6.3 Complexity of the Incremental Projection Algorithm

The average time complexity of the *Incremental Projection Algorithm* is $\Omega(\sqrt{n}n)$. If there is no degree of vertices great than M, the time complexity of the *Incremental Projection Algorithm* is $O(n)$. Given a graph $G=(V,E)$, where $V=\{p_1, p_2, ..., p_n\}$ and $E=\{e_1,e_2,...,e_m\}$, let d_i denote the degree of the vertex p_i $(i=1,2,...,n)$ and $d=\max\{d_1,d_2,...,d_n\}$. At first, the time spent on statements one and two in the algorithm is constant. For statement three, it has one loop in which the index variable j changes from 2 to n. For each vertex p_j, at most d_i edges need to be manipulated. Suppose the time for manipulating each edge is not greater than T_0. Then the time spent on vertex p_j is $d_i * T_0$, which is less than $d * T_0$. Therefore, the total time spent on the loop does not exceed $n * d * T_0$. Because the average degree is $(n+1)/2$ and $(n+1)/2 \geq \sqrt{n}$, it implies the average time complexity is $\Omega(\sqrt{n}n)$. If $d \leq M(const)$, then $n * d * T_0 \leq M * T_0 * n$, which means $O(n)$.

6.4 Scalability of the Incremental Projection Algorithm

Scalability is the ability of something designed to operate at one measure of size to operate successfully at other sizes, usually at large size. For the scalability of the *Incremental Projection Algorithm*, two aspects need to be considered. The first aspect is the increase of the vertices in a graph. The second aspect is the increase the number of degree.

Theoretically, since the stability of the algorithm is not affected by increasing the number of the vertices, the algorithm never crashes as the vertices increase under the capacity of the

hardware of a computer. On the other hand, the degree of a vertex can not exceed $n-1$, where the number n is the size of a graph. Therefore, the angle increment $\theta = (2\pi)/(n-3)$ can not be zero. So the connectors at a vertex are finite and can be distinguished. Practically, there is no case that the algorithm can not handle. As long as the screen of a computer permits it, whatever a graph inputted, the algorithm can produce a reasonable 3D drawing of the graph.

If the vertices of a graph are too crowded, however, the 3D drawing of the graph may not be displayed clearly with ease. Some vertical line segments seem overlapped from some perspectives. For this reason, when inputting the vertices, it is necessary to set them apart from each other to prevent the 3D drawing from being over crowded.

Figure 6.6 is an example of a graph with 70 vertices and 200 edges. The 3D display gives an impression of the scalability of the algorithm.

