3D Visualization of Software Architectures

by
Stephen H. Watson, B.A.Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Engineering

Ottawa-Carleton Institute of Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University,
Ottawa, Ontario
December, 1996

© copyright
1996, Stephen H. Watson

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your title Votre référence

Our title Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

—

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

—

ISBN 0-612-19524-4

Dissertation Abstracts International and Masters Abstracts International are arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation or thesis. Enter the corresponding four-digit code in the spaces provided.

0544 UMI
SERIALS CODE

THE HUMANITIES AND SOCIAL SCIENCES

Psychology 0629

Psychology	0525
Reading	0533
Religions	0527
Science	0714
Summary	0530
Social Sciences	0534
Sociology of	0346
Special	0523
Teacher Training	0530
Technology	0710
Tests and Measurements	0288
Vocational	0747

LANGUAGE, LITERATURE AND

LANGUAGE, LITERATURE AND LINGUISTICS	
Language	
General	0671
Arabic	0281
English	0291
Modern	0291
Literature	
General	0401
Classical	0294
Comparative	0295
Mixed	0297
Modern	0297
African	0316
American	0291
Asian	0305
Canadian (English)	0353
Canadian (French)	0353
Czech	0402
German	0311
Latin American	0312
Middle Eastern	0315
Romanian	0313
Slavic and East European	0314

—

Ancient	0679
Medieval	0681
Modern	0682
Black	0688
African	7031
Asia, Australia and Oceania	0692
Canadian	0694
European	0695
Latin American	0696
Middle Eastern	0697

History of Science 0675

Library Studies	03357
History of Science	03355
Philosophy	03798
Philosophy	
General	06151
International Law and	
Public Administration	06141
Public Administration	06117
Education	06114
Social Work	04552
Sociology	
General	06236
Community and Psychology	
Demography	07138
Classical and Social Studies	06311
Individual and Family	
Individual and Family	06238
Individual and Family	
Individual and Family	06239
Individual and Family	06230
Social Structure and	
Development	07000
Theory and Methods	03444
Theory and Methods	03444
Urban and Regional Planning	07999
Women's Studies	04553

ENGINEERING

Goodyear _____ 007

Geology	0350
Geography	0370
Hydrology	0390
Mathematics	0410
Metallurgy	0345
Polymers	0420
Physics	0410
Physiology	0930
Psychology	0940
Physical Geography	0340
Physical Oceanography	0415
HEALTH AND ENVIRONMENTAL SCIENCES	
Environmental Sciences	0760
Health Sciences	
General	0560
Anatomy	0570
Chemistry	0720
Dentistry	0565
Dietetics	0350
Nursing Management	0760
Occupational Development	0765
Immunology	0920
Medicine and Surgery	0560
Mental Health	0567
Nutrition	0570
Optics	0350
Ophthalmics and Gynaecology	0570
Occupational Health and Safety	0765
Pharmacy	0570
Physiology	0930
Psychology	0940
Pharmacology	0940

•

Ceramics	0537
Aerospace	0538
Agribusiness	0539
Astronomy	0540
Automotive	0541
Chemical	0542
Civil	0543
Electronics and Electrical	0544
Food and Thermodynamics	0545
Industrial	0546
Marine	0547
Mathematical Sciences	0548
Medical	0549
Metallurgy	0550
Mining	0551
Nuclear	0552
Physics	0553
Publications	0554
Sanitary and Municipal	0555
System Science	0556
Operations Research	0557
Statistics	0558
Teaching Technology	0559
Teaching Technology	0560
Technology	0561
Transportation	0562
Urban Planning	0563
Urban Planning	0564
Urban Planning	0565
Urban Planning	0566
Urban Planning	0567
Urban Planning	0568
Urban Planning	0569
Urban Planning	0570
Urban Planning	0571
Urban Planning	0572
Urban Planning	0573
Urban Planning	0574
Urban Planning	0575
Urban Planning	0576
Urban Planning	0577
Urban Planning	0578
Urban Planning	0579
Urban Planning	0580
Urban Planning	0581
Urban Planning	0582
Urban Planning	0583
Urban Planning	0584
Urban Planning	0585
Urban Planning	0586
Urban Planning	0587
Urban Planning	0588
Urban Planning	0589
Urban Planning	0590
Urban Planning	0591
Urban Planning	0592
Urban Planning	0593
Urban Planning	0594
Urban Planning	0595
Urban Planning	0596
Urban Planning	0597
Urban Planning	0598
Urban Planning	0599
Urban Planning	0600
Urban Planning	0601
Urban Planning	0602
Urban Planning	0603
Urban Planning	0604
Urban Planning	0605
Urban Planning	0606
Urban Planning	0607
Urban Planning	0608
Urban Planning	0609
Urban Planning	0610
Urban Planning	0611
Urban Planning	0612
Urban Planning	0613
Urban Planning	0614
Urban Planning	0615
Urban Planning	0616
Urban Planning	0617
Urban Planning	0618
Urban Planning	0619
Urban Planning	0620
Urban Planning	0621
Urban Planning	0622
Urban Planning	0623
Urban Planning	0624
Urban Planning	0625
Urban Planning	0626
Urban Planning	0627
Urban Planning	0628
Urban Planning	0629
Urban Planning	0630
Urban Planning	0631
Urban Planning	0632
Urban Planning	0633
Urban Planning	0634
Urban Planning	0635
Urban Planning	0636
Urban Planning	0637
Urban Planning	0638
Urban Planning	0639
Urban Planning	0640
Urban Planning	0641
Urban Planning	0642
Urban Planning	0643
Urban Planning	0644
Urban Planning	0645
Urban Planning	0646
Urban Planning	0647
Urban Planning	0648
Urban Planning	0649
Urban Planning	0650
Urban Planning	0651
Urban Planning	0652
Urban Planning	0653
Urban Planning	0654
Urban Planning	0655
Urban Planning	0656
Urban Planning	0657
Urban Planning	0658
Urban Planning	0659
Urban Planning	0660
Urban Planning	0661
Urban Planning	0662
Urban Planning	0663
Urban Planning	0664
Urban Planning	0665
Urban Planning	0666
Urban Planning	0667
Urban Planning	0668
Urban Planning	0669
Urban Planning	0670
Urban Planning	0671
Urban Planning	0672
Urban Planning	0673
Urban Planning	0674
Urban Planning	0675
Urban Planning	0676
Urban Planning	0677
Urban Planning	0678
Urban Planning	0679
Urban Planning	0680
Urban Planning	0681
Urban Planning	0682
Urban Planning	0683
Urban Planning	0684
Urban Planning	0685
Urban Planning	0686
Urban Planning	0687
Urban Planning	0688
Urban Planning	0689
Urban Planning	0690
Urban Planning	0691
Urban Planning	0692
Urban Planning	0693
Urban Planning	0694
Urban Planning	0695
Urban Planning	0696
Urban Planning	0697
Urban Planning	0698
Urban Planning	0699
Urban Planning	0700
Urban Planning	0701
Urban Planning	0702
Urban Planning	0703
Urban Planning	0704
Urban Planning	0705
Urban Planning	0706
Urban Planning	0707
Urban Planning	0708
Urban Planning	0709
Urban Planning	0710
Urban Planning	0711
Urban Planning	0712
Urban Planning	0713
Urban Planning	0714
Urban Planning	0715
Urban Planning	0716
Urban Planning	0717
Urban Planning	0718
Urban Planning	0719
Urban Planning	0720
Urban Planning	0721
Urban Planning	0722
Urban Planning	0723
Urban Planning	0724
Urban Planning	0725
Urban Planning	0726
Urban Planning	0727
Urban Planning	0728
Urban Planning	0729
Urban Planning	0730
Urban Planning	0731
Urban Planning	0732
Urban Planning	0733
Urban Planning	0734
Urban Planning	0735
Urban Planning	0736
Urban Planning	0737
Urban Planning	0738
Urban Planning	0739
Urban Planning	0740
Urban Planning	0741
Urban Planning	0742
Urban Planning	0743
Urban Planning	0744
Urban Planning	0745
Urban Planning	0746
Urban Planning	0747
Urban Planning	0748
Urban Planning	0749
Urban Planning	0750
Urban Planning	0751
Urban Planning	0752
Urban Planning	0753
Urban Planning	0754
Urban Planning	0755
Urban Planning	0756
Urban Planning	0757
Urban Planning	0758
Urban Planning	0759
Urban Planning	0760
Urban Planning	0761
Urban Planning	0762
Urban Planning	0763
Urban Planning	0764
Urban Planning	0765
Urban Planning	0766
Urban Planning	0767
Urban Planning	0768
Urban Planning	0769
Urban Planning	0770
Urban Planning	0771
Urban Planning	0772
Urban Planning	0773
Urban Planning	0774
Urban Planning	0775
Urban Planning	0776
Urban Planning	0777
Urban Planning	0778
Urban Planning	0779
Urban Planning	0780
Urban Planning	0781
Urban Planning	0782
Urban Planning	0783
Urban Planning	0784
Urban Planning	0785
Urban Planning	0786
Urban Planning	0787
Urban Planning	0788
Urban Planning	0789
Urban Planning	0790
Urban Planning	0791
Urban Planning	0792
Urban Planning	0793
Urban Planning	0794
Urban Planning	0795
Urban Planning	0796
Urban Planning	0797
Urban Planning	0798
Urban Planning	0799
Urban Planning	0800

Physical therapy 0382
P.L.E. Month 0473

Physical Therapy	0382
Public Health	0673
Radiology	0674
Rehabilitation	0675

Psychology _____ 0349
Education _____ 0672

Psychology _____ 0349
Psychometrics _____ 0432
Total _____ 0461

Carleton University
Ottawa, Canada K1S 5J7

50 1942-1992

Thesis contains black and white and/or coloured graphs/tables/photographs which when microfilmed may lose their significance. The hardcopy of the thesis is available upon request from Carleton University Library.

The undersigned recommend to the Faculty of Graduate Studies and
Research acceptance of the thesis

3D Visualization of Software Architectures

submitted by Stephen H. Watson, B.A.Sc.
in partial requirement of the requirements for
the degree of Master of Engineering

Thesis Supervisor

Chair, Department of Systems and Computer Engineering

Carleton University

Abstract

The need to simultaneously show many aspects of a software system strains the limits of two-dimensional visualization. Three-dimensional visualization has been considered promising, but past experience indicates that it is difficult to apply 3D techniques in a useful way. This thesis argues that useful 3D requires the careful choice of a spatial organization paradigm, and proposes the Scaffold Model, an adaptation of the Slot-and-Team Model to a 3D environment. The Scaffold Model uses the vertical dimension to represent both the multiple-part-of relationship, and levels of conceptual abstraction. The use of Simulated Annealing as a heuristic to automatically arrange the elements of the display is discussed. A prototype 3D visualization tool, SMV, is described. Experiment indicates that the Scaffold Model is an effective way to simultaneously view several relationships between parts of a system by use of the third dimension.

Acknowledgments

"No man is an island" wrote John Donne, and few things of any worth were ever produced by individuals acting in isolation. I would like to thank a few of the many people who contributed in some way to this thesis.

My supervisor, Ray Buhr, who put me up to it in the first place, gave me some ideas, and pointed me to a large stack of literature with which to start my search.

Ron Casselman, who listened to my ideas, suggested some of his own, and provided many helpful comments.

Darlene Hebert, who kept extricating me from red tape.

My employer, Bell-Northern Research, on whose equipment the final drafts of this thesis were prepared and printed.

The folks who shared the Real-Time lab with me during the 1993-1994 term, in particular: Alex Hubbard, Erik Dravnieks, Greg Franks, for tutoring me in the intricacies of various Unix programs and utilities. Also: John Bryant, who preceded me on the 3D project, and gave me the benefit of his experience with IRIX and Inventor.

The regular population of the Carleton Mature and Part-time Students' Centre, for coffee and popcorn, and conversations which had *nothing* to do with computers.

Dave, Pam, Phil, Lynn, Vic and Jayne for all their moral support.

My sons Trevyn and Nic, for being themselves; and reminding me that, after all, school is really not the most important thing in life.

Lastly, and mostly, my wife Seanna, for supporting me in every sense of the word during this time. She also proof-read several drafts of this thesis and offered much excellent advice, the value of which is in no way diminished by the fact that I frequently chose to ignore it.

Table of Contents

Chapter 1:Introduction	1
1.1 Motivation	1
1.2 Issues in Visualization	1
1.3 Limitations of Two Dimensions	3
1.4 A 3D Approach	5
1.5 Experiments with the Scaffold Model.....	7
1.6 Contributions.....	7
1.7 Outline.....	7
Chapter 2:Introduction to 3D Graphics	9
2.1 3D Rendering: a Brief Tutorial.....	9
2.2 Changing the Viewpoint.....	12
2.3 Computational Complexity	15
Chapter 3:Uses of 3D for Software Visualization	17
3.1 3D Examples	18
3.2 Requirements for Useful 3D.....	27
Chapter 4:The Scaffold Model	28
4.1 Scope of the Model.....	28
4.2 Elements of the Model	28
4.3 Displaying the Scaffold Model.....	31
4.4 Graphical Representation of the Scaffold Model	37
Chapter 5:The Scaffold Model Viewer	39
5.1 Features of SMV	40
Chapter 6:ANNEAL: A Simulated Annealing Tool	46
6.1 Simulated Annealing	46

6.2	Annealing the Scaffold Model.....	49
6.3	Extensions for Containment	52
Chapter 7:	Case Studies	54
7.1	HotDraw	54
7.2	Scaffold Model of HotDraw	56
7.3	HotDraw Use Cases in Scaffold Model.....	62
7.4	Anneal	64
Chapter 8:	Conclusions and Future Direction	78
8.1	Summary	78
8.2	Conclusions	79
8.3	Suggestions for Further Work	80
References.....		82
Appendix A:	Syntax and Semantics	85
A.1	Example Model: HotDraw	87
A.2	Example Model: Anneal.....	88
Appendix B:	Experimental Viewers	90
B.1	Crane	90
B.2	ScamVu	91

