VISUALIZING THREE-DIMENSIONAL GRAPH DRAWINGS

SEBASTIAN HANLON
Bachelor of Science, University of Lethbridge, 2004

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

VISUALIZING THREE-DIMENSIONAL GRAPH DRAWINGS

SEBASTIAN HANLON

Approved:

Signature	*Date*
Supervisor: Dr. Stephen Wismath	
Committee Member: Dr. Howard Cheng	
Committee Member: Dr. Timothy Pope	
Committee Member: Dr. Daniela Sirbu	
External Examiner: Dr. David Bremner	
Chair, Thesis Examination Committee: Dr. Marc Roussel	

Abstract

The GLuskap system for interactive three-dimensional graph drawing applies techniques of scientific visualization and interactive systems to the construction, display, and analysis of graph drawings. Important features of the system include support for large-screen stereographic 3D display with immersive head-tracking and motion-tracked interactive 3D wand control. A distributed rendering architecture contributes to the portability of the system, with user control performed on a laptop computer without specialized graphics hardware. An interface for implementing graph drawing layout and analysis algorithms in the Python programming language is also provided. This thesis describes comprehensively the work on the system by the author—this work includes the design and implementation of the major features described above. Further directions for continued development and research in cognitive tools for graph drawing research are also suggested.

Acknowledgments

Firstly, I would like to thank my advisor, Dr. Stephen Wismath, for encouraging me to pursue graduate studies in computer science—without him, this thesis would not exist. He has furthermore been supportive and enthusiastic throughout the course of my studies; one could not ask for a better supervisor. I also wish to thank the rest of my thesis committee and the external examiner for their advice and quality assurance efforts.

I gratefully acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) for providing major funding in support of this research.

Thanks are also extended to my friends and colleagues for providing support, advice, and encouragement as I completed this thesis; particularly Susan Beaver, Paul Dawson, Barry Gergel, David Lenz, Brian McFee, Elspeth Nickle, Ed Pollard, Tiffany Proudfoot, Patrick Stewart, and Katrina Templeton. Thanks as well to the development communities supporting the Twisted and wxPython packages.

Special thanks are due to my very good friend Kim Hansen for shoring up my mental stability and motivation throughout the research and writing process.

Finally, I thank my parents Vincent and Teresa Hanlon and my brother Matt for their continual and loving support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since the early 1990s, the increasing power and diminishing cost of computer processors (and the related availability of specialized 3D graphics acceleration hardware) has brought high-performance graphics realization capability to commodity computing platforms. Hardware capable of supporting dynamic visualization of detailed virtual scenes and objects is available at a low cost, and standard APIs (including OpenGL) allow applications to be written and deployed in a portable manner.

While the development of graphics technology has been driven in large part by the consumer market for electronic entertainment, the 3D capability so widely used to sell games also benefits researchers and developers in less commercial disciplines. Research into *information visualization* techniques has been stimulated by the capability of computer systems to facilitate interactive exploration of data sets, with users issuing queries and having the results displayed in graphical form with a minimum of waiting.

The popularity of 3D graphics displays has also stimulated research in the field of three-dimensional *graph drawing*. In two dimensions, only the class of *planar graphs* can be drawn without edge crossings. This restriction is removed in three dimensions—all graphs can be drawn in three dimensions without edge crossings, so considerable research is fo-

cused instead on reducing the volume required (determined asymptotically) to draw various classes of graphs.

1.1 Motivations

Software tools designed specifically for working with graph layouts in three dimensions are relatively rare. Most are focused on implementing particular algorithms or classes of algorithms. For interactive experimentation with layout ideas or communication of visual ideas between individual researchers, it is often easier to use limited physical models (for example, Styrofoam balls and wire to represent vertices and edges) than to write formal implementations of prototype layout algorithms or to use a two-dimensional interface to manipulate a three-dimensional drawing.

The idea behind the GLuskap system here described is to develop an interface for interacting with three-dimensional graph drawings that uses principles of interactive systems and data visualization to best effect. We wish to build a tool that provides an effective, naturalistic interface on the order of Styrofoam balls and pipe cleaners while retaining the advantages of digital modeling: virtually unlimited resources, zero-cost data replication and communication, and the ability to automate and extend nearly any aspect of the process.

1.2 The GLuskap VR System

The GLuskap system for interactive three-dimensional graph drawing has been developed at the University of Lethbridge under the direction of Dr. Stephen Wismath. Building on the existing GLuskap software product written in the Python programming language, this system integrates the application software with a specialized hardware system to sup-

port 3D graph drawing research activities in a large-screen (6.0' $\times$ 8.0', 1.83m $\times$ 2.44m) rear-projected stereographic environment with immersive features. Both head-tracking and wand-tracked interaction are supported.

The system is reasonably portable to maximize efficient use of space and also increasing potential uses of the system in research, visualization, and teaching roles. To achieve this portability requirement, a networked rendering architecture is supported by the GLuskap software and is implemented using a laptop and two small form factor computers connected over Ethernet. The computing resources, motion tracking equipment, and projectors and optical equipment (without the large screen) pack down into three travel cases; the large rear projection screen and frame requires two additional packing tubes for storage or transport. The system can also be adapted to work with a lenticular front-projection screen if required.

In the course of my thesis work, I have been solely responsible for the development and maintenance of the GLuskap software program as it has been enhanced to support the visualization and interaction technologies described in Chapters 3 and 4. My work has also included the design and implementation of the hardware system that supports the large-screen immersive 3D interface.

1.3 Applications Of 3D Graph Drawing

The algorithms and approaches of three-dimensional graph drawing can be put to use in several practical areas. As a visualization technique for relational data, 3D drawing techniques are of obvious utility when working with data sets involving multi-dimensional data. These include, but are not limited to, scenarios where data objects are associated with positional attributes that can be translated into the virtual space directly.

Large data set visualization More generally, extending the display media into three dimensions allows users to effectively interpret and manipulate larger data sets than would otherwise be possible, especially in an interactive situation where the user can alter the perspective of the data display. 3D organizational techniques like *cone trees* (introduced by Robertson *et al.* [RMC91]) and Munzner's H3 [Mun97] have been developed to maximize efficiency for displaying large quantities of data.

Displaying data in virtual environments provides the user with a context in which to integrate large data sets. Much like visiting a new building or town, the brain can build mental maps using the spatial relationships between features and landmarks.

Space minimization for wire-routing As advances in VLSI construction processes enable circuits to be built in multiple layers, three-dimensional graph drawing techniques (especially in the area of orthogonal drawings) have become relevant in this area. The connections between individual components on a chip die are easily represented as an undirected graph. Research in this area is typically focused on finding layouts which may be constrained in one or more dimensions (preserving the overall planar nature of the chip package) while minimizing the length of individual wire connections and the overall surface area and volume of the circuit.

1.4 Structure Of This Document

The remainder of this thesis is divided into five chapters. The first part of Chapter 2 introduces the field of three-dimensional graph drawing and discusses some results from the literature pertaining to the area of graph drawing algorithms which GLuskap is designed to work with. Next, a brief treatment of human visual perception and interaction principles is given. Attention is focused on those characteristics which are relevant to the design

and implementation of the GLuskap interactive interface. Finally, different types of existing software for three-dimensional graph drawing are surveyed. Some related products, including the antecedent version of the GLuskap software, are reviewed briefly.

Chapter 3 describes concepts considered in the design of the GLuskap system. Particular requirements for data visualization techniques incorporated directly into the software are described. The accommodations required to maximize the portability of the system, particularly in producing stereographic output on a large projection screen, are also covered.

Detailed coverage of the construction and implementation of the system is provided in Chapter 4. An in-depth examination of the component hierarchy, control flow, and data-flow architecture of the GLuskap software is given here; this includes details of the implementation of the networked stereographic rendering system and the mechanics of data flow for handling the Flock of Birds motion tracking system. All aspects of the large-screen interactive user interface are described, along with the programming interface for plug-in scripting of drawing algorithms in Python. The chapter concludes with a discussion of software engineering and development techniques used in the construction of the GLuskap application software.

In the fifth chapter, the work is evaluated in retrospect. The features of the GLuskap system are compared with other software products for interactive three-dimensional graph drawing, including the previous version of the GLuskap software. The visualization and interaction techniques implemented in the current system are examined in comparison to alternative strategies and pertinent ideas from the related literature.

Chapter 6 presents some concluding remarks on what has been accomplished through the development of the GLuskap system and the accompanying research and documentation, including this thesis. Ready opportunities for additional work to extend the usefulness of this product, as well as directions for continued research into methods for providing cog-

nitive support to researchers in graph drawing, are described here.

Chapter 2

Background

2.1 Graph Drawing

The study of graphs is fundamentally concerned with the relationships between distinct entities. Graphs can represent thought processes, bureaucratic organizational hierarchies, networks of various kinds, state diagrams for automata, procedural flowcharts, digital logic circuits, and many other concepts. In all of these contexts, it is their relational nature that makes the graph data structure applicable. Here we will primarily concern ourselves not with the domain-specific uses of graphs, but instead consider graphs independent of any particular context in which they may be used.

Abstracted from any deeper semantic content, we call the entities *nodes* or *vertices,* and the relations between them *edges.* Behzad and Chartrand offer the following formal definition of a graph:

> A *graph* G (sometimes called an *ordinary graph)* is a finite, non-empty set V together with a (possibly empty) set E (disjoint from V) of two-element subsets of (distinct) elements of V. Each element of V is referred to as a *vertex* and V itself as the *vertex set* of G; the members of the *edge set* E are called *edges*. By

an *element of a graph* we shall mean a vertex or an edge. [BC71]

For each edge $e = \{u, v\}$, e is said to be *incident* to u and v. u and v are said to be *adjacent*, and *joined* by e. The *degree* of a vertex v is the number of edges incident to v. The graph $K_n = (V, E)$ with $n = |V|$ vertices, where each vertex u is adjacent to every other vertex v $(\forall u, v \in V)(u \neq v \rightarrow \{u, v\} \in E)$ is called the *complete graph*.

A variant data structure, the *directed graph,* is formed from a set V of vertices and a set E of ordered pairs (rather than unordered two-element subsets) of vertices. In this kind of graph, the edge $e = (u, v)$ has *source* u and *target* v. Though there is substantial research on, and many applications for, directed graphs, in this thesis we will assume that graphs are undirected except where specified.

Within this document, we will use the term *vertices* exclusively to avoid ambiguity with the components of the GLuskap networked display system as described in Sections 3.1.3 and 4.2.3.

An ordinary graph can be represented equivalently as a list of vertices and a list of edges, an adjacency list, or a two-valued adjacency matrix. Figures 2.1, 2.2, and 2.3 are equivalent representations of the same graph.

These representations are precise and unambiguous, and it is clear to see how they form the basis for common data structures used to represent graphs for processing by algorithms and applications. They are less accessible, though, to comprehension by the human reader. We therefore introduce a fourth representation: an example of a *graph drawing* equivalent to the three previous representations is given in Figure 2.4.

The advantages of this visual representation in a cognitive context are obvious. By representing the vertices as circles and drawing lines connecting adjacent vertices, we produce a *node-link diagram*. All the information present in Figures 2.1, 2.2, and 2.3 is preserved, and the relationships between individual vertices are clearly visible—as is the overall struc-

V: $v_1, v_2, v_3, v_4, v_5, v_6$
E: $\{v_1, v_2\}, \{v_1, v_3\}, \{v_3, v_2\}, \{v_3, v_4\}, \{v_4, v_5\}, \{v_4, v_6\}, \{v_5, v_6\}$

Figure 2.1: Example Graph: Vertex and Edge List

v_1: v_2, v_3
v_2: v_1, v_3
v_3: v_1, v_2, v_4
v_4: v_3, v_5, v_6
v_5: v_4, v_6
v_6: v_4, v_5

Figure 2.2: Example Graph: Adjacency List

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

Figure 2.3: Example Graph: Adjacency Matrix

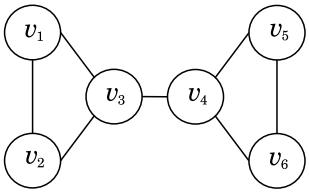

Figure 2.4: Example Graph: Node-Link Diagram

ture of the graph, that of two groups $\{v_1, v_2, v_3\}$ and $\{v_4, v_5, v_6\}$ connected by the single edge $\{v_3, v_4\}$.

Ware [War04, p. 23] claims that data, in general, can be divided into entities and relationships; entities are the objects we are interested in and their context is formed of the relations between them. What constitutes an *object* is subject to definition on a case by case basis. For example, a hockey player is an object; so too are a hockey team and the city they play in. In software engineering, a software program, a specific use case, and a data table may all be conceptual objects at different levels of the design process. As long as information can be structured into objects and connections between them, this approach is valid. Object-oriented programming techniques and the languages that support them represent a formalization of this paradigm, though in many cases the relationships between objects are not well-defined in programming language structures.

Even if we stop short of attempting to fit *all* possible data sets into the object-relationship paradigm, it is clear that a great number of real-world scenarios lend themselves to representation and analysis as graphs. Graph drawing therefore provides a useful infrastructure for the visualization and cognitive comprehension of these data. Continuous lines between objects represent the abstract idea of "connectedness" in a powerful way, creating conceptual linkages that dominate those induced by simple proximity or similarity of colour, size, or shape [PR94].

2.1.1 2D Graph Drawing

When used for data visualization, the elements of a graph drawing form a *visual grammar* (that is, a framework for interpreting the symbols of a diagram) capable of expressing relationships both simple and complex. The overwhelming majority of research in the area of graph drawing concerns two-dimensional drawings in the plane. This is the most natural

context for communicating ideas—we are surrounded by written and illustrated materials in planar forms. As we will discuss in Section 2.2.1, the human visual system is well-adapted to extracting information from two-dimensional representations, the three-dimensionality of the natural world notwithstanding.

What makes a good drawing algorithm? Graph drawing algorithms can be assessed as to the "comprehensibility" of the drawings they produce—a "good" drawing should be easy to read accurately. This is obviously a difficult concept to define precisely, but Sugiyama [Sug02] has compiled a list of drawing rules based on the criteria of Batini *et al.* [BFN85] produced by the analysis of diagrams constructed by human designers in actual use. Sugiyama divides this list into *structural rules* and *semantic rules*.

Structural rules are concerned with the expression of graph-theoretic properties, including the minimization of edge crossings and edge bends, the accurate representation of symmetries and isomorphic subgraphs, and the minimization of total drawing area and total edge length. Semantic rules are imposed by the user or attributed properties of the graph elements, including rules about placement of specific vertices relative to each other or relative to the drawing area (for example, placing specified vertices near the center of the drawing).

From this perspective, the task of a graph drawing algorithm is to solve a priority ranked set of optimization problems derived from some set of these structural and semantic rules.

Some algorithms are optimized for, or even restricted to, drawing specific classes of graphs; for example, there is a great deal of literature on tree drawing algorithms. Other algorithms make compromises in an attempt to maximize their suitability for large classes of graphs. Many physics-based or energy-minimization models fall into this category; for examples, see [FR91, DH96, Noa03].

Di Battista *et al.* [BETT99] and Kaufmann and Wagner [KW01] provide comprehensive

coverage of the established methods and algorithms for drawing a number of classes of graphs in two dimensions.

Geometric results Two-dimensional drawings motivate the graph theory study of *planarity*—whether or not a given graph can be drawn in the plane with no edge crossings. In fact, Fáry [Fár48] establishes that all planar graphs can be drawn using *straight line* edges without crossings. If vertices are restricted to lie on integer grid points, then it becomes possible to assess drawing algorithms on the basis of the area of the grid required to draw the graph. The area is typically calculated relative to the number of vertices in the graph (n), and represents the size of a rectangular bounding box with sides parallel to the coordinate axes which completely encloses the graph drawing.

De Fraysseix *et al.* [dFPP88, dFPP90] and Schnyder [Sch90] independently prove that all planar graphs can be drawn on a grid of size $O(n^2)$. This bound is tight (*i.e.* $\Theta(n^2)$) for some families of planar graphs. For other classes of planar graphs, the area can be improved asymptotically—for example, rooted trees of constant degree have a straight-line downward grid drawing in area $O(n \log^2 n)$ [SKC96], and complete, AVL, and Fibonacci trees can be drawn under the same conditions in area $O(n)$ [CP95, Tre96].

2.1.2 3D Graph Drawing

The capability of computer graphics systems to create and display pseudo-realistic synthetic objects and virtual three-dimensional spaces is well established. Three-dimensional display techniques are used in industry for visualization and interactive design. The extension of graph drawing in computer science to three dimensions is therefore a natural progression.

We continue from the previous topic by considering *three-dimensional integer grid drawings*. The fundamental requirements that vertices be placed on integer grid points

and that edges must not cross are preserved. Rather than measuring the area of a two-dimensional bounding box of a drawing, we must now determine the size of a drawing as the volume of a rectangular prism (with all edges parallel to the coordinate axes) large enough to enclose the entire drawing.

Integer grid drawings Cohen *et al.* [CELR97] present the "moment curve" algorithm, which provides a straight-line crossing-free integer grid drawing in three dimensions of any graph, in volume $O(n^3)$. They additionally prove that the complete graph K_n has the volume bound $\Omega(n^3)$, indicating that the bound is tight for unrestricted ordinary graphs. Better results can be obtained for restricted classes of graphs; in particular, they prove that complete bipartite graphs can be drawn in volume $O(n^2)$.

Calamoneri and Sterbini [CS97] follow up Cohen *et al.* by establishing a lower volume bound of $\Omega(n\sqrt{n})$ for k-colourable graphs (for fixed $k \geq 2$), and provide a three-dimensional drawing algorithm for 2-, 3-, and 4-colourable graphs in volume $O(n^2)$. They hypothesize that *all* k-colourable graphs can also be drawn in volume $O(n^2)$. Pach *et al.* [PTT97] prove this conjecture, and also show that $O(n^2)$ cannot be beaten, closing the gap between the lower bound of Calamoneri and Sterbini and the established upper bound for this class of graphs.

A linear volume result in three dimensions ($O(n)$) is shown by Felsner *et al.* [FLW03] for all *prism-drawable* graphs. These are graphs which can be drawn without crossings by placing vertices along the spines of a regular three-dimensional triangular prism, where the edges are constrained to lie on the facets of the prism. All outerplanar graphs are prism-drawable, though not all planar graphs are prism-drawable. Dujmović and Wood [DW04] significantly show that all planar graphs can be drawn with an upper bound of $O(n\sqrt{n})$ volume. A gap remains between this result and the trivial lower bound of $\Omega(n)$ for planar graphs which have $O(n)$ vertices and $O(n)$ edges; this is an important open problem in the

field of three-dimensional graph drawing.

Bent edge results Improved volume bounds can also be obtained if we relax the straight line edge constraint. Morin and Wood [MW04] define a *polyline grid-drawing* as having vertices placed at integer grid points and edges represented as a sequence of straight line segments. The bend points of these polyline edges are also constrained to occupy integer grid points. Discrete polyline edges must not intersect each other. Morin and Wood define a *b-bend drawing* to be a polyline drawing with at most b bends per edge—note that a straight-line drawing is exactly a 0-bend drawing.

Even if we allow an unlimited number of bends, Bose *et al.* [BCMW04] show that the $\Omega(n^2)$ lower bound of Pach *et al.* [PTT97] still holds for all grid drawings of K_n. Dyck *et al.* [DJN+04] further demonstrate that this lower bound is achievable with a maximum of two bends per edge. They also propose a 1-bend drawing algorithm, but without asymptotic improvement over the 0-bend $O(n^3)$ case.

Morin and Wood [MW04] develop a novel algorithm for 1-bend drawings, lowering the upper bound to $O(n^3/\log^2 n)$. Devilliers *et al.* [DEL+05] recently developed a new algorithm based on the Morin and Wood construction that reduces the upper volume bound for 1-bend drawings further still to $O(n^2\sqrt{n})$, though a gap still remains between this result and the $\Omega(n^2)$ lower bound.

Orthogonal drawings There has been considerable investigation of *orthogonal grid drawings* in two dimensions [BETT99, Ch.5], in which edges are constructed from a sequence of horizontal and vertical line segments. This field of study is motivated in part by application to component placement issues and circuit routing in VLSI design [Lei80, KvL85] as well as aesthetic preference for rectilinearity in diagram construction [BRT84, BFN85, Tam85, NT90].