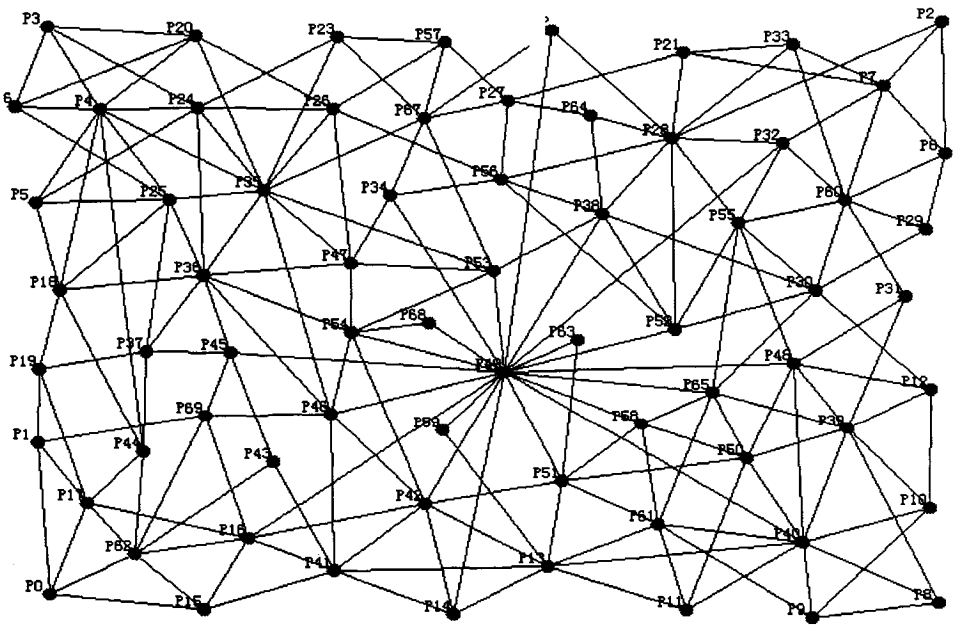


Figure 6.6 Example for scalability of the *Incremental Projection algorithm*

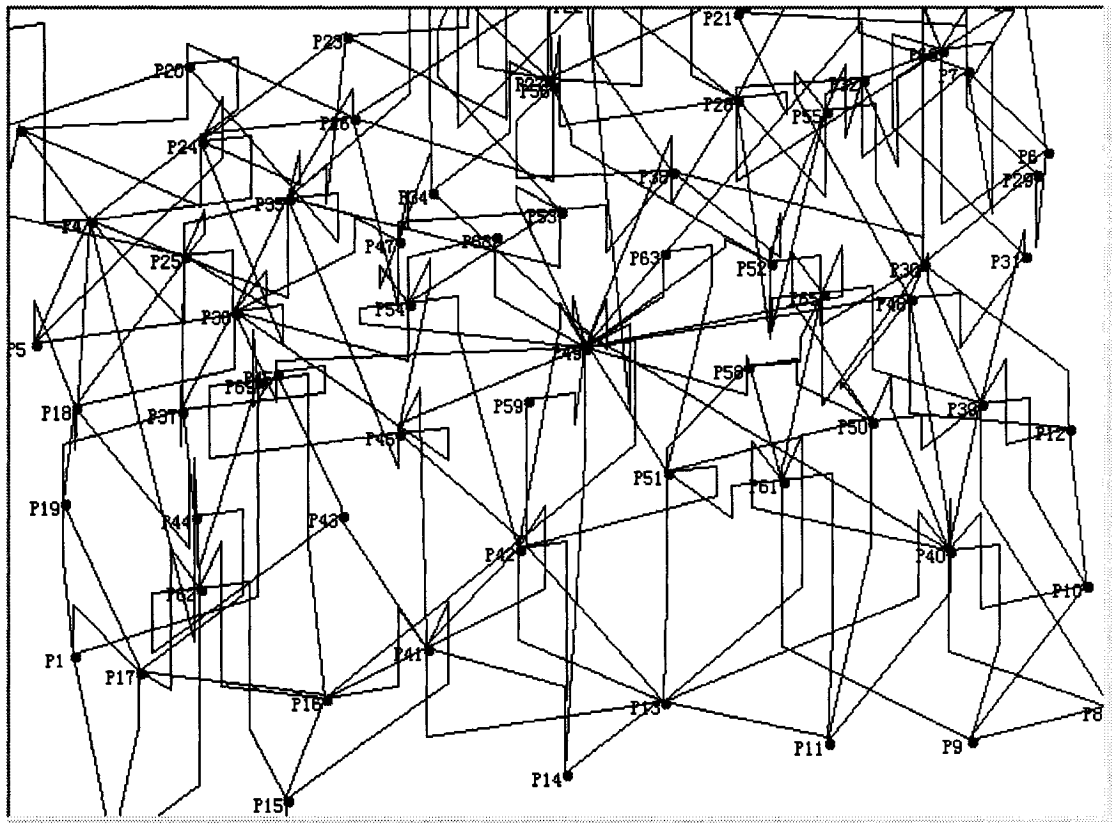


Figure 6.7 The 3D drawing of the graph with 70 vertices and 200 edges

Actually, the *Incremental Projection Algorithm* can be used to manipulate any graph with any size, if the hardware of a computer allows it.

6.5 Testing for the Depth-Height Buffer Algorithm

The *Depth-Height Buffer Algorithm* is designed to eliminate the hidden surfaces of the objects in a multi-layered network. Before implementing the algorithm, it is necessary to verify whether the algorithm is correct and functional. To this end, some tests are applied.

6.5.1 Structure testing

The purpose of structure testing is to check the correctness of an algorithm. Usually, the statement coverage test, branch coverage test, condition/branch coverage test, and path coverage test are used.

(1) **Statement Coverage Testing:**

Statement coverage testing focuses on the statements of an algorithm to check whether every statement is executed at least once. In fact, the first statement of the algorithm is executed because it is just a preparation step.

