1

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

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

Acquisitions and Bibliographic Services

Direction des services bibliographiques

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Layered Fisheye View Techniques for Understanding of Large Software Systems

by

Charles Qi Cui, B.C.S.

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

**Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University,
Ottawa, Ontario, Canada
March 1993**

©1993, Charles Q. Cui

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

Author's Acknowledgement

Author's Acknowledgement

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 0-315-84102-8

Canada

Name

Charles R. Cui

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

U·M·I

SUBJECT TERM

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language
General 0679
Ancient 0289
Linguistics 0290
Modern 0291
Literature
General 0401
Classical 0294
Comparative 0295
Medieval 0297
Modern 0298
African 0316
American 0591
Asian 0305
Canadian (English) 0352
Canadian (French) 0355
English 0593
Germanic 0311
Latin American 0312
Middle Eastern 0315
Romance 0313
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
General 0318
Biblical Studies 0321
Clergy 0319
History of 0320
Philosophy of 0322
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
Archaeology 0324
Cultural 0326
Physical 0327
Business Administration
General 0310
Accounting 0272
Banking 0770
Management 0454
Marketing 0338
Canadian Studies 0385
Economics
General 0501
Agricultural 0503
Commerce-Business 0505
Finance 0508
History 0509
Labor 0510
Theory 0511
Folklore 0358
Geography 0366
Gerontology 0351
History
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
General 0615
International Law and Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family Studies 0628
Industrial and Labor Relations 0629
Public and Social Welfare 0630
Social Structure and Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
General 0473
Agronomy 0285
Animal Culture and Nutrition 0475
Animal Pathology 0476
Food Science and Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Range Management 0777
Wood Technology 0746
Biology
General 0306
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0379
Ecology 0429
Entomology 0353
Genetics 0369
Limnology 0793
Microbiology 0410
Molecular 0307
Neuroscience 0317
Oncology 0416
Physiology 0433
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics
General 0786
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Palaeobotany 0345
Palaeoecology 0426
Paleontology 0418
Paleozoology 0985
Petrology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0300
Chemotherapy 0992
Dentistry 0567
Education 0350
Hospital Management 0769
Human Development 0758
Immunology 0982
Medicine and Surgery 0564
Mental Health 0347
Nursing 0769
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and Therapy 0354
Ophthalmology 0381
Pathology 0571
Pharmacology 0419
Pharmacy 0572
Physical Therapy 0382
Public Health 0573
Radiology 0574
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
Chemistry
General 0485
Agricultural 0749
Analytical 0486
Biochemistry 0487
Inorganic 0488
Nuclear 0738
Organic 0490
Pharmaceutical 0491
Physical 0494
Polymer 0495
Radiation 0754
Mathematics 0405
Physics
General 0605
Acoustics 0986
Astronomy and Astrophysics 0606
Atmospheric Science 0608
Atomic 0748
Electronics and Electricity 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0756
Solid State 0611
Statistics 0463

Applied Sciences

Applied Mechanics 0346
Computer Science 0984

Engineering
General 0537
Aerospace 0538
Agricultural 0539
Automotive 0540
Biomedical 0541
Chemical 0542
Civil 0543
Electronics and Electrical 0544
Heat and Thermodynamics 0348
Hydraulic 0545
Industrial 0546
Marine 0547
Materials Science 0794
Mechanical 0548
Metallurgy 0743
Mining 0551
Nuclear 0552
Packaging 0549
Petroleum 0765
Sanitary and Municipal System Science 0554
Geotechnology 0790
Operations Research 0428
Plastics Technology 0796
Textile Technology 0795

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

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

Layered Fisheye View Techniques for Understanding of Large Software Systems

submitted by Charles Qi Cui, B.C.S.
in partial fulfilment of the requirements for
the degree of Master of Computer Science.