Three-dimensional orthogonal graph drawings can be considered a subset of polyline grid drawings, where edge segments are constrained to grid lines (parallel to one of the coordinate axes at integer grid intervals). These have been studied in depth and several results pertain.

If vertices are represented by points in three dimensions, it is trivial to show that only graphs with maximum degree six can be drawn orthogonally without crossings: there are only six "faces" on which incident edges can contact each vertex. Orthogonal drawings of graphs of higher degree can still be made without crossings if vertices are represented as three-dimensional boxes or line segments spanning multiple grid points [BSWW99, PT99].

A lower bound for the volume of three-dimensional orthogonal drawings of arbitrary degree (with vertices represented by boxes) is established by Biedl *et al.* [BSWW99]. This lower bound for drawings of K_n with an arbitrary number of bends per edge is $\Omega(n^2\sqrt{n})$; a matching upper bound is also shown if 3 or more bends per edge are permitted.

Eades *et al.* [ESW00] describe a set of 5 drawing algorithms for three-dimensional orthogonal graph drawings with maximum degree six. If 7 bends per edge are allowed, a drawing with volume $O(n\sqrt{n})$ is produced. Similarly, the 6-bend drawing requires volume $O(n^2)$, the 5-bend drawing $O(n^2\sqrt{n})$, and for 3 bends per edge, $O(n^3)$ volume is used. Their fifth algorithm provides a 3-bend drawing in volume $O(n^2)$, but only for graphs of maximum degree 4.

The $O(n\sqrt{n})$ volume for the 7-bend case is tight: Kolmogorov and Barzdin [KB67] and Rosenberg [Ros83] prove a lower bound of $\Omega(n\sqrt{n})$.

Papakostas and Tollis [PT99] provide a linear time algorithm for a 3-bend orthogonal drawing of graphs of maximum degree six in volume $O(n^3)$, with a significant constant factor improvement over the 3-bend layout of [ESW00]. They also provide a two-bend orthogonal drawing algorithm for graphs of arbitrary degree (using solid boxes to represent vertices). Both algorithms are incremental such that vertices are added to the graph on-line.

Nomura *et al.* [NTU05] show in a recent paper that all outerplanar graphs with maximum degree six and with no triangles have three-dimensional orthogonal drawings without bends.

2.2 Visualization and Interaction

While a thorough treatment of human visual perception, data visualization practices, and interactive systems cannot be accomplished within the scope of this thesis, this section will attempt to introduce concepts relevant to the work presented in later chapters. Readers are referred to Ware's *Information Visualization* [War04] for a more comprehensive study of these topics.

2.2.1 Basics of Perception

An effective model for studying human visual perception is presented by Gibson [Gib86]. The environment around us can contain multiple sources of light and countless objects that absorb, reflect, and scatter light. In obtaining information about that environment and the objects within it, though, an individual observer is restricted to the rays of light which arrive at a single viewpoint from all directions. The information contained in the structure of these light rays and the way in which they change over time constitutes what Gibson calls the *ambient optical array.* This limited subset of the total light rays present in the environment nonetheless allows the observer to extract useful information about their surroundings.

Ware [War04] suggests as a metaphor the projection of a portion of the ambient optical array into two dimensions as though it were observed through a rectangular pane of glass. Isolated in this way, a portion of the array can be modeled and simulated by computer graphics techniques, allowing us to create virtual spaces with varying levels of naturalistic behaviour and appearance.

Figure 2.5: *La Trahison des Images* (Magritte, 1929)

The ability of the brain to effectively interpret a two-dimensional projection in this way is shown by our ability to recognize objects portrayed in photographs, illustrations, and even line drawings. Magritte's famous painting *La Trahison des Images* ("The Treason of Images") (Figure 2.5) provides a clever yet succinct demonstration of this principle—as the caption ("This is not a pipe") points out, the image is not a pipe (rather, it is *of* a pipe), yet the object is easily recognized.

Though the evolutionary background for the human visual system is entirely based on extracting useful information about objects in the nearby environment, the adaptive neural mechanisms which have developed to perform these perceptive tasks are capable of effectively interpreting information presented with varying degrees of abstraction. It is this capability that makes the entire field of visual design and information visualization possible.

It is important to keep in mind that most of the perceptual mechanisms of the visual system operate pre-consciously. We consciously perceive, recognize, and interact with objects, not patches of light in the visual field. Even such fundamental concepts as colour and brightness are pre-processed heavily. Designed by evolution to ascertain the properties of objects from the characteristics of their surfaces by way of reflected light, and to function

effectively in a wide variety of circumstances, the human visual system[1] discards most information about the quality, quantity, or absolute wavelength of light in the environment.

The shape of objects The three-dimensional shape of individual objects is understood to be perceived using a combination of factors. The silhouette of the object provides a general indication of the shape, and Marr [Mar82] suggests that the visual perception system incorporates certain assumptions that allow the three-dimensional shape of objects to be extrapolated from silhouette information.

The way that light falls on and is reflected by objects also reveals their shapes. As mentioned, light in real environments is created and altered by reflection and transmission countless times before it reaches the observer. Ware [War04, Ch.2] describes a simplified model of surface illumination and shading that is useful in understanding how light interacts with many common types of surface.

In constructing visual simulations, use of simplifications like this may actually be more productive than attempting to model all possible light interactions in a scene, as it mirrors our understanding of the assumptions the visual system of the brain makes in interpreting the surface characteristics of objects. This model identifies four factors that influence the amount and quality of the light reaching the observer from a point on an object's surface:

Lambertian shading Lambertian scattering takes place when light penetrates the surface of an object and interacts with the pigments contained therein. Some wavelengths of the light are absorbed, and others are reflected in all directions from the surface of the object. This produces the characteristic colour of the pigmented surface and the smooth shading patterns typical of simple three-dimensional graphics displays, which reveal the general curvature and angular features of objects and surfaces. The

[1] And the visual systems of most, if not all, animals as well.

amount of light reflected in this way depends on the orientation of each point on a surface relative to the light source.

Specular highlights Specular highlights are caused by light from some source being reflected directly by the surface without reaching the pigments contained within. This light is reflected at, or very close to, the angle of incidence (like a mirror), and has the colour of the illuminant. Specular reflections can highlight minor surface features that "catch" the light, but these may require direct illumination from a specific angle or range of angles to be visible.

Ambient light Ambient light models the character of the light that has been reflected by surrounding objects, rather than coming directly from a specific light source. In computer graphics simulations ambient illumination is often simplified to a constant.

Cast shadows Objects cast shadows on each other and on themselves, in the opposite direction from an illuminant. Shadow casting can reveal minor details, similar to specular highlights, and also contributes to perception of the relative size and positioning of multiple objects, though it is dependent on the position and orientation of the light source.

Similar assumptions allow the visual perception system of the brain to infer *shape-from-shading*. It integrates shape-from-shading data with the silhouette contour of the object. Surface textures can also provide shape cues, especially if the textures form linear or grid-like patterns. The use of contours on geographic maps to indicate terrain features and elevation is a common example.

Three-dimensional scenes As mentioned earlier, objects move through the ambient optical array according to physics-derived rules. In particular, movement of the observer

creates *visual flow fields*: as the observer moves forward, objects in the center of the field of view tend to increase in size and move toward the edges of the field before disappearing. These flow patterns are interpreted by the visual system and contribute to a sense of movement through the scene.

Similarly, the visual system uses a set of rules to draw conclusions about the *depth* (distance from the observer) of objects. These are known as *depth cues* [War04, Ch.8]. Consider these examples:

Occlusion One object placed in front of another (from the point of view of the observer) will occlude the more distant object.

Size gradient Identical objects placed at different distances from the observer will appear to be different sizes, the nearer objects occupying a proportionally larger area of the visual field.

Linear perspective Parallel lines will converge toward the horizon.

Motion parallax The perceived motion of objects perpendicular to the direction of travel of an observer in motion: nearer objects move faster across the visual field than those more distant.

Ware [War04] separates depth cues into three categories based on the observing conditions required. Many cues can be correctly interpreted even from still images (*e.g.* linear perspective, occlusion, object size gradient); these are called *monocular static* cues. *Monocular dynamic* cues (such as motion parallax) require an animated display, and *binocular* cues make use of the differences between the views captured simultaneously by two eyes separated in space.

The majority of identified important depth cues are classified as monocular static. This speaks to the power of photographs and illustrations to convey the appropriate ideas of

space and relative object positions. For the visualization designer, this also means that much can be accomplished even with little in the way of graphics hardware, if the cues included are chosen carefully. Occlusion is a powerful indication of whether one object is nearer than another, yet if the two objects do not overlap in the visual field it is useless. Linear perspective and the texture gradients formed on ground surfaces establish a depth gradient through a large area, but are less specific for nearby objects, especially if the objects do not contact the ground directly or cast shadows on it. While not all depth cues must be used in every visualization context, those that are implemented should be chosen to support and reinforce each other.

2.2.2 Perception and Computer Graphics

Moving beyond the static display of still images with the use of dynamic graphic displays and specialized hardware, we can make use of the powerful dynamic and binocular depth cues mentioned in the previous section. We describe the advantages and considerations of these techniques.

Structure from motion The human visual system is capable of integrating visual information over time, and using motion over time to augment depth perception. Two major depth cues fall under *structure-from-motion*: motion parallax and kinetic depth effect. Both are amenable to simulation in a computer graphics display.

Motion parallax is the effect briefly described earlier as the perceived motion of objects when the observer is in motion. Looking perpendicular to the direction of travel, a velocity gradient is seen, with objects nearer to the observer moving faster than those closer to the horizon. As the observer looks forward along the direction of travel, a different kind of parallax field is observed, where objects closer to the center of the field of view move slowly toward the edges of the field of view, and move more quickly as they approach the

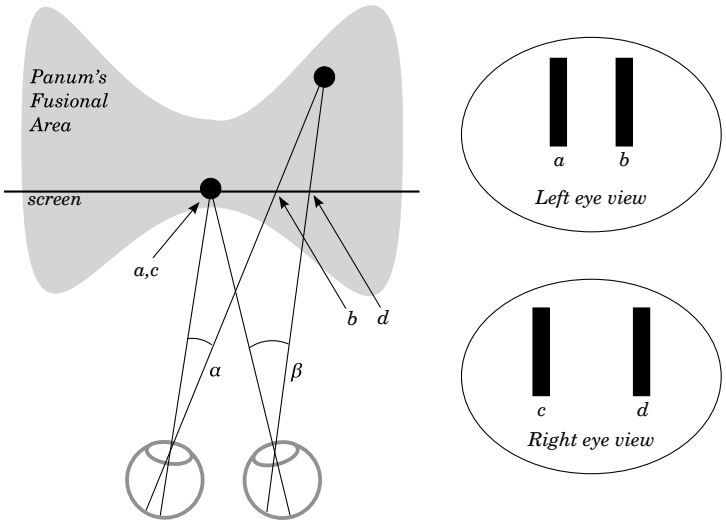

Figure 2.6: Stereoscopic Depth Cue Example (Adapted from [War04])

edges before disappearing from view.

The *kinetic depth effect*, described by Wallach and O'Connell [WO53], occurs as a result of a visual system assumption that objects are rigid in 3D space. When an object is smoothly rotated in space, its three-dimensional shape is rapidly perceived. Wallach and O'Connell use the example of the shadow cast on a flat surface by a bent wire: an observer seeing the shadow will perceive only a bent line, but if the wire is rotated, the observer will be able to discern its three-dimensional form.

Stereoscopy Stereographic three-dimensional viewing is often described as "real 3D." Stereoscopy is indeed an effective depth cue and is useful for many visualization tasks. While it has limitations, we have decided it worthwhile to include as a major feature of the GLuskap package and therefore describe the principles and issues associated with stereographic visualization here.

Figure 2.6 shows a simple stereo display showing two vertical lines, one of which is at the same depth as the screen and the other situated some distance behind it. The eyes

in the example are fixated on the nearer line. The *screen disparity* between the two lines is the difference between the distance *a–b* and the distance *c–d*; this is used by the visual system of the brain to infer the *angular disparity* between the two angles α and β. By trigonometric methods,[2] the brain then perceives the two lines as being at different depths. The two images become *fused* in the visual field (the observer perceives two lines, not two distinct pairs of lines).

If the disparity between the left and right views is too great, the visual system is unable to fuse them as a single image with depth perception, and *diplopia* (or "double vision") occurs. The region within which objects can be effectively fused, relative to the observer and the screen, is known as *Panum's fusional area* and is shaded in the figure. The fusion of objects which are out of focus (due to the fixation of the eyes on another object at a different distance) or in the peripheral visual field is easier for the brain to accomplish—however computer graphics displays typically do not simulate focal depth of field, leading to out-of-depth objects being displayed in sharp focus and thus prone to diplopia. Patterson and Martin [PM92] discuss some of the factors regulating the size of the fusional area.

The mechanisms controlling the *convergence* of the eyes to fixate on an object of interest, and those controlling the focusing of the lenses are very tightly linked. In a natural setting, this ensures the rapid adaptation of the eyes, maintaining proper focus while tracking moving subjects or scanning many objects in rapid sequence. While stereographic computer graphics displays require vergence accommodation to fixate on objects at different depths in a virtual scene, the entire display lies at a uniform focal depth. This brings the linked systems into conflict, as the vergence accommodation cues a change in focal depth which must be suppressed in order to maintain proper focal accommodation. This *vergence-focus problem* can cause eyestrain especially if the vergence disparities in the

[2] The trigonometric calculations are obviously not consciously performed—rather, this information is processed preconsciously by the visual centers of the brain.

scene are large and the system is used for extended periods of time.

While high-performance 3D graphics adapters capable of rendering detailed scenes at high resolution have become relatively inexpensive commodity items, stereographic display requires specialized hardware, including so-called "workstation" graphics adapters, LCD or polarized glasses, and multiple projector systems. More information on the particular requirements for stereographic 3D will be found in the following chapters.

2.2.3 Interactive Systems

Interactive data systems operate in terms of *feedback loops*: the system presents some data representation, the user responds by modifying the data or the parameters of the representation, and the system updates the display accordingly. Ware [War04, Ch.10] explores some of the principles and models used in construction and analysis of interactive visualization systems. These can be extended in most cases to apply to interactive graph drawing software, where we are interested not only in visualizing existing data sets but also creating and modifying graph drawings.

Feedback loops Modeling the human-computer interaction interface as a nested set of goal-directed loops echoes the layered processing model of the visual perception system of the brain [War04, Ch.1]. At the highest level, a goal is specified. Through successively lower levels, the task of achieving this goal is broken down into sets of more easily achievable sub-problems, until at the lowest level individual actions are taken towards completing the task. Each level of cognitive processing operates on a shorter time base than the level above it.

Integrating computer systems into the problem-solving process creates loops that pass back and forth between the user and the interactive system. Ware characterizes the lowest level of interaction as the *data manipulation loop*, where individual objects are selected and

altered. This loop has the shortest timebase; the user issues commands and feedback should be given as rapidly as possible, for delays at this level are multiplied by the frequency with which these simple actions take place. Even a fraction of a second of lag per operation can be disruptive to the user.

The *exploration and navigation loop* structures the interaction with larger data sets. The *visual working memory* of the average user stores about four simple objects or features at a time, but aggregating multiple features as objects themselves allows four times as much data to be held in this way [LV97]. For working with larger data structures, the realization of semantic relations between objects (allowing aggregation) and the formation of mental maps and models must be facilitated by the interactive system.

At an even higher level of abstraction, and on a correspondingly longer time base, is the general *problem solving loop*. As the investigation progresses, data may be added or re-processed to emphasize different attributes, and new interactive tools may be added to the software to allow increased user freedom. Cycles of exploration and interaction are then re-launched and the investigative process continues. The goal in designing such cognitive visualization systems is that they will be useful tools for expanding the capacity of users to solve problems.

Bederson [Bed04] characterizes the optimal experience for skilled users as "flow"—when a tool is used naturally as an extension of the user's body without distracting from the task at hand. Users are able to focus on the work and maintain responsive control over the operation of the system. Timely response speed (the faster the better) and clear feedback is essential to maintaining a sense of "flow".

A similar structure of feedback loops is an effective model for the interactive construction, analysis, and manipulation of graph drawing algorithms. In the middle layers of the structure, there is not only a process of exploration and navigation but also direct and indirect manipulation of the underlying data: graph structures and geometric drawing

representations thereof.

Fitts' law and interface lag At the data manipulation level, selecting objects on a two-dimensional display or in space is a fundamental and frequently occurring task in many applications. *Fitts' law* is a formulation of the time taken to select a target with a fixed position and size.

$$\text{Selection time} = a + b \log_2 \left(\frac{D}{W} + 1.0 \right)$$

where D is the distance to be covered to reach the center of the target, W is the width of the target, and a and b are empirical constants [War04]. The *Index of Performance* (IP), $1/b$, is measured in "bits per second." While it is specific to individual users, typical values are about 4 bits per second. The difficulty of a given task is estimated by the logarithmic term $log_2 (D/W + 1.0)$, in units of "bits"—larger, closer targets can be selected more quickly than smaller targets further away.

Ware and Balakrishnan [WB94] adapted Fitts' law to incorporate the effects of both system lag (time from input device movement to display feedback) and human response time:

$$\text{Mean time} = a + b(\text{Human Time} + \text{Machine Lag}) \log_2 \left(\frac{D}{W} + 1.0 \right)$$

Ware points out that for a constant amount of latency (as is typical in most interactive systems, especially those invoking the processing overhead of three-dimensional input devices and stereoscopic displays), the net effect of the lag increases with small targets and can make precision selection tasks more difficult. Reducing the end-to-end latency of the visualization system and increasing the effective size of selection targets are two possible strategies to aid the user in these tasks.

Fitts' Law effects also apply to 3D interactive systems as well as 2D interfaces, though they have not been precisely quantified in the three-dimensional case [WB94].

2.3 Interactive 3D Graph Drawing

2.3.1 Software Products for 3D Graph Drawing

In the last 10 years, computer technology has advanced to the point where the real-time visualization of 3D graphics is possible with inexpensive commodity hardware. This has greatly facilitated the investigation of three-dimensional graph drawings. Software tools for research in this area fall into three rough categories: general purpose 3D modeling, specialized 3D graph visualization software, and general purpose 3D graph drawing software.

General purpose 3D modeling software A large body of software exists for the creation and visualization of general three-dimensional objects and scenes. Included in this category are industrial CAD/CAM and drafting packages such as AutoCAD; artistic graphics and animation tools including 3D Studio MAX, Lightwave, Blender, and POV-Ray; and VRML editing and viewing software. Many of these software packages can be used to produce high quality visualizations of three-dimensional graph drawings. However, they typically lack the interface functionality and high-level data semantics to be effectively used for the interactive creation and manipulation of graph drawings, or the programmatic implementation of graph drawing algorithms.

Specialized graph drawing software There exists a large market for graph visualization software, primarily focused on two-dimensional drawings and constructed to meet the needs of disciplines as diverse as computer network management, social science research, and molecular biology. Tom Sawyer Software leads the industry in this area, focused exclusively on providing commercial graph visualization software. Less common are applications using three-dimensional graph drawing methods, which we speculate is due to the more recent emergence of the field and the technological requirements for displaying and

manipulating three-dimensional objects.

One such visualization product is the Tulip package, described by Auber as a "huge graph visualization framework" [Aub04]. Optimized for use with graphs containing up to 1,000,000 elements, Tulip implements algorithms for displaying general graphs, trees, and clustered graphs in two and three dimensions, as well as interactive clustering and additional techniques appropriate to analysis of large data sets. It is freely available with source code and is designed to be extensible for a variety of application domains. Tulip is written in C++, using the Qt and OpenGL libraries for interface.

An even more specialized example is CrocoCosmos, "part of a comprehensive experimental software analysis tool set to support analysis, comprehension, and quality assessment of large object-oriented programs" [LN04]. CrocoCosmos maps program entities (methods, attributes, classes, files, and subsystems) onto vertices of a graph, and hierarchical organization and relations between entities are mapped onto edges.

By drawing on information visualization principles, these graph entities are attributed with metrics corresponding to the size, degree, etc. of the program components they represent. The specific representation functions used are under the control of the user. Force-directed (energy-based) methods are used to produce a three-dimensional layout. This is because of the general amenability of such methods to the graphs which will be created by the mapping algorithms, without any prior knowledge of specific graph theoretic properties that would allow the use of optimized layouts.