List of Figures

Figure 1:	HotDraw Teams, with equivalence arcs [2].....	3
Figure 2:	ObjecTime Display	4
Figure 3:	Rendering Geometry.....	10
Figure 4:	Rendering a Facet	11
Figure 5:	Camera Translations	13
Figure 6:	Camera Rotations.....	13
Figure 7:	Zooming In and Out.....	15
Figure 8:	Object-Oriented System as a Layered Graph	18
Figure 9:	Class Hierarchy vs. Method Names (after [5])	20
Figure 10:	Ball-and-Stick Model.....	21
Figure 11:	Cone Tree [13]	25
Figure 12:	Fsn directory browser	26
Figure 13:	Structural Elements of the Scaffold Model.....	29
Figure 14:	Layering and Layer Folding	30
Figure 15:	Scaffold Model Representations	32
Figure 16:	Team Re-use by Inheritance	33
Figure 17:	Inheritance and Layering	34
Figure 18:	Team Elaboration Through Inheritance	35
Figure 19:	Display of Elaborated Teams.....	36
Figure 20:	Scaffold Model Viewer Window	39
Figure 21:	Fish-eye Expansion of a Node	42
Figure 22:	Nodes with Subgraphs	52
Figure 23:	Decomposition of Hierarchical Graph for Annealing.....	53
Figure 24:	Full Derivation of Model-View-Controller Team.....	55
Figure 25:	HotDraw Central Teams	57
Figure 26:	HotDraw scaffold with SmallTalk layer showing.....	58
Figure 27:	HotDraw scaffold with SmallTalk layer showing (alternate view) ...	59
Figure 28:	Expansion of DrawingView (close-up, looking down from above)..	60
Figure 29:	HotDraw scaffold with DrawingView expanded.....	61
Figure 30:	EditorMVC: loadDrawing Use case Path.....	63
Figure 31:	Anneal Class Hierarchy with layers.....	64
Figure 32:	Anneal: top-level view, with moveNode use case path	67
Figure 33:	Anneal with theGraph expanded.....	68
Figure 34:	Obscured view in Scaffold display	69
Figure 35:	NodeEdge team.....	70

Figure 36:	Geometry layer view of NodeEdge team.....	71
Figure 37:	NodeEdge team: origins in Primitive layer.....	72
Figure 38:	Anneal with displayGraph use case map	74
Figure 39:	Use Case Map Animation, showing slot-filling.....	75
Figure 40:	displayGraph path with theGraph expanded.....	76
Figure 41:	displayGraph path with theGraph expanded (alternate view)	77
Figure 42:	Crane Controls	91

Chapter 1: Introduction

1.1 Motivation

For as long as there has been software, its designers have been drawing pictures to explain to themselves, and others, how it works. The motivation for this is deep-seated, as expressed by biologist Stephen Jay Gould:

“Primates are visual animals. No other group of mammals relies so strongly on sight. Our attraction to images as a source of understanding is both primal and pervasive. Writing, with its linear sequencing of ideas, is a historical afterthought in the history of human cognition.” [1]

It is universally agreed that program code, being essentially a specialized form of writing, is inadequate as a source of understanding of the system being constructed. Various *visualization* schemes have been used over the years, each intended to illustrate some aspect of a software system. While the earlier schemes were simply diagrams executed on paper (e.g. flowcharts), visualization has recently become more powerful with the advent of CASE tools running on engineering workstations (e.g. ObjecTime, StateMate). Large software systems contain many components, each of which may have several aspects of interest, and may be associated with other components by various different relationships. Attempting to display all the items of interest at any one time strains the limits of what can usefully be displayed on a workstation screen, or comprehended by a human observer.

1.2 Issues in Visualization

Attempting to understand a system by reading its code imposes a cognitive burden on the human mind: the observer must translate the linear, written information into a mental model of the components, and their interrelationships. At some point, keeping track of these mental models, and elaborating them further, becomes difficult, as there is only so much detail that a human can keep in mind at one time. Visualization attempts to address this by

allowing the user to perceive significant components and relationships directly -- literally, "at a glance" -- thus shifting some of the cognitive load to the perceptual system [13].

A practical visualization system should attempt to meet the following goals:

1. To display as much of the system as is practical on the screen;
2. To show relationships between different components;
3. To show all necessary aspects of each component. Most systems (and components thereof) have multiple aspects, each being important in a particular context, and all of them together being necessary to understanding the system.

Designers, and others seeking to understand a system, do not only require information about the static structure of the system: they also need to know "how it runs", about sequencing of activity in response to certain stimuli. Accordingly, an additional goal must be:

4. To show large-scale behaviour patterns across the system.

Problems which need to be solved in pursuit of these goals are:

1. *Display Arrangement*: arranging the various components, and relationships between components, for easy understanding. It must also make efficient use of limited screen space.
2. *Context vs. Detail*: allowing the user to focus on the details of a particular part without losing track of how that part fits into the whole.
3. *Navigation*: when the model is too large for the display device, allowing the user to move efficiently through the model without losing track of where he/she is. One aspect of this is that the display should change gradually, rather than abruptly, when the viewpoint is moved. A sudden jump may cause the user to lose track of where he/she is in the model. A slower "scroll" action, over the period of a second or two, allows the user to keep track of where the viewpoint is moving to, as the old context is gradually replaced by the new one.

1.3 Limitations of Two Dimensions

An example of the limitations on 2D representation is given in Figure 1, which represents the central teams of HotDraw (a drawing editor written in Smalltalk), analyzed according to the Role Architectural Model¹. The heavy lines with arrows at both ends represent equivalence arcs, indicating that the components pointed to are, in fact, the same component appearing in more than one role in different teams. The arcs are confusing in themselves, and preclude the possibility of more information (e.g. use case maps) being added to the diagram, as it stands. The diagram is clearly limited in the number of different relationships and aspects it is capable of representing at any one time.

One common approach to the task of displaying multiple aspects of software is to break

Figure 1: HotDraw Teams, with equivalence arcs [2]

¹. Regrettably, some of the scanned-in figures are not very clear. They are intended to illustrate general concepts, rather than for detailed study.

them up into multiple windows. Examples of this approach are GraphTrace [3] and ObjectTime [4].

GraphTrace animates the behaviour of an object-oriented program, and thus enables a user to perceive both its structure and behaviour. For a single object, it can display data values, method invocation, inheritance (both ancestors and descendants), and *part-of* relationships. By animating the display to show execution, the user can see which objects are made use of, which methods are invoked, and at what level of the inheritance hierarchy they are implemented. However, GraphTrace displays these aspects in separate windows. If several objects were being examined simultaneously, it would be up to the user to keep track (presumably by reading the window titles) of which windows represented the same object. In the case of a sufficiently complex display, the probable result is confusion on the part of the user.

Figure 2: ObjectTime Display

The ObjecTime tool also attempts to present several aspects of system components at one time (Figure 2), and supports behavioural animation. First, there is the block diagram which shows the components, and messaging paths between them (possibly with several levels of decomposition, as components may themselves contain internal structure). Next, each component embodies a state machine which partially specifies its behaviour (stimulus to state-change). Finally, there is the detailed code inside each state, and along the state-transition arcs, showing what messages are to be sent in response to stimuli. Again, when several components are being examined at once, the display becomes cluttered, and the information about each one is fragmented among a number of windows.

While multiple windows are useful, they impose a cognitive burden on the user, in that the relationships between windows must be understood intellectually. As pointed out by Koike [5], when each window presents a different model, representing a different aspect of a system (or part thereof), it forces the user to mentally construct a composite model which synthesizes all these aspects into one. The effort involved (and the possibility of confusion) is compounded in the case where several multi-aspect entities are being displayed at once. The problem has been stated succinctly by F.P. Brooks:

...software is very difficult to visualize. Whether one diagrams control flow, variable-scope nesting, variable cross-references, data-flow, hierarchical data structures, or whatever, one feels only one dimension of the intricately interlocked software elephant. If one superimposes all the diagrams generated by the many relevant views, it is difficult to extract any global overview [6].

Thus, we have returned to the problem we started with (albeit at a higher level). 2D visualization will go only so far in shifting the cognitive load to the perceptual system.

1.4 A 3D Approach

This thesis argues that the above concerns, for at least some interesting aspects of software systems, can be addressed by the appropriate use of three-dimensional representation on a workstation specialized for rapid 3D rendering. Specifically, I develop the *Scaffold Model*, which attempts to display the *operational architecture* of an object-oriented system as a

collection of vertical pillars (the objects), joined to one another at various altitudes by horizontal arcs and polygons (various relationships). Further, it represents levels of conceptual abstraction found in an object-oriented system as horizontal slices (known as *layers*) through the model. This shows several aspects of the system simultaneously and in one window, while avoiding the clutter associated with 2D displays. It allows the user to perceive relationships directly, rather than being forced to infer them.

In arranging elements of a display, a choice must be made as to how to organize them. For each ordinate (two in a 2D display, three in 3D), elements can be placed according to some property of the item, or they can be arranged solely for the convenience of viewing. In this thesis, I use the term *semantic placement* to indicate that the location of an item in a model is determined by some property of the item itself (even if it is only the collation order of the item's name), and the term *aesthetic placement* to indicate that location is determined solely by display considerations, and is unrelated to any intrinsic property of the item.

The scaffold model incorporates both semantic and aesthetic arrangement. It was considered desirable to automate the layout process as much as possible, and so the optimization heuristic *simulated annealing* is used. Model components are positioned according to a set of criteria chosen to reflect readability of the resulting display.

The Context-vs.-Detail issue is addressed through the use of fish-eyes (as described in [7]).

There are 3D input devices on the market, however these were not available, and so were not used. Navigation through the model, as displayed on the screen, is accomplished by providing the user with a mouse-driven interface incorporating the following movement paradigms:

- Constrained virtual track-ball, allowing the entire model to be rotated, but not tilted to the side.
- Dolly in/out, zoom in/out and pan.
- Logarithmic seek-to-point-of-interest [8].

1.5 Experiments with the Scaffold Model

The above concepts were tested by creating various displays on a Silicon Graphics workstation.

Simulated annealing proved to be a satisfactory method to perform a first-pass layout of a system, producing layouts topologically similar to those produced by hand. Some adjustment of the final output still proves useful.

Two experimental tools, ScamVu and Crane, incorporating most of the concepts outlined in the previous section, were developed, and used to visualize two object-oriented programs (HotDraw and Anncal). Experience gained in creating and using these viewers leads to a suggested specification for a complete tool, SMV.

1.6 Contributions

The contributions of this thesis are:

1. Representation of software architecture as a *layered hierarchical hypergraph* (see Section 4.4).
2. The use of simulated annealing, as a layout method for hierarchical hypergraphs.
3. Demonstration of the concept of 3D for the display of layering and multiple-part-of relationships.
4. Demonstration of the concept of Constrained Virtual Trackball and point-of-interest logarithmic movement as navigation paradigms.

My own experience in developing this experimental tool, and informal feedback from others, persuades me that the concepts outlined above are effective ways of using 3D for the visualization of software architectures.

1.7 Outline

This thesis is organized as follows:

Chapter 2 contains a brief explanation of the basics of 3D rendering.

Chapter 3 surveys previous work in the field of 3D data visualization.

Chapter 4 describes the Scaffold Model.

Chapter 5 describes the proposed Scaffold Model Viewer tool (SMV).

Chapter 6 gives background on Simulated Annealing, and describes the application of the method to the task of arranging the Scaffold Model for understandability.

Chapter 7 presents case studies of two object-oriented systems: Anneal and HotDraw.

Chapter 8 presents conclusions, and suggests directions for future work.

The Appendices describe the elements of the SMV model description language, list the SMV sources for the case studies, and describe the experimental tools developed to test the Scaffold Model Viewer concepts.

Chapter 2: Introduction to 3D Graphics

2.1 3D Rendering: a Brief Tutorial

2.1.1 View Geometry

Rendering is the process of projecting the mathematical description of a three-dimensional scene onto the pixels of a two-dimensional computer screen. The basic elements of this projection are the *camera*, the *viewing volume*, and the *scene elements* (see Figure 3). The camera and the scene elements (i.e. the virtual objects being viewed) are located in a coordinate system called the world space. Since the camera may move in world space, a second coordinate system called *view coordinates* may be defined, anchored to the camera. The viewing volume is that part of world space which the camera “sees”, and which is rendered on the screen. The boundaries of the viewing volume are called the *clipping planes*, as objects (or parts thereof) which lie outside them are clipped off. Particularly important are the near and far clipping planes, which are not encountered in normal vision: objects closer than the near plane, or farther than the far plane, will not be rendered, although in real life they would be visible. The viewing plane corresponds to the computer screen itself, through which the user looks into a virtual world.

A commonly used projection, which approximates normal vision, is the *perspective projection*. In this, the viewing volume is a truncated pyramid (called a *frustum*), with the camera situated at the apex and looking along the central axis of the pyramid

Figure 3: Rendering Geometry

The scene itself consists of mathematical descriptions of various objects, together with descriptions of their surface properties, and the locations and intensities of light sources which illuminate the scene.

In its simplest form, rendering consists of dividing each scene object into facets, and mathematically projecting each facet on to the viewing plane, so that it appears as a polygon (see Figure 4). This polygon is then filled with a colour which depends on the surface properties, the angle between the facet normal and line of sight, and the colours and locations of virtual light sources in the scene. Realistic effects are achieved by then interpolating the assigned colours of adjacent facets, so as to produce a smooth shading across edges (this is especially useful when approximating curved surfaces by a large number of flat facets.) Additional

computation is required to determine which facets should not be rendered because they are on the back surfaces (i.e. facing away from the camera) of objects, or are hidden by other objects (known as back-face culling and hidden-surface removal).

Figure 4: Rendering a Facet

As in real life, light sources are necessary in order for objects in a scene to be visible. A scene description must include specification of light sources as to location, intensity, colour, and type. Among possible types of light source are:

1. Point Light: illumination source which is considered to radiate in all directions from a single point in space.
2. Spot Light: like a point light, but "masked off" in most directions, so that illumination diverges from the source as a cone of light.

3. Directional Light: a broad, parallel beam of light which illuminates the whole scene from one direction

2.1.2 Transparency

It is possible to model partially transparent objects in 3D rendering. Two main methods are:

1. Screen door transparency. In this, when a facet is rendered, only some fraction of the pixels lying within that screen polygon are set to the facet colour (e.g. if the object is 50% transparent, then only one-half of the pixels are used). The rest are ignored, and will take on the colour(s) of objects further back in the scene. This is a fast method of displaying transparency, but produces a "grainy" appearance.
2. Blended transparency: In this method, all pixels in the screen polygon are coloured, however their colour will be a blend of the colour of the foreground transparent object and those of the objects behind it. This method is slower, but produces a more realistic and aesthetically pleasing result.

2.2 Changing the Viewpoint

Different parts of the scene can be viewed by moving or altering the viewing volume (with camera attached) in various ways. This corresponds to moving the virtual camera relative to the scene elements (in some cases it is conceptually simpler to think of leaving the camera and volume stationary and moving the scene elements as a unit), or changing its virtual optics.

Primitive translations possible are:

1. Dolly in/out: translation of the camera along the line of sight, towards (in), or away from (out) the scene.
2. Dolly left/right/up/down: translation of the camera in a direction perpendicular to the line of sight, but with the camera orientation remaining constant.

These translations are shown in Figure 5.