Thesis Supervisor

Chairman,
School of Computer Science

School of Computer Science
Carleton University
March 28, 1993

Abstract

This thesis introduces a layered-fisheye-view technique and presents a prototype system called SnakeTool, which uses this technique to visualize large software systems. SnakeTool represents large software systems at a high level of abstraction, the level of component and timethread. It presents nesting-diagram-like pictures to illustrate the system structure, and uses timethreads to depict the system behaviour. The layered fisheye view technique is developed to address two significant problems in the area of large software system visualization: accommodating of the large amount of information in a limited screen space, and preserving adequate contextual information. The main idea is to nonlinearly distort the views: the interesting and important components are expanded and shown in detail and the less important components are squeezed and shown in less detail to preserve context. This technique is implemented and incorporated in SnakeTool. Experimental results show that the layered fisheye view is effective to the visualization of large hierarchically structured systems with macroscopic execution threads.

Acknowledgment

First, I would like to thank Dr. R.J.A. Buhr, my thesis supervisor, for his valuable advice, timely guidance, and continual encouragement. Without which, it is unlikely that I would have completed this work.

I would also thank Ron Casselman for the many discussions we had, for his good ideas and suggestions throughout my thesis work. Mike Petras of Bell Northern Research followed my work with interest and offered useful insights.

Special thanks to Shenghan, for her understanding, encouragement and support. She was the first reader of each chapter, and made many valuable comments on my writing.

Finally, I would like to thank Carleton University and Bell Northern Research for the financial support they provided.

Table of Contents

Chapter 1	Introduction	5
1.1	Background and Motivation	5
1.2	Objectives	7
1.3	Overview of the thesis work	8
1.4	Contributions	9
1.5	Thesis Outline	10
Chapter 2	Related Work	11
2.1	Program Visualization	11
2.1.1	Static Aspects of Program Visualization	12
2.1.2	Dynamic Aspects of Program Visualization	18
2.1.3	Program Representation	20
2.2	Information Visualization	21
2.3	On the Viewing Techniques	27
2.3.1	Flat Layout	27
2.3.2	Windowing	29
2.3.3	Leveling	30
2.3.4	Distorted Views	30
2.3.5	Three Dimensional Views	33
2.3.6	Animation	33
2.3.7	Sound	34
Chapter 3	SnakeTool	35
3.1	Design Considerations	35
3.2	System Representations	36
3.2.1	Representation of The System Structure	37
3.2.2	Representation of System Behaviour	38
3.3	Layered Fisheye Views: What the User Sees	40
3.4	Interactivity	45
3.5	Presentation of Software Systems: An Example	48
3.6	Summary	55
Chapter 4	Layered Fisheye Views	56
4.1	Generalized Fisheye Views	56
4.2	Layered Fisheye Views	60

4.2.1	Terminology	60
4.2.2	Non-linear Magnification	62
4.2.3	Perspective Projection	64
4.2.4	Locking and the Layered Fisheye View Formation	65
4.2.5	The Algorithm	70
4.2.6	Transformation on Arcs	72
4.3	Discussion	72
4.3.1	Fisheye Views of Hierarchical Structures	73
4.3.2	Multiple Focuses	76
Chapter 5	Implementation	78
5.1	The TimeBench Framework	78
5.1.1	System Structure in TimeBench	78
5.1.2	The TimeBench Viewing Mechanism	81
5.2	Implementing Timethread	82
5.2.1	Logical	82
5.2.2	Graphical	83
5.2.3	Toolcomponent	84
5.3	Applying the Layered Fisheye View Technique to the Box-and-Timethread Diagram	85
5.3.1	Initial Layout	85
5.3.2	Handling the Event	86
5.3.3	Transforming the Coordinates of Nodes and Arcs	87
5.3.4	Updating the View	92
5.4	Some User Interface Issues	93
5.4.1	Response Time	93
5.4.2	Animating the Focus Opening	94
5.4.3	Color Coding	95
Chapter 6	Conclusion	96
6.1	Summary	96
6.2	Conclusions and Future Work	98
Appendix A	TimeBench Class Hierarchy with the Extended Classes of SnakeTool	104