To eliminate the hidden surfaces according to the z-coordinate, sorting the objects by their z-coordinate must be done. Thus statement two is not skipped.

For objects with the same z-coordinate, if displayed randomly, some parts of objects will be blocked by those with a smaller y-coordinate. In this case, the display will be unnatural. So, for those objects with the same z-coordinate, sorting them by their y-coordinates will result in a better display. Therefore, grouping the objects by z-coordinate and sorting the objects group by group by y-coordinates must be done in the algorithm. Then, statement three is the key step that can not be omitted.

The purpose of statement four is to put the sorted objects all together, then to display the 3D drawing of a multi-layered network according to the new order, thus, statement four must be executed.

(2) **Branch Coverage Testing**:

Branch coverage testing focuses on testing branches in an algorithm. Since there is just one main branch in the algorithm, the branch is definitely covered.

(3) **Condition/Branch Coverage Testing:**

This focuses on testing whether every branch is invoked at least once and all possible combinations of conditions in decisions are exercised. Again, for the same reason, as in branch coverage testing, every branch is invoked.

(4) **Path Coverage Testing**:

Path coverage testing considers all possible local paths in an algorithm and leads to test cases aimed at exercising each path. In the *Depth-Height Buffer Algorithm*, there are two local paths. One is at the first loop before z-coordinate sorting. It is definitely exercised because all z-coordinates of the objects need to be taken. The second local path is at the loop for y-coordinate sorting for every group. It is also exercised. So, all paths are covered.

6.5.2 Functional testing (Black box testing)

Black box testing focuses on the functionalities of an algorithm or software. Usually, cases are inputted and the corresponding outputs are then checked to see whether the outputs are coincident to expected results. Some functional testing results are shown below:

Case 1: *Input*: A planar network with 12 nodes, 15 links among these nodes, and 4 circles.

Output: A 3D drawing of the network with IP router icon, ATM icon, SONET icon, and DWDM switch icon. Different icons are on the different layers. The front icons can not be blocked by the back icons and the higher level icons can not be blocked by the lower level icons.

The network is drawn as the following picture:

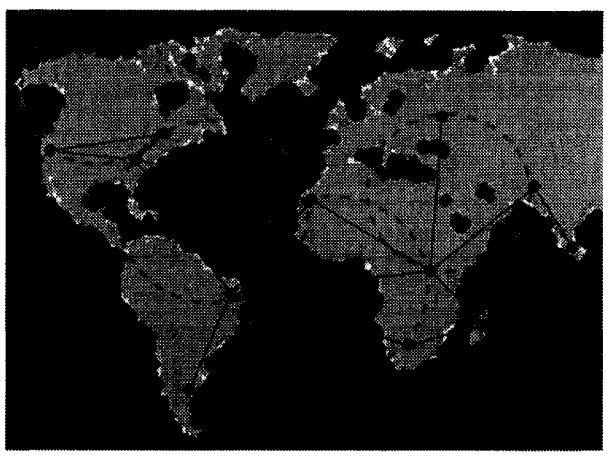


Figure 6.8 A network with 12 nodes, 15 links, and 4 circles

The 3D drawing of the network is shown as follows:

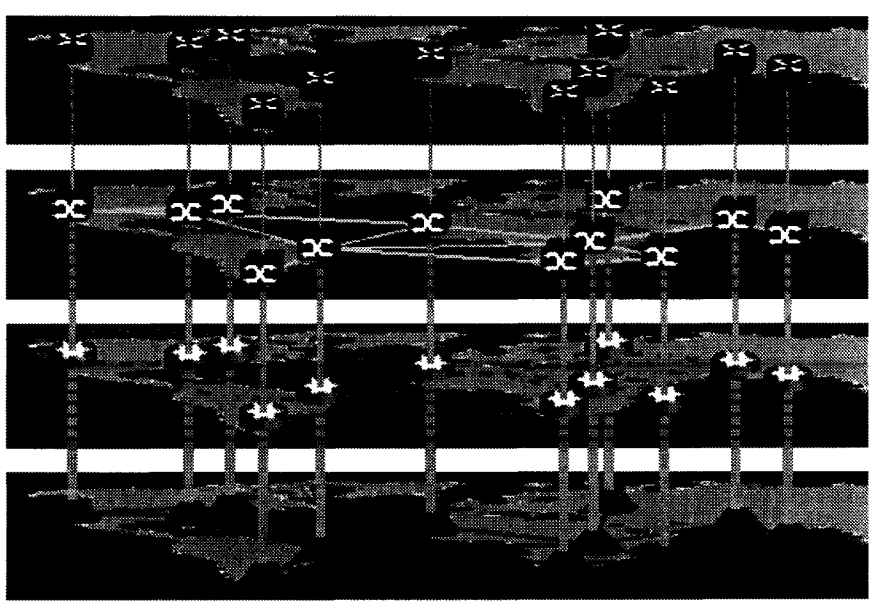


Figure 6.9 A 3D drawing of a network with *zy-buffer algorithm*

The figure above has the following features:

(1) The icons are grouped by their y-coordinates and displayed on different layers. The IP router icon, ATM icon, SONET icon, and DWDM switch icon are set on the top layer, second layer, third layer, and bottom layer, respectively.

(2) The topology of the original network is kept.

(3) Icons on the same layer are displayed in turn from the icons with larger z-coordinates to the icons with smaller z-coordinates. The front icon blocks the back icon instead of the back icon blocking the front icon. The hidden portions of the icons can not be seen. So, the function of eliminating hidden objects is realized.

(4) Every vertical line segment is cut into two pieces, one is visible and another is blocked by the plane layer, which is displayed by the dashed line segment.

This case shows that the *Depth-Height Buffer Algorithm* is working.

Case 2: *Input*: A planar network with 7 nodes, 9 links among these nodes, and 4 circles.

Output: A 3D drawing of the network with IP router icon, ATM icon, SONET icon, and DWDM switch icon.

But in this case, the *Depth-Height Buffer Algorithm* is disabled on purpose. What will happen can be seen from the following figures:

Figure 6.10 A network with 7 nodes, 9 links, and 4 circles

The 3D drawing of the network without using the *Depth-Height Buffer Algorithm*, is shown as follows:

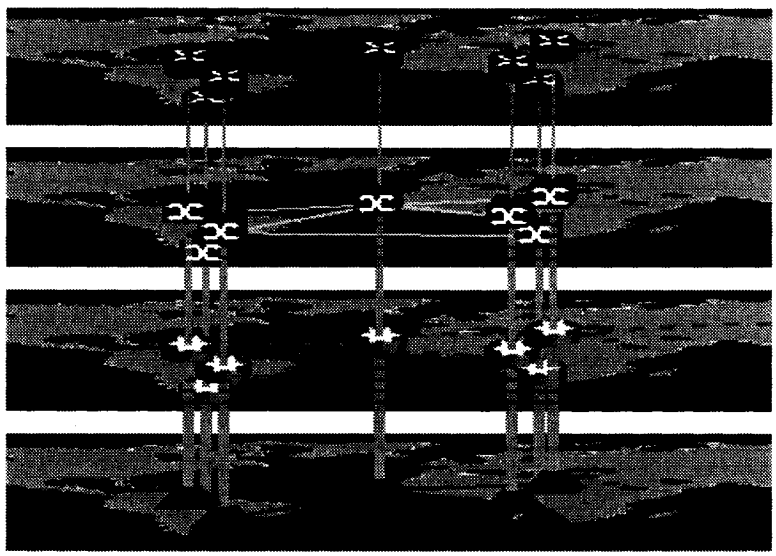


Figure 6.11 A 3D drawing of the network without the *zy-buffer algorithm*

The figure above shows that some 3D icons with larger z-coordinates were displayed in front of other icons with smaller z-coordinates. As a result, the 3D drawing looks unnatural. Therefore, if the 3D drawing of a multi-layered network is produced without using the *Depth-Height Buffer Algorithm*, the display obtained is not correct. Whereas, if the *Depth-Height*

Buffer Algorithm is employed in a 3D drawing, the display is correct as shown in Case 1. Thus the functionality of the *Depth-Height Buffer Algorithm* is verified.

6. 6 Stability of the Depth-Height Buffer Algorithm

The stability of an algorithm focuses on the output of an algorithm. Suppose that the input of an algorithm is unchanged. If the algorithm is run some times and the outputs are variable, then the algorithm is not stable. It can be verified that the *Depth-Height Buffer Algorithm* for 3D display of a network is stable.