Figure 5: Camera Translations

There are three possible primitive rotations which the camera can undergo (see Figure 6):

1. Pan: rotation of the camera about its vertical axis, i.e. turning from side-to-side (also known as "yaw").
2. Tilt: rotation of the camera about a horizontal axis perpendicular to the line of sight (i.e. "pitching" up or down).
3. Roll: rotation of the camera about the line-of-sight.

Figure 6: Camera Rotations

A compound motion can be defined by combining the above primitive motions. For example, the camera may be revolved about a fixed point in the space, while simultaneously being rotated to keep it directed at that point. To the viewer, this appears as if the scene elements are being rotated about an axis through the scene (which is actually the simpler way to conceptualize it).

In the preceding transformations, the viewing volume may be thought of as a rigid object fixed to the front of the camera, and so moving with it. The final transformation modifies the view volume itself:

Zoom in/out: this corresponds to what, with a real camera, would be changing the optics to increase or decrease the magnification, but without actually moving the camera. In terms of the view geometry, it involves increasing or decreasing the angle of the viewing frustum (Figure 7). The effect of zooming in is to magnify a small portion of the scene, while excluding everything else. Zooming out produces a wide-angle effect, squeezing more of the scene onto the screen. The effect of zooming in and out appears similar to dollying in or out (in that scene objects become larger or smaller), however zooming produces distortions

of the image. Zooming in tends to “flatten” the image from back to front, while zooming out tends to exaggerate the apparent depth of the scene, and to distort objects near the edges.

Figure 7: Zooming In and Out

Viewing and moving through a complete scene involves combining the above transformations in various ways.

2.3 Computational Complexity

The amount of computation involved in 3D rendering is considerable, and increases with the amount of detail and degree of realism desired¹. The rendering process involves projecting many facets onto the view plane, calculating positions and surface colours. Moving and rotating objects, or the camera, requires performing 3D transformations represented by

¹. For example, the frontespiece (see Abstract page) was generated by raytracing, a rendering technique (more elaborate than the one described here) which produces highly realistic images by simulating the actual physical processes of light reflection and refraction in a scene. Using the hobbyist program POV-Ray, the image required over an hour to render on a 100MHz Pentium PC – some four orders of magnitude too slow for interactive work.

arithmetic involving 4-by-4 matrices. Rendering time for a simple scene could take several minutes even on a fast processor. Clearly, this is not useful when the desire is to allow a user to browse and explore a scene.

For this reason, computers used to create and manipulate 3D images in real time include specialized processing hardware to perform the matrix and vector calculations rapidly. Even so, the Silicon Graphics Indigo workstation used to produce the images contained in this thesis took several seconds to update the screen on some of the more complex scenes (especially when blended transparency was used). This is an unacceptably long delay; it leads to abrupt changes in viewpoint, which impair the user's ability to keep track of location.

Chapter 3: Uses of 3D for Software Visualization

Little work has been done in the 3D visualization of software. The visualization of software is a specific case of the general visualization of complex structured data. There are several motivations for using 3D visualization techniques:

- To fit more information about the system onto a screen (since some components can be behind others).
- To show more information about the *relationships* between components.

There are several ways in which 3D techniques can elucidate relationships:

1. The extra dimension allows inter-component arcs to cross without touching, thus alleviating the visual clutter and confusion between lines present on a dense 2D diagram.
2. In schemes where components are to be classified according to certain categories, the extra dimension allows for one more level of classification than does a two-dimensional matrix.
3. To enable the relationships to be perceived directly, rather than inferred intellectually. This is sometimes referred to as “shifting cognitive load to the perceptual system” [13]. For example, for a user to have to keep in mind that a symbol displayed in one panel represents the same component as another symbol in a different panel requires a mental effort which distracts her/him from thinking about the system as a whole. Adding an arc indicating identity may be impossible, or may lead to cluttering of the display. However, if by use of a 3D model the symbols can be combined (say, by having the different aspects of the component appear at different depths, or on different sides of a polyhedral symbol), the user can see the identity directly.

3.1 3D Examples

3.1.1 Structuring Space

Ossher [14] suggests a scheme for the conceptual organization of software systems based on directed graphs which are arranged in layers (Figure 8), the objective being to specify relationships such as “uses” between objects. Although not intended as the basis of a CASE visualization tool, it illustrates some interesting aspects of 3D organization of information. Each vertical slice represents an *object* (ten are shown here). Each horizontal slice represents a *view* of the system, as it seen by the other objects (for simplicity, all objects except *a*, *b*, and *db* are considered utilities, and are merged into a composite view called *u*). Each node represents a view of an object, as it is seen by another object. For example, the node in intersection of the vertical *db* object slice and the horizontal *b* view represents *db* (a database object) as seen by *b*, a user object. The directed arcs indicate that the implementation

Figure 2-8: Layered Graph

Figure 8: Object-Oriented System as a Layered Graph

of the source node uses the destination node (Ossher's *ImpUses* relation). The Layered Graph approach illustrates the use of semantic placement in two ordinates (object and view), with the third (going into the page) being left free for aesthetic placement.

In [15], Ossher discusses a browser for a multi-dimensional matrix which stores information about a class hierarchy. The scheme starts with a basic two-dimensional table, in which one axis is data type, and the other is generic function (corresponding roughly to interface methods). Thus, the user can tell at a glance whether a particular type supports a particular interface. Higher dimensions are brought into play to handle *method subdivisions*. Examples of subdivisions are overloading by parameter type or value, or discrimination by the value of an environment variable. A given method (i.e. at a particular location in the type/function table) may be subdivided according to several different criteria, necessitating the use of three or more ordinates to describe the location of a specific implementation. Ossher's browser tool did not attempt to display the database in three dimensions; rather each entry in the basic two-dimensional matrix is a location from which further matrices can be popped up.

A similar approach is taken by Koike [5], in which an inheritance hierarchy is drawn (in 2D) in the customary "inverted tree" style, but the third dimension is then used to define "method planes". Methods having the same name (whatever class they are defined in) are

assigned to the same plane. A cube, floating in space at the intersection of a method plane and a line projected from a class on the inheritance diagram, indicates that the class defines a method with that name (see Figure 9), and helps to make method overrides easier to spot.

Figure 9: Class Hierarchy vs. Method Names (after [5])

In the same paper, Koike also proposes a 3D system for visualizing message passing among processors in a real-time system. It is common to diagram such systems in 2D as a collection of parallel lines (representing the processors) directed along the time ordinate, with messages represented as arcs passing between them at the appropriate times. Koike's system simply lifts the diagram out of the plane, and allows the processor time-lines to be distributed in space (but still remaining parallel to one another). While this approach eliminates some of the clutter (due to message arcs crossing over processor time-lines to which they do not connect), it does not make it easier to perceive message patterns as a whole. In addition, the example used in the paper to illustrate the value of the approach (a comparison of message passing patterns between a normal system and a damaged one)

seems contrived: while in the given case, the difference between the two systems is obvious, it is not clear that the same would hold true for other systems, or for other malfunctions of the same system.

3.1.2 Nodes and Arcs in 3D

Some attempts to apply 3D to visualization of data structures have simply attempted to extend into three dimensions the common nodes-and-arcs diagrams already being used in two dimensions.

Inspired by the success of 3D visualization in molecular chemistry, Reid [16] has proposed a “ball-and-stick” model to browse the database of large software projects (see Figure 10). Data about individual entities would be encapsulated in nodes located in 3-space. Relationships such as compilation dependency and “uses” would be represented by arcs joining the nodes. He argues that the user, by manipulating and “flying through” the network, will be able to comprehend its structure in the same way a chemist understands a molecular struc-

Figure 10: Ball-and-Stick Model

ture by looking at a model. Reid also suggests clustering some nodes inside super-nodes in order to reduce the amount of clutter the user must deal with at once. Super-nodes can be opened (in a separate display window) to reveal their internal structure as necessary.

Unfortunately, chemists and software designers have different needs: chemists find 3D models useful precisely because they are interested in overall structure, or in determining ways in which two molecules might fit together in the course of a reaction. As a paradigm for software visualization, the premise of [16] has not, in general, been born out in practice. It suffers from several difficulties:

- Software is not a physical object, and so (unlike a molecular model) has no natural physical “shape”. A user can perceive the structure of a ball-and-stick molecule from the outside, but the “shape” of a ball-and-stick software model does not necessarily convey information about the system in the same way as the shape of a molecule conveys information about its chemical properties.
- A user moving into the model to focus on a particular object of interest can become lost and disoriented.
- It may be difficult to find a place where the object of interest can be viewed without intervening foreground objects being in the way.
- Other balls and sticks provide a distracting background to the objects of current interest.

One implementation of the nodes-and-arcs paradigm in 3D is SemNet [17], a tool for displaying and exploring a Prolog knowledge database. In SemNet, nodes represent knowledge elements in the database and arcs represent relations between them (for example, one application has been to store linguistic rules and messages passed between rule-groups for natural language research). There are two schemes available for the placement of nodes:

1. *Semantic placement*: Knowledge elements are assigned numeric values based on some properties of the element. If three numbers can be assigned to an element, each reflecting some different aspect of it, they can be used as the coordinates of the node in 3-space. Placing elements in this way allows a user to see clusters of similar elements, or to find an element with particular characteristics, by using the mapping between position and semantics.

2. *Aesthetic placement*: The designers of SemNet also examined schemes in which nodes which are more strongly connected to each other are placed closer together than nodes which are weakly connected. This proved useful because it minimized the lengths of the connecting arcs (which otherwise tended to obscure the view of the elements), and revealed clusters of strongly connected nodes (thus yielding a sense of the macroscopic structure of the database). For some applications (e.g. the natural language project) such macroscopic structure information may be useful. Several methods for performing the arrangement were tried, including one similar to simulated annealing.

SemNet also attempts to address the context-vs.-detail problem by use of the fish-eye-view paradigm. First, elements are grouped into a hierarchy of clusters. When exploring the knowledge base, only the closest elements are seen as individuals; more distant elements are depicted as hidden inside clusters or meta-clusters. This is potentially a powerful technique, however it would probably be best if the clustering be done intelligently, based on some characteristic of the elements, or on connectedness (in SemNet it is done arbitrarily by successive divisions of the space into octants).

The authors of [17] do not indicate whether a problem was encountered with users becoming disoriented while navigating through the SemNet display.

3.1.3 Head-Coupling and Stereopsis

Investigations by Arthur Booth and Ware [18] indicate that users are able to extract useful information from 3D node-and-arc displays, provided special equipment is used. They experimented both with *stereopsis* and *head-coupling*.

Stereopsis attempts to give perceived depth to the image by simulating natural binocular vision. It uses a rendering process which produces two slightly different images of the scene during alternate video frames. The user wears special glasses, having LCD shutters synchronized to the frame rate, so that each eye sees only one image. The visual system combines the images into a single stereo image which seems to possess depth.

In head-coupling, the user wears a "hat" which is coupled by a system of levers to a sensor which determines the position and orientation of the user's head. This information is used to modify the virtual camera position, thus rotating the scene on the screen. Such rotation gives the user the subjective impression of looking over and around a solid object as he moves his/her head to different viewing positions.

The task set the experimental subjects was that of searching for a connection between two nodes in a 3D graph under various combinations of head-coupling and stereopsis, while time and accuracy were recorded. While the different viewing conditions did not appear to affect the task performance speed very much, they had a marked effect on accuracy. When simply viewing the static picture (i.e. no 3D effect at all; equivalent to 2D picture), the error rate was 21.8%, while adding stereo brought the error rate down to 14.7%. Head-coupling offered impressive performance gains, with an error rate of 2.7% for head-coupling alone, and 1.3% when combined with stereopsis.

The lack of such elaborate hardware (and user discomfort while wearing it) means that head-coupling is not widely used (and was not used in the work of this thesis). However, since the main effect of head-coupling is to rotate the scene on the screen, at least some of its benefits should be realizable by giving users an interface which permits them to move and rotate the scene themselves using hand controls.

3.1.4 Tree Browsers

Experience with node-and-arc models suggests that 3D may be more effective if the third (and possibly the second and first) dimension is used for some very specific purpose (i.e. not just to give one more degree of freedom in the placement of arcs and nodes).

Robertson et al designed the Cone Tree tool (Figure 11), to allow the user to browse hierarchically-organized information (in this example, a UNIX directory tree). The parent-child relationships between the components is clear. Moreover, the object of interest can be rotated to the front, leaving the rest in the background (visible, but not intrusive), partially resolving the context-detail tension. This also allows more information to be displayed on the screen as compared with a flat display in which the branches at each level are splayed

out sideways. Here, the vertical dimension is used to indicate branching level on the tree (semantic placement). The horizontal plane is best described in polar, rather than rectangular, coordinates: items can be arranged around the circumference according to any appropriate semantic (file-name, creation date), while the radius of the cone itself is determined by the display width (i.e. by aesthetics).

Figure 11: Cone Tree [13]

The Fsn browser (Figure 12), from Silicon Graphics, displays the directory hierarchy on a flat plane, but with differences which distinguish it from the tree-style browsers which come with many GUIs: the plane is placed visually at an angle to the user, who “flies” over it using the mouse as a speed and direction control (side-panel controls allow the user to adjust the altitude and look-down angle). Directories appear as boxes on the plane, connected to their subdirectories by lines drawn on the plane. Files are represented as towers sitting atop the directory boxes, with the tower height indicating its size. Tower colour indicates the age of the file. Fsn incorporates a fish-eye effect (addressing the context-detail tension): towers farther away from the user are compressed laterally, thus allowing more of them to fit in the width of the display. But as the user flies towards a particular directory, it expands sideways, and is roughly square by the time the user is in a convenient location for examining it in detail (at the same time, text giving the file names appears). A slight difficulty

arises in that the user can only look "up" the tree (i.e. towards the leaves), thus possibly losing track of her/his location in the tree. This is alleviated by the provision of a pop-up overview panel, which marks the user's current location. A rapid "seek" behaviour is available: clicking on a particular item causes the viewpoint to fly rapidly toward the item, stopping in front of it.

Figure 12: Fsn directory browser

The experience of using Fsn is that of flying over a flat terrain covered with towers of various heights, following connecting lines to get to the tower of interest. That this is an effective paradigm for the visualization of tree-searching may be gauged from the fact that Fsn was used in the popular movie *Jurassic Park* to convey the concept to a lay audience. In this case, the vertical dimension (Y-axis) represents file size, the in/out direction (Z-axis) represents tree depth, and the subdirectories are arranged alphabetically from left to right (X-axis). Thus, all three directions carry semantic significance. Fsn demonstrates effective use of spatial assignment, colour, fish-eye, and navigational aids.

3.2 Requirements for Useful 3D

While Cone Tree and Fsn show good use of 3D to visualize relationships in structured data, software systems (unlike file systems) are not tree-structured. Software is probably more naturally represented by the ball-and-stick paradigm, the limitations of which have been discussed above. With reference to the general visualization problems outlined in Section 1.2, requirements for useful 3D software visualization are:

1. Display Arrangement:

- Structure the space by using semantic placement in at least one ordinate, and otherwise group strongly connected elements close together. This promotes understanding of relationships between components.