CrocoCosmos supports a high-performance graph viewing application that uses OpenGL for display.

General purpose 3D graph drawing software 3DCube by Patrignani and Vargiu [PV97] can be described as a general-purpose graph drawing package, though it primarily supports the development of orthogonal graph drawing algorithms. It also includes the straight-line

"moment curve" algorithm [CELR97]. The interface provides a three-dimensional perspective view and allows the user to specify the appearance of nodes and edges. Graph drawing algorithms can be animated, and selected views of individual three- dimensional drawings can be saved and recalled. 3DCube is written in C++ using the Motif and graPHIGS interface libraries.

The OrthoPak 3D software was developed at the University of Lethbridge to work with 3D orthogonal grid drawings of graphs [CEGW98]. It supports graphs of arbitrary vertex degree (using three-dimensional boxes to represent vertices) as well as graphs with maximum vertex degree constrained to 6 (required to avoid crossings when vertices are represented by points). Several 3D orthogonal drawing algorithms are implemented. Output is produced in VRML format for 3D viewing. The LEDA libraries are used for interface.

Dwyer and Eckersley [DE04] have developed the WilmaScope package for three-dimensional graph editing and visualization, including directed graphs and graphs with clustered vertices. It provides several force-directed 3D layout algorithms, as well as support for using the Graphviz DOT program for layered 2D drawings [EGK+04]. An interface for writing additional "layout engines" is provided. Customizing the appearance of graph elements and the parameters of the layout algorithms is supported. WilmaScope is written in Java and uses the Java3D library for interface.

2.3.2 Historical GLuskap

As of September 2004, the GLuskap package had been developed at the University of Lethbridge as a general-purpose three dimensional graph drawing package, capable of manipulating and drawing arbitrary undirected graphs. It will henceforth be referred to as "GLuskap 2.4" where necessary to differentiate it from the enhanced GLuskap system and software which is described in the following chapters.

Motivation and capabilities GLuskap is designed to assist in three dimensional graph drawing research by allowing the interactive construction of graph layouts with the vertex positions specified exactly. GLuskap includes support for edges defined by polylines in three dimensions, or "bent edges", with the position of bend points specified precisely.

This allows the construction of volume-bounded layouts as described in Section 2.1.2 including the straight-line layout of Cohen *et al.* [CELR97], the 1-bend layouts of Morin and Wood [MW04] and Devilliers *et al.* [DEL+05], and the 2-bend layout of Dyck *et al.* [DJN+04].

Import and export with the standard GML and GraphML file formats (described in [Him96] and [BEL04] respectively) is supported. Capture and export of perspective views of graph drawings directly from the user interface is possible, while high-quality raytracings are possible through export to POV-Ray scene description files which can be rendered at high resolutions. While production of vector (EPS) format drawings using the technique described by Kilgard [Kil97] is possible due to GLuskap's use of OpenGL, it has not yet been implemented.

Implementation The code-base has been rewritten and significantly revised since its initial release in 2003 [DJN+03]. GLuskap 2.4 (and all subsequent development) is written in Python, and makes use of the OpenGL and wxPython libraries for user interface.

The Python language was chosen for its high-level object-oriented data structures, robust exception handling, and ease of cross-platform portability [Lut96]. As a result, GLuskap is developed and tested on Microsoft Windows, Linux, and Apple OS X platforms with no change in functionality. As well, Python's straightforward syntax makes it ideal for implementation of graph drawing algorithms in a readable way. High CPU load calculations that would suffer under the performance constraints of an interpreted language can be delegated to modules written in C or C++ [ADH+01, GMHW03].

Avenues for expansion The GLuskap 2.4 software package serves as the base upon which the GLuskap system described in the following chapters is built. A plug-in scripting system has been added to allow graph algorithms to be executed within the GLuskap interface. The stereographic 3D output and user input subsystems have been expanded considerably to support an immersive workspace for the construction and visualization of three-dimensional graph drawings, following the principles of interactive visualization systems.

Chapter 3

Design

In this chapter, the design requirements and considerations for the construction of an interactive virtual reality system are described. The main focus is on the particular requirements of interactive graph drawing for abstract graphs, making the hardware system both effective and transportable, and ensuring the flexibility and portability of the software components.

3.1 Software System

3.1.1 GLuskap Visualization Enhancements

The original GLuskap software has been augmented and altered for use in an interactive virtual reality system, following established principles of visualization and interaction.

Textured objects Gibson's ecological optics maintains that natural objects are perceived as surfaces, and that these surfaces necessarily have *texture*—the "structure of the surface," as it appears to the viewer [Gib86]. Perfectly uniform smoothly shaded objects do not exist in the real world, their appearance in many virtual environment simulations and computer models notwithstanding. Not only does the mere presence of a surface texture lend realism

to a virtual object, characteristic textures can convey information about the attributes of the object to the user. The contours of a regular pattern are also useful cues to the surface shape of an object.

Stereographic 3D display provides a further motivation to use textured objects. Recall that stereoscopic depth is resolved by the brain using the angular disparity between the left and right views of the same point on a particular object (Section 2.2.2). With smooth shading, the only features for which depth can be determined are the edges of an object. Consider a sphere: it will be perceived as having a different depth than surrounding objects. Inside its boundaries, though, it is equivalent to a disc. Adding visible texture to virtual objects provides additional points of reference on the surface that can be resolved by the viewer to determine the 3D shape of the object.

To this end, the rendering system in GLuskap applies simple regular texture maps to all three-dimensional objects in the virtual space.

Rapid zooming To focus on one part of a graph drawing, it is useful to be able to move rapidly through the virtual space towards some target. It is also useful to be able to identify some named (yet possibly out of view) element of the graph drawing and bring it into view. These tasks are facilitated by the inclusion of a "rapid zooming" feature, based on the Point of Interest movement interface developed by Mackinlay *et al.* [MCR90].

In GLuskap, selecting an element of the graph drawing (vertex or edge) and activating the rapid zoom mechanism causes the viewpoint to move quickly and smoothly towards the selected target, terminating with the object of interest highlighted and centered in the display. The rate of travel through the virtual space is logarithmic, slowing down as the viewpoint approaches the selected target. In contrast with simply jumping to a new viewpoint, zooming in this way has the advantage of preserving the user's sense of presence and augmenting the user's mental map of the virtual space.

Embedded three-dimensional cursor To facilitate actions in the three-dimensional virtual space, including selecting, positioning, and relocating graph objects, an "embedded" pointer has been added to the GLuskap interface to be used with three-dimensional input devices. Unlike the windowing system cursor which operates in two dimensions, between the user and the three-dimensional perspective view, the embedded pointer is incorporated into the virtual space with a real three-dimensional position and orientation. Movement and positioning of the cursor is relative to the viewpoint and view direction, so that the cursor remains in view even if the perspective changes.

3.1.2 GLuskap Stereographics Support

In a conventional (monocular) 3D perspective display, a viewpoint, view direction (alternately "view plane normal" or *VPN),* and angular field of view are established inside the virtual environment and used to produce a rendered display of the virtual scene (Figure 3.1). Movement of the user inside the virtual environment is simulated by moving the viewpoint and changing the orientation of the VPN [SWN+03].

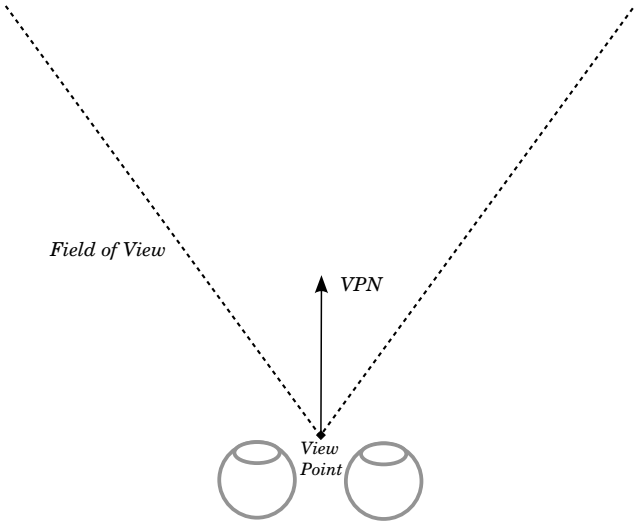

Figure 3.1: Conventional 3D perspective display

To enable stereographic 3D display, it is necessary to simultaneously produce two rendered views of the virtual environment: one for each of the left and right eyes, as shown in Figure 3.2. The perspective for each of these views is shifted perpendicular to the VPN, corresponding to the binocular separation of the eyes. As well, the VPN for each view is rotated inward slightly to simulate the convergence of the eyes on objects of interest, resulting in a virtual *convergence plane,* at which virtual objects will appear to be at the same depth as the physical screen.

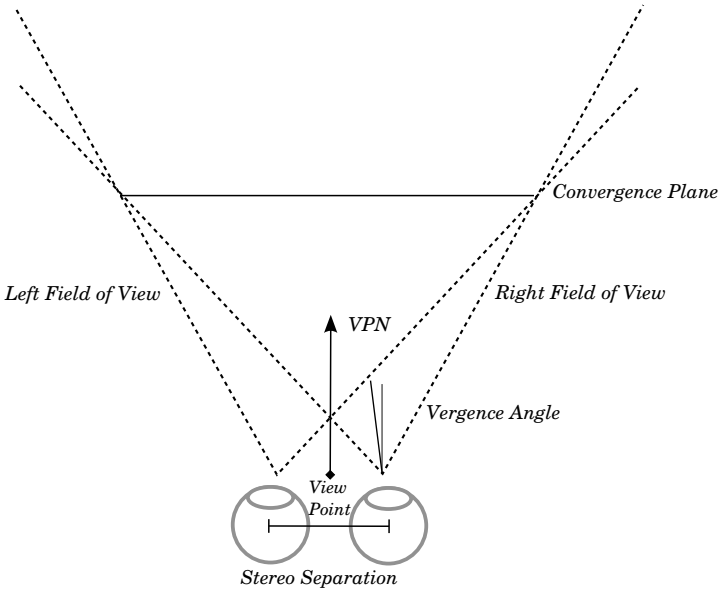

Figure 3.2: Stereographic 3D display

The GLuskap software supports two general modes of stereographic display: anaglyphic (red-blue) separation applied in software, and OpenGL stereography requiring specialized hardware.

Anaglyphic stereo For maximum hardware compatibility, GLuskap supports stereographic display using anaglyphs [SM03]. In this mode, the left and right views have red and blue filters applied, respectively, in software. Stereoscopic depth effects can be therefore per-

ceived while using a standard computer monitor or a single data projector in conjunction with readily available and inexpensive red/blue filter glasses.

Anaglyphic stereography does impose significant restrictions on the use of colour for semantic purposes in the display. The use of inexpensive glasses can also result in crosstalk (diminished left/right channel separation) due to mismatched colour calibration between the display device and the particular colouration of the filter elements in use.

OpenGL quad-buffered stereographics GLuskap also supports stereography using the standard OpenGL interface, called *quad-buffering* [SWN+03]. Four frame buffers are used so that the standard double-buffered drawing technique can be used with both the left and right view being held in video memory simultaneously. This requires a video adapter with the appropriate hardware capabilities. Typically, an adapter of this type will produce *active* (or "frame-sequential") stereographics requiring LCD shutter glasses to view, as the display alternates rapidly between the left and right views. Some adapters also support *passive* stereographics, where both left and right views are rendered simultaneously to independent output devices.

3.1.3 Network Rendering

To drive a large-screen projection system using dual projectors and polarized filters, as described in Section 3.2.2, passive stereo output is required. For maximum portability, the use of a laptop computer is desirable, yet very few laptop computers are equipped with video hardware capable of driving two projectors simultaneously or supporting OpenGL stereo.

Paquet and Viktor [PV04] describe the use of two laptop computers coupled by an IEEE1394 "Firewire" link to produce passive stereographics under a similar portability requirement. One computer acts as the primary node, maintaining the state of the simulation,

processing input, and rendering one of the stereo views. Rendering of the other stereo view is delegated to the second computer; simulation data and synchronization are transferred over the IEEE1394 connection. This configuration is depicted in Figure 3.3, with arrowheads indicating the direction of data flow.

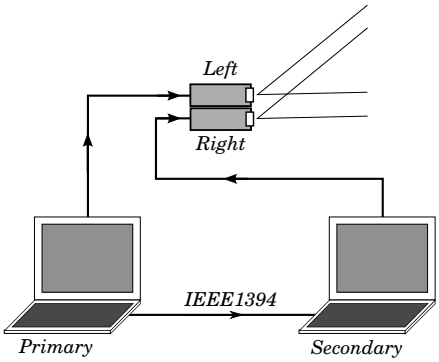

Figure 3.3: Two Laptops for Portable Passive Stereographics

In this scenario, frame-by-frame synchronization between the two computers is required to compensate for the communication delay—otherwise it is possible that the secondary "lags behind" the primary in updates to the display.

Synchronization and latency concerns To reduce the need for precise synchronization, our alternate configuration for portable passive stereo uses a single primary node and **two** secondary nodes, connected using standard 100BaseTX ethernet. Each secondary node is responsible for producing one of the left or right perspective views. Any latency introduced in the data flow path from the primary node to the projection display will be symmetric. See Figure 3.4.

Distributing the production of the display images in this way necessarily introduces some latency into the processing and display of user interaction and other updates to the simulation. This latency must be kept to a minimum, as small amounts of delay (especially

in hand-pointer tracking) can make positioning and selection tasks more difficult [WB94].

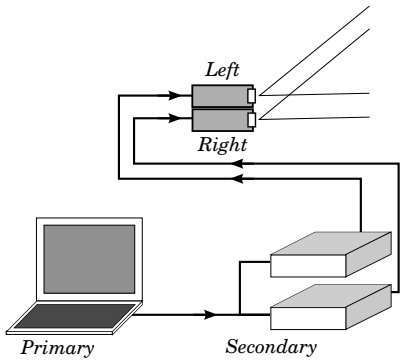

Figure 3.4: Primary and Two Secondary Nodes for Portable Passive Stereographics

3.1.4 Interaction Principles for Large-Screen Interface

While the standard windowing interface for GLuskap includes a large number of widgets for modifying the graph drawing and manipulating properties of the visualization, the large screen interactive interface has been simplified considerably allowing the user to focus on the tasks of constructing and manipulating graph drawings in three dimensions.

The active 3D pointer, described in section 3.1.1, allows the user to directly specify positional information and move objects in three dimensions. In addition to the positional tracking information, the particular hardware pointing device used includes 3 buttons and a two-dimensional analog input device [Ascb]. With these control widgets, it is possible to create a simple interface to common interaction tasks by using the 3D pointer in combination with context-sensitive menus.

Graph tasks including adding, removing, positioning, and resizing graph elements are chosen from context-sensitive menus. A heads-up overlay provides feedback regarding the current task, the position of the pointer, and any error conditions. The menu interface is designed to balance the number of clicks required to perform any specific task with the typ-

ical use frequency for that task. Each button is assigned a function to be consistent within the interface and to correspond to traditional conventions of windowing system interfaces where possible. For example, the left button is used to select objects and confirm choices, and the right button is used to activate context-specific menus.

To reduce the incidence of error in positioning graph objects, a "snap" function is provided to restrict positional input to the nearest point on a uniform three-dimensional grid. As most three-dimensional graph drawing layout algorithms constrain vertex positions to integer grid points, this compromise is acceptable. The snap function can be set to an arbitrary level of granularity depending on the user requirement.

Moving the viewpoint to examine or manipulate the graph drawing from an alternate angle is often useful. In addition to head tracking, rotation and zooming of the graph model is linked to the two-dimensional input widget on the pointer device.

3.1.5 Plug-in Architecture for Layout Algorithms

Providing some method for graph drawing layout algorithms to be coded in software by users motivates the construction of a "plug-in" interface for the GLuskap software. Some previous packages have integrated a selection of algorithms into the program code [PV97, CEGW98]. In this case, modifying the particular algorithm implementations or adding new algorithms requires editing the core source code of the software package. This is cumbersome in the best case and impossible in the worst case, where the source code is unavailable due to license restrictions or lack of maintenance.

One approach would be to allow users to code algorithms in a scripting language like Lua [IdFC03], which is designed for integration of interpreted scripting into an application coded in C/C++. This requires establishing a "dialect" of scripting commands and interfaces available to the user which is distinct from the interface used by the authors to

implement core software functionality. This may cause bugs to appear if these two interfaces are not kept consistent through the maintenance and further development of the core product. Furthermore, the capabilities of the external scripting language may limit the scripting functionality available to the user.

As GLuskap is written in Python, though, a third option is available. User scripts can be written directly in Python, accessing the same standard interfaces used by the core software, while still being loaded and interpreted at run time as independent modules. Care must be taken in developing the standard interface in a consistent and usable fashion, but this effort is doubly rewarding: it benefits both the user writing independent scripts and the developer integrating other components of the core product with these same interfaces.

Capabilities In a three-dimensional graph drawing application such as GLuskap, the user is able to modify three inter-related types of data: the structure of a graph (vertices and edges), the three-dimensional embedding of the graph (positions, sizes, and visual attributes of the vertices and edges), and characteristics of the virtual scene and display in which the graph is presented (position of the viewpoint, colour of the background, whether secondary visual aids like coordinate axes are present).

An ideal plug-in scripting system should provide an interface to control all three of these facets. Facilities should be provided to add and remove graph elements, to alter their positional and display attributes, and to change the properties of the three-dimensional view. The GLuskap scripting system meets all of these requirements.

Examples of plug-in scripts that could be developed using this interface include testing for planarity or other graph-theoretic properties [BM04b], presenting specific 2D or 3D graph layout algorithms [CELR97], or to test heuristics for optimal viewpoints of 3D graph drawings [EHW97].

3.2 Hardware System

The hardware requirements for an interactive visualization system include one or more output devices, one or more input devices, and a computer processing system to maintain the state of the visualization while incorporating user input.

3.2.1 Display

All computer graphics systems use some kind of display device. Recent developments in computer technology allow the use of real time interactive three-dimensional graphic displays. Standard desktop PC monitors allow the presentation of high resolution data, and in some cases can be used for stereographic 3D display.

When working with interactive virtual reality systems, though, it is desirable to provide the user as much as possible with a sense of presence in the virtual environment. A head-mounted display (HMD) that restricts the user's view to the virtual environment and allows the user's head orientation to be tracked and simulated can be used to this end. It has also been shown that the use of a large projection screen can provide an immersive experience comparable to a HMD, and for a lower cost [PCS+00].

While a large projection screen requires a similarly large operating area, it is possible to construct the screen and projection apparatus in such a way that they can be easily disassembled and stored compactly when the system is not in use. The construction of such a portable system is described by Arns [Arn03].

3.2.2 Projection Setup

Polarization and dual projection In the large screen stereographic configuration, two projectors are used for display of both the left and right views simultaneously. To separate the two projected views and direct them to the appropriate eyes, optical polarization filters

are applied to the projected images in conjunction with matched polarized glasses worn by the user. There are two commonly available varieties of polarizing filters: linear and circular.

Linear polarization filters can be used for this task, though their effectiveness depends on the user's head remaining level with respect to the alignment of the projection filter. Inadvertent tilting of the user's head can allow the user's left eye to view the right view and vice versa, diminishing the stereo depth effect.

Circular polarization filters, incorporating a quarter-wave plate in addition to the linear polarizing element, are preferred, as they make the stereo effect robust against shifts in user head orientation [Arn03].

Care must be taken to select a screen material that preserves the polarization of the projected images. A white wall or standard white projection screen is designed for maximum viewing angle and will scatter the polarization on reflection and is therefore of no use for this application. If a front projection system is to be used, a silver lenticular screen will effectively preserve polarization [BFLR00].

A front projection system suffers from shadows being cast on the projected image by the user if they stand between the projector(s) and the screen, as is typical for a large-screen interactive visualization system. For this reason a rear projection system is preferred, as in [Arn03], despite the larger operating area required. The use of a screen material that preserves polarization on transmission is therefore required.