In the first step of the *Depth-Height Buffer Algorithm*, all objects are numbered and their characteristic points (for example, the point with smallest z-coordinate or the point in the center of the objects) are identified. Though the characteristic point on an object may vary subject to the method used, the relative positions of the objects may not be changed. In step 2, since the *Depth-Height Buffer Algorithm* employs a stable sorting algorithm to sort the objects by their y-coordinates, the algorithm gives a stable order of the objects. In step 3, grouping the objects according to their y-coordinate can not change the order of the objects. For each group, sorting the objects according to their z-coordinates with a stable sorting algorithm will not produce two distinct orders of the objects. Therefore, the result matrix in step 4 in the *Depth-Height Buffer Algorithm* is fixed. Thus, the *Depth-Height Buffer Algorithm* is stable theoretically.

Depth-Height Buffer Algorithm is stable in practice. In fact, for any network, the 3D drawing of the network is the same whatever how many times the algorithm is repeated. Therefore, the *Depth-Height Buffer Algorithm* is stable.

6.7 Complexity of the Depth-Height Buffer Algorithm

Suppose there are N objects in a network. In step 1 of the *Depth-Height Buffer Algorithm*, for each object, the finding of the point with the smallest z-coordinate just needs linear time if the objects are formed by facets. Actually, if an object is formed by some curved surfaces, it can be tiled by some facets. If we employ the merge sorting or quick sorting algorithm to sort the objects by their y-coordinate, the time complexity will be $O(N \log N)$. For each group CM_i, to sort the smallest z-coordinates the time needed will be $O(|CM_i| \log |CM_i|)$ (i=1,2,...,k). Since $|CM_i| \leq n$ and $\sum_{1}^{k}(|CM_i|) = N$, the total sorting job needs $O(N \log N)$ + $\sum_{1}^{k} O(|CM_i| \log |CM_i|) \leq O(N \log N) + \sum_{1}^{k} O(|CM_i| \log N) \leq O(N \log N)$ + $O((\sum_{1}^{k}|CM_i|)\log N) = O(N \log N)$. Therefore, the overall time complexity is $O(N \log N)$.

6.8 Scalability of the Depth-Height Buffer Algorithm

Since the *Depth-Height Buffer Algorithm* has no limitation to the size of a network, it can be used to process the objects of any multi-layered network if the hardware of a computer can support it. There has yet to be a case where the *Depth-Height Buffer Algorithm* has failed to work.

Chapter 7

Conclusions

7.1 Summary of the Thesis

This thesis presents two new algorithms for 3D graph drawing and network display. The first algorithm, the *Incremental Projection Algorithm*, is a general graph drawing approach for any graphs with arbitrary vertex degree. It can represent a graph in 3D space without any edge crossing. The bends produced by the Incremental Projection Algorithm is at most two. As to the time complexity, the average time complexity of the *Incremental Projection Algorithm* is $\Omega(\sqrt{N}N)$. If there is no degree of vertices great than M, the time complexity of the *Incremental Projection Algorithm* is $O(N)$, where the integer N is the number of vertices in the graph. Therefore, the *Incremental Projection Algorithm* is an efficient algorithm for 3D graph drawing. A package of the algorithm is implemented, which shows that the *Incremental Projection Algorithm* works well. The correctness and functionality are verified by testing.

The second algorithm, the *Depth-Height Buffer Algorithm*, is designed for special multi-layered network display in 3D space. It is actually a hidden objects elimination method. With the algorithm, a package is developed to display the multi-layered communication network with IP routers, ATM, SONET, and DWDM switches. The time complexity of the algorithm is approximately $O(N \log N)$. A package of the algorithm is implemented, which shows that the algorithm works well. The correctness and functionality are verified by testing as well.

The thesis discusses the methods and techniques for 2D graph drawing as well. In bi-layered crossing reduction, the thesis presents a new method called *minimizing angle approach*. This approach can be used to reduce the crossings in a bi-layered plane graph in practice. Examples show that the *minimizing angle approach* works in some circumstances.