2. Context vs. Detail:

- Use "clumping" mechanisms (such as fish-eye views).
- Find a way to eliminate foreground and background clutter.
- Use contrasting colours to help the user distinguish between figure and background, and between different kinds of objects.

3. Navigation:

- Structure the space using semantic placement. This provides the user with visual cues to maintain a sense both of direction and location.
- Provide the user with interface capabilities which enable rapid, but controlled movement of the viewpoint through the model, and to focus in on an element of interest. Implicit in interface considerations is the requirement for *responsiveness*: there should be no annoying delays between user actions and display updates, as these interfere with the feedback the user needs to control the 3D movement effectively.

Chapter 4: The Scaffold Model

4.1 Scope of the Model

The Scaffold Model, as originated by Casselman [19], and extended in this thesis, is a three-dimensional version of the slot model [21]. It describes a software architecture in terms of several structural elements: *slot*, *object*, *team*, and the organizational paradigm *layer*. Causality flows can be depicted by *use-case paths*.

The slot model expresses the architecture of a system in terms of teams and the objects which, at run-time, fill the constituent slots in the team. The pools and installation arcs of the model, together with use-case paths, show the dynamics of such slot-filling. The scaffold model shows teams and their members, and additionally shows relationships between slots in different teams which are filled by the same object at run-time, and gives some sense of the re-use of objects and behaviour patterns. While some clues about the classes from which the objects are built can be gleaned from the model, it does not attempt to explicitly combine the class hierarchy view of the system with the slot/team view.

4.2 Elements of the Model

Teams represent collaborating groups of objects. A team definition is the basic template for creating the teams of the architectural model. A team is defined by specifying the interaction which takes place between its members, known as “slots”, and the interfaces by which those slots support the interaction. Slots may be understood to be implemented as classes by writing method bodies which carry out the interaction, but note, that the explicit mapping of slot to class is outside the scope of the scaffold model. Objects are instantiations of classes. Similarly, teams are instantiations of team definitions, and are thus considered to be concrete entities. They are specified in terms of the concrete objects which fill the slots given in the team’s definition. In order to fill a slot, an object must be of a class which supports the interface and behaviour given for that slot. A team is considered to be instantiated at run-time when:

1. Objects capable of filling the slots given in the team's definition have been created, and
2. Those objects have been connected to the other team partners (the exact mechanism of such connection is outside the scope of the scaffold model: for some suggestions, see [21], chapter 8).

The relationships between the model elements are summarized in Figure 13.

Figure 13: Structural Elements of the Scaffold Model

This shows how slots (at design time) translate into concrete objects at run time. The “classes” item is included here to show how objects capable of filling slots could be created in whatever object-oriented language is being used, but such implementation is considered to lie outside the scaffold model as such.

4.2.1 Layering

Layering is not so much an element of the Scaffold Model as it is a paradigm used to organize the other elements. It is used to indicate *infrastructure*, that is, other parts of the system (in which the model-user is not currently interested) which provide services to the part in which the user is currently interested [20]. The definition of a layer, then, is somewhat user-defined: a layer boundary can be placed wherever the user finds it convenient to divide the system into conceptual blocks. Layering is commonly used in data communications, where the exchange is considered as taking place between peer entities, with the details of the link deliberately hidden in lower layers. With this in mind, I offer the following definition:

layer: a part of the system which provides services to other parts while hiding the details of those services' implementation. The services provided are so pervasive that they are best made implicit in the presentation of the service-users. It especially includes entities which provide low-level interaction mechanisms which are used as part of interactions between higher-level entities.

For example, an operating system may be seen as a layer below the application. A second use of layering may be pragmatic: the designer of a system may have been given an existing framework or library, and told to build the system around it. The framework then becomes a layer below the system he/she is designing. The application itself would be considered the top-most layer.

Layering is useful as a means of information hiding. Consider the interaction shown at left in Figure 14(a) below, in which an activity proceeds from component A to component B by way of components C, D, E and F. Components A and B are assigned to Layer 2, the rest to Layer 1. If the details of the interaction with C, D, E and F are not currently of interest, then the diagram may be redrawn as in Figure 14(b), with Layer 1 "folded" -- its presence visible, but unobtrusive.

Figure 14: Layering and Layer Folding

4.3 Displaying the Scaffold Model

The abstractions to be displayed are:

1. *Teams*: Teams of collaborating objects are displayed as labelled, blue polygonal platforms, with their participating slots at the vertices (see next item). Teams can be replicated (when there are multiple objects in one or more of the slots, each participating in a separate instantiation of the team). This is depicted by displaying multiple platforms, stacked vertically and closely spaced.
2. *Slots*: The slots are depicted as vertical pillars at the corners of the team platforms. A solid pillar indicates a static or "fixed" object, i.e. the same object fills that slot throughout the life of the system. Slots which are filled dynamically over time by a succession of different objects are shown as wireframe pillars (called "placeholders", since they "hold the place" for a real object which is not always present in that slot). Collections or pools of objects are displayed as bundles of pillars, with the number of pillars in a bundle giving a rough idea of the relative size of the collection. The name of the object floats in space above the pillar.
3. *Relationships*: The inter-object relationships defined in the Scaffold Model, and their representations, are:
 - *Maintains*: A given object may maintain a pool of other objects. This is shown as a green arc from the maintaining object, ending in a "barrel" encircling the collection. An exception to this occurs when the collection is considered to be contained by another object. In this case, the enclosing object is implicitly considered to be the maintainer, and no "maintains" arc is displayed.
 - *Comes-from*: A given slot (usually a placeholder) is to be filled with an object (often coming from a collection). This is shown as a red arc with a cone-shaped arrowhead, extending from the source object(s) to the destination slot.
 - *Containment*: Objects may contain other objects and teams. Geometry representing the contained structures is placed inside the containing object. In order to view it, it is necessary to expand the container and turn it transparent. Relationships between enclosed objects and objects in the outer decomposition are not shown unless the enclosure has been thus opened.

4. *Layering*: Layer boundaries are represented by transparent tiled floors cutting horizontally through the object pillars. Floors are labelled in one corner with the name of the layer they represent. Each tile is square, but with one corner cut off diagonally. As well as demarcating the layers, this provides an orientation cue to the user.

Figure 15 shows examples of each of the display elements. Here, **Team1** represents a collaboration between **Object3**, **Object4** and **Object5**. **Object4** is a placeholder which will be filled at run-time by an object coming from the collection **Object2**, which is maintained by **Object1**.

Figure 15: Scaffold Model Representations

4.3.1 Inheritance, Layering and Collaboration

Layering may be used to represent levels of *conceptual abstraction*. Object-oriented systems are frequently built by first defining a number of “primitive” classes embodying very basic concepts and functions. These are then used, through inheritance or composition, to build a another set of classes embodying higher-level concepts. These classes may then be further sub-classed or used to build a third set, and so on. The result is that, if the class hierarchy is drawn, it can be divided horizontally into layers.

Suppose we have a team, called TeamAB, with slots A and B (see Figure 16) and that these slot definitions are implemented by classes A1 and B1, respectively. If classes A1 and B1 are subclassed to produce A2 and B2, then objects of these classes will also be able to participate in TeamAB.

Figure 16: Team Re-use by Inheritance

Note that classes A2 and B2 are able to take advantage of the interaction implemented in the base classes *without needing, or having, access to the implementation of that interaction*. Thus, the base classes are providing a service (the ability for higher-level objects to participate in a particular team) while hiding the details of its implementation at a lower level. By the definition given above, we can say that the base classes, and the team definition which they implement, belong to a low layer, and are used at a higher layer. The situation can be illustrated as shown in Figure 17(a), below (note that the layers, and thus the inheritance relationship, are shown inverted with respect to Figure 16). Here the objects *anA2*¹ and *anB2* are instantiations of classes A2 and B2, and fill the slots of TeamAB (thus

instantiating it as *aTeamAB*). Note that the objects *anA2* and *aB2* extend across the layer boundary, since parts of these objects are defined in Layer 1 (classes A1 and B1), and parts in Layer 2 (classes A2 and B2).

Figure 17(b) shows Layer 1 folded up. This has the effect of “collapsing” the inheritance relationships, and bringing *aTeamAB* up to the top layer.

Figure 17: Inheritance and Layering

A further elaboration is possible: a team defined at a higher layer may make use of one from a lower layer. Consider the case that, an interaction is implemented by some low-layer base classes. Higher-layer derived classes each inherit the behaviour, and elaborate it in some way, possibly even extending the interaction to involve a greater number of objects (i.e. more slots). The high-layer interaction itself defines a team, different from, but making use

¹. The examples follow the convention of naming objects by prefixing the indefinite article ("a" or "an") to the name of the class they are instances of.

of the low-layer team. In Figure 18a, we have slots A and B (defining TeamAB), elaborated to give A' and B', and adding a third slot C to give TeamABC. Figure 18b shows the implementation of this in terms of the class hierarchy.

Figure 18: Team Elaboration Through Inheritance

4.3.2 Layer Assignment and Display

The scaffold model *assigns* slots (and, by implication, team definitions) to layers. However, SMV *displays* concrete objects, rather than slots, and team instantiations, rather than definitions. In what layers should objects and teams be displayed?

The answer is: an object should be displayed in all layers in which a slot is defined which that object is capable of filling. It is displayed as a pillar extending through the full thickness of the layer. It may sometimes occur that, an object may have slots in two non-adjacent layers, but not in the intervening layer(s). In this case, the two sections of the object pillar are connected by a thin pillar through the intervening layer(s).

A general principle to follow for displaying relations between objects is that no important relation should be invisible, regardless of the state of the display. The consequences of this are examined below.

Teams, in general, should be displayed in the layer in which they are defined. There are exceptions to this, however. Consider the case illustrated in Figures 16 and 17: TeamAB is defined at Layer 1, but used at Layer 2. If the rule were followed strictly, that teams should only be displayed in the layer-of-definition, then *aTeamAB* would be invisible in Figure 17b. Since this team presumably represents some important interaction between the objects, this is undesirable. For this reason, a team which would otherwise be invisible must be “brought up” to the next available layer.

Figure 19: Display of Elaborated Teams

However, there is an exception to this rule, also. Figure 19 shows an instantiation of the case presented in Figure 18. A display of Layer 2 only (Figure 19a) would show instantiations of TeamABC, but need not show the instantiation of TeamAB as well, since it is implicitly subsumed in TeamABC (i.e. it is the kind of detail we are trying to hide using

layering). If both Layer 2 and Layer 1 are to be displayed, then both TeamABC and TeamAB should be displayed (Figure 19b), as layer expansion indicates that we now wish to examine the details.

Other relationships between objects (comes-from, maintains) should follow the same rules as for teams.

4.4 Graphical Representation of the Scaffold Model

The Scaffold Model represents a system as a *3D layered hierarchical hypergraph*¹, with the objects forming the nodes, and their various interactions forming the edges. A hypergraph is defined as:

hypergraph: a set of vertices (or nodes) V , and a set $E = \{E_1, E_2, E_3, \dots\}$ of subsets of V . The members E_i of E are called hyperedges, although those containing only two nodes can simply be called edges (after [22]).

In this case, the nodes are slot-filling objects, with edges and hyperedges forming the various relations ("maintains", "comes-from" and teams).

The hypergraph generated by the scaffold model is hierarchical in that some of the nodes themselves contain sub-graphs (see Figure 22, on page 52 for an example). The third dimension is used to allow hyperedges to cross each other without interference: nodes extend vertically (i.e. in the case of Figure 22, out of the plane of the paper), and can make connections to each other at various altitudes.

In the scaffold model, the vertical dimension is a hybrid of semantic and aesthetic placement. First, the vertical ordinate is divided into discrete layers, and each element is assigned to a layer. However, within each layer the vertical placement is determined by the input provided by the user, purely for ease of viewing.

¹ The "hierarchical hypergraph" concept, as used here, is a restricted case of Harel's "higraph" concept, described in [24], with nodes corresponding to the "blobs" of a higraph. Nodes, in the scaffold model, are restricted to being either disjoint or totally enclosed by another node, whereas blobs in a higraph are also permitted to partially overlap one another.

Arrangement of elements in the two horizontal dimensions is done by aesthetic placement, with the intent being to minimize edge crossings, and clarify relationships between objects.

4.4.1 Scaffold Model and Class Hierarchy

Classes are not an explicit part of the scaffold model, and the model does not attempt to show the inheritance hierarchy explicitly. Nevertheless, some information about classes is available. If the class hierarchy, as written, can be divided into conceptual layers, then seeing that an object passes through a particular layer tells one that object is derived from one or more of the classes assigned to that layer. This does not give a precise picture of the hierarchy, but may give clues as to where to look for more information about an object of interest.

Chapter 5: The Scaffold Model Viewer

This section describes a tool for viewing the Scaffold Model, called SMV. Not all features described here have been implemented in a working tool. The experimental programs ScamVu¹ and Crane², which have been written, embody the basic principles (see Appendix B for a description of the operation of each viewer). They are written in C++ using the SGI Inventor library, and run on an SGI Indigo workstation.

Figure 20: Scaffold Model Viewer Window

The Scaffold Model display attempts to replace the equivalence arcs of, for example, Fig-

¹ "ScamVu": another short form of Scaffold Model Viewer, but "cam" also stands for "camera".

² "Crane": metaphor for the camera movement scheme used internally, in which the camera is treated as if mounted on an extensible arm, which is free to pivot about a point in the scene.

ure 1, with vertical connectors, and to represent layering by dividing the 3D space horizontally into actual layers. Entities are to be displayed in the layers to which they have some relevance. Implicit in the model is the assumption that it is interactive, being displayed on a workstation with controls to alter the perspective of the display. Different colours are used to help distinguish the elements from one another. An example of the SMV window is shown in Figure 20.

Input to SMV is from a text file containing a description of the model. The semantics of the description language are given in Appendix A.

5.1 Features of SMV

For consistency of nomenclature with the library used to build the Scaffold Model Viewer, the following convention will apply in this discussion: the X-direction points horizontally from left to right across the scene, the Y-direction points vertically upwards, and the Z-direction points out of the plane of the screen or paper towards the viewer (i.e. right-handed coordinate system).

The interface of the Scaffold Model Viewer (SMV) is mouse-driven, with some features being controlled by use of the menus, or other widgets located in the window trim, while others are controlled through direct mouse interaction with the display area itself. The display area interaction has two primary operating modes, which are toggled by clicking on the “eye” button at the upper right of the window trim: viewing and editing. In viewing mode, the mouse is used to move the viewpoint around or through the model. Editing mode allows the user to modify the display in certain ways.