It should be noted that so-called "virtual rear projection" systems incorporating multiple projectors to alleviate the occlusion problem of front projection without the space requirements of true rear projection have been developed. [SACR03]. The processing and synchronization requirements to implement such techniques for stereoscopic 3D projection have not been investigated in the literature at this time, and were considered to be unreasonable for the purposes of the GLuskap system.

Calibration of overlapping displays Whether front or rear projection is used, two projected images are viewed simultaneously on the same area of the screen, and the optical centers of the projectors are necessarily separated by some small distance. As most data projectors are wider than they are tall, this distance is minimized by stacking the projectors vertically rather than placing them side by side.

If no correction is applied, the usable area of the display is restricted to the overlapping projection area and thus requires that some of the usable pixels of each projector be wasted. Figure 3.5(a).

If the projectors are angled inward, toward the axis of projection, the projected images can be made to overlap more completely. This has the side effect of producing "keystoning" as the axis of projection for each projector is no longer perpendicular to the plane of the screen. Many projectors have mechanisms to alleviate keystoning effects digitally. Figure 3.5(b).

Commercial products are also available that use optical lens shift to align the projected images at the point of projection, though at an additional cost. Figure 3.5(c).

3.2.3 User Tracking

Tracking the position and orientation of the user further enhances a sense of presence in the virtual environment, with the coupling of user head tracking to the viewpoint position being particularly important. Lateral movement of the viewpoint (that is, movement parallel to the screen plane) provides the user with depth information from motion parallax [BM04a].

3.2.4 Input Devices

Implicit in the definition of an interactive system is the means to manipulate data in real time and have the results displayed. Historically, the GLuskap system has supported in-

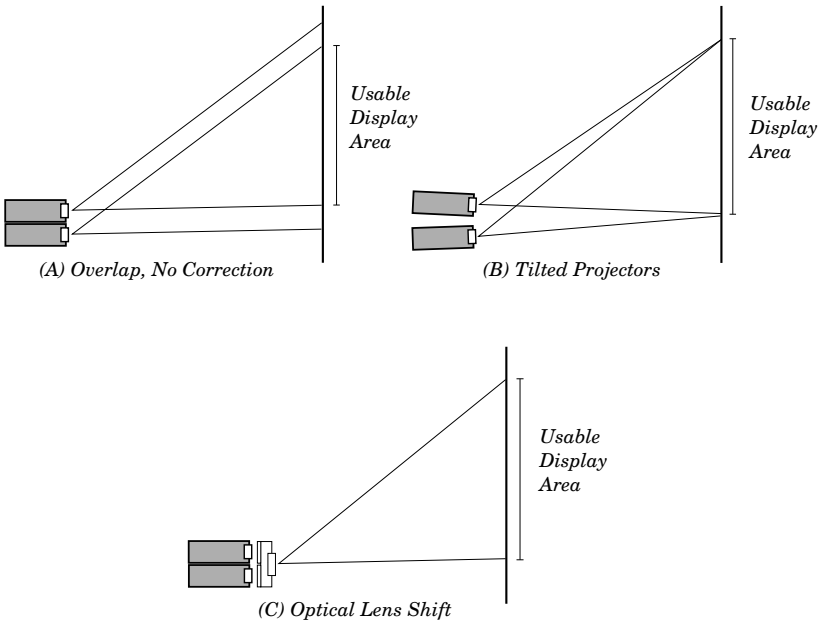

Figure 3.5: Overlapping Alignment of Two Projectors

teraction with keyboard and two-dimensional mouse in a standard window environment [DHW04].

Wand tracking for interaction The use of a real three-dimensional input device coupled to a cursor embedded in the virtual environment allows natural manipulation of virtual objects by direct translation and positioning in three dimensions. The position of the cursor is always translated in the virtual space so that it remains relative to the current viewpoint, as the input device moves relative to the physical screen.

More complex actions, such as those requiring alphanumeric input, are still performed using the keyboard and mouse in the windowing environment.

3.2.5 Processing and Rendering System

The hardware system is based around a single computer serving as the *primary node.* This computer runs the GLuskap software and is responsible for handling input from the user through the mouse and keyboard, as well as 3D tracking and input systems. Depending on output requirements and available hardware, the primary node may directly produce a conventional 3D view or a stereoscopic 3D view, or manage the output of a stereoscopic 3D view from connected *secondary nodes.* These secondary nodes are individual computers which run a specialized version of the GLuskap software and produce synchronized 3D views of the virtual environment managed by the primary node.

Chapter 4

Implementation

In this chapter, the implementation details of the GLuskap system are described. The "GLuskap system" includes the hardware and software components used to provide the large-screen stereographic 3D interactive interface. The GLuskap software is used as part of this system and includes enhanced functionality to support specific features of the integrated system, though it remains usable as a standalone product.

4.1 The Visualization Hardware System

The hardware components of the GLuskap system are sketched in Figure 4.1. The system includes three computers (one primary node and two secondary nodes), two motion tracking controllers, two data projectors, and a large rear-projection screen. The function and roles of these components will be described here.

4.1.1 Screen

The GLuskap system uses rear projection for the large screen interactive interface. The screen assembly is designed to be easily disassembled and reassembled for transport in a

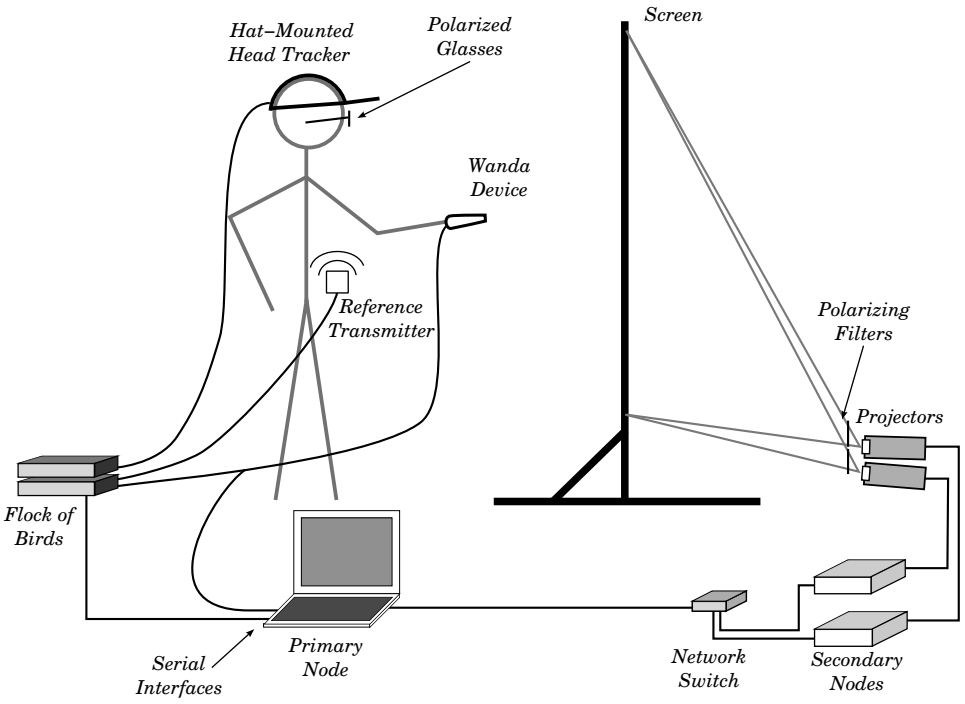

Figure 4.1: GLuskap System Hardware Components (not to scale)

relatively small package, and is patterned on the design developed for the Envision Center at Purdue University [Arn03].

The screen frame is constructed from modular extruded aluminum parts from 80/20. The projection area is approximately 6 feet (1.83m) high and 8 feet (2.44m) across in the standard 4:3 display aspect ratio; Stewart Filmscreen 100 material is used for the projection surface.

4.1.2 Projectors and Polarized Optics

Paired Dell MP2300 DLP data projectors are used for stereoscopic projection. One projector is connected to the video output of each of the secondary nodes (described below).

These projectors have a maximum resolution of 1024×768 pixels and output brightness of 2300 lumens each. The projectors were chosen for their high brightness, value for money, and compact size.

Custom adjustable stands have been constructed to enable the projectors to be stacked on top of each other, minimizing the distance between projection lenses, and so that the projectors can be precisely aligned to overlap. These stands are collapsible for transport.

Circular polarization filters are placed between the projectors and the projection screen. In conjunction with matched polarized glasses, the filters allow the projection of the left and right views to overlap. The views are then separated by the user's glasses and directed to the correct eyes.

4.1.3 Computers

The primary node of the GLuskap system, as implemented, is a Dell Precision M20 laptop running Microsoft Windows XP and was chosen for portability, hardware compatibility, and capability reasons. The Precision M20 features an ATI FireGL 3100 "workstation" graphics adapter, capable of OpenGL quad-buffered stereographics if an external CRT display is used. It also includes an onboard RS-232 serial interface, required for connection to the Flock of Birds motion tracking system. A second serial interface, to connect the input widgets of the Wanda device, is provided by a USB to serial adapter. The laptop also features an onboard 100BaseTX Ethernet adapter.

Both secondary nodes are Apple Mac Mini computers running Mac OS version 10.4. Chosen for their compact form factor and portability, the Mac Mini includes an ATI Radeon 9000 video adapter fully supporting OpenGL graphics. They also include onboard 100BaseTX Ethernet. The secondary nodes are connected to the primary node over by a 100BaseTX Ethernet switch operating at 100Mbit/s.

The software configuration of the computer systems is described in Section 4.2.3.

4.1.4 Motion Tracking

Three-dimensional position and orientation tracking is performed by a Flock of Birds system from Ascension Technology [Asca]. Two 6-degree-of-freedom[1] (6DOF) sensors are managed by the system: one tracks the position of the handheld Wanda device used for interaction, and the other is attached to a hat worn by the user to track their head movements.

The Flock of Birds system uses DC magnetic tracking technology, where magnetic fields aligned on the X, Y, and Z axes are sequentially generated by the reference transmitter. The strength of these fields at the sensor position is used to calculate the position and orientation of the sensors relative to the transmitter. The Flock configuration used by the GLuskap system is capable of tracking the positions of the sensors within a 3.94 foot (1.2 m) range of the reference transmitter.

Each sensor is connected to a separate controller chassis. The two controller units are connected together by a high-speed RS-485 serial bus, and one of the controllers is connected to the primary node by a 115.2Kbps RS-232 serial link. This controller also drives the reference transmitter. All initialization and tracking data communication between the Flock and the GLuskap software on the primary node takes place over the RS-232 serial connection.

Wanda device The Wanda device is a handheld pointing and control device that includes a 6DOF motion tracking sensor, three buttons mounted left to right, and a two-dimensional "HulaPoint" analog input controller mounted below the buttons [Ascb]. The motion tracking sensor is used for positional and movement input to the GLuskap system; the control widgets are used to interact with the fullscreen immersive interface (see Section 4.2.4).

[1] Three positional coordinates (X, Y, Z) and three angular coordinates (pitch, yaw, and roll).

For consistency, the buttons on the Wanda device will be referred to as the *left, middle,* and *right Wanda buttons.*

4.2 Interactive Graph Drawing Software

As an interactive system, GLuskap processes input from the user in combination with existing data and then provides feedback to the user. An outline of the data flow through the software is shown in Figure 4.2. User input is processed directly from serial peripherals (tracking system) and the mouse and keyboard, as well as through more complex actions accessed through the windowing user interface or by executing a plug-in script. The updated display is drawn by the GLCanvas object at the bottom of the diagram using the OpenGL interface.

Rectangular boxes in the diagram represent software object modules, heavy edges indicate the direction of data flow, and dashed edges indicate the direction of procedure calls (for example, the WandaCtrl object *initiates* the procedure call for a data flow *from* the serial interface). We will call this behaviour *retrieving data*. The other possibility is that the object initiating the procedure call *delivers data* to the receiver.

4.2.1 Multiple Files, Multiple Contexts

For convenience, the GLuskap program allows multiple graph drawings to be loaded simultaneously, though at this time data cannot be exchanged between them directly. The graph data structure, current 3D camera and view settings, and various other stateful data specific to a particular graph are bound together in a data structure called a GraphContext (shown as a grey shaded box in Figure 4.2), and abbreviated to "context" where unambiguous.

While in theory there is no limit on the number of contexts that can be open at any given time, only one context can be viewed or modified at any given time. This is represented in

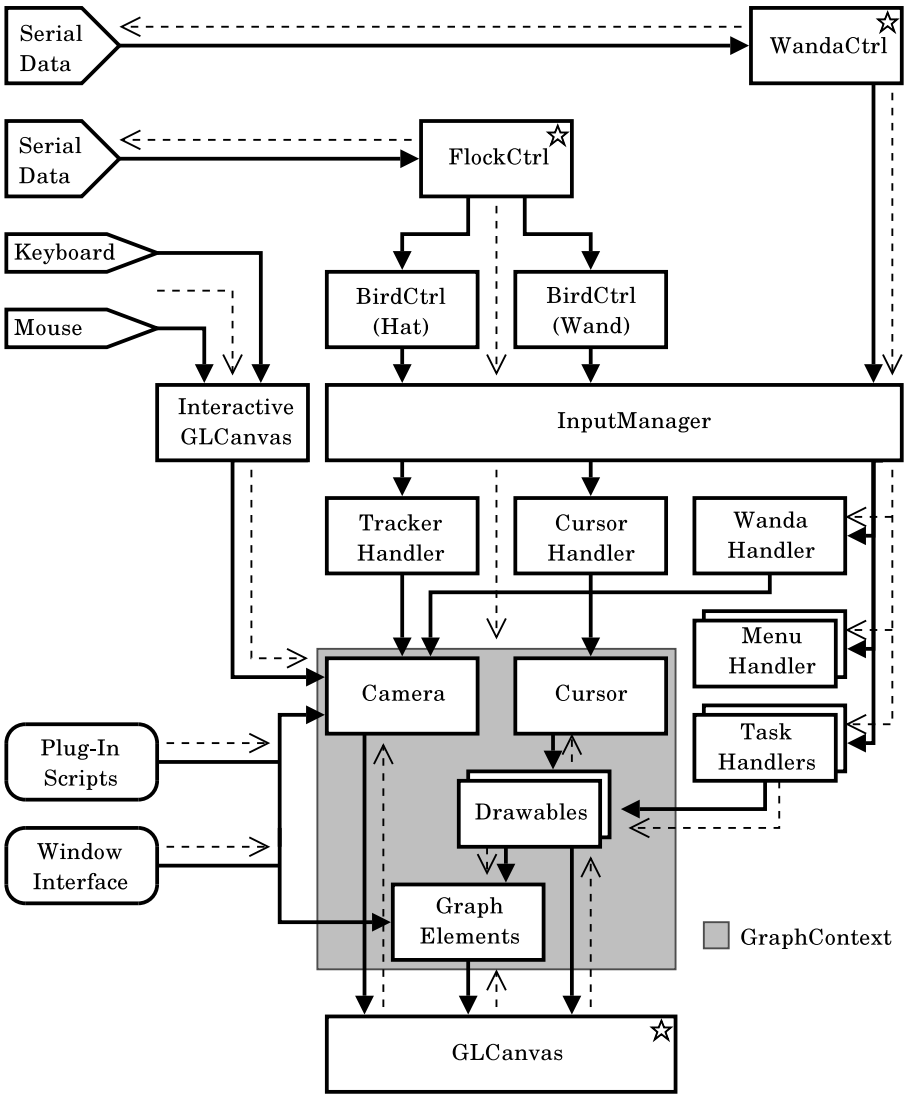

Figure 4.2: GLuskap Data Flows

the window system interface by use of a tab selection widget with one tab per open context, and in the software by an object called the ContextManager (not pictured for simplicity). In a single instance of GLuskap, there exists one and only one ContextManager object, and it maintains a list of all open contexts as well as a pointer to the *active context*—the context containing the graph drawing that is currently being displayed and to which all user input, plug-in scripts, and other graph-specific operations are directed.

The ContextManager ensures that at any given time, if at least one context is open, then a valid active context is maintained. If no contexts are open, the ContextManager disables user interface elements related to graph manipulation and navigation and invalidates the active context pointer.

All code procedures that interact with graph data or viewing parameters must first query the ContextManager to determine the active context, and if no active context exists (no graph drawings are open), then the procedure should be aborted. This layer of indirection allows most code modules to interface with a single graph drawing and disregard any other contexts that may be open at a given time.

4.2.2 Asynchronous Core

It can be seen in Figure 4.2 that the procedure call graph (dashed line edges) is not a directed tree—there are multiple nodes with out-edges that have no in-edges. In particular, the FlockCtrl, WandaCtrl, and GLCanvas nodes are marked with a star. These objects initiate procedure calls to retrieve or deliver data without being called by some other program object. Instead, they have *update methods* that are activated periodically by the main loop of the GLuskap program. After processing new data and updating other program objects, each update method terminates and returns control to the main loop. This allows other parts of the software to be activated, either in response to external events or on a periodic

schedule.

This ability to respond to asynchronous events in multiple areas of the program flow is fundamental, both to the window system user interface and the networking layer, and also to allow them to work together. Using a *cooperative* scheduling system to manage concurrency, as described, requires accommodation from the code perspective: any long-running procedure must be explicitly broken up into parts that can be scheduled so that the procedure does not block the operation of the rest of the system.

An alternative approach would be to use *threads*, which push the responsibility of managing concurrency onto the operating system. Threading comes at a cost, though: communication between concurrent threads and synchronization of operations becomes more difficult to implement and even more difficult to test for robustness. As both the wxPython interface and Twisted networking libraries support explicit cooperative scheduling, it was decided to use this approach for the main GLuskap concurrency scheduling system.

Rendering The GLCanvas object is responsible for initiating updates of the display. The target frame rate R_T of the 3D graph view defaults to 20 fps (frames per second) but can be customized by the user. The target frame rate is used by the GLuskap main loop to set the scheduling interval ($1/R_T$ seconds) between calls to the GLCanvas update method.

When the GLCanvas update method is called, the ContextManager is queried to determine the active context, and the *draw* method on the GraphContext is called. This *draw* method sets up the OpenGL viewing frustum according to the state of the Camera object, then draws the objects in the scene representing the elements of the three-dimensional graph drawing (the Graph object). Finally, additional interface elements including the heads-up display, ground plane, and 3D cursor are drawn.

Diverse input devices Several avenues are available to the user for navigating and manipulating the active graph drawing. Keystrokes and mouse actions for navigation sent to the 3D view pane are handled first by the window system. These events are subsequently relayed to the Interactive GLCanvas,[2] which adjusts the parameters of the Camera object.

The user can also use the menus and interface widgets present in the window system interface to activate graph manipulation or navigation functions, *e.g.* rapid zooming (see Section 3.1.1). These functions operate directly on the Camera and Graph objects. Plug-in scripts function in much the same way and are described in detail in Section 4.2.5.

Input from the Flock of Birds tracking system is received over a hardware serial interface. Unlike keyboard and mouse input, which is also used for general input to the window system and thus necessarily mediated by the wxPython interface layer, the GLuskap software handles serial communication directly. The position and orientation data from the two tracking sensors is multiplexed into a single serial communications line by the Flock hardware and is retrieved from the serial interface and demultiplexed by the FlockCtrl object. The FlockCtrl object is also used to configure and initialize the tracking hardware on program start.

Position and orientation tracking data is delivered to each of the two BirdControl objects; at program initialization time, the InputManager registers handler objects with each of the BirdControl objects so that they can deliver data when it is received. These handlers (Tracker Handler and Cursor Handler) perform all scaling and transformation operations that are not specific to a particular GraphContext and deliver the position and orientation updates to the Camera and Cursor objects of the active context so that they can be reflected in the next display update.

Input from the control widgets mounted on the Wanda device is processed through a

[2] A subclass of GLCanvas with support for these interaction events—except in the case of a remote client (Section 4.2.3), the GLCanvas used for drawing is actually an instance of Interactive GLCanvas and performs both roles.

second serial interface in a similar manner to the Flock system. Status data for these widgets is encoded using the standard serial mouse protocol [Ascb]. These data are retrieved by the WandaControl object from the serial interface and delivered to the InputManager.

Unlike the positional tracking data from the wand and head-tracking sensors, the Wanda widget inputs are used both for navigation in space and to control the interactive context menu system (see Section 4.2.4). The InputManager maintains a stack of Handler objects: Wanda Handler is the default; Menu Handlers and Task Handlers (there are multiple kinds of each, corresponding to different graph tasks) are pushed on the stack and popped when completed. The InputManager delivers Wanda input events to the Handler on top of this stack.