7.2 Further Work

The *Incremental Projection Algorithm* is a general 3D graph drawing approach. Though it is efficient to display graphs in 3D space, 3D drawing seems to lack aesthetic sense. Therefore, increasing symmetry and decreasing space occupation are two future tasks.

The projection method used in the thesis and software is just the parallel projection. Software implemented with the perspective projection method needs to be developed as an alternative to multi-layered network display. With the perspective projection method, one can change view position with ease to view the 3D graph from different positions and angles.

The line segments that connect the nodes and object icons need to be improved with 3D bars to increase the 3D effect in the package. The objects displayed need to be rendered with texture mapping so that the object is displayed with material effect.

Evaluation experiments with human judges (a task-based evaluation) are interesting works to do in the future.

The crossing reduction in a layered 2D graph is a very interesting topic. It is not only theoretically important but also has many applications in practice. Though the *minimizing angle approach* presented in the thesis proposed a new method for crossing reduction, there is still much work to be done to implement the method in practical application. For instance, the *minimizing angle approach* may get just the local minimum. How to get the global minimum is a more practical question that needs to be solved in the future. Another thing in need of investigation is the interrelation between the *minimizing angle approach* and the barycenter heuristics or the median heuristics.

Bibliography

[1] Jünger, M., Mutzel, P., *Graph Drawing Software,* Springer-Verlag, 2004.

[2] Tamara Munzner and Eric Hoffman and K. Claffy and Bill Fenner, Visualizing the Global Topology of the MBone, Proceedings of the 1996 IEEE Symposium on Information Visualization, pp. 85-92, October 28-29 1996, San Francisco, CA, 1996.

[3] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, Ioannis G. Tollis, *Graph drawing: Algorithms for the Visualization of Graph*, Prentice-Hall, Inc. 1999.

[4] J. Brown, A. McGregor, and H.-W. Braun, *Network performance visualization: Insight through animation.* in *Proceedings of the Passive and Active Measurement Workshop*, 2002.

[5] Hassan Gomaa, *Designing Concurrent, Distributed, And Real-Time Applications with UML*, Addison Wesley, New York, 2000.

[6] David D. Riley, *The Object of Java*, Addison Wesley, New York, 2002.

[7] Bertolazzi, P., Di Battista, G., Liotta, G., Mannino, C. (1994) Upward drawings of triconnected digraphs. Algorithmica 6 (12), 476–497

[8] Booth, K., Lueker, G. (1976) Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. Journal of Computer and System Sciences 13, 335–379

[9] Peter E., Qing-Wen Feng and Xuemin Lin, *Straight-Line Drawing Algorithms for Hierarchical Graphs and Clustered Graphs*. LNCS, Springer-Verlag , London UK, pp.113-128,1996.

[10] Buchheim, C., J¨unger, M., Leipert, S. (2001) A fast layout algorithm for *k*level graphs. In: J.

Marks (ed.) Graph Drawing '00, Lecture Notes in Computer Science 1984, Springer-Verlag, 229–240

[11] Chiba, N., Nishizeki, T., Abe, S., Ozawa T. (1985) A linear algorithm for embedding planar graphs using PQ-trees. Journal of Computer System Science, 30 (1), 54–76.

[12] Cook, W. J., Cunningham, W. H., Pulleyblank, W. R., Schrijver, A. (1998) Combinatorial Optimization. John Wiley & Sons

[13] Dahlhaus, E. (1998) A linear time algorithm to recognize clustered graphs and its parallelization. In: C. L. Lucchesi and A. V. Moura (eds.) Latin '98, Lecture Notes in Computer Science 1380, Springer-Verlag, 239–248

[14] Eades, P., Wormald, N. (1994) Edge crossings in drawings of bipartite graphs. Algorithmica 11, 379–403

[15] Feng, Q. W., Cohen, R. F., Eades, P. (1995) Planarity for clustered graphs. In: P. Spirakis (ed.) Algorithms – ESA '95, Lecture Notes in Computer Science, 979, Springer-Verlag, 213–226

[16] Fößmeier, U., Kaufmann, M. (1996) Drawing high degree graphs with low bend numbers. In: F. J. Brandenburg (ed.) Graph Drawing '95, Lecture Notes in Computer Science 1027, Springer-Verlag, 254–266

[17] Garey, M. R., Johnson, D. S. (1983) Crossing number is NP-complete. SIAM J. Algebraic Discrete Methods 4, 312–316