List of Figures

Figure 1	PegeSys	13
Figure 2	The structure presentation of two subsystems of a DBMS	14
Figure 3	Visualization of list data structures in KAESTLE	15
Figure 4	FooScope: a program is described as a network of functions	16
Figure 5	A structure diagram in the PV prototype	17
Figure 6	A visualization of a binary tree in PV prototype	17
Figure 7	FooScope: a snapshot of a program execution	18
Figure 8	KAESTLE: generating a sequence of snapshots of a list	19
Figure 9	Perspective Wall: Visualizing a file structure	23
Figure 10	Cone Trees	23
Figure 11	SemNet: the complete knowledge base of Prolog modules.....	25
Figure 12	SemNet: A fisheye view of a complete knowledge base	26
Figure 13	Tree-Map: Tree-Map with 1000 files	28
Figure 14	A City Map of the United States	32
Figure 15	A graphical fisheye view of Figure 14	32
Figure 16	A nesting-diagram-like depiction of a system structure	37
Figure 17	Timethreads, the system microscopic behaviour	40
Figure 18	The top-level view of a hierarchical structured system.	42
Figure 19	A layered fisheye view	43
Figure 20	A layered fisheye view, one more layer down than the previous figure ...	44
Figure 21	Invoking the Lock function from a box specific menu.....	47
Figure 22	Localizing the view transition	48
Figure 23	The top-level flat view	49
Figure 24	A layered fisheye view	50
Figure 25	A layered fisheye view (one more layer down than Figure 24)	51
Figure 26	Fisheye views with multiple foci	52
Figure 27	Layered fisheye views	53
Figure 28	The process of exploring a target system	54
Figure 29	A flat view of a C program	57
Figure 30	A fisheye view of the C program in Figure 23	58
Figure 31	A conventional tree presentation of a system	61
Figure 32	Layered fisheye view presentation of Figure 31	61
Figure 33	Initial layout	62

Figure 34	Non-linear magnification	63
Figure 35	Two steps of perspective projection	65
Figure 36	Transformation sequence of the layered fisheye view formation	66
Figure 37	Maginification Transformation.....	67
Figure 38	A hierarchical structure with 100 nodes	74
Figure 39	A generalized fisheye view of Figure 38	74
Figure 40	A graphical fisheye view of Figure 38	75
Figure 41	A layered fisheye view of Figure 38.	76
Figure 42	Class hierarchy of the TimeBench graphical component	79
Figure 43	The TimeBench internal representation of a simple system	80

Chapter 1 Introduction

Large software systems are inherently complex. In addition to the difficulties in their design and development, it is also intensely difficult for individual designers to understand their structures and behaviors. Evidence from industry shows that better tools are in need to enhance presentation, learning and understanding of large software systems. Encouraged by many interesting and successful visualization systems in other domains such as scientific visualization, much research has been directed towards developing new techniques for the visualization of large software systems. This thesis introduces a layered-fisheye-view technique and presents a prototype system called SnakeTool, which uses this technique to visualize large software systems.

1.1 Background and Motivation

It has long been argued that computer programs are best comprehended by observing the graphical illustration of their structure and behaviour[4][31]. With the wide use of workstations, program visualization has received considerable interest. Many program visualization systems have been developed to use graphical images to help the understanding of small programs.

Large software system visualization is closely related to program visualization. They share the same objective of using the technology of interactive graphics to enhance the presentation and understanding of computer programs. However, the significant distinction between the amounts of information in their targets requires different approaches in term of presenting computer programs at different levels of abstraction. Unlike program visualization which focuses on the depiction of program execution details, such as code and data structures, large software system visualization concerns how to illustrate software structure and behaviour at a higher level of abstraction.

Another related area is information visualization which uses graphical illustration to help people understand large data collections. In particular, techniques such as 3D display [16][17][35], interactive animation[5][8][27], and distorted views[18][25][35] have been

exploited to visualize large structured information systems, and the results are very interesting. Since large software systems are often viewed as hierarchically structured systems, the existing information visualization techniques would be helpful to the development of new techniques for their visualization. However, these techniques only deal with structural information, not behavioural information, which is equally important to visualizing software. Moreover, they are not adequate in preserving contextual information which always plays a crucial role in the understanding of large software systems.

The understanding of large software systems is a