Some, though not all, of the functionality described here has been implemented in experimental viewers called ScamVu and Crane. Much of the functionality of the ScamVu viewer comes from the Inventor library class SoXtExaminerViewer, with which it is implemented.

5.1.1 Layer Display

(Implemented in ScamVu).

The Layers menu allows the user to choose which layers to display. Since the organization of entities in lower layers usually only makes sense in terms of the higher-layer entities which they serve, layer display obeys a top-down rule, i.e. selecting a particular layer for display causes every layer from the top, down to and including the selected layer, to be displayed.

5.1.2 Fish-eye

(Implemented in *ScamVu* and *Crane*).

Objects which contain internal details are marked by an expansion handle, in the form of an inverted yellow cone. When the handle is clicked, and SMV is in Edit mode, the cylinder turns transparent, and expands horizontally to reveal its internal structures. Other model elements move radially away from the expansion centre. The handle remains visible, and clicking it again collapses the fish-eye.

Furnas [23] has described a generalized fisheye view, in which the *degree-of-interest* (DOI) of an object in the display is given by its *a priori importance* (A_i) and its distance from the current point of interest. The DOI of an object translates roughly into its displayed size.

The approach taken by SMV differs from Furnas' in that the distance term is ignored, and all objects at the same level of decomposition are accorded the same DOI (and thus, are displayed the same size). When an object is expanded, its internal components are "promoted" to the same DOI as the outer level. Accordingly, the expansion is controlled by an "equal density" algorithm, which attempts to make the overall density of nodes approximately equal in the expanded and non-expanded states. Two parameters must be calculated to control the fish-eye: the magnification of the expanded node, and the resulting distortion of the surrounding space (which determines the amount by which other objects are moved away).

The magnification m is calculated as: $m = \sqrt{\frac{nA_i}{Na}}$.

where:

A and N are: the area of the outer decomposition (before the expansion takes place), and the number of objects in the outer decomposition, respectively.

a and n are: the unexpanded area of the node, and the number of contained sub-objects respectively.

This makes the density of internal objects in the expanded node equal to the density of objects in the outer decomposition before the expansion took place. A node which has a normal (i.e. unexpanded) radius r will have a radius, when expanded of $r' = mr$ (see Figure 21, below).

Figure 21: Fish-eye Expansion of a Node

It remains to calculate the amount by which to move other objects away from the expansion centre. The constraint that the average density of nodes in the rest of the graph remain the same implies that the total graph area outside the expanded node must also not change. Consider an external node such as B in Figure 21, lying at a location described by a vector R relative to the expansion centre. We define an annulus surrounding Node A, of initial out-

er radius $R = |\mathbf{R}|$, and inner radius r . When the node expands, the annulus expands with it to a new outer radius $R' = |\mathbf{R}'|$, which must be chosen such that the total area of the annulus remains the same. The pre-expansion area of the annulus is:

$$A - a = \pi (R^2 - r^2) ,$$

and the post-expansion area is:

$$A' - a' = \pi (R'^2 - r'^2) .$$

Setting these equal and solving for R' , we obtain:

$$R' = \sqrt{R^2 + r^2(m^2 - 1)}$$

The displacement of node B is called the *fish-eye vector* $\mathbf{F}$, which is given by:

$$\mathbf{F} = \mathbf{R}' - \mathbf{R} = \mathbf{R} \left(\frac{R'}{R} - 1 \right)$$

In the case where there are several fish-eyes open simultaneously, the displacement of a node will be the sum of the fish-eye vectors produced by each of the expanded nodes.

5.1.3 Constrained Virtual Trackball

Motion in SMV takes place with respect to a point in the scene called the *point-of-interest* (*POI*), which can optionally be marked with a small set of arrows pointing in the directions of the major axes (providing a directional cue). There are two methods of moving through the model: Constrained Virtual Trackball, and Logarithmic Seek-to-point-of-interest.

In the virtual trackball paradigm, the scene being viewed is considered to lie inside a transparent sphere, centered on the POI, which can be rotated in any direction. There are two ways of rotating the sphere: by use of thumbwheels located in the window trim (labelled

“Rotx” and “Roty” in Figure 20), or (when in viewing mode) by “grabbing” the sphere with the mouse (left button depressed) and moving it in the desired direction. In addition, the distance between the viewpoint and the POI can be varied using the “Dolly In/Out” thumbwheel, or the mouse (left and middle buttons depressed). In keeping with the argument made in [8], the in/out motion is logarithmic with respect to the POI: the closer the viewpoint is to the POI, the slower the motion will be. This allows rapid movement while the viewpoint is far from the POI, but fine adjustment when moving in close. The viewpoint can move arbitrarily close to the POI, but never right through it.

SMV introduces a constraint on the virtual trackball: it is desired that the “up” direction of the model always be maintained. In the unconstrained trackball, the model can be tilted in any direction, or even turned upside down. The constrained trackball permits the model to be tilted towards or away from the camera, but not to the side. Another way of visualizing this is in terms of camera movement: the camera can be moved to any location, and can be pointed up or down (pitch) and left or right (yaw) (in order to keep it looking at the POI), but it may not be rolled about the axis along which it is looking. ScamVu uses an unconstrained trackball provided by the Inventor library. The constrained trackball type of movement was implemented in Crane, though under keyboard (rather than mouse) control.

The POI can itself be moved by using the mouse (middle button pressed) to drag it to the desired location in the model.

5.1.4 Seek to Point-of-Interest

(*Implemented in ScamVu*). A second method for changing the POI is to put the viewer into seek mode, by clicking the Seek button (lower right window trim). Now, clicking on an object on the display area will cause that object to become the new POI, and the camera will undergo a compound motion to reposition the view-point to look at that object. This involves both rotating the camera to point in the new direction, and moving the camera in or out so that the distance to the new POI is the same as the distance from which the old POI was being viewed. At each stage of the seek animation, the “camera up” constraint is maintained. The seek animation is implemented in ScamVu, but without the “camera up” constraint.

5.1.5 Query

Sometimes, in the course of exploring a model in SMV, it may occur that the label on an item of interest is no longer in view, or else that more detailed information is desired about the item. For this purpose, textual output is appropriate. SMV has a query function which operates when in edit mode: clicking on an object or team causes a window to appear giving the item's name, and various information about it (the window should be dismissable using the mouse). This includes the type of the item (e.g. for an object: its class, and for a team: the team definition it instantiates), and inheritance information.

Query is partially implemented in ScamVu: no new window is created, and the text information is written to the standard output.

5.1.6 Display Pruning

(Implemented in ScamVu). No matter how carefully a 3D display is arranged, it is inevitable that at times, something will be in the way of an object the user wants to look at. This problem can be alleviated by turning the occluding objects transparent. This might be done automatically, by detecting the presence of objects lying in a cone between the viewing plane and the POI. In ScamVu a simpler and more flexible approach is used: the user can make any object disappear simply by clicking the middle mouse button on it. The object will also become transparent to all other mouse functions (i.e. Query and Seek-to-POI). All pruned objects can be brought back by refreshing the layer (pull down Layers menu and select the layer).

Chapter 6: ANNEAL: A Simulated Annealing Tool

In the scaffold model, the Y dimension is assigned a meaning (to differentiate between layers). However, the X and Z ordinates are left free for aesthetic arrangement of the model elements. It was desired that the layout process be automated as much as possible, and so the simulated annealing heuristic is used to create an initial layout of a model. In [25], Davidson and Harel describe their use of simulated annealing to arrange general connected graphs for clarity of presentation. This research adapts their implementation, and extends it to hierarchical hypergraphs. An arrangement tool, ANNEAL, was written to act as a pre-processor for model data, to assign the X-Z positions to the objects which form the nodes of the graph. The tool runs the graph (which initially has no node positions associated with it) through a simulated annealing to position the nodes. It also allows manual editing, using the mouse, of the node positions either during or after the anneal. For purposes of the annealing, the Y-dimension is ignored, and the model is treated as a two-dimensional hypergraph.

6.1 Simulated Annealing

Simulated annealing (abbreviated as SA) was first devised in 1953, as a mathematical model of the physical process of annealing [9], but it was 30 years before it was proposed as a general technique for combinatorial optimization [10]. Since then SA has found application in a wide variety of disciplines ranging from pure mathematics (Travelling Salesman Problem) to VLSI design (block placement and interconnect routing) [11][12]. In systems which can take on a large (possibly infinite) number of possible configurations, but for which there is no simple analytical method which yields the optimum one, SA stochastically searches the configuration space and finds one which is probably near the global optimum. In this work, I have applied SA to optimize the placement of display elements with respect to certain aesthetic criteria.

In order to understand the use of simulated annealing as an optimization method, it is helpful to understand its relationship to the physical process of which it is an analog.

6.1.1 Background on Physical Annealing

The annealing of an ingot of material proceeds by first raising the ingot to a high temperature (to allow the atoms to move freely and release any crystal defects), and then slowly lowering the temperature. The temperature must be lowered slowly enough that the ingot remains in *thermal equilibrium* at each stage, or else defects (which represent local areas of strain energy) will be frozen in, and the total energy will not be minimized.

Thermal equilibrium is characterized by the condition that the total energy E of the system falls on the *Boltzmann Distribution*:

$$p(E) = e^{-E/kT} \quad (\text{EQ 1})$$

where:

E is the energy of the system.

T is the temperature, and

k is Boltzmann's constant.

If the temperature is lowered arbitrarily close to absolute zero, the ingot will then be in its lowest possible energy state.

6.1.2 Optimization by Simulated Annealing

To apply simulated annealing to an optimization problem, several items must be defined:

1. *Cost function*: for each configuration C , the cost function can be evaluated to give the measure of "goodness" of the configuration (with lower cost being better). The cost is the analog of the energy of a physical system, and will be symbolized here by $E(C)$.
2. *Perturbation function*: a rule used to generate a new configuration from an existing one. The *neighbourhood* is then defined as the set of all configurations reachable from a starting configuration by one iteration of the perturbation function.

3. *Control parameter*: this is the analog of temperature, and is used as described below. It is symbolized here by T . The *cooling schedule* is a scheme for lowering T from an initially high value to a lower, final value.

One method of searching for an optimum would be to start the system in some random configuration (call this C), and then perturb it to generate a new configuration C' . If the new configuration has a higher cost than the old (i.e. $E(C') > E(C)$), then discard C' , otherwise, replace C by C' . In either case, repeat the procedure until no further improvement can be obtained. However, this algorithm is liable to become trapped in a local minimum of the cost function which is far from the global minimum we are seeking. Simulated annealing avoids this problem by sometimes allowing "up-hill" moves (i.e. accepting perturbations which increase the cost) to escape from such minima. Up-hill moves are accepted with a probability given by the *Metropolis criterion*:

$$p(\text{accept-uphill-move}) = e^{-|E(C') - E(C)|/T} \quad (\text{EQ 2})$$

(Here the units of E and T have been chosen such that $k=$ unity).

It can be shown that, for a given value of T , this criterion leads to a Boltzmann distribution for E , as given in Equation 1, and thus simulates the condition of thermal equilibrium.

In order to use simulated annealing, it remains to define the *cooling schedule*, which consists of:

1. The choice of an initial (high) value for the control parameter T .
2. A scheme for lowering T such that the system remains in (simulated) thermal equilibrium.
3. A stopping criterion for the anneal.

The initial value of T is not critical as long as it is sufficiently high to guarantee good randomization of the system. There are two general schemes for decreasing T . These are:

1. The *homogeneous* algorithm, in which T is decreased in discrete stages, being held constant during each stage while the main loop is repeated to allow the system to come back into equilibrium at the new value of T . In this case it is necessary to determine, at each cooling stage, when enough passes have been made at that stage to guarantee equilibrium, and how large the decrement in T should be between stages.
2. The *inhomogeneous* algorithm, in which T is decreased by a small amount after each execution of the main loop. The decrement is small enough that the system is not far from equilibrium at each new value of T .

The stopping criterion should ideally be “when the cost function reaches its lowest possible value”. However, the optimum value of the cost function is in principle unobtainable for the class of problems of interest, and so in practice the stopping criterion is either: when the cost has shown no appreciable decrease over the course of several cooling stages, or when some predefined iteration limit is reached.

A lengthy and rigorous mathematical background on the SA algorithm, discussion of several applications, and an extensive bibliography, are given in [12].

6.2 Annealing the Scaffold Model

6.2.1 Cost Function

For the purpose of arranging the hypergraph which represents a scaffold model, the cost function embodies aesthetic criteria for viewing the display: undesirable features are penalized. Different aesthetics may be given different weights to reflect features the viewer considers important for clarity. The components considered in the cost function are:

1. *Node Spacing*: It is desirable that nodes be spaced out more-or-less evenly. This is done by including in the cost function a term proportional to the sum of the reciprocal squares of all the distances between every pair of nodes in the graph (whether they are directly connected or not). The effect of this is to make all the nodes “repel” each other. Additionally, each node can have an individual weight assigned to it, making it repel other nodes more strongly.

2. *Edge Length and Hyperedge Size:* It is desirable that edges and hyperedges not be allowed to become too long, or too large. Edge lengths are controlled by adding a term to the cost function proportional to the sum of the squares of the lengths of each edge. A hyperedge is treated as an all-to-all connection of its member nodes, and so contributes a cost equal to the sum of the squares of the distances between all pairs of member nodes. The effect is to make edges and hyperedges behave like rubber bands or sheets, trying to collapse to their smallest length or size. Individual edges and hyperedges may be assigned different weights, causing some to contract more strongly than others.
3. *Edge-Edge and Edge-Hyperedge Crossings:* To avoid confusing networks of crossing lines, it is desirable that edge crossings be minimized. Accordingly, whenever the crossing of two edges is detected, a fixed quantity is added to the cost function. Similarly, when an edge crosses a hyperedge, an amount proportional to the length of the edge which lies inside the polygon is added to the cost.

The result of annealing with such a multi-term cost function is a compromise between the various aesthetic criteria. The relative weight of each of the above components can be adjusted to emphasize one aesthetic over the others.

There is no term in the cost function penalizing overlap between hyperedges, or for non-member nodes which lie within the bounding polygon of a hyperedge. It was found that the program produced satisfactory results without them, and so they were omitted to save computation time.

In order to prevent nodes from wandering off the screen, Davidson and Harel included a term which caused nodes to be repelled from the borders. However, this author found that it tended to produce graphs which were squashed in areas which lay near the borders, and so the borders term was eliminated. Instead, the display window of the anneal tool is automatically scaled and translated so as to track the graph's location and size as it anneals.

6.2.2 Perturbation Function

In each pass through the graph, each node is perturbed in turn and the new value of the cost function evaluated, with the result being accepted or rejected by the Metropolis Criterion

(Equation 2 on page 48).