Networking A framework for multi-node passive stereographics was introduced in Section 3.1.3. Although it is not shown explicitly in Figure 4.2, the network layer integrates into the asynchronous data flow model. A full description of the network transport system is given in Section 4.2.3.

The ContextManager and GraphContexts and all component objects (everything inside the shaded grey GraphContext box) are shared between the primary node and the secondary nodes. Updates to the attributes of these objects, and the creation and deletion of component objects (*e.g.* vertices and edges in the Graph) are delivered by the Twisted layer to their copies on the secondary nodes. The GLCanvas on each secondary node then retrieves the data from the local object copies when drawing as described above.

4.2.3 Networked Multi-Headed Display with the Twisted framework

Twisted is a widely used framework for Python network application development, using the well-known reactor pattern for asynchronous event handling [Sch95]. It includes modules supporting many common internet protocols as well as a framework for implementing

new protocols and services. Of particular utility is the "Perspective Broker" remote object framework which allows complex objects to be accessed or copied over the network in a *translucent* manner: "while remote object calls have different semantics than local ones, the similarities in semantics are mirrored by similarities in the syntax" [ZL02].

With Perspective Broker, complex data structures can be shared with or accessed by remote nodes through the networking layer with relatively few changes to existing code (compared to creating an object serialization and update protocol from scratch) and a minimum of overhead. The translucency is accomplished by using *proxy objects* which represent remote objects and provide object methods (*e.g.* *callRemote*) that allow remote procedures to be accessed. Data returned from remote procedure calls, where required, are handled asynchronously using the standard Twisted mechanism of returning *Deferred* objects to which callback functions are then attached. These functions are called by the Perspective Broker layer with the return data values once they are available. A full treatment of the Deferred asynchronous event handling mechanism is well beyond the scope of this thesis—for more on this topic, consult the Twisted documentation [Twi] or "Twisted Network Programming Essentials" by Abe Fettig [Fet05].

Twisted also provides a flexible authentication infrastructure to regulate connections and access to objects through the network interface. The GLuskap system typically uses an isolated network segment, though, and peer authentication mechanisms were deemed unnecessary in this case. If the system were to be regularly exposed to an open network while in use, the Twisted credential mechanism could be used to restrict access to authorized clients.

Nodes and network setup As described in Section 4.1.3, the distributed passive stereographic display uses a primary node and two secondary nodes. The primary node laptop runs the full GLuskap software; both secondary nodes run a different program called

the *GLuskap network client.* The Twisted Perspective Broker protocol runs over standard TCP/IP.

In order to initialize the system, the GLuskap software is started on the primary node. As the secondary nodes have no directly connected input devices, VNC software is used to connect to the console interface on each secondary node from the primary node and start the GLuskap network client. The network clients immediately connect to the instance of GLuskap running on the primary node.

A weakness of this system is that if the primary instance of GLuskap needs to be restarted for any reason, the network client instances must be re-connected to the new primary instance through the use of the VNC remote console interface [RSFWH98]. An alternative solution under consideration would allow the "network client" software to listen for connections initiated by the primary node. In this scenario, the secondary nodes would return to an idle state in case of primary node shutdown or connection failure. The underlying Twisted subsystem and Perspective Broker object sharing protocol do not make a functional distinction between "server" and "client" after the TCP/IP connection is established, so no changes to the data structure and object sharing code would be required.

Sharing the graph data structure over the network A bandwidth-intensive strategy for producing the stereo 3D views on the secondary nodes would be to render each perspective view on the primary node and deliver them to the secondary nodes as video streams to be displayed. Even if the frames are rendered using accelerated graphics hardware on the primary node, the computational load on the primary node is at least doubled[3] and the network bandwidth consumed rises quadratically as the display resolution increases (while a 1024×768 pixel display has twice the linear resolution of a 512×384 pixel

[3] If the perspective view destined for one of the secondary nodes is re-used for the local display on the primary node. In order to present a proper monocular view (without the left- or right-eye bias present in the perspective views displayed by the secondary nodes), the primary node would have to render three times as many frames.

display of the same physical size, it contains four times as many pixels.) While there exist streaming video compression algorithms, a small movement of the camera viewpoint in the virtual scene can result in changes to the entire frame (hampering the effectiveness of compression algorithms that transmit the difference between frames), and high-resolution video compression algorithms tend to consume large amounts of CPU cycles, especially in the compression step. The net outcome of this approach would likely be to unnecessarily overload the primary node and the network bandwidth.

It is possible to avoid this by communicating the parameters of the objects and the scene to be displayed rather than fully realized 3D views. The Perspective Broker system provides several mechanisms for translucent communication between objects across the network channel. A first naïve approach would be to expose the GraphContext data structure to data retrieval procedure calls through the network. For each frame rendered by the GLCanvas on a network client, it would request the required data from the GraphContext of the primary instance. While each client would be guaranteed to have the freshest object data for each frame, considerable amounts of network bandwidth would be wasted delivering duplicate data for successive frames; in the majority of cases only a few attributes are changed on a few objects from frame to frame, if at all.

To eliminate a good deal of the redundant data transfer in these two cases, we can additionally make our data objects *Copyable* (a superclass, part of the Perspective Broker framework). Making our existing objects Copyable requires only that they inherit from the class Copyable and provide a *getState* method that selects the attributes which are to be transmitted to the proxy object on the remote system when the object is first requested and when it is updated. By making the Graph, Camera, Drawables, GraphContext, and other related objects Copyable, their attributes and relations need not be transmitted over the network to each client for each frame. The contents of these objects are updated only when changes are made on the primary node, saving a considerable amount of bandwidth

and allowing frames to be rendered using the accelerated hardware of the secondary nodes.

Even if we use Copyable objects, an update of a single property on a complex object will require that the entire object be serialized and transmitted to both remote clients. Perspective Broker provides an even more efficient solution for data structures involving complex objects: the Cacheable superclass. In addition to using the *getState* method to initially copy the state of the object to the remote proxy, Cacheable objects must also inform their remote proxies (using a remote procedure call) about changes to attributes. In this way, only the new or changed data needs to be propagated across the network layer; how much or how little data is sent is determined by the implementing object.

The GLuskap software uses a Cacheable implementation as described above. The ContextManager is shared by the primary node to all connected secondary nodes, along with all GraphContexts and component objects and the ContextManager pointer to the active context. This allows multiple graphs to be loaded and switched from the window interface; once the context and data of a loaded graph has been transmitted to the secondary nodes, switching to display it requires only changing the active context pointer and reflecting the change on the remote nodes. The dataflows to shared objects are handled by *update* remote procedure calls from the objects on the primary node to their remote proxies.

Interface implementation with Drawables The Drawable class is conceptually simple: Drawable objects can be added to a list stored on a GraphContext. When each frame is being rendered, the *draw* method on the Drawable object will be called by the GraphContext. Drawable objects are Cacheable and can thus be passed through the Perspective Broker layer to the secondary nodes.

The power of Drawables is expanded considerably by granting them access to the other parts of the GraphContext data structure. For instance, a Drawable object can query the current position of the Cursor so as to draw its associated 3D objects in proximity to the

Cursor. This is in fact how the interactive object placement mechanism described in Section 4.2.4 is implemented: the Task Handler creates a Drawable that tracks the position of the Cursor and draws a semi-transparent vertex[4] at the nearest grid point. When the user clicks the left Wanda button to signify acceptance of the vertex position, the Task Handler retrieves the last used vertex position from the Drawable before destroying the Drawable and creates a vertex in the graph at the same position. This data exchange is shown in Figure 4.2 as a data edge from the Drawables directly to the graph elements for simplicity.

Drawables are used for drawing graph objects and other associated display objects (*e.g.* selection highlights) for interactive selection and positioning tasks, and also to display the interactive context menus (also described in Section 4.2.4).

Latency and synchronization issues On an isolated network segment, the network latency for small update packets is minimal (less than 10ms). With available unidirectional throughput estimated at 40Mbit/s, 225KB of data can be transmitted to both secondary hosts in 100ms (including 10ms latency but not processing delay).

While individual data updates must necessarily be initiated and transmitted sequentially (to one secondary node and then the other), most frame-to-frame updates such as repositioning individual vertices or moving the camera or 3D cursor involve small amounts of data per packet. In practice this means that both secondary nodes remain synchronized such that the ordering of updates is undetectable by the user.

4.2.4 3D Navigation and Interaction

GLuskap 2.4 features a standard window system interface (using the wxPython library) with embedded OpenGL three-dimensional graph display. In the new GLuskap system, the primary node retains the window system interface for comprehensive control (see Figure 4.3).

[4] Using the same underlying procedure used for drawing the existing graph vertices.

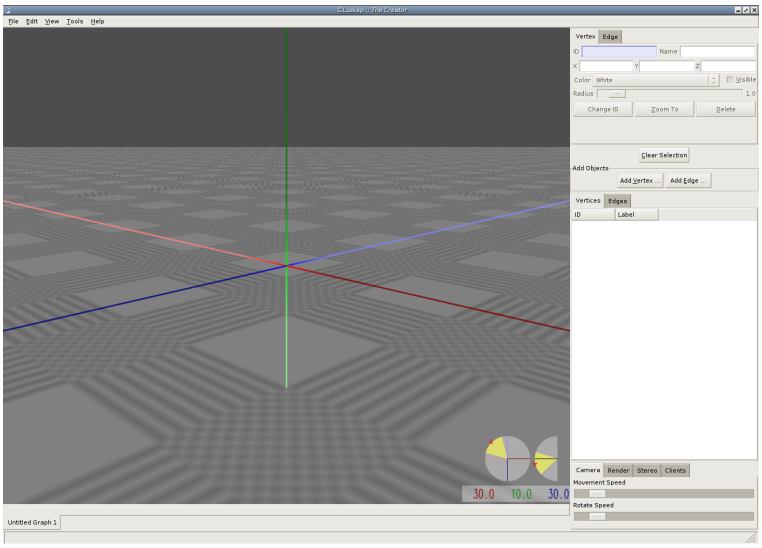

Figure 4.3: Primary Node Window System Interface

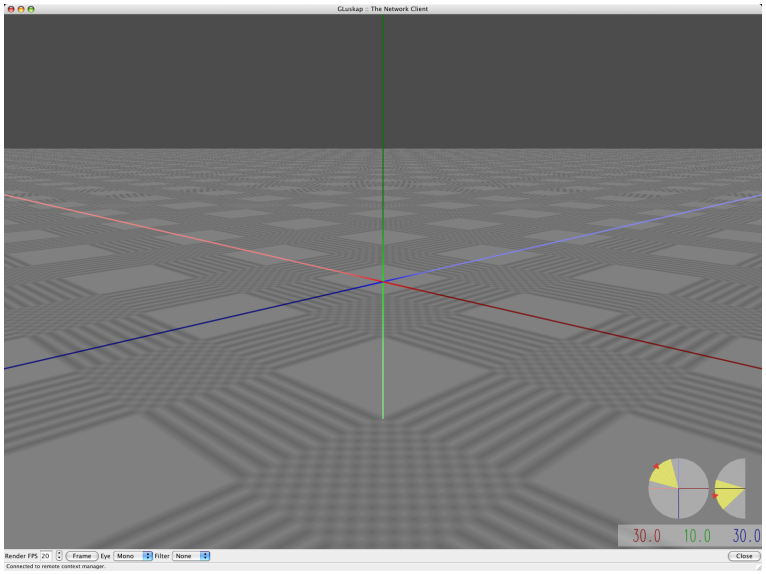

Figure 4.4: Fullscreen 3D Interface

A new context-sensitive interface has also been added allowing the user to perform interactive navigation and manipulation tasks using the Wanda device, without resorting to the window system interface.

This full-screen environment, shown in Figure 4.4, eliminates most of the window system GUI interface widgets. The window system elements that are present are not used for interaction, only for initial configuration of the GLuskap network client software during system initialization. Attempting to use window system widgets for interaction (menus, prompts, selections, etc.) is complicated by the network layer interposed between the user input devices and the display device: for example, presenting a dialog box would require the software to synchronize the operation, appearance, and destruction of a native window system dialog box between both secondary nodes. The wxPython interface to the window system does not provide for the fine-grained "remote control" over native widgets that would be required to accomplish this effectively.

All interactive elements are drawn within the OpenGL frame by the GLuskap software. By using Drawable objects that can be shared across the network layer as Cacheables and drawn by the GLCanvas on each computer, the GLuskap system can provide a dynamic interface that is consistent across primary and both secondary nodes.

Active HUD for feedback Along the lower margin of the OpenGL frame, a "Heads-Up Display" (HUD) is superimposed. This display of status elements supplements the perspective graph view by informing the user of the three-dimensional position of the cursor, the orientation of the camera view relative to selected vertices and the coordinate axes, and provides text feedback on the progress of interactive operations. These are drawn as semi-transparent surfaces at the depth of the camera, so they overlay other elements of the scene at the same display position. Stereoscopically, the HUD appears to lie at the same depth as the screen.

Basic 3D navigation User navigation through the virtual environment allows three-dimensional graph drawings to be visualized from different viewpoints and also allows the user to work more closely with one part of a large drawing. In the GLuskap window interface, the orientation of the camera is controlled by dragging with the mouse on the OpenGL frame, and linear movement of the camera is controlled by the keyboard. The challenge in developing a navigation interface for the fullscreen interface is to maximize freedom of movement while maintaining consistency with a limited input device.

Using the wand or head position tracking data for direct navigational control breaks the 1:1 coupling between natural head and hand movements and their analogs (camera and cursor) in the virtual display; in particular, the effectiveness of head movement for absolute movement and rotation is hampered by the limited range of motion (within a 1.2m radius of the transmitter) and the fact that if a user turns his head too far to one side or another, he will lose sight of the screen.

We can use the mouse and keyboard in the window system interface to control the view even when the large-screen interface is being used. Mouse input through the window system necessarily uses the window system pointer, though, and requiring one to alternate one's focus between the large-screen 3D interface and the window system interface on the primary node to keep track of the position of that pointer negates much of the advantage of using the fullscreen interface.

The primary navigation control system for the large-screen interface therefore involves the controls present on the Wanda device. With only two input axes available on the physical directional control widget, the functionality of this input must be constrained to two of the six degrees of freedom for three-dimensional navigation. This limited functionality could be extended by allowing the user to switch between subsets such that two DOF are accessible at any given time.

As implemented, GLuskap provides a navigation control interface that conforms some-

what to the "turntable" metaphor described by Ware [War04, Ch. 10], belonging to the larger class of "world-in-hand" interface metaphors. Movement on the left-right (X) axis of the directional widget maps to "orbital" movement of the viewpoint around the centre point[5] of the graph: from the user's point of view it appears as though the graph model is being rotated in place. The forward-backward (Y) axis of the widget is mapped to forward and backward movement, which in the typical case where the view is centered on the graph model is equivalent to zooming in on the part of the graph nearest to the camera. For simplicity, modal selection of alternate movement bindings for this input device is not implemented; this results in some changes in camera orientation and position being directly available only through the window system interface running on the primary node.

Although hampered by the lack of a 3 or more DOF input control scheme on the hand-held controller, this configuration allows the graph drawing to be evaluated from many different viewpoints and allows most parts of the drawing to be brought into range for manipulation with the cursor (adding and modifying graph elements). The "turntable" orbital rotation feature also enhances the three-dimensional perception of the graph model through the kinetic depth effect (Section 2.2.2). The Wanda buttons are used for interactive tasks as described below.

Head-tracking for direction and position offset Movements of the user's head are tracked by a Flock sensor affixed to an adjustable-fit ballcap. The distance between the hat sensor and a reference point is used as an offset to the position of the viewpoint in the virtual space. This reference point defaults to the position of the Flock reference transmitter and can be zeroed (set to the current position of the sensor) by the user on request. Rotations of the user's head in the up-down and left-right directions are also applied to the camera view; this is also relative to a calibrated reference position. With a large-screen

[5] Determined by calculating the mean of all vertex positions.

display, rotational head tracking is useful only as far as the user can maintain the screen in the user's field of view.

This allows naturally occurring minor shifts in position and orientation, as well as deliberate head movements, to be reflected in the three-dimensional display. Animation of motion parallax effects is an effective depth cue, independent of and complimentary to the stereo depth effect and other cues (see Section 2.2.2).

Interactive highlight-and-select The most common graph manipulation tasks, selecting and positioning graph elements, have been made the most easily accessible in the fullscreen interface. The left Wanda button is used consistently for selecting and confirming actions, consistent with left mouse button usage in popular window systems.

The 3D cursor represents the Wanda device as a conical pointer with a three-dimensional coordinate position and orientation in the virtual scene. One approach to selecting objects, logically extending two-dimensional conventions, is to force the user to position the cursor so that its tip is enclosed by the three-dimensional volume of the object to be selected and then click the left Wanda button. This conforms to the Fitts' Law model of difficulty, and is affected by human and machine lag as described in Section 2.2.3. As well, evaluation of the distance from the cursor to a target object in the depth direction is more difficult than the horizontal or vertical distance, even with stereoscopic and other depth cues.

The task can be made easier in two ways. First, we can provide feedback to the user indicating when an object is *selectable*; that is, when an input click will result in that object being selected. In the above example, an object is selectable when the tip of the pointer is enclosed by the object's three-dimensional surface. Research indicates that for selection tasks, both visual and auditory feedback are effective, especially in visually stressful conditions [AMH95, FG00]. Visual feedback in the form of "mouse-over" highlighting of interface elements is found in many two-dimensional interfaces, and the extension to high-

lighting three-dimensional objects when they are selectable is straightforward. Auditory feedback can be as rudimentary as a "click" or other simple sound played when an object becomes selectable and still be effective.

Second, we can enlarge the three-dimensional *picking volume* within which the cursor must be positioned to make an object selectable. With appropriate interaction feedback, the enlarged selection area need not have a direct visual representation in the interface—it is experienced by the user only in that an object becomes selectable when the cursor is within a certain distance. In the case where the cursor is within the picking volume of two or more objects, the closest object to the cursor is selectable and should be highlighted.

Both visual feedback and target size enlargement are implemented in the GLuskap interface to enhance the user's performance in selecting objects and mitigating the effects of end-to-end system lag. If no object is already selected, holding down the left Wanda button engages a selection mode for picking vertices and edges. In this mode the nearest object to the cursor, within a *picking tolerance* (a set distance from the cursor, effectively equivalent to the amount by which the picking volume of each object is enlarged), is highlighted. If an object is highlighted when the button is released, that object is selected and a selection highlight (distinguishable from the picking highlight) remains on the object.

Auditory feedback is not included in GLuskap at this time.

Selection serves to designate graph objects to be acted upon by various operations. Clicking the middle Wanda button while an object is selected will immediately clear the selection.

Drag-moving vertices If a vertex object is selected, holding down the left Wanda button engages a *drag movement* mode, where the vertex is moved (with visual feedback) corresponding to movements of the cursor. Edge segments cannot be directly moved as they are necessarily tied to the positions of their incident vertices, though the bend points of bent

edges can be selected and moved like vertices.

GLuskap implements a *snap* feature for object placement: the position of the cursor is filtered to the nearest point on a three-dimensional grid of fixed intervals. This is especially appropriate in the three dimensional graph drawing application domain as vertices are typically constrained to lie on integer grid points (Section 2.1.2). The snap interval for placing objects can be set by the user depending on situational requirements. If required, object position coordinates can be specified with arbitrary precision[6] by keyboard input through the window system interface.

Interactive object creation New graph objects are created by selecting the type of object to be created from the root context menu (described above). In the case of a new vertex, a semitransparent "ghost" vertex is created and follows the cursor position, appearing at the nearest snap grid point. The coordinate position of the new vertex is displayed in the left side of the HUD. The new vertex can be fixed in place by clicking the left Wanda button: the ghost vertex will be replaced by a standard vertex.

Creating a new edge is a more involved procedure. After selecting the "Add Edge" menu option, the user must select a vertex (following the same closest object to cursor highlight-and-select procedure as described above). A second vertex must then be selected, and an edge is created and displayed between the two selected vertices. Status feedback and prompts are given in the HUD during the procedure. If the user attempts to create a loop (an edge incident to only one vertex) or a redundant edge (between two already-adjacent vertices), an error message is displayed. The task can be aborted at any time by clicking the middle Wanda button.