[18] Gansner, E. R., Koutso.os, E., North, S. C., Vo, K. P. (1993) A technique for drawing directed graphs. IEEE Transactions on Software Engineering 19, 214–230

[19] Garg, A., Tamassia, R. (1995) On the computational complexity of upward and rectilinear planarity testing. In: R. Tamassia and I. G. Tollis (eds.) Graph Drawing '94, Lecture Notes in Computer Science 894, Springer-Verlag, 286–297

[20] Healy, P., Kuusik, A. (1999) The vertex-exchange graph: a new concept for multi-level crossing minimization. In: J. Kratochv' (ed.) Graph Drawing '99, Lecture Notes in Computer

Science 1731, Springer-Verlag, 205–216

[21] Hopcroft, J., Tarjan, R. E. (1974) E.cient, *Planarity testing.* Journal of the ACM 21, 549–568.

[22] Jünger, M., Mutzel, P. (1997) 2-layer straight line crossing minimization: performance of exact and heuristic algorithms. Journal of Graph Algorithms and Applications 1, 1–25

[23] Lempel, A., Even, S., Cederbaum, I. (1967), *An algorithm for planarity testing of graphs.* Theory of Graphs: International Symposium: Rome, July 1966, Gordon and Breach, New York, 215–232

[24] Mehlhorn K., Mutzel, P. (1996) On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. Algorithmica 16, 233–242

[25] Patrignani, M. (2001) On the complexity of orthogonal compaction. Computational Geometry: Theory and Applications 19 (1), 47–67

[26] Reingold, E., Tilford, J. (1981) Tidier drawing of trees. IEEE Transactions on Software Engineering **7**, 223–228

[27] Sugiyama, K., Tagawa, S.,Toda, M. (1981) Methods for visual understanding of hierarchical system structures. IEEE Transactions on Systems, Man, and Cybernetics 11, 109–125

[28] Sugiyama, K., Misue, K. (1991), *Visualization of structural information: automatic drawing of compound digraphs.* IEEE Transactions on Systems, Man, and Cybernetics 4 (21),876–893

[29] Supowit, K. J., Reingold, E. M. (1983) The complexity of drawing trees nicely. Acta Inform. 18, 377–392

[30] Tamassia, R. (1987) On embedding a graph in the grid with the minimum number of bends. SIAM Journal on Computing 16 (3), 421–444

[31] Tamassia, R., Di Battista, G., Batini, C. (1988) Automatic graph drawing and readability of diagrams. IEEE Transactions on Systems, Man, and Cybernetics, 18, 61–79

[32] Waddle, V., Malhotra, A. (1999) An E log E line crossing algorithm for leveled graphs. In: J. Kratochv′ (ed.) Graph Drawing '99, Lecture Notes in Computer Science 1731, Springer-Verlag,

59–70

[33] Bertolazzi, P., Di Battista, G., Didimo, W. (1998) Quasi-upward planarity. In: S. H. Whitesides (ed.) Graph Drawing '98, Lecture Notes in Computer Science 1547, Springer-Verlag, 15–29

[34] Bertolazzi, P., Di Battista, G., Mannino, C., Tamassia, R. (1998) Optimal upward planarity testing of single-source digraphs. SIAM Journal on Computing, 27, 132–169

[35] Brandes, U., K¨opf, B. (2002) Fast and simple horizontal coordinate assignment. In: P. Mutzel, M. J¨unger, S. Leipert (eds.) Graph Drawing '01, Lecture Notes in Computer Science 2265, Springer-Verlag, 31–44

[36] Batini, C., Nardelli, E., Tamassia, R. (1986) A layout algorithm for data flow diagrams. IEEE Transactions on Software Engineering SE-12 (4), 538–546

[37] T.Fruchterman and E. Reingold, *Graph Drawing by Force-directed Placement*, Softw.-Proct. Exp., 21,No.11,1120-1164,1991.

[38] Eades, P. (1984). A heuristic for graph drawing. Congressus Numerantium, 42, 149-160.

[39] I. Bruss and A. Frick, Fast Interactive 3D Visualization, Proc. of Workshop GD'95, LNCS1027, Springer-Verlag 1995, pp 99-110.