Two perturbation methods are available, with the method used being chosen at random for each perturbation. One method is that the node being perturbed is moved by a random distance (up to a maximum) in a random direction. The second method is that the node swaps positions with another randomly chosen node. It was found that allowing node swaps was more effective at untangling complex networks of criss-crossed edges than was only allowing simple moves.

6.2.3 Cooling Schedule

The initial value of the control parameter T can either be specified by the input data, or determined automatically. In the automatic algorithm, the graph (initially arranged randomly) is perturbed a large number of times, and the average change in cost caused by a single perturbation is calculated. T is then set equal to this value. Recalling Equation 2, and setting T equal to the expected value of $E(C') - E(C)$, we have:

$$p(\text{accept-uphill-move}) = e^{-1} \approx 0.37 \quad (\text{EQ 3})$$

which gives a reasonably high probability of accepting uphill transitions, thus avoiding getting caught in a local minimum right at the start of the annealing process.

Following the practice of Davidson and Harel, the cooling schedule is homogeneous. The control parameter is lowered in a number of stages, with a number of complete passes through the graph being made at each stage. Between stages, the control parameter is decreased by a factor γ , the value of which is an input parameter (in practice, a value of 0.75 yielded acceptable results). The number of passes at each stage, and the number of stages (also input parameters) are likely to be different for every graph, and must be determined by experiment: a graph which "freezes" early in the process, while still in a very messy state, is a clear indication that the cooling is too rapid (i.e., either γ , or the number of passes at each temperature, must be increased). The only penalty for cooling too slowly is that the annealing takes longer than necessary.

6.3 Extensions for Containment

The scaffold model allows objects to contain other objects and their interrelationships (both within the container, and with external objects). In the corresponding hypergraph representing the model, nodes are allowed to contain sub-graphs. Edges may connect nodes across levels of decomposition, as shown in Figure 22. The circles represent nodes, and nodes containing subgraphs are shown expanded so the internal structure can be seen. Some edges (for example AC) connect nodes at the same level in the hierarchy. Others cross node boundaries to connect nodes at different levels (AG, for example), or connect nodes at the same level, but in different decompositions (e.g. DE).

Figure 22: Nodes with Subgraphs

Attempting to anneal all nodes in the graph at once would be time-consuming. For this reason, the anneal proceeds in a top-down fashion through the hierarchy: first the outermost decomposition is annealed, and then the internal parts of each node, and so on down

through all levels of decomposition. At each level, the internal structure of the nodes are ignored. Edges which come from an enclosed node are considered as connecting to the enclosing node. When a subgraph (i.e. one contained in a node) is annealed, “dummy” nodes, fixed to the enclosure’s perimeter, are generated to connect edges entering from the outside to the internal nodes. Thus the problem of annealing the entire hierarchical hypergraph is broken down into that of annealing a number of smaller hypergraphs. As the computational complexity of the annealing heuristic is quadratic in the number of nodes and edges, breaking up the graph in this way saves time. The successive decompositions used in annealing the graph of Figure 22 are shown in Figure 23 (the “dummy” nodes are shown in gray).

Figure 23: Decomposition of Hierarchical Graph for Annealing

Chapter 7: Case Studies

This chapter presents case studies on the application of the Scaffold Model to the visualization of two object-oriented programs: HotDraw and Anneal. A note on the colour figures is in order: they were generated as screen dumps from an SGI workstation, and some loss of detail and colour contrast inevitably results from the reproduction process. Also, a static reproduction cannot convey the same information as a model with which the viewer can interact -- our visual systems are adapted to detect movement, and a detail which may be difficult to make out against the background when stationary often becomes obvious as soon as the displayed model is rotated slightly.

7.1 HotDraw

HotDraw is an object-oriented SmallTalk framework for creating 2D graphical applications. The framework can be extended and customized by building on the existing class hierarchy, taking advantage of interactions already defined. An analysis according to the Role Architecture Model (an early version of the slot model) already exists [2], part of which was shown in Figure 1, on page 3. This section demonstrates the application of the Scaffold Model to explication of the HotDraw framework and its functioning.

7.1.1 HotDraw Teams

The SmallTalk language [26] is itself a framework, defining a number of classes, which embody various important interactions. The `Object` class, from which all others are derived, defines the concept of *dependency*. This is the concept that, during the execution of a program, the state of some objects depends on the state of other objects, and that the state of the dependents should be updated whenever the "source" object's state changes. Each instance of the `Object` class maintains a list of other objects which depend on it. When an object's state is changed, it notifies all of its dependents by sending an `update` message to each of them. In slot model analysis, this interaction defines the *Dependency team*.

The Dependency team is used and extended in the Model-View-Controller (MVC) paradigm [27], a construct used frequently in SmallTalk user interfaces. This represents a three-slot team containing:

1. A Model object encapsulating some data.
2. One or more dependent View objects which display the data (possibly each in a different way).
3. A Controller object which senses user input, and directs the Model to change its data values (i.e. its state) appropriately. Since all the Views depend on the Model, all the displays of the Model data are updated automatically.

Figure 24: Full Derivation of Model-View-Controller Team

`Model`, `View` and `Controller` are abstract classes, and a given application makes use of them by subclassing and defining specific behaviours. The result, in terms of teams, is a three-level construct: Dependency being extended to MVC, being further extended to a specific implementation of MVC for the application (see Figure 24). Here, the SmallTalk language system is considered the base layer, while the application is the layer above this.

7.2 Scaffold Model of HotDraw

The HotDraw framework makes extensive use of MVC teams, many of which “overlap” (i.e. more than one MVC team makes use of the same object). The Scaffold Model applies 3D techniques to elucidate the relationships (see Figure 25).

The four teams visible are: **ToolMVC** (to the left), **EditorMVC**, **DrawingMVC**, and **ToolAction** (to the right, stacked above each other). The last three all share two objects in their Controller and View slots: **DrawingController** and **DrawingView**. The **DrawingEditor** object is shared between the **EditorMVC** and **ToolMVC** teams, filling the Model slot in both teams. **ToolAction** is not a MVC team, and its third slot requires a Tool object, **CurrentTool**, a placeholder which is filled from the **Tools** collection.

It would be difficult to display such an overlapping structure in 2D in such a way that the relationships between the various parts are clear, but the 3D representation makes them naturally apparent. Note that the team “platforms” are rendered to be somewhat transparent, so as not to obscure elements behind them (unfortunately, this does not always show up well in reproductions).

Figure 26: HotDraw scaffold with SmallTalk layer showing

Figures 26 and 27 show the HotDraw scaffold with the Smalltalk layer display turned on. Figure 26 views the model from approximately the same vantage point as Figure 25, to allow comparison, while Figure 27 is from the opposite side. Note that layer floors, like teams, are also partially transparent, to allow the user to see objects in lower layers. Here we can see the Dependency and generic MVC teams, located in the SmallTalk layer, beneath each application-specific MVC team, as described in Figure 24.

Figure 27: HotDraw scaffold with SmallTalk layer showing (alternate view)

Other features to note are:

- The collections **Tools** and **Drawings**, both maintained by **DrawingEditor** (green arc-and-collar symbols).

- The optional slots **CurrentTool** (in **ToolAction**) and **CurrentDrawing** (Model slot in **DrawingMVC**), which are filled by concrete objects coming from the above collections (red installation arrows).

CurrentDrawing also maintains a collection, **Figures** -- a drawing is, after all, composed of figures. Figure objects do not participate in a team except when some operation (e.g. copy, move, cut, re-size) is to be performed on them. In order to operate on a figure (or a group of figures), they must first be selected, which causes small "handle" markers to appear on them on the screen. The visual handles are themselves represented by software objects. The association between handles and figures is maintained within the **DrawingView** object. This relation is made visible (see Figures 28 and 29) if **DrawingView** is expanded, by clicking on the yellow cone atop the cylinder.

Figure 28: Expansion of **DrawingView**
(close-up, looking down from above)

Figure 29: HotDraw scaffold with DrawingView expanded

This reveals that several figures (**SelectedFigures**) participate with several **Handles** in multiple **FigureSelection** teams. An installation arc now stretches back to the external **Figures** pool to show the source of the concrete objects filling the optional **SelectedFigures** slot. Note (compare Figure 25 to Figure 29) that the expansion of **DrawingView** (at the right of Figure 29) has caused a distortion of the rest of the diagram to allow **SelectedFigures** and **Handles** to appear the same size as the other objects. The overlapping teams **ToolAction**, **DrawingMVC** and **EditorMVC** are visible to the upper centre of the model, each with one corner touching the boundary of the expanded **DrawingView** object.

7.3 HotDraw Use Cases in Scaffold Model

7.3.1 Use CaseDisplay and Animation

The UseCases menu (not implemented in the experimental viewers) allows the user to choose which use case paths to display.

Use case display (when in edit mode) also provides two additional mechanisms for control of layer display and fish-eye: fold icons and stubs. Where a use case map passes through a layer fold (see Figure 14b), it is marked by a layer fold icon. Clicking on the icon causes the display to be rebuilt so that the layer is now shown unfolded, with the path's full course visible. Similarly, if a path undergoes some complex interaction inside an object, this will be abbreviated to a stub when the object is shown unexpanded. Clicking on the stub invokes fish-eye on the object (in the same way as if the object's expansion handle were clicked).

In order to better show the behaviour of the system, clicking on a use case start sphere will cause a cone-shaped indicator to begin to move along the path. As it moves, it will invoke the movement of objects into slots (e.g. a wireframe placeholder will become solid), or trigger other displayed use cases as it passes.

7.3.2 Use Case Paths and Layer Folds

Figure 30, shows one of the use cases associated with the EditorMVC team, superimposed on part of the HotDraw scaffold. The loadDrawing path starts (at the pink sphere to the right of centre) with user input (the selection of a new drawing to edit, as sensed by Drawing-

Controller), causing DrawingEditor (the Model) to be changed (i.e. a new drawing is loaded into the editor), in turn causing DrawingViewer to be updated (i.e. the new drawing to be displayed in the window). The path terminates at the disk at left.

Figure 30: EditorMVC: loadDrawing Use case Path

The segment of use case from Model to View involves a Dependency team defined down in the SmallTalk layer. The presence of this team is signified by the green layer-fold icon (intended to resemble a layer tile). The use of layer-fold icons will be discussed in the next section.

```

classDiagram
    class Primitive {
        ArrayIterator<T>
    }
    class Geometry {
        GVertex
    }
    class Display {
        ViewPort
    }
    class Graph {
    }

    Primitive <|-- ArrayIteratorT["ArrayIterator<T>"]
    Primitive <|-- CircleIterator["CircleIterator<>"]
    Primitive <|-- ArrayT["Array<T>"]
    Primitive <|-- StackT["Stack<T>"]
    Primitive <|-- RetCounter["RetCounter"]
    Primitive <|-- GSource
    Primitive <|-- GDependent

    Geometry <|-- GVertex
    Geometry <|-- GIntersection
    Geometry <|-- GPoint
    Geometry <|-- GLineSegment
    Geometry <|-- GPolygon

    Display <|-- ViewPort
    Display <|-- PlotWindow
    Display <|-- DGroup
    Display <|-- DPoint
    Display <|-- DLineSegment
    Display <|-- DPolygon

    Graph <|-- GraphComponent
    Graph <|-- Hyperedge
    Graph <|-- Node
    Graph <|-- Edge
    Graph <|-- EntryNode

    Graph <|-- GraphViewer
    GraphViewer <|-- AnnealAction

    CircleIterator --> Graph
    ArrayT --> Graph
    StackT --> Graph
    RetCounter --> Graph
    GSource --> Graph
    GDependent --> Graph
    GVertex --> Graph
    GIntersection --> Graph
    GPoint --> Graph
    GLineSegment --> Graph
    GPolygon --> Graph
    ViewPort --> Graph
    PlotWindow --> Graph
    DGroup --> Graph
    DPoint --> Graph
    DLineSegment --> Graph
    DPolygon --> Graph
    GraphComponent --> Graph
    Hyperedge --> Graph
    Node --> Graph
    Edge --> Graph
    EntryNode --> Graph
    GraphViewer --> Graph
    AnnealAction --> GraphViewer
  
```

The diagram illustrates the class hierarchy and relationships within the GraphViz library. It is organized into four main categories: Primitive, Geometry, Display, and Graph. The Graph category is the most complex, containing numerous classes that represent the core data structures and algorithms of the library.

Figure 31: Anneal Class Hierarchy with layers

7.4.1 Layers and Teams in Anneal

The Anneal class hierarchy divides into four layers (see Figure 31), which are described below:

1. *Primitive*: contains classes embodying a few simple concepts which are used as basic tools throughout the framework. It includes:
 - A template `Array` class, and two iterator classes `ArrayIterator` and `CircIterator` for use with it. The interaction of `Array` with `ArrayIterator` defines the *Iterator team*.
 - The `RefCounter` class, which implements a reference-counting mechanism to perform memory management for transient objects.
 - The classes `GSource` and `GDependent`, which partially implement the slots of a *Dependency team*, similar to that found in SmallTalk. `GSource` maintains an `Array` of `GDependents`, which it keeps updated by calling the `GDependent::notify()` method in response to changes in its own state (this is done within the `GSource::changed()` method). `GSource` is derived from `RefCounter`, because it is frequently used in a context where a source object should be deleted automatically when all its dependents have been detached. `GDependent::notify()` is abstract and subclasses must provide an implementation for it.
2. *Geometry*: defines classes used to perform basic geometrical constructions and calculations. It includes:
 - `Vector`, which holds the (x,y) value of a 2-vector, and defines basic vector arithmetic operations.
 - `GPoint`, which encapsulates a `Vector`, and expresses the concept of a location in 2D Cartesian coordinates. It is also derived from `GSource`, as points are used to define the location and shape of lines and polygons (which are then dependent on it). It defines methods such as `setPos`, `moveTo` and `moveBy` which control the way in which a point's location may be set and changed. When one of these methods is invoked, a `GPoint` calls `changed()` on itself.
 - `GLine`, which is simply a data structure holding the equation of a line in slope-intercept form.
 - `GLineSegment`, which is a `GLine` bounded by two `GPoints`. It is derived from `GDependent` so as to make use of the *Dependency team* definition: when one of the two end-points is moved, the `GLineSegment::notify()` method causes positions of the ends to be fetched, and the equation of the line to be recalculated:

```

class GLine {
    // calcEqn takes references to two GPoints and calculates the
    //equation of a line through them
    void calcEqn( const GPoint& p, const GPoint& q ) {
        /* Calculate slope & intercept */
    }
};

class GLineSegment : public GLine, public GDependent {
    GPoint*end[2]; // Pointer to end-points
    /*.....*/
    //Re-calculate equation
    void notify( const GSource& ) { calcEqn( *end[0], *end[2] ); }
};

```

This implementation of `notify()` creates an *elaboration* (see Section 4.3.1 on page 32) of the `Dependency` team definition, which is called the `LineSegment` team definition.