[6] Within the limitations of platform floating-point accuracy.

Selected object	*Menu options*
None	Add vertex
	Add edge
Vertex or bend point	Delete
Edge (plain)	Add bend
	Delete
Edge (subedge)	Add bend
	Select parent
Edge (superedge)	Delete

Table 4.1: Context menu structure

Context menus with Wanda 2D widget Most tasks are initiated through context-sensitive menus that are drawn as semi-transparent surfaces in the middle of the OpenGL display. The right Wanda button is designated for context-sensitive operations. If no menu is currently displayed, clicking the right Wanda button will bring up a context menu appropriate to the current selection status. The forward-backward (Y) axis of the Wanda navigation widget is used to move the highlight between menu selections; clicking the left (select) or right (context) Wanda buttons will select the currently highlighted menu option. Clicking the middle Wanda button while a context menu or task initiated from a menu is active will dismiss the menu or cancel the action. Table 4.1 lists the context menu options displayed for each type of selected object.

When no objects are selected, a root menu is presented that allows the user to initiate creation of graph objects (described below). The context menu for an edge or vertex lists several actions that can be performed on the selected object.

Several menus for edges are listed in Table 4.1; different menu options are presented for a selected edge depending on the type of edge. A *plain edge* is a straight-line edge without bends. *Subedges* and *superedges* are used in the construction of bent edges in polyline grid drawings: a subedge is a single straight-line segment incident to a vertex and a bend

point or two bend points, whereas a superedge represents the sequence of subedges forming a complete bent edge incident to two vertices. A subedge necessarily has a "parent" superedge, and each superedge must contain at least two subedges.

A plain edge can be "bent" by choosing the "add bend" option from the edge context menu. A bend point will be added to the edge and placed by the user, similar to adding a new vertex. A plain edge can also be deleted from the graph.

Like plain edges, subedges can be selected as described under "Interactive highlight-and-select", but superedges cannot be directly selected. A selected subedge instead provides the option to select its parent superedge. While a bend can be added to a given subedge, a subedge cannot be deleted directly. To reduce the number of segments in an edge, the user must remove bend points instead; they can be selected and removed like vertices.

Bend points cannot be added to superedges (again, due to ambiguity). Superedges can, however, be removed from the graph.

4.2.5 Plug-In Architecture

The GLuskap plug-in scripting framework provides a mechanism for users and developers to implement fully functional graph processing and drawing algorithms while remaining insulated from the core GLuskap code. Plug-ins are self-contained Python scripts that are launched from within the GLuskap user interface and have access to the active GraphContext, including the Camera and all graph elements. Even in the case where GLuskap is distributed as a packaged executable file, rather than as editable Python source, plug-in scripts can still be created and edited as plain source files.

Plug-in scripts can interact with the user. A script can request a value to be entered, a selection to be made from a list of choices, or simple yes-or-no confirmation decisions. Mes-

sages can also be displayed to the user, either through interactive mechanisms or through the status area of the heads-up display.

Using the standard Python import mechanisms, a plug-in script file can import modules from the GLuskap codebase or from the Python standard library. In the latter case, only a subset of the standard library may be available if GLuskap is being run from a packaged executable file, as this distribution format does not require a full Python runtime environment to be installed on the user's system. Only components of the standard library that are used by the GLuskap core code, or are explicitly designated for inclusion by the developer performing the packaging, will be available. These include standard math functionality, string and array handling, and colorspace transformations. The GLuskap package also includes a selection of three-dimensional vector operations (including addition, subtraction, multiplication and division by scalars, normalization, arbitrary axis rotation, dot product) that can be used by scripts.

Python's exception handling is used throughout the GLuskap codebase including the plug-in scripting subsystem. Any unhandled exceptions raised during execution of a plugin will interrupt the execution of the plugin and exception information (type of exception, value, and stack trace) will be displayed to the user to aid in debugging the plug-in script. More details on Python exception handling are available in the official documentation [vR05, Sec. 4.2].

API details A plug-in script file must provide a *run* function, which is called by the GLuskap program when the plug-in is launched by the user. This function is passed two objects by reference: the GraphContext, and a UI object that provides methods for interacting with the user.

Graph and Camera instances are accessible as properties of the GraphContext. These objects are the same data structures used by the GLuskap core to represent the graph draw-

ing and the viewpoint parameters respectively.

The Graph data structure is based around a list of vertices and a list of edges. These lists contain Vertices and Edges: code objects that contain methods and attributes appropriate to each type of graph element. Vertices have an *id* attribute (character string) that is required to be unique among all vertices in the graph. Edges are uniquely specified by their *source* and *target* Vertices— while GLuskap does not support true directed graphs, the source and target notation is used for convenience. The bend points of bent edges are instances of DummyVertex, a subclass of Vertex, and are stored in the graph vertex list. Likewise, the subedges and superedges used to implement bent edges (described in Section 4.2.4) are variants of Edge and are stored in the graph edge list.

Where possible, the methods and attributes related to Edges and Vertices are located on those objects. However, in some cases (particularly where an operation would involve adding or deleting graph elements, for example adding a bend point to an edge), the methods are located on the Graph object such that the method has access to the vertex and edge lists.

Adding elements to the graph is a two-step process. First, a new object of the desired type (Vertex or Edge) is created and its attributes set. Next, a method is called on the Graph object (*e.g.* *addVertex*) to add the new element to the graph. At this step, the element will be checked for uniqueness (*id* of a vertex, *source* and *target* of an edge) and the object will be added.

Likewise, elements can be removed from the graph by use of the *removeVertex* or *removeEdge* methods on the Graph object. These methods remove the specified object from the vertex or edge list of the graph and ensure that references to the removed object from other graph elements are also removed. If the calling code maintains a reference to the removed object, it will be preserved, otherwise it will be garbage collected.

Because of optimizations in the OpenGL drawing pipeline, updates to the Graph object

and any Vertices and Edges must be announced by setting a *dirty* flag on the Graph. All methods on the Graph object will set this flag automatically. In cases where attributes of Vertices or Edges are being modified directly (color, size, position, etc.), the following syntax is used:

```
def run(context, UI):
    # Grab the first vertex from the graph vertex list
    G = context.graph
    v = G.vertices[0]

    # This will not update the dirty flag:
    v.pos = (0.0, 1.0, -1.0)

    # This will:
    G.modVertex(v).pos = (0.0, 1.0, -1.0)
```

The *modVertex* and *modEdge* methods of the graph class first check that the specified object is present in the graph, set the *dirty* flag, and then return the object. This allows the inline usage of the *mod* methods as shown.

Latency and synchronization issues In the current state of implementation, using plug-in scripts with the networked rendering setup can result in temporary yet noticable desynchronization of the networked displays. It is hypothesized that this undesirable behaviour stems from the method by which updates to Cacheable objects (including the graph data structure) are handled.

To ease integration with existing code, these objects were adapted to the Perspective Broker framework by adding a call to the Cacheable update method on every change to the properties of the object (see Section 4.2.3). This ensures that individual objects are updated

quickly on the secondary nodes. Under interactive operating conditions, the processing time and network latency for these updates is minimized by comparison with the reaction time delay between user operations.

When a plug-in script is executed, though, many graph objects may be created, updated, or removed in a very short period of time. The overhead of each operation being processed separately can compromise the responsiveness of the large-screen display for a period of seconds. This is highly detrimental to the user experience.

One possible solution would be to providing an interface for plug-in scripts to explicitly notify the networking subsystem of large numbers of updates to cached objects. Temporarily disabling the automatic processing of property updates in favor of updating the entire data structure following a batch of updates could minimize the per-transaction overhead of the Perspective Broker mechanisms. Further investigation of this area is required.

4.3 Software Engineering Techniques

The GLuskap system is a complex piece of software with several integrated subsystems. The codebase comprises more than 11,000 lines of Python and is the product of three years of development by four programmers (at different stages of development).

Brooks [Bro95] differentiates between *programs, programming systems,* and *programming products.* A program performs a useful function, but only for the developer in the environment in which it was constructed. The programming product is a program made portable and robust, maintainable and usable by others, and comprehensively tested and documented—and it costs on the order of three times as much to develop as the simple program it is based on. A programming system is an assemblage of subprograms, integrated to perform a larger task or set of tasks. In this case, the added complexity comes from defining interfaces for each subprogram such that they can reliably communicate between

themselves, and from the system integration testing that is required in addition to the testing of each subprogram in isolation. The programming system again costs three times as much as a single program.

The *programming system product,* then, is the logical intersection of these two extensions to the simple program—an integration of subprograms that is maintainable, reliable, tested, and documented. By Brooks' estimation, in a development context the programming system product costs nine times as much as the simple program that may have been at its core. It is the contention of the author that the GLuskap software falls into this most useful and most costly of categories. As described below, GLuskap integrates several interdependent subsystems using defined interfaces. Furthermore, efforts are made to refine the product for distribution and operation in a variety of environments by many users.

The development of these complex products requires care and attention to a degree that simple programs do not. No software product is perfect; here we define perfection in the sense of performing all possible tasks with 100% correct functionality. New uses for existing software are conceived; new algorithms are developed that improve on the speed, reliability, or functionality of existing implementations. For these and other reasons, software products go through a *life cycle*. After the initial development and testing, the software is released. User feedback and evolving requirements stimulate further development and testing, leading to additional releases. An entire field of research into *software engineering* is focused on methods and processes for software development that improve quality and reliability.

With a small user base and development team, the GLuskap development process has not adopted a formal software engineering process. Effective concepts and techniques have been adopted where possible in order to increase the maintainability and overall quality of the software product.

4.3.1 Object Orientation and Modular Interfaces

While it supports several programming paradigms, including functional and traditional procedural methodologies, the Python language has a well-developed object model and is optimally suited to object-oriented programming. The critical wxPython and Twisted libraries both use object-oriented interfaces. While the Python interface to OpenGL mirrors the standard C/C++ procedural interface, access to OpenGL routines has been isolated in relevant GLuskap code objects, which form a consistent wrapper around the procedural aspects.

It is natural, then, that GLuskap development should follow an object-oriented approach, building the software product from *code objects* which tie together *data objects* with applicable *methods*. While each code object implements an interface with which it interacts with other objects, the relationships between these objects can be organized in such a way as to provide straightforward and consistent interfaces between *subsystems*.

GLuskap interfaces Consistency of interface minimizes the amount of overhead required on the part of the developer in understanding the details of the structure and implementation of the subsystem. As an example, GLuskap represents graph data structures internally as cross-referenced lists of vertices and edges along with other associated data, bound up in the Graph, Vertex, and Edge classes. In some cases, simplicity of implementation has been sacrificed for optimization of execution speed.

Simply adding a bend point to an edge requires a considerable amount of bookkeeping in order to properly handle different circumstances and to ensure that the data structure remains internally consistent. From the point of view of a developer not working with the Graph class implementation, though, the interface to this task is encapsulated in the Graph.*bendEdge()* method, which takes an Edge object and the coordinates of the new bend point as parameters and returns the new DummyVertex object or raises an exception if the operation cannot be performed. It is the responsibility of the Graph implementor to

ensure that the code behind this interface behaves "as advertised."

Code re-use Consistent interfaces and *factoring* of abstract routines allow for effective *code re-use:* Python's data type model is both dynamic and strict [vR05, Sec. 3.1], maximizing the potential for abstraction of algorithms and data types while minimizing unpredictable results. Effective code re-use has been shown to reduce the incidence of errors in software development [TDB92, BBM96]. Furthermore, errors can be more rapidly found and fixed in re-used procedures than in multiple implementations scattered through the code base.

Figure 4.2 gives an outline of the code objects involved in the data flow process of the interactive user interface. Other subsystems, like the wxPython structures for managing the window system interface, the Twisted network connection management infrastructure, and the routines for loading and saving graph data from supported file formats, communicate with these objects over straightforward object-oriented interfaces.

4.3.2 Change Management With Subversion

As the aphorism has it, Rome was not built in a day. The software development process is similarly time-consuming and incremental. New routines are implemented, tested, and debugged. Existing routines are at turns revised, re-tested, factored, and combined. These processes often take place over time intervals that exceed the span of human working memory. Maintaining a grasp on the motivations and history of changes in a software implementation is aided by the use of a *change management* strategy.

To a certain extent, changes can be documented by annotating the source code with embedded comments, but if this is used exclusively, the accumulation of revision documentation over time would become cumbersome, redundant, and negatively affect the readability of the code. Attempting to preserve the full history of revisions made to individual lines of

source code in this way would be inconceivable, due to both the heavy responsibility placed on the programmer and the sheer volume of historical comments that would be generated.

The need for a systematic change management solution is magnified when more than one programmer works concurrently on the same project. Changes to the source code become a form of communication between developers. Not only the history but the provenance of changes must be recorded. Individuals collaborating in development of the same code modules may attempt incompatible changes to the code at the same time—how is this to be resolved?

Change management software There is a long history of change management software products designed to address these issues, dating to Rochkind's Source Code Control System [Roc75]. SCCS was largely superseded by Tichy's Revision Control System [Tic85], which was then integrated into the CVS package which supports multiple developers working in a networked environment [Ced03]. CVS is still widely used by many development teams, especially in the free/open source software community, but suffers from several key weaknesses.

All these products have core principles in common. Sets of changes to source code files are stored in sequence and the *revision history* is accessible to the user. When changes are *committed,* that is, the change management system is instructed to save the current state of a file or files in the revision history, the user is prompted to attach some descriptive comments to this set of changes. These comments are useful not only for producing a human-readable summary of the changes to the software over a given time period, but also for the developer in understanding the motivations behind each particular set of changes.

Subversion and CVS Subversion was created as a direct replacement for CVS, designed to correct the deficiencies of CVS without drastic changes to the general philosophy and

workflow processes. This approach makes it easy for users to migrate from one system to the other. Like CVS, Subversion uses a network-accessable *repository* from which developers *check out* a copy of the source code; changes that they make must be *committed* to the repository in order to be shared with others. Where CVS supports only tracking of revisions on individual files (not directories) and does not support the renaming or relocation of files,[7] Subversion manages files, directories, and metadata by tracking revision numbers across the entire repository tree. The Subversion repository access protocol is improved in both efficiency (only differences, not entire files, are sent across the network) and robustness (commit operations are *atomic*— a set of changes is committed together or not at all) [CS02, CSFP05].

GLuskap development and Subversion Through the development cycle of GLuskap 2.4, change management was introduced using CVS. After a short period wherein the disadvantages of the latter system were encountered, the project was migrated to Subversion.

The "branching" feature (common to both CVS and Subversion) allows multiple versions of the same software product to be maintained concurrently. The GLuskap 2.4 release and the latest development version (the "trunk") of the GLuskap system are both accessible in the repository. When changes are made to a part of the source code that is identical in both versions, these changes can be easily applied to both branches of the repository. When the development version of the GLuskap system is deemed ready, it will be copied into a new branch. Testing in preparation for a release can be carried out and fixes committed to this branch, while new features continue to be added on the trunk.

While GLuskap is usually distributed to end users as a self-contained package precompiled for a specific platform, or as a source code archive of a particular revision, interested users can obtain development snapshots directly from the Subversion repository over the

[7] These operations result in the loss of version history for the affected files.

Internet.

Documentation for GLuskap, including the user's manual, is also maintained with Subversion. LaTeX files in particular are well suited to revision control, as the differences between revisions are human readable (unlike the binary file formats used by some other document preparation and word processing packages). In this way, changes to the documentation can be annotated and tracked; "released" editions can be preserved while revisions and new material are tracked on a separate branch. Managing contributions from multiple authors is similarly straightforward.

Chapter 5

Evaluation

5.1 Other Interactive Graph Drawing Systems

It is useful to examine the GLuskap software in the context of other interactive 3D graph drawing products. GLuskap is the only package of those examined that has explicit support for large-screen stereographic visualization and specialized three-dimensional input devices. This sets it apart, certainly, but we assess here the feature set and capabilities of GLuskap relative to a selection of comparable software packages.

5.1.1 Historical GLuskap

Improvements in capability and usability have been made to the GLuskap software in the process of adapting it for use in the large-screen immersive 3D system. The software package remains usable independent of the specialized hardware used in the GLuskap system, though some features are necessarily dependent on the presence of specific hardware components. GLuskap is freely available and licensed under the GNU General Public License [FSF91].

Window system interface GLuskap 2.4 and the new GLuskap software provide graphical windowed interfaces using the wxPython libraries and with an interactive 3D view rendered in an embedded OpenGL frame. The look and feel of the GLuskap window interface has not been substantially changed, though in some places widgets have been added or reorganized to support new features.

Both products are tested on Microsoft Windows, Linux, and Macintosh OS X.

Three-dimensional input systems True three-dimensional control over object positioning and selection has been made possible through the support for the position-tracked Wanda input device. The interfaces created to support this particular hardware configuration, including the representation of the input device as a pointer embedded in the virtual 3D space, are designed to be extensible. Support for additional hardware devices and configurations is feasible with the development of software modules that adapt the data format of new devices to the existing interfaces.

The new GLuskap interface to the Wanda device and Flock of Birds systems are supported only under Windows and Linux; Mac OS X systems typically lack the hardware serial interfaces required.

Plug-in system GLuskap 2.4 included some simple layout algorithms (including random and circular placement of vertices and a simple force-directed model). These were included as features of the core GLuskap software. For the user to modify or add new integrated algorithms of this type is non-trivial and involves modifications to the core classes of the GLuskap implementation.

The new GLuskap system introduces a plug-in scripting framework to address this. A selection of these scripts are distributed with the software package, including the straight-line "moment curve" algorithm of Cohen *et al.* [CELR97], the one-bend drawing algorithm

of Morin and Wood [MW04], the two-bend algorithm of Dyck *et al.* [DJN+04], and the simple force-directed spring embedding algorithm written for GLuskap 2.4 repackaged as a plug-in script. Along with some trivial plugins that demonstrate various features of the interface, the included algorithmic plugins are useful examples for users who wish to construct their own layout plugins.

File format support The file import and export modules have not changed significantly in the new GLuskap. GraphML [BEL04] and GML [Him96] are supported, in addition to the native MG2 file format.

Graph feature support The new GLuskap software, like GLuskap 2.4, supports ordinary undirected graphs without loops (edges with the same vertex for both endpoints) or multi-edges (two or more edges connecting the same pair of vertices). Both straight-line and bent (poly-line) edges are supported. Directed edge support is seen as a priority for future development; while a full implementation is not included at this time, steps toward adapting the codebase to this goal have been taken.

Both new and old packages support the attribution of vertices with size (floating-point ≥ 0) and color (floating-point RGB) values as well as arbitrary string labels. Edges can be attributed with size and color values. Support for changing the shape of the 3D representations of vertices and edges is not supported.

5.1.2 WilmaScope

WilmaScope is a 3D graph visualization package developed by Tim Dwyer [DE04] and others at the University of Sydney, Australia. Where GLuskap is designed primarily for working with drawing algorithms that specify explicit positions for vertices (and edge bend points), WilmaScope is based around several force-directed models for visualizing arbitrary

graphs. It is freely available under the GNU Lesser General Public License [FSF99].

Window system interface WilmaScope is written in Java and uses the Java Swing UI toolkit to provide a portable cross-platform interface. The Java3D interface to OpenGL is used for the animated 3D graph display. WilmaScope supports Microsoft Windows, Linux, Macintosh OS X, and Unix environments.

Whereas the wxPython library used by GLuskap provides a uniform programming interface but uses each window environment's native widgets, the WilmaScope user interface is identical on all supported platforms but is inconsistent with the appearance of native applications in any specific environment. The latter is typical of many Java applications. Which approach is to be preferred is a point of contention among developers and interface designers.

WilmaScope requires that the Java runtime environment and the Java3D package be installed on the user's system; the latest Windows distribution packages include the Java runtime to be optionally installed if not already present. For Windows and Mac OS X systems, GLuskap is distributed in standalone packages that require no external libraries. This comes at a cost of increased download size (5.6MB for self-installing Windows package vs. 473KB for source distribution). By comparison, the WilmaScope 3.1 install package for the Windows platform is 62.4MB.