[40] I. Cruz and J. Twarog, 3D Graph Drawing with Simulated Annealing, Proc. of Workshop GD'95, LNCS1027, Springer-Verlag 1995, pp 162-165.

[41] A. Papakostas, Ioannis G. Tollis, Incremental Orthogonal Graph Drawing in Three Dimensions, GD '97, Rome, Italy, September 18-20, 1997, Proceedings. LNCS1353, Springer 1997, pp 52-63.

[42] D. Harel and Y. Koren. Graph drawing by high-dimensional embedding. Proc. Graph Drawing, 2002.

[43] Peter Rodgers. *graph drawing techniques for geographic visualization*, In Alan MacEachren, Menno-Jan Kraak, and Jason Dykes, editors, *Exploring geovisualization*. Elsevier, January 2004.

[44] Kishalay Kundu, Chad Sessions, Marie desJardins, and Penny Rheingans, *Three-dimensional visualization of hierarchical task network plans*, Proceedings of the Third International NASA Workshop on Planning and Scheduling for Space, Houston, Texas, October 27-29, 2002.

[46] Brown, J., A. McGregor, and H-W. Braun. Network performance visualization: insight through animation. *Proceedings PAM2000 Passive and Active Measurement Workshop,* Hamilton, New Zealand, pp. 33-41, Apr. 2000.

[47] Eleftherios E. Koutsofios, Stephen C. North, Russell Truscott, Daniel A. Keim *Visualizing large-scale telecommunication networks and services*. Proceedings of IEEE Visualization '99, 457-461.

[48] Wilhelm Barth, Micheal Jünger, and Petra Mutzel, *Simple and Efficient Bilayer Cross Counting*,Lecture Notes in Computer Science, Springer Verlag (2002) 130 - 141.

[49] Michael Junger, Petra Mutzel, *2-layer straight line crossing minimization*, Journal of Graph Algorithms and Applications 1, 1-25.

[50] K. Sugiyama, S. Tagawa, and M. Toda, *Methods for visual understanding of hierachical system structures*. IEEE Transactions on Systems, Man and Cybernetics 11(1981)109-125.

[51] Xiao Yu Li and Matthias F. Stallmann, *New bounds on the barycenter heuristic for bipartite graph drawing*, Information Processing Letters, Vol.82, Issue 6, 30 June 2002, Pages 293-298.

[52] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, Ioannis G. Tollis, *Graph Drawing:Algorithms for the Visualization of Graphs*, Prentice Hall, 1999, ISBN 0-13-301615-3.

[53] Michael Junger, Petra Mutzel, *Graph Drawing Software*, Springer-Verlag New York, 2003, ISBN: 3540008810.

[54] P.Eades, B. McKay, and N. Wormald. *On an edge crossing problem*, In Proceedings of the 9th Australian Computer Science Conference, pages 327-334. Australian National University,1986.

[55] M. R. Garey, D. S. Johnson, *Crossing number is NP-complete*, SIAM Journal on Algebraic and Discrete Methods, 4 (1983), 312-316.

[56] ILOG JViews 5.5, *Graph Layout User's Manual*, 2002.

[57] P. Eades and D. Kelly, *Heuristics for Reducing Crossing in 2-Layered networks*, Ars Combin., 21.A, 89-98,1986.

[58] Jünger, M., Mutzel, P., 2-layer Straightline Crossing Minimization, Journal of Graph Algorithms and Applications, 1, no.1, 1-25,1997.

[59] F. S. Hill, *Computer Graphics*, Prentice-Hall,Inc., Upper Saddle River, New Jersey, 2001.

[60] Donald Hearn & M. Pauline Baker, *Computer Graphics*, Prentice-Hall,Inc., Englewood Cliffs, New Jersey, 1986.

[61] Phong, B-T, *Illumination for Computer Generated Images*, Communications of the ACM 12:311-317, 1975.

[62] Blinn, J.E. *Model of light reflection for computer synthesized pictures*, computer Graphics 11(2), 1977.

[63] Coffman, E. G., Graham, R. L. (1972) Optimal scheduling for two processor systems. Acta Informatica 1, 200–213.

[64] A.Papakostas, I.G. Tollis, Algorithm for Incremental Orthogonal Graph Drawing in Three Dimensions, Vol.3,no.4,pp.88-115(1999).