- `GPolygon`, which maintains an `Array` of the `GPoint` objects which comprise its vertices. It also has a `Dependency` relationship with its source points: the vertices are supposed to be kept sorted in counter-clockwise order (as seen from the centroid of the polygon). The implementation of `GPolygon::notify()` (called whenever a vertex point is moved) is to re-sort the array of points. This elaboration of `Dependency` constitutes the `Polygon` team definition.
 - `GVertex` is an `ArrayIterator` specialized to refer to the vertices of a `GPolygon`. It is used to perform various geometric constructions and operations on polygons.
3. *Display*: contains classes used to display geometric forms on the screen. They are:
- `PlotWindow`, a data structure which maintains the current mapping between the world coordinates of the geometrical objects being displayed, and the pixel coordinates of the screen. The actual mapping data may only be accessed through controlled methods, which (for example) return screen coordinates of a point given in world coordinates.
 - `Displayable`, a class which defines only a colour, and a virtual `draw( PlotWindow )` method. To create a concrete displayable object, the `Displayable` class must be combined (through multiple inheritance) with one of the `Geometry` classes, and the `draw()` method implemented to use the `PlotWindow` data to draw the object at the proper location and size. The collaboration use made of a `PlotWindow` by a `Displayable` during the operation of `draw()` is considered to constitute the `Display` team.

4. *Graph*: contains classes pertaining to connected hypergraphs, such as Node, Edge and Hyperedge. The following sections discuss the use of the Scaffold Model Viewer to explore how the functions of this top layer (the application) are derived from those of the lower layers.

7.4.2 The Architecture of Anneal

Figure 32 shows a top-level view of the architecture of the Anneal program. It consists of a team **ViewAndEdit**, which defines the interaction between **theGraph** (an instance of Graph), **theWindow** (a PlotWindow object) and **mainAnnealer** (an instance of AnnealAction). The **ViewAndEdit** team (an instance of the ViewAndEdit team definition) allows for the display of the Graph in the PlotWindow, and its alteration by an AnnealAction.

Figure 32: Anneal: top-level view, with moveNode use case path

The place-holder object **PW1** is filled from **theWindow**, and its use is discussed in Section 7.4.3.

The **moveNode** use case map starts at **mainAnnealer**, and goes through a stub in **theGraph**. Since we are interested in behaviour, our exploration of this architecture will be carried out by following this path.

Figure 33: Anneal with theGraph expanded

In Figure 33, **theGraph** has been expanded (by clicking either on its expansion handle, or on the **moveNode** stub), revealing that it contains collections of Nodes, Edges and Hyper-edges. The **moveNode** use case path enters the **theNodes** collection (but only influences one Node), passes to **theEdges** (where it affects several Edges) and exits from **theGraph**. It splits (and-fork) as it crosses as moving one Node may affect several Edges. Var-

ious teams connect the internal components of **theGraph**. In Figure 34 we are attempting to move in and examine the details of the area between **theNodes** and **theEdges**. However, **theHypers** is in the way.

Figure 34: Obscured view in Scaffold display

Using the pruning facility (Section 5.1.6 on page 45) to turn the occluding object transparent, we obtain a close-up view of the **NodeEdge** team which associates Nodes with Edges (Figure 35). This is the “bridge” by which the **moveNode** path passes from **theNodes** to **theEdges**. The layer-fold icon in the middle of this path indicates that there are details of the interaction hidden at a lower layer.

Figure 35: NodeEdge team

Clicking this icon allows one to investigate the nature of this team more deeply by displaying lower layers of the model. In Figure 36, we are displaying down to the Geometry layer, and our vantage point has also moved to a location within the Geometry layer. This allows us too see that the **NodeEdge** team is defined here, and not in the Graph layer (and note that the path now crosses over here). A query informs the user that **NodeEdge** is an instance of the **LineSegment** team definition (here, the query windows have been added afterwards).

Figure 36: Geometry layer view of NodeEdge team.

There is still a layer-fold icon on the NodeEdge team, indicating that there is yet lower-level detail to be examined. In Figure 37, this is expanded to show that the LineSegment team definition has its roots down in the Primitive layer (camera location is the same as in Figure 36, but now pointing downwards through the Geometry floor). The team labelled [NodeEdge] is an instance of Dependency, which is elaborated in the definition of LineSegment. A glance at the inheritance hierarchy (or, a query of the model), shows that theNodes and theEdges are derived from classes which support the proper slot interfaces and behaviours for LineSegment and Dependency.

Figure 37: NodeEdge team: origins in Primitive layer

From the cross-over, the `moveNode` paths proceed up the `theEdges` column to the Geometry layer, where it spawns use cases back across `NodeEdge` into `theNodes`. This represents the query which affected `GLineSegments` make of their end-points in order to recalculate their position

Knowledge of the team and class relationships spanning several layers gives us understanding of how the Anneal program works: looking at the code for the `Node`, `Edge` and `AnnealAction` classes only, there is no information as to how the relationships between `theNodes` and `theEdges` are maintained. The Scaffold Model explains how this relationship is maintained by the `LineSegment` team definition, and that this (which in turn uses the `Dependency` mechanism) keeps the states of individual `Edge` objects continually updated as the nodes are moved in the course of annealing.

7.4.3 Optional Slots and Use Case Maps

In the architectural model, optional slots represent placeholders: team members for which an interface has been defined, but whose precise identity may change over the course of time.

Consider the top-level of Anneal again, this time with the `displayGraph` use case showing (below).

Figure 38: Anneal with `displayGraph` use case map

Clicking on the start of the `displayGraph` use case causes the path to be animated (Figure 39): a token passes along it, and the PW1 slot is filled from the `theWindow` object.

Figure 39: Use Case Map Animation, showing slot-filling

The display of the **theGraph** object follows the nested structure of the **Graph** class itself. Nodes **Edge** and **Hyperedge** are all **Displayables** which implement the virtual **Draw()** method. Class **Graph** groups them within **DGroup** objects (Arrays of **Displayable**), which implements **draw()** to call the **draw()** method of each array item. **Graph::draw()** simply calls **draw()** on each of the collections in turn. The **Display** team exists only until the **draw()** method exits, and so the **PlotWindow** is considered to be a placeholder slot in an optional team, being filled when **draw()** is invoked with a **PlotWindow** passed as parameter. The complete use case map, with **theGraph** expanded, can be seen in Figure 40 and Figure 41.

Figure 40: displayGraph path with theGraph expanded

Figure 41: displayGraph path with theGraph expanded (alternate view)

Chapter 8: Conclusions and Future Direction

8.1 Summary

The designers and maintainers of large software systems use various visualization tools to help them to understand their systems. Large software systems not only have many components, but each component may have several aspects, and diverse relations to other components. The amount and complexity of information taxes the limits of what can usefully be shown on a 2D screen. Displaying information in a piece-meal fashion (multiple windows) results in display clutter and fragmentation of the information, which a human observer must then mentally re-integrate in order to comprehend the system structure. One possible solution to this problem is to display the system in three dimensions, thereby increasing the effective "space" through which objects and relations may be spread. The recent availability of the necessary hardware to make real-time rendering practical has opened this up as a promising area of research.

While the available literature records some success with simply expanding existing 2D techniques (based on node-and-arc diagrams) into 3-space, doing this does not immediately solve the display-space problem. Since objects tend to occlude other objects, the extra amount of "space" in a 3D scene is not actually all available at the same time for the display of additional information. A survey of recent applications of 3D techniques for visualizing complex data sets suggest that it may be more useful to give the additional dimension a specific use.

8.1.1 Conceptual Framework for 3D

The objective of this thesis has been to develop a particular 3D representation of a software model (the Team and Slot model), and to demonstrate the principles involved through a prototype display tool. The Scaffold Model has been developed and described, and comprises the following abstractions:

- **Layers:** conceptual partitioning of the model into different functional domains, here representing different levels of abstraction in an inheritance hierarchy.
- **Team Definitions and Roles:** specifications of collaboration groups of interacting objects, which can be used conceptually to understand and/or build a system.
- **Teams and Objects:** concrete instances of Team Definitions and Roles, as they exist in a running program.

Objects are rendered as vertical pillars, with teams as polygonal platforms suspended between them. Layer divisions are shown as tiled floors cutting horizontally through the scene.

8.1.2 Proposed 3D Tool

The desirable features of a 3D browsing tool, SMV, for the Scaffold Model have been described, and several of these are demonstrated in the experimental viewers Crane and SeamVu. The features proposed are:

- Layer Display selection.
- Fish-eye expansion of objects containing internal detail.
- Use Case Map display and animation.
- Constrained Virtual Trackball with point-of-interest logarithmic movement as a navigational paradigm.
- User pruning of foreground objects.

A related sub-problem in visualization is that of arranging the elements of the data set in such a way as to minimize clutter and enhance understanding. This research has treated this as a problem in optimization of a function embodying appropriate aesthetic criteria, and demonstrates the simulated annealing heuristic as the optimization method. This is implemented as a pre-processor for Scaffold Model data.

8.2 Conclusions

The Scaffold Model shows promise as a basis for display of the Role Architectural Model in 3D. Using the third dimension allows:

- Natural separation of levels of abstraction in an object-oriented system.

- Avoidance of tangled relationships between objects within layers.

The proposed Scaffold Model Viewer, with Layer Display selection, fish-eye expansion of objects, Constrained Virtual Trackball movement, and user-activated pruning, allows efficient navigation through the Scaffold Model, providing orientation cues, and the means to expose or hide detail at will.

The ability to superimpose Use Case maps on the model allows us to visualize run-time behaviour, both by tracing the flow of activity over time, and through different layers, and by showing the process of role-filling.

Simulated Annealing proved to be a satisfactory method of arranging the model elements for clear display.

8.3 Suggestions for Further Work

8.3.1 Scaffold Model

The SMV tool has not actually been built. While most of the basic concepts have been tried out in the experimental viewers Crane and ScamVu, some remain to be tested:

1. Query: The query function has been partially implemented, with the description being dumped to the standard output¹. It should be properly implemented so that the text appears in pop-up windows on the model. This will allow experimentation to determine the best way of presenting this information (and allowing the user to get rid of it when it is no longer needed).
2. Use case display: Use case maps are not implemented at all in either of the experimental viewers². Displaying them, under user control, or even allowing the user to trace them manually, would add considerable power to the tool's ability to show the behav-

¹. The query windows appearing on figures in Chapter 7 were added manually.

². Use case maps appearing on figures were created by displaying the model in ScamVu, saving the geometry to an Inventor script file, editing the file to add the path geometry, then displaying the modified scene with a generic SGI viewer.

jour of object-oriented architectures. This would necessitate the creation of semantics for defining use case paths through the system, and adding these to the SMV script language (described in Appendix A).

3. Use case animation: a further extension to use case display would be to implement the animation feature, including showing the filling of placeholder roles.

The Scaffold Model attempts to describe the design concepts behind a system. It would be useful to couple this more closely with the system-as-written, by associating more concrete information with the objects, classes and teams. For example, querying an object pillar might also give access to a class browser, and to method lists and code listings. A team query might produce a precise specification of the pattern of method invocations among the team partners.

The experimental tools described here-in were created with C++. Since they were completed, Virtual Reality Modelling Language (VRML) has been invented as a generic tool for creating interactive "smart" 3D models, and is attracting great interest in the VR and World Wide Web communities. VRML is something of a hybrid between the SGI Inventor scene-description language and Hyper-Text Markup Language (the primary medium of the World Wide Web). It includes primitives to support animation, and to provide HTML-style links to view other scenes, or HTML documents, or to modify the scene by executing scripts written in the Java language. VRML version 2 [28] appears to have most of the semantic power necessary to implement the Scaffold Model concepts. VRML/Java, together with a VRML viewer incorporating appropriate navigational paradigms, would probably be the tools of choice for future work on the Scaffold Model, by allowing the designer to concentrate on modelling, rather than programming.

8.3.2 Display Arrangement

The display arrangement is performed by simulated annealing. This could be improved by speeding it up (possibly by choosing the cooling schedule more carefully). It is also possible that some other optimization heuristic would produce better results (for example, a genetic algorithm).

References

- [1] Stephen Jay Gould. "A Tale of Three Pictures" in *Eight Little Piggies*, pg. 427. W.W.Norton & Co. 1993.
- [2] R.S. Casselman. *A Role-Based Architectural Model Applied to Object-Oriented Systems*. Master's Thesis, Dept. of Systems and Computer Engineering, Carleton University, OCIEE-93-11.
- [3] Michael F. Kleyn and Paul C. Gingrich. "GraphTrace -- Understanding Object-Oriented Systems Using Concurrently Animated Views", *Proceedings of OOPSLA '88*, pp. 191-205.
- [4] *ObjecTime Manual*. ObjecTime Limited.
- [5] Hideki Koike "The Role of Another Spatial Dimension in Software Visualization", *ACM Transactions on Information Systems* 11(3):266-286, July 1993.
- [6] Frederick P. Brooks Jr. "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer* 20(4):10-19, April, 1987.
- [7] Charles Qi Cui. *Layered Fish-eye View Techniques for Understanding of Large Software Systems*. M.C.S. Thesis, School of Computer Science, Carleton University, 1993.
- [8] Jock D. MacKinlay, George G. Robertson and Stuart K. Card. "Rapid Controlled Movement Through a Virtual 3D Workspace", *Computer Graphics* 24(4):171-176, August 1990.
- [9] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *Journal of Chemical Physics*, vol. 21, pp.1087-1092, 1953.
- [10] S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671-680, May 1983.
- [11] P. Siarry, L. Bergzoni, and G. Dreyfus, "Thermodynamic Optimization of Block Placement," *IEEE Transactions on Computer-Aided Design*, CAD-6(2):211-221, March 1987.