Three-dimensional input systems WilmaScope does not support any external 3D input devices for position input or navigation. This is less of a hindrance when considering WilmaScope's focus on visualization with force-directed models; navigation in three dimensions is more easily accomplished in a two-dimensional input context than interaction under the same circumstances. Specific positioning of vertices in three dimensions can be done by dragging vertices with the mouse; in order to change the position of a vertex in the

depth direction, the graph view can be rotated to view the graph from the "side" and the vertex then moved into place.

Plug-in system Several types of plug-ins can be written in Java by extending parts of the WilmaScope class structure. New graph generation algorithms, layout structures, and analysis algorithms can all be implemented through separate specific plug-in interfaces. By contrast, the GLuskap plug-in architecture provides a generic interface where scripts can be written to perform all these functions.

Layout algorithms in WilmaScope are applied continuously as graph elements are added or removed; in GLuskap, a plug-in must be re-executed after any changes to the graph data. The GLuskap plug-in interface also provides simple hooks to allow the script to interact with the user, presenting choices, requesting values or confirmation of actions, and displaying messages.

WilmaScope additionally supports plug-ins that modify the function and appearance of the graph display view; this functionality is not present in GLuskap.

A simple CORBA [Vin97] interface is also implemented in WilmaScope, allowing scripts to be written in any language that provides CORBA bindings. Example Python scripts using this interface are provided with the software package.

File format support Both the standard GML file format and an XML-based native format (WXG) are supported by WilmaScope. Unlike GLuskap, the XML-based standard GraphML format is not supported.

Graph feature support WilmaScope supports undirected and directed graphs and includes support for multi-edges. Polyline edges (as featured in GLuskap) are not explicitly supported. Several types of directed edge with distinct visual representations can be used

to represent semantic properties of edges, *e.g.* three-dimensional drawings of UML class diagrams [RJB99].

Clustering of vertices is well-supported. Users can designate clusters as vertex-induced subgraphs; the cluster is then visually differentiated with an enclosing translucent sphere. A nested hierarchy of clusters can be created. Several types of cluster are supported, with different appearance and layout of the vertices within the cluster.

5.1.3 OrthoPak 3D

OrthoPak 3D is an older package developed at the University of Lethbridge for work on 3D orthogonal graph drawing algorithms. Written in C++ and using the LEDA[1] GraphWin for the interactive interface, OrthoPak 3D is included here partly as an illustration of the relative dearth of software for interactive three-dimensional graph drawing (as opposed to software designed for data visualization). While OrthoPak 3D is described as being "available free of charge for teaching and research purposes" [CEGW98], the LEDA libraries are now commercially licensed and this restricts the effective availability of the software. Binaries are provided for Solaris and Linux environments.

Window system interface The main interactive window interface to OrthoPak 3D is based around a LEDA GraphWin and is two-dimensional in essence. The graph data structure can be modified, new graphs created from several patterns (complete, random, planar, etc.), and a variety of LEDA graph library functions is provided.

The included three-dimensional orthogonal drawing algorithms (described in [CEGW98]) are accessed from the GUI. Each outputs a drawing of the current graph using the particular selected algorithm as a VRML file which must be displayed using a separate viewer (not part of the OrthoPak 3D software).

[1] Library of Efficient Data structures and Algorithms; see [MN99].

Three-dimensional input systems There is no capacity for direct three-dimensional position input or navigation; this is consistent with the two-dimensional nature of the GraphWin interface.

Plug-in system No integrated system for extending the functionality of OrthoPak 3D is provided. New graph algorithms must be written in C++ (using the LEDA graph data structures) and the application recompiled.

File format support Reading and writing to and from the LEDA native graph format is supported, as is the standard GML format. Three-dimensional drawings are exported as VRML, as indicated above.

Graph feature support The underlying LEDA package supports directed and undirected graphs, including loops and multi-edges. Vertex clusters and explicit specification of hierarchical graph features is not provided. Vertices and edges can be attributed by color, shape, size, as well as user-specified arbitrary string values.

5.2 Best Practices For Visualization and Interaction

The intended use of the GLuskap system is as a cognitive tool for researchers in the area of three-dimensional graph drawing; such researchers form the natural population in which to assess the performance of the system. To best evaluate the performance of the GLuskap system and the various techniques implemented to enhance the capabilities and ease of the user, an in-depth user study would be required.

As such a study is beyond the scope of this thesis, the visualization and interaction features of the GLuskap system will be examined in light of related research and alternative implementation strategies in this area.

5.2.1 3D Graph Visualization Results In The Literature

It has been shown by Ware and Franck [WF96] that stereoscopic and kinetic enhancements contribute greatly to increased user performance on path-tracing tasks in three-dimensional graph drawing. They demonstrate a factor of 2.2 improvement in the complexity of the data that can be understood when head-tracking information is incorporated to produce motion feedback in the display, and a factor of 1.6 improvement (compared to baseline) when a stereoscopic display is used. When both technologies are combined, a factor of 3 improvement over baseline performance is observed.

There are significant differences between the test apparatus in this study and the GLuskap system. In particular, Ware and Franck used a desktop computer monitor for display, whereas the GLuskap system uses a large-screen display to approximate an immersive environment. The set of user tasks that are typically performed in the GLuskap environment is also much more complex than simple path-tracing: users interactively create, manipulate, and cognitively evaluate graph data structures and their three-dimensional drawings.

Despite these differences, their result would seem to indicate that further investigation of how these techniques affect user performance in interactive graph drawing tasks is warranted. Indeed, Ware and Franck themselves advance the case for use of advanced 3D graphics techniques in diverse applications involving complex data structures.

5.2.2 Three-Dimensional Cues

Ware [War04] identifies techniques that can be used in a graphics display to augment the perception of virtual objects and scenes as three-dimensional. These are related to the depth cues discussed in Section 2.2.1; here we consider the implementation and effectiveness of techniques that produce these depth cues in a graphics display. Stereoscopy will be considered separately in the next section.

Perspective and occlusion The GLuskap OpenGL display implements standard 3D graphics techniques for display of virtual scenes and objects using a linear perspective transformation. Z-buffering is used to ensure that near objects properly occlude more distant objects [SWN+03]. A ground plane is optionally drawn in the scene with texture to establish a sense of scale.

The usefulness of the ground plane as a depth cue is enhanced by techniques that establish the position of virtual objects on the plane. This is trivial for objects that contact or intersect the plane, as most objects in the natural world do (due to gravity). Objects that float in space can be "anchored" through the use of pseudorealistic shadows cast on the ground plane, or more explicitly by the use of vertical lines that extend from the object to the plane. The latter technique, while less "realistic", is easily understood by users and can also aid the accurate perception of an object's vertical distance from the plane; Kim *et al.* [KTS93] show that it is comparable to stereopsis in its effectiveness at showing 3D position.

Both of these techniques require additional computational power (shadows more so than vertical lines) and contribute to the cluttering of the display. Though there is no significant obstacle to doing so, neither have been implemented in the GLuskap interface at this time.

Illumination shading Basic simulation of shading from directional illumination is easily achieved with OpenGL. Based on the conclusions of Ramachandran [Ram88] regarding the model used by the brain in determining shape from shading, it is likely best results can be obtained from this technique by using a single virtual light source illuminating objects from above. GLuskap uses a single light source positioned directly behind the user. This is judged to be adequate as the objects used for three-dimensional graph drawings (spheres and cylinders) are simple; no semantic content is encoded in the shape of the individual objects which would be revealed through shape-from-shading.

If GLuskap were to be expanded to allow attribution of vertices and edges with shape information of sufficient subtlety, the lighting model could be enhanced to optimize user discrimination of subtleties of object shape.

Texture gradients and contours As discussed in Section 3.1.1, regular texture patterns help communicate the three-dimensional shape of irregular objects, as well as adding more generally to the perceived realism of the virtual object. Texture also greatly aids the stereoscopic depth effect by providing more discrete points of similarity that can be fused by the visual system.

The GLuskap drawing engine applies a regular checkerboard- and grid-like texture to all graph objects as well as the ground plane. The texture pattern is scaled so as to be visible at typical object sizes and working distances. Furthermore, the pattern is stretched along the long axis of the three-dimensional cylinders which represent edges, emphasizing their linear shape. Figure 5.1 presents a detail from a graph drawing, showing the texture patterns used. Though no formal studies have been conducted of the GLuskap implementation, it has been the experience of the author that texturing increases the effectiveness of stereoscopic viewing markedly.

5.2.3 Stereoscopy

GLuskap implements stereoscopy following established practices for OpenGL [Bou02], but is subject to several conceptual and practical issues. Ware [War04, Ch.8] outlines these common problems, including diplopia, frame cancellation, diminished stereographic effect with distant objects, and vergence-focus conflicts.

Cyclopean scale, a technique involving automatic manipulation of the parameters of the stereoscopic visualization in a transparent way, is described by Ware *et al.* [WGP98]. While it alleviates many of the aforementioned problems, the implementation of cyclopean

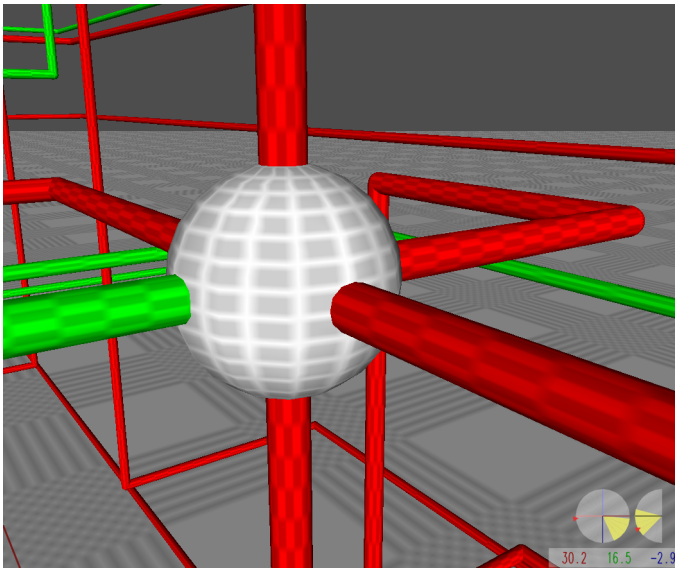

Figure 5.1: GLuskap Texture Patterns

scale requires the real-time assessment of the distance to the nearest object in the scene. The authors suggest that this can be accomplished in a straightforward fashion by sampling the OpenGL depth buffer (typically used to ensure that nearer objects occlude those more distant).

Though cyclopean scale has not yet been implemented in the GLuskap interface. the potential enhancement to the effectiveness of the stereographic display makes it a good candidate for future development.

5.2.4 Interactive Pointer and Menu Interface

The direct coupling of three-dimensional pointer movement to the corresponding movements of the physical wand device is a somewhat naïve approach to the underlying problem of selecting, positioning, and manipulating the attributes of three-dimensional structures in a consistent and straightforward way.

A possible alternative for the selection of objects is adapted from the two-dimensional

interface display of the three-dimensional scene. In the two-dimensional context, objects are selected with the mouse pointer which lies "in front of" the entire three-dimensional scene. If multiple objects lie on the line of sight emanating from the position of the mouse pointer, the nearest (*i.e.* visible) object is selected. By projecting a visible ray away from the user from the position of the embedded three-dimensional pointer and selecting the nearest object intersecting this "beam", the user would be able to clearly and consistently select objects that would not be within "reach" of the user under the current selection paradigm (see Section 4.2.4, "Interactive highlight-and-select").

The incorporation of a secondary input device for the non-dominant hand could also enhance the effectiveness of the GLuskap interface. In performing many real-life tasks, both dominant and non-dominant hands are used. Guiard [Gui87] presents the *kinematic chain* as a model for understanding the differences in functional role between the two hands: one hand establishes a frame of reference within which the other can act—imagine one hand holding a notebook while the other hand holds the pen for writing.

Two-handed control systems have been investigated in depth for two-dimensional interface design [BSP+93, SFB94, KFBB97]. The use of these techniques in the construction of three-dimensional interactive interfaces is less well-developed, though there do exist some implementations [SG97, CFH97].

5.3 Software Engineering Techniques

5.3.1 Development and Planning Model

As an exercise in software development process management, the GLuskap project has had mixed success. The GLuskap 2.4 development process was constrained to stabilizing and overhauling the existing codebase with a minimal set of new features. By contrast, the

continuing development of the GLuskap system has been characterized by the prioritization of new feature development over rigorous documentation and diligence. The push to implement the capabilities required for the GLuskap system, particularly the networked multi-head rendering system and the interface to the Flock of Birds motion tracking system, has resulted in maintenance of other lower-priority software subsystems being neglected.

Having only one developer participating in development, while contributing to conceptual consistency (the design is the product of a single mind), has definite disadvantages in the development and maintenance of complex systems. It is the experience of the author that working with a second programmer, particularly one who is similarly immersed in the codebase and development process, allows for the critical refinement of both design concepts and module code. This is supported by the incorporation of *pair programming* into the Extreme Programming software engineering paradigm [Bec00]: pair programming is the formalization of assigning two programmers to work simultaneously on the same code, to the extent of sharing a single workstation.

Change control with Subversion The aforementioned process control difficulties notwithstanding, the use of the Subversion version control system to help manage the development process has proved quite valuable. The implementation of significant bundles of related functionality was eased by the ability to create a *branch* and implement a contained set of changes without affecting the primary source code. Once complete and tested, the modifications to a branch can then be *merged* back into the main source code in a controlled manner.

Also valuable to the development and debugging process is the tracking of particular changes to the source code on a line-by-line, day-by-day basis. In combination with the ability to "roll back" the progress of development to any point at which changes were committed, this provides a powerful aid to debugging. When an undesired change behaviour

is discovered, the software can be methodically tested backwards in time until proper behaviour is found; the bug can then be readily isolated.

5.3.2 Code and Structure Model

Best efforts have been made to ensure orthogonality and effective demarcation of boundaries between particular subsystems and code modules within the GLuskap program structure. Without metrics to perform a quantitative analysis of complexity and entanglement on the GLuskap code base, it is difficult to draw strong conclusions about the effectiveness of this effort. The following examples are presented to provide some degree of insight into the structuring of the codebase.

Graph data structure Fundamental to the operation of the GLuskap software is the data structure used to represent the graph (and by convenience, the three-dimensional embedding and attributions thereof). Measures have been taken to ensure that the objects composing the graph data structure have a minimal set of dependancies and certainly do not reference any interface-specific components (such as wxPython or OpenGL). As a result, the graph structure can be tested independent of the main GLuskap program, and a regression test suite has been established to ensure consistency and reliability of this critical component.

In the GLuskap 2.4 release, the OpenGL commands creating the virtual representations of the graph objects were associated with the graph object classes themselves. During the development process of the GLuskap system (since version 2.4), all drawing commands were factored out of the graph data structure objects. This opens up the future possibility of easily enhancing the 3D display to use alternate and varied shapes to represent graph components.

Flexible input management system In order to manage multiple sources of input data for navigation and manipulation in a straightforward manner, a producer-consumer model was used for constructing the input management system. Before the Flock of Birds system is initialized, for example, function callbacks on the InputManager object are registered with the BirdCtl objects. These callbacks are then activated whenever the Flock devices have updated positional data.

The separation of the Flock system from the core operations of the graph and user interface through the InputManager infrastructure means that the incorporation of additional support drivers for diverse input devices in future should be straightforward.

Chapter 6

Conclusions

Upon review of the previous chapter, it is apparent that there are still significant opportunities for improvement of the GLuskap system and the underlying software. The significance of the extant work should not be underestimated, though: there exists in the literature no other three-dimensional graph drawing product designed or adapted to work with a head-tracked large-screen display with three-dimensional navigation and input using wand tracking.

The hardware-software system described in the preceding chapters is a demonstration of the potential use of large-scale 3D graphics and interactive systems technology in graph drawing research. The GLuskap system functions as a cognitive tool, allowing the user to externalize relatively abstract concepts in three dimensions. This allows not only further machine-mediated exploration of the problem space, but facilitates communication of these ideas to other individuals, either in the same physical space or by digital transmission.

Challenges of the Implementation The GLuskap software package draws on several distinct libraries with discrete interfaces; combining the wxPython and Twisted packages, which both provide their own asynchronous event handling mechanisms and event loops,

had the potential to be problematic. This particular overlap in responsibility was resolved by integrating a new reactor architecture from the unreleased development version of Twisted. In order to interface with the Flock of Birds hardware tracking system, no suitable library was available, and I had to construct my own serial interface modules that integrated with the GLuskap asynchronous core and provided translation and an object-oriented data delivery interface for the position and orientation data.

Throughout the course of the implementation, I made several attempts to design and implement an effective system for animation of graph drawing algorithms and other demonstrations. Preserving the readability of an algorithm implementation would seem to require a synchronous execution model, as control should flow through the algorithm linearly. Although it is discussed further below, I bring special attention to the animation problem here as the considerable effort spent attempting to innovate a solution to this problem unfortunately is not reflected in the system as it is currently implemented.

6.1 Future Work

The development of the GLuskap system to this point has opened several potential directions for additional research and development.

6.1.1 Window system interface

Considerable work has been invested into supporting the GLuskap large-screen interactive interface and the attendant specialized hardware. While GLuskap remains usable as a standalone product for viewing and modifying 3D graph drawings with a standard window system interface, comparatively little attention has been given to developing an optimal user experience in the window system.

Zeleznik and Forsberk [ZF99] and Smith *et al.* [SSS01] describe innovative models and

techniques for interaction in three dimensions using the standard two-dimensional input methods available on most desktop computers. Finding a natural and effective method of performing 3D navigation and manipulation tasks with a two-dimensional input device is a productive avenue of development for GLuskap, as specialized 3D input hardware is rare compared to the ubiquitous mouse interface.

6.1.2 User studies on interface

In keeping with the focus on developing GLuskap as an effective tool[1] for graph drawing research, a user study is indicated so as to provide definitive evaluation of the GLuskap interfaces (large-screen and window system) in comparison to each other as well as other interactive graph drawing systems.

The GLuskap interactive large-screen interface is likely the component of the system which would benefit most from a user study. A human research program in this area would serve to quantitatively assess the effectiveness of the interface elements as currently implemented for the performance of common graph manipulation and visualization tasks. It would also provide a qualitative framework to gather ideas for enhancement and revision of both the design assumptions and the implementation to cater more particularly to the specific needs of graph drawing researchers.

The internal component architecture of the GLuskap software allows for alternative input and display concepts to be implemented and evaluated without substantial modification of other subsystems. Alternate concepts for selection and navigation, display of graph elements, or general look and interactive feel of the large-screen and window system interfaces can be integrated into the software.

[1] In the Bederson [Bed04] sense: see Section 2.2.3.

6.1.3 Plug-In Programming Interface

Ideally, the plug-in architecture will allow users to implement graph drawing and layout algorithms in a straightforward and intuitive style. Readability and writability of plug-in scripts are priorities for ongoing development in this area, as they contribute to the ability to check the script implementation for correctness against more abstract original sources (*e.g.* pseudocode or rough outline in a paper). To meet these requirements, the programming interface should be made as simple as possible. Care must be taken, though, in abstracting the complexity of the interface, such that the mechanisms of the interface itself do not suffer in maintainability as a result.

6.1.4 Presentation Enhancements

The interpretation and analysis of graph drawings, and 3D layouts in particular, can be made easier for the user through a variety of techniques that extend beyond the simple display and interaction model. For example, it is often the case that the step-by-step procedure of a layout algorithm is animated for presentation as a video. The process of creating even a short animation using standard 3D graphics tools can be a painstaking process, taking hours or days of development time for a single video; this time cost cannot be amortized over subsequent videos which must be made fresh.

Animations There are compelling reasons to augment the GLuskap software for the creation and presentation of animations. It would seem a natural enhancement to the plug-in scripting interface: the step-by-step procedural nature of a script realizing a graph layout lends itself to being animated simply by inserting appropriate time delays between steps of the process. While conventional 3D graphics software packages may include facilities to (*e.g.*) link objects together to facilitate animating the movement of vertices along with

their incident edges, the GLuskap software inherently understands the semantics of graphs and graph drawings. An integrated animation system would also capitalize on the existing large-screen stereographic display capabilities of the GLuskap system.