- [12] P.J.M. van Laarhoven and E.H.L. Aarts, *Simulated Annealing: Theory and Applications*, Reidel, Dordrecht, 1987.
- [13] George G. Robertson, Jock D. MacKinlay and Stuart K. Card, "Cone Trees: Animated 3D Visualizations of Hierarchical Information", *Proceedings of ACM CHI '91*, pp. 189-194.
- [14] Harold L. Ossher, *A New Program Structuring Mechanism*, Ph.D. Thesis, Stanford University, 1984. Published as Technical Report STAN-CS-85-1078, Stanford University.
- [15] Harold L. Ossher, "Multi-Dimensional Organization and Browsing of Object-Oriented Systems", *Proceedings of the 1990 International Conference on Computer Languages* pp. 128-135. IEEE, 1990.
- [16] Peter Reid, "Dynamic Interactive Display of Large Data Structures", in *Graphics Tools for Software Engineers*, pp. 60-70. A. Kilgour & R. Earnshaw, ed. Cambridge University Press, 1989.
- [17] Kim M. Fairchild, Steven E. Poltrock, George W. Furnas "SemNet: Three-Dimensional Graphic Representations of Large Knowledge Bases" in *Cognitive Science and its Applications for Human-Computer Interaction*, Raymonde Guindon, ed. Lawrence Erlbaum Associates, Inc., 1988.
- [18] K.W. Arthur, K.S. Booth, and C. Ware, "Evaluating 3D Task Performance for Fish Tank Virtual Worlds", *ACM Transactions on Information Systems*, 11(3):239-265, July 1993.
- [19] John Bryant, "Experiments in 3D Software Visualization", Dept. of Systems and Computer Engineering, Carleton University, TR-SCE-93-33.
- [20] R.J.A. Buhr and R.S. Casselman, *Architecture of the Whole: with Roles and Timethreads*, Dept. of Systems and Computer Engineering, Carleton University, TR-SCE-94-07.
- [21] R.J.A. Buhr and R.S. Casselman, *High-Level Design of Object-Oriented and Real-Time Systems: A Unified Approach with Use Case Maps*, Prentice-Hall, 1995 (in press).

- [22] "Hypergraph". *Encyclopaedia of Mathematics*, Kluwer Academic Publishers, 1989.
- [23] George Furnas. "Generalized Fisheye Views". *Proceedings of ACM CHI '86*. pp 16-23.
- [24] David Harel. "On Visual Formalisms". *Communications of the ACM* 31(5):514-529, May 1988.
- [25] Ron Davidson and David Harel. "Drawing Graphs Nicely Using Simulated Annealing". July 1989, unpub.
- [26] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1985.
- [27] G.E. Krasner and S.T. Pope. "A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80". *Journal of Object-Oriented Programming*, Aug/Sept. 1989, pp. 26-49
- [28] David R. Nadeau. Course Notes: "Introduction to VRML 2.0". SIGGRAPH 96.

Appendix A: Scaffold Model Input

The viewers ScamVu and Crane (and the hypothetical SMV) take input in the form of a text file specifying the model. This appendix describes the syntax and semantics used to specify an architecture in the Scaffold Model, and lists the input specifications of the two case study examples, HotDraw and Anneal.

A.1 Syntax and Semantics

The input language has a key-word for each element of the model. In the following description, literal text is shown as: **TEXT**, parameters to be supplied are shown as: *parameter*, and optional elements are enclosed in square brackets: *[optional]*. Letter case is ignored in key-words, but is significant in string parameters (thus: "parameter" and "ParaMeter" are distinct). Spaces between tokens are ignored.

1. Layers:

LAYER (*layer-name*)

This specifies that all entities defined after this statement belong to the layer *layer-name*, until another **LAYER** statement is encountered.

2. Slots:

SLOT (*new-slot-name*) [**ISA** (*parent-slot-name*)]

This defines a Slot called *new-slot-name*, and optionally specifies that this slot is an elaboration of a previously-defined slot called *parent-slot-name*.

3. Objects:

OBJECT (*obj-name*[, *num*][, **FIXEDPLACE**]) **ISA** (*slot-name*) **AT** (*x*, *z*)
[**COMESFROM** (*source-obj-name*)] [**MAINTAINS** (*pool-name*)] [{.....}]

This specifies an object named *obj-name* which is capable of filling slot *slot-name*, and is positioned in the horizontal plane at (x,z) . The optional *num* parameter indicates that the object is a pool or collection, and should be displayed as a bundle of *num* cylinders (default value is 1). The temporal attribute of the object is indicated by the choice of **FIXED** or **PLACE** (i.e. placeholder), with the default being **FIXED**.

The optional **COMESFROM** qualifier indicates that this (usually place-holder) object is installed at run-time from another object (or pool), called *source-obj-name*. The **MAINTAINS** qualifier indicates that this object maintains a pool, called *pool-name*, of other objects.

If the object being defined contains internal detail, this is indicated by the open brace, followed by a series of **OBJECT** and **TEAM** statements describing the contents, and terminated by the close-brace.

4. Defining Teams:

TEAMDEF (*td-name*) **HASSLOTS** (*slot1, slot2,....*)

This creates the definition of a team, called *td-name*, of a team having slots named *slot1*, *slot2*, etc., which must have been previously defined by **SLOT** statements.

5. Instantiating Teams:

TEAM (*team-name* [, *num*]) **ISA** (*td-name*) **HASSLOTS** (*obj1, obj2,....*)

This specifies the instantiation, named *team-name*, of a previously specified team definition *td-name*, with objects *obj1*, *obj2*, etc., filling the slots. The objects given must have been defined as being capable of filling the respective slots given in the **TEAM-DEF** of *td-name*. The optional *num* parameter indicates that the team is replicated, and should be displayed as *num* platforms.

A.2 Example Model: HotDraw

```
# HotDraw Scaffold Model
Layer (SmallTalk)
  SLOT (Source)
  SLOT (Dependant)
  TEAMDEF (Dependancy) hasslots (Source, Dependant)
Layer (HotDraw)
  SLOT (Controller)
  SLOT (Model) isa (Source)
  SLOT (View) isa (Dependant)
  TEAMDEF (MVC) hasslots (Model, View, Controller)
  SLOT (EvtHdlr)
  SLOT (Tool) isa (Model)
  SLOT (Canvas)
  TEAMDEF (ToolAction) hasslots (EvtHdlr, Tool, Canvas)
  SLOT (Figure)
  OBJECT (Figures.FIXED.7) isa (Figure) at(-130,18)
  OBJECT (Drawings.FIXED.7) isa (Model) at(-137,-17)
  OBJECT (CurrentDrawing.PLACE.1) isa (Model) at(-116,-8) COMESFROM(Draw-
ings) MAINTAINS (Figures)
  SLOT (DwgVwr) isa (View, Canvas)
  OBJECT (DrawingView.FIXED.1) isa (DwgVwr) at(-100,10) {
    OBJECT (SelectedFigures.PLACE.3) isa (Figure) at (-48,119) comesfrom(Figures)
    SLOT (Handle)
    OBJECT (Handles.PLACE.3) isa (Handle) at(-35,110)
    TEAMDEF(HdlFig) hasslots(Handle, Figure)
    TEAM(FigureSelection.3) isa(HdlFig) hasslots(Handles, SelectedFigures)
  }
  OBJECT (ToolPaletteController.FIXED.1) isa (Controller) at (-124,-45)
  OBJECT (ToolPaletteView.FIXED.1) isa (View) at (-111,-52)
  OBJECT (Tools.FIXED.7) isa (Tool) at(-82,-56)
  OBJECT (CurrentTool.PLACE.1) isa (Tool) at(-83,-29) COMESFROM (Tools)
  OBJECT (DrawingEditor.FIXED.1) isa (Model) at(-109,-35) MAINTAINS
(Tools, Drawings)
  SLOT (DwgCtrlr) isa (Controller, EvtHdlr)
  OBJECT (DrawingController.FIXED.1) isa (DwgCtrlr) at(-97,-23)
  TEAM (ToolDep) isa (Dependancy) hasslots (DrawingEditor, ToolPaletteView)
  TEAM (ToolMVC) isa(MVC) hasslots(DrawingEditor, ToolPaletteView, ToolPaletteCon-
troller) includes (ToolDep)
  TEAM (EditorMVC) isa(MVC) hasslots(DrawingEditor, DrawingView, DrawingControl-
ler)
```

```

TEAM (EditorDep) isa (Dependency) hasslots (DrawingEditor.DrawingView) includ-
edby (EditorMVC)
TEAM (DrawingMVC) isa(MVC) hasslots(CurrentDrawing.DrawingView.Drawing-
Controller)
TEAM (DwgDep) isa (Dependency) hasslots (CurrentDrawing.DrawingView) includ-
edby (DrawingMVC)
TEAM (ToolAction) isa (ToolAction) hasslots(DrawingController.CurrentTool.Drawing-
View)

```

A.3 Example Model: Anneal

Anneal Scaffold Model

```

Layer (Primitive)
  slot (Source)
  slot (Dependant)
  teamdef (Dependency) hasSlots(Source,Dependant)
  slot (Array)
  slot (ArrayIterator)
  teamdef (Iteration) hasSlots(Array,ArrayIterator)
  slot (CircularIterator) isa (ArrayIterator)
  teamdef (CircIteration) hasSlots (Array,CircularIterator)
Layer (Geometry)
  slot (Point) isa (Source)
  slot (Line) isa (Dependant)
  teamdef (LineSegment) hasSlots (Point,Line)
  slot (Polygon) isa
  teamdef (Polygon) hasSlots (Point,Polygon)
  slot (Vertex) isa (ArrayIterator)
  teamdef (Vertex) hasSlots(Polygon,Vertex)
  slot (hullFinder)
Layer (Display)
  slot (Displayable)
  slot (PlotWindow)
  teamdef (Display) hasSlots (Displayable,PlotWindow)
  slot (DPoint) isa (Displayable.Point)
  slot (DLine) isa (Displayable.Line)
  slot (DPolygon) isa (Displayable.Polygon)
Layer (Graph)
  slot (Node) isa (DPoint)
  slot (Edge) isa (DLine)
  slot (Hyperedge) isa(Displayable)
  slot (Graph)

```

```

slot (Viewer)
slot (Anneal.Action) isA (Viewer)
teamdef (ViewAndEdit) hasSlots (Graph.Viewer.PlotWindow)
#
# Nodes, Edges, Groups:
object (mainAnnealer.FIXED) isA (Anneal.Action) at (-70,222)
object (theWindow.FIXED) isA (PlotWindow) at (-49,221)
object (PW2.PLACE) isA (PlotWindow) at (-42,200) comesFrom(theWindow)
object (theGraph.FIXED) isA (Graph) at (-60,206) {
  object (theNodes.7.FIXED) isA (Node) at (-15,35)
  object (PWN.PLACE) isA (PlotWindow) at (50,22) comesFrom(PW2)
  team (DN.7) isA (Display) hasSlots(theNodes.PWN)
  object (theEdges.FIXED.7) isA (Edge) at (14,-1)
  object (PWE.PLACE) isA (PlotWindow) at (45,-5) comesFrom(PW2)
  team (DE.7) isA (Display) hasSlots(theEdges.PWE)
  object (theHypers.FIXED.7) isA (Hyperedge) at (27,57) {
    object (theNodes2.FIXED.7) isA (Node) at (-14,-30) comesFrom(theNodes)
    object (hull.FIXED) isA (Polygon) at (-13,-29)
    team (poly) isA (Polygon) hasSlots(theNodes2.hull)
  }
  object (PWH.PLACE) isA (PlotWindow) at (60,50) comesfrom(PW2)
  team (DH.7) isA (Display) hasSlots(theHypers.PWH)
  object (findHull) isA (hullFinder) at (-7,58)
  team (NodeEdge.7) isA (LineSegment) hasSlots (theNodes,theEdges)
  team (NEdep.7) isA (Dependency) hasSlots (theNodes,theEdges) includedby
(NodeEdge)
}
team (ViewAndEdit) isA (ViewAndEdit) hasSlots(theGraph.mainAnnealer.theWin-
dow)

```

Appendix B: Experimental Viewers

The two experimental viewers, Crane and ScamVu, run on a Silicon Graphics IRIS Indigo workstation. Their features and use are described in this Appendix. Both viewers take as their input a file containing a Scaffold Model description, in the syntax described in Appendix A. The suggested file extension for scaffold model source is ".smv".

B.1 Crane

Crane is invoked from the UNIX prompt:

```
>Crane <SMV-file> &
```

where <SMV-file> is the name of the file containing the model to be displayed. The Crane viewer window will appear, with the model displayed in it. Crane does not support showing and hiding of different layers, and so all the layers of the model are always displayed. There is also no Point-of-Interest marker, however there is a point-light located at the POI, and a spot light directed towards it. Crane implements the constrained trackball motion paradigm (Section 5.1.3 on page 43), under keypad control. To exit Crane, double-click on the X-window close handle (upper left corner of the window trim).

Controls are as follows:

1. Mouse:

Left Button: Clicking left on most objects causes the POI to move to that object. The viewer does not immediately move the POI to the centre of the view (it jumps there when the next command is entered). Clicking left on an expansion handle (inverted yellow cone) opens or closes the object's fish-eye.

Middle Button: Click middle on an object to "prune" it, i.e. turn it transparent (see Section 5.1.6 on page 45). The object can be made to reappear by expanding or collapsing a fish-eye (note: this is not a very good interface for this purpose).

2. Keypad: the numeric keypad supports all the movement functions of the Crane viewer, as shown in the keymap below (note that not all keys have functions).

Figure 42: Crane Controls

NumLock	/	*	- ZoomOut
7 Dolly In	8 Rotate Up	9 Expand Spot	+ Zoom In
4 Rotate Left	5	6 Rotate Right	
1 Dolly Out	2 Rotate Down	3 Contract Spot	Enter
0		Del	

3. Other keys:

X/Z: Move the spot-light towards/away from the POI

C/D: Move the near clipping plane in/out.

W: Write the scene geometry to an Inventor file (file type ".so").

B.2 ScamVu

An image of the ScamVu window is shown in Figure 20, on page 39. ScamVu is invoked from the UNIX prompt:

```
>ScamVu &
```

The ScamVu window will appear. The File menu is used to load a scaffold model for viewing, and to exit the program (File|Quit). Choosing File|Load causes a dialog to pop up allowing the user to choose a model for viewing. Once the model is loaded, the Layers menu becomes active, and allows the user to select which layers to view. Initially, no layers are

selected, and the user must select one or more before anything will become visible. Several buttons appear in the right-hand window trim:

1. Eye in profile(at top): When "open", window is in Viewing Mode. When closed, it is in Edit Mode. Clicking the left mouse button on it toggles the setting.
2. Question Mark: Brings up help on the viewer.
3. House: Returns the user to previously-set vantage point.
4. House-with-Arrow: Saves the current vantage point (return here by pressing House button).
5. Eye with Arrows: Set the camera to a position where something can be seen (use it at the start, or if the user has become completely lost).
6. Seek button: Click left here (mouse changes to "gun-sight" icon) and then click on the new POI. Camera will move and re-orient to focus on this spot.

The mouse behaves differently depending on whether the viewer is in View or Edit mode:

1. Left button: In Move mode, click-and-drag causes the model to rotate (in an unconstrained way). In Edit mode, it opens and closes fish-eyes (click on yellow cone), and performs queries on objects and teams (information is written to standard output).
2. Middle button: In Move mode, click-and-drag moves the POI. In Edit mode, it prunes objects.
3. Left + Middle: Performs Dolly In/Out.
4. Right: Brings up help and preferences menu.