The major obstacle thus far to implementing such an animation system within the GLuskap plug-in architecture has been the reconciliation of the asynchronous nature of the GLuskap software (see Section 4.2.2) with the desired simplicity of the scripting interface (see above). The essence of animation is inserting a delay between sequential actions so that they can be seen to take place one after the other. In a synchronous execution model, this can be accomplished relatively easily and transparently by sleeping (suspending execution) after updating the display but before proceeding to the next step of the animation. In the asynchronous case, though, the animation script cannot simply sleep, as this would block the processing of asynchronous network and GUI events. Instead, it must relinquish control to the scheduling system and provide a mechanism to resume processing when the scheduler returns control after the specified amount of time has elapsed.

Explicitly breaking down the control flow of plug-in scripts such that they can be scheduled in small pieces has a highly detrimental effect on the straightforward readability of the resulting script. An alternate technique is to pass scheduling responsibilities to the operating system by using threading. As discussed in Section 4.2.2, though, the use of threads creates a nontrivial amount of overhead in the protection of data structures and can become difficult to test effectively.

Non-Graph Objects A further useful enhancement to the GLuskap package for presentation and visualization purposes is the incorporation of objects into the virtual space that are not strictly part of the graph drawing. These *non-graph objects* include anything that is not a vertex or an edge—at present, GLuskap provides a ground plane and a set of coordinate axes. Allowing the user to create additional non-graph objects (even in simple geometric

shapes such as planes and lines of arbitrary position and orientation) would facilitate the perception of organizing principles in the graph drawing and draw attention to specific features. The positioning of groups of vertices to lie in particular planes or on particular lines, as is typical of many 3D graph drawing layout algorithms, can be easily conveyed in this way.

The effect of this capability is enhanced if these objects can be stored and transmitted as part of the graph drawing; this will require an extension to the GLuskap native file format. Providing an interface to manipulate these objects from the plug-in scripting interface is both natural and desirable, especially if animation capabilities are provided.

Bibliography

[ADH+01] D. Ascher, P. F. Dubois, K. Hinson, J. Hugunin, and T. Oliphant. Numerical Python. Technical Report UCRL-MA-128569, Lawrence Livermore National Laboratory, Livermore, CA, 2001.

[AMH95] Motoyuki Akamatsu, I. Scott Mackenzie, and Thierry Hasbroucg. A comparison of tactical, auditory, and visual feedback in a pointing task using a mouse-type device. *Ergonomics*, 38(4):816–27, April 1995.

[Arn03] Laura Arns. RP system. http://www.envision.purdue.edu/RPsystem.html, 2003.

[Asca] Flock of birds. http://www.ascension-tech.com/products/flockofbirds.php. Distributed by Ascension Technology Corporation.

[Ascb] Wanda device. http://www.ascension-tech.com/products/wanda.php. Distributed by Ascension Technology Corporation.

[Aub04] David Auber. Tulip – A huge graph visualization framework. In Jünger and Mutzel [JM04], pages 105–126.

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39(10):104–116, 1996.

[BC71] Mehdi Behzad and Gary Chartrand. *Introduction to the Theory of Graphs*. Allyn and Bacon, Inc., Boston, MA, 1971.

[BCMW04] Prosenjit Bose, Jurek Czyzowicz, Pat Morin, and David R. Wood. The maximum number of edges in a three-dimensional grid-drawing. *Journal of Graph Algorithms and Applications*, 8(1):21–26, 2004.

[Bec00] Kent Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Bed04] Benjamin B. Bederson. Interfaces for staying in the flow. *Ubiquity*, 5(27), September 2004.

[BEL04] Ulrik Brandes, Markus Eiglsperger, and Jürgen Lerner. GraphML primer. http://graphml.graphdrawing.org/primer/graphml-primer.html, June 2004.

[BETT99] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the visualization of graphs*. Prentice Hall, New Jersey, 1999.

[BFLR00] Dan Bennett, Paul A. Farrell, Michael A. Lee, and Arden Ruttan. A low cost commodity based system for group viewing of 3D images. In S. Klasky and S. Thorpe, editors, *Proceedings of VDE2000, Visualization Development Environments 2000*, pages 145–149, 2000.

[BFN85] C. Batini, L. Furlani, and E. Nardelli. What is a good diagram? – A pragmatic approach. In *Proceedings of the 4th International Conference on the Entity Relationship Approach*, pages 312–319, Chicago, 1985.

[BM04a] Breght R. Boschker and Jurriaan D. Mulder. Lateral head tracking in desktop virtual reality. In Sabine Coquillart and Martin Göbel, editors, *Eurographics Symposium on Virtual Environments EGVE'04*, pages 45–54. Eurographics, June 2004.

[BM04b] John M. Boyer and Wendy J. Myrvold. On the cutting edge: Simplified $O(n)$ planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.

[Bou02] Paul Bourke. 3D stereo rendering using openGL (and GLUT). http://astronomy.swin.edu.au/~pbourke/opengl/stereogl/, May 2002.

[Bro95] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA, 20th anniversary edition, 1995.

[BRT84] C. Batini, M. Ralamo, and R. Tamassia. Computer aided layout of entity-relationship diagrams. *Journal of Systems and Software*, 4:163–173, 1984.

[BSP+93] E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose. Tool glasses and magic lenses: The see-through interface. In *Proceedings of SIGGRAPH '93*, pages 73–80. ACM, 1993.

[BSWW99] T. Biedl, T. Shermer, S. Whitesides, and S. Wismath. Bounds for orthogonal 3-D graph drawing. *Journal of Graph Algorithms and Applications*, 3(4):63–79, 1999.

[Ced03] Per Cederqvist. *Version Management with CVS*. Network Theory Ltd., Bristol, UK, 2003.

[CEGW98] M. Closson, H. Everett, S. Gartshore, and S. K. Wismath. ArrangePak, OrthoPak, and VisPak 2.0. Technical Report CS-01-98, University of Lethbridge, 1998.

[CELR97] R. F. Cohen, P. Eades, T. Lin, and F. Ruskey. Three-dimensional graph drawing. *Algorithmica*, 17:199–208, 1997.

[CFH97] Lawrence D. Cutler, Bernd Frölich, and Pat Hanrahan. Two-handed direct manipulation on the responsive workbench. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 107–114, New York, NY, USA, 1997. ACM Press.

[CP95] P. Crescenzi and A. Piperno. Optimal-area upward drawings of AVL trees. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes in Computer Science*, pages 307–317. Springer-Verlag, 1995.

[CS97] Tiziana Calamoneri and Andrea Sterbini. Drawing 2-, 3- and 4-colorable graphs in $O(n^2)$ volume. In *GD '96: Proceedings of the Symposium on Graph Drawing*, pages 53–62, London, UK, 1997. Springer-Verlag.

[CS02] Ben Collins-Sussman. The Subversion project: Building a better CVS. *Linux Journal*, 2002(94):3, 2002.

[CSFP05] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly and Associates, Sebastopol, CA, USA, 2005.

[DE04] Tim Dwyer and Peter Eckersley. WilmaScope – A 3D graph visualization system. In Jünger and Mutzel [JM04], pages 55–76.

[DEL+05] Olivier Devilliers, Hazel Everett, Sylvain Lazard, Maria Pentcheva, and Stephen Wismath. Drawing K_n in three dimensions with one bend per edge. In Patrick Healy and Nikola S. Nikolov, editors, *Proceedings of the 13th International Symposium on Graph Drawing (GD 2005)*, Limerick, Ireland, September 2005. Springer-Verlag.

[dFPP88] H. de Fraysseix, J. Pach, and R. Pollack. Small sets supporting Fary embeddings of planar graphs. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computation*, pages 426–433, 1988.

[dFPP90] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[DH96] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[DHW04] Breanne Dyck, Sebastian Hanlon, and Stephen K. Wismath. GLuskap 2.4 user's manual. Technical Report CS-03-04, University of Lethbridge, 2004.

[DJN+03] Breanne Dyck, Jill Joevenazzo, Elspeth Nickle, Jon Wilsdon, and Stephen Wismath. GLuskap: Visualization and manipulation of graph drawings in 3-dimensions. http://www.cs.uleth.ca/~vpak/gluskap/docs/software.ps, 2003.

[DJN+04] Breanne Dyck, Jill Joevenazzo, Elspeth Nickle, Jon Wilsdon, and Stephen K. Wismath. Drawing K_n in three dimensions with two bends per edge. Technical Report CS-01-04, University of Lethbridge, 2004.

[DW04] Vida Dujmović and David R. Wood. Three-dimensional grid drawings with sub-quadratic volume. In János Pach, editor, *Towards a Theory of Geometric Graphs*, number 342 in Contemporary Mathematics, pages 55–66. American Mathematical Society, 2004.

[EGK+04] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and Dynagraph – static and dynamic graph drawing tools. In Jünger and Mutzel [JM04], pages 127–148.

[EHW97] Peter Eades, Michael E. Houle, and Richard Webber. Finding the best viewpoints for three-dimensional graph drawings. In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 87–98, London, UK, 1997. Springer-Verlag.

[ESW00] Peter Eades, Antonios Symvonis, and Sue Whitesides. Three-dimensional orthogonal graph drawing algorithms. *Discrete Applied Mathematics*, 103(1-3):55–87, 2000.

[Fár48] I. Fáry. On straight line representation of planar graphs. *Acta Univ. Szeged. Sect. Sci. Math.*, 11:229–233, 1948.

[Fet05] Abe Fettig. *Twisted Network Programming Essentials*. O'Reilly and Associates, Sebastopol, CA, USA, October 2005.

[FG00] Julie Fraser and Carl Gutwin. The effects of feedback on targeting performance in visually stressed conditions. In *Graphics Interface*, pages 19–26, May 2000.

[FLW03] Stefan Felsner, Giuseppe Liotta, and Stephen Wismath. Straight-line drawings on restricted integer grids in two and three dimensions. *Journal of Graph Algorithms and Applications*, 7(4):363–398, 2003.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software–Practice and Experience*, 21(11):1129–1164, 1991.

[FSF91] *GNU General Public License*. Free Software Foundation, Boston, MA, June 1991. Version 2.

[FSF99] *GNU Lesser General Public License*. Free Software Foundation, Boston, MA, February 1999. Version 2.1.

[Gib86] James J. Gibson. *The Ecological Approach to Visual Perception*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986.

[GMHW03] Perry Greenfield, Jay Todd Miller, Jin-chung Hsu, and Richard L. White. numarray: A new scientific array package for Python. http://www.stsci.edu/resources/software_hardware/numarray/papers/pycon2003.pdf, March 2003.

[Gui87] Y. Guiard. Asymmetric division of labor in skilled bimanual action: The kinematic chain as a model. *Journal of Motor Behavior*, 19:486–517, 1987.

[Him96] M. Himsolt. GML: Graph modelling language. Manuscript, Universität Passau, Innstraße 33, 94030 Passau, Germany, 1996. Available at http://infosun.fmi.uni-passau.de/Graphlet/GML/gml-tr.html.

[IdFC03] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua 5.0 reference manual. Technical Report MCC-14/03, PUC-Rio, 2003.

[Ipp05] Bob Ippolito. py2app - convert python scripts into standalone mac os x applications. http://undefined.org/python/py2app.html, 2005.

[JM04] Michael Jünger and Petra Mutzel, editors. *Graph Drawing Software*. Springer-Verlag, Berlin, Heidelberg, 2004.

[KB67] A. N. Kolmogorov and Ya. M. Barzdin. On the realization of nets in 3-dimensional space. *Problems in Cybernetics*, 8:261–268, March 1967.

[KFBB97] G. Kurtenbach, G. Fitzmaurice, T. Baudel, and B. Buxton. The design and evaluation of a GUI paradigm based on two-hands, tablets, and transparency. In *Proceedings of CHI '97*, pages 35–42. ACM, 1997.

[Kil97] Mark Kilgard. Achieving quality PostScript output for OpenGL. http://www.opengl.org/resources/code/rendering/mjktips/Feedback.html, April 1997.

[KTS93] Won S. Kim, Frank Tendick, and Lawrence Stark. Visual enhancements in pick-and-place tasks. In Stephen R. Ellis, Mark K. Kaiser, and Arthur J. Grunwald, editors, *Pictorial communication in virtual and real environments (2nd ed.)*, pages 265–282. Taylor & Francis, Inc., Bristol, PA, USA, 1993.

[KvL85] M. R. Kramer and J. van Leeuwen. The complexity of wire-routing and finding minimum area layouts for arbitrary VLSI circuits. In F. P. Preparata, editor, *Advances in Computing Research*, volume 2, pages 128–146. JAI Press, Greenwich, CT, 1985.

[KW01] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[Lei80] C. E. Leiserson. Area-efficient graph layouts (for VLSI). In *Proceedings of the IEEE Symposium on the Foundations of Computer Science*, pages 270–281, 1980.

[LN04] Claus Lewerentz and Andreas Noack. CrocoCosmos – 3D visualization of large object-oriented programs. In Jünger and Mutzel [JM04], pages 279–298.

[Lut96] Mark Lutz. *Programming Python*. O'Reilly and Associates, Sebastopol, CA, USA, 1996.

[LV97] S. J. Luck and E. K. Vogel. The capacity of visual working memory for features and conjunctions. *Nature*, 390(6657):279–281, November 1997.

[Mar82] David Marr. *Vision*. W. H. Freeman and Company, New York, 1982.

[MCR90] Jock D. Mackinlay, Stuart K. Card, and George G. Robertson. Rapid controlled movement through a virtual 3D workspace. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 171–176, New York, NY, USA, 1990. ACM Press.

[MN99] Kurt Melhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 1999.

[Mun97] Tamara Munzner. H3: laying out large directed graphs in 3D hyperbolic space. In *INFOVIS '97: Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97)*, page 2, Washington, DC, USA, 1997. IEEE Computer Society.

[MW04] Pat Morin and David R. Wood. Three-dimensional 1-bend graph drawings. In *Proceedings of the 16th Canadian Conference on Computational Geometry (CCCG'04)*, pages 40–43, 2004.

[Noa03] Andreas Noack. An energy model for visual graph clustering. In Giuseppe Liotta, editor, *Graph Drawing (Proc. GD '03)*, pages 425–436, Berlin, 2003. Springer-Verlag.

[NT90] J. Nummenmaa and J. Tuomi. Constructing layouts for ER-diagrams from visibility representations. In *Proceedings of the 9th International Conference on the Entity-Relationship Approach*, pages 303–317, 1990.

[NTU05] Kumiko Nomura, Satoshi Tayu, and Shuichi Ueno. On the orthogonal drawing of outerplanar graphs. *IEICE Trans Fundamentals*, E88-A(6):1583–1588, 2005.

[PCS+00] Emilee Patrick, Dennis Cosgrove, Aleksandra Slavkovic, Jennifer A. Rode, Thom Verratti, and Greg Chiselko. Using a large projection screen as an alternative to head-mounted displays for virtual environments. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 478–485, New York, NY, USA, 2000. ACM Press.

[PM92] R. Patterson and W. L. Martin. Human stereopsis. *Human Factors*, 34(6):669–692, 1992.

[PR94] S. E. Palmer and I. Rock. Rethinking perceptual organization: The role of uniform connectedness. *Psychonomic Bulletin and Review*, 1(1):29–55, 1994.

[PT99] Achilleas Papakostas and Ioannis G. Tollis. Algorithms for incremental orthogonal graph drawing in three dimensions. *Journal of Graph Algorithms and Applications*, 3(4):81–115, 1999.

[PTT97] János Pach, Torsten Thiele, and Géza Tóth. Three-dimensional grid drawings of graphs. In *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 47–51, London, UK, 1997. Springer-Verlag.

[PV97] Maurizio Patrignani and Francesco Vargiu. 3DCube: A tool for three dimensional graph drawing. In Giuseppe Di Battista, editor, *Proc. 5th Int. Symp. Graph Drawing, GD*, number 1353, pages 284–290. Springer-Verlag, 1997.

[PV04] Eric Paquet and Herna L. Viktor. Visualisation, exploration and characterization of virtual collections. In *XXth Congress of the International Society for Photogrammetry and Remote Sensing, Commission V*, pages 597–602, July 2004.

[Ram88] V. S. Ramachandran. Perception of shape from shading. *Nature*, 331:163–166, 1988.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, 1999.

[RMC91] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone trees: animated 3D visualizations of hierarchical information. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 189–194, New York, NY, USA, 1991. ACM Press.

[Roc75] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.

[Ros83] A. L. Rosenberg. Three-dimensional VLSI: A case study. *Journal of the ACM*, 30(2):397–416, 1983.

[RSFWH98] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[SACR03] Jay Summet, Gregory D. Abowd, Gregory M. Corso, and James M. Rehg. Virtual rear projection: A comparison study of projection techniques for large interactive displays. Technical Report 03-36, Georgia Institute of Technology, 2003.

[Sch90] W. Schnyder. Embedding planar graphs on the grid. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 138–148, 1990.

[Sch95] Douglas C. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74, 1995.

[SFB94] M. C. Stone, K. Fishkin, and E. A. Bier. The movable filter as a user interface tool. In *Proceedings of CHI '94*, pages 306–312. ACM, 1994.

[SG97] Zsolt Szalavari and Michael Gervautz. The personal interaction panel - a two-handed interface for augmented reality. *Computer Graphics Forum*, 16(3):335–346, 1997.

[SKC96] C.-S. Shin, S. K. Kim, and K.-Y. Chwa. Area-efficient algorithms for upward straight-line tree drawings. In *Proceedings of the 2nd International Computing and Combinatorics: COCOON'96*, volume 1090 of *Lecture Notes in Computer Science*, pages 106–116. Springer-Verlag, 1996.

[SM03] William R. Sanders and David F. McAllister. Producing anaglyphs from synthetic images. In Andrew J. Woods, Mark T. Bolas, John O. Merritt, and Stephen A. Benton, editors, *Stereoscopic Displays and Virtual Reality Systems X*, volume 5006, pages 348–358. SPIE, 2003.

[SSS01] Graham Smith, Tim Salzman, and Wolfgang Stuerzlinger. 3D scene manipulation with 2D devices and constraints. In *Proceedings of GRIN'01: Graphics Interface 2001*, pages 135–142, Toronto, Ont., Canada, Canada, 2001. Canadian Information Processing Society.

[Sug02] Kozo Sugiyama. *Graph Drawing and Applications for Software and Knowledge Engineers*. World Scientific, New York, 2002.

[SWN+03] Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis, and Mary Beth Sheridan. *OpenGL Programming Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 4th edition, 2003.

[Tam85] R. Tamassia. New layout techniques for entity-relationship diagrams. *Proceedings of the 4th International Conference on the Entity-Relationship Approach*, pages 304–311, 1985.

[TDB92] W. M. Thomas, A. Delis, and V. R. Basili. An evaluation of ada source code reuse. In *Proceedings of the 11th Ada-Europe international conference on Ada: moving towards 2000*, pages 80–91, London, UK, 1992. Springer-Verlag.

[Tic85] Walter F. Tichy. RCS—A system for version control. *Software—Practice and Experience*, 15(7):637–654, 1985.

[Tre96] L. Trevisan. A note on minimum-area upward drawing of complete and Fibonacci trees. *Information Processing Letters*, 57(5):231–236, 1996.

[Twi] Asynchronous programming with Twisted. http://twistedmatrix.com/projects/core/documentation/howto/async.html.

[Vin97] Steve Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[vR05] Guido van Rossum. Python reference manual. http://docs.python.org/ref/ref.html, September 2005.

[War04] Colin Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann, San Francisco, 2nd edition, 2004.

[WB94] Colin Ware and Ravin Balakrishnan. Reaching for objects in VR displays: lag and frame rate. *ACM Transactions on Computer-Human Interaction*, 1(4):331–356, 1994.

[WF96] C. Ware and G. Franck. Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Transactions on Graphics*, 15(2):121–140, 1996.

[WGP98] Colin Ware, Cyril Gobrecht, and Mark Andrew Paton. Dynamic adjustment of stereo display parameters. *IEEE Transactions on Systems, Man, and Cybernetics: Part A*, 28(1):56–65, 1998.

[WO53] H. Wallach and D. H. O'Connell. The kinetic depth effect. *Journal of Experimental Psychology*, 45:205–217, 1953.

[ZF99] Robert Zeleznik and Andrew Forsberg. Unicam—2D gestural camera controls for 3D environments. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 169–173, New York, NY, USA, 1999. ACM Press.

[ZL02] Moshe Zadka and Glyph Lefkowitz. The Twisted network framework. In *Proceedings of the Tenth International Python Conference*, 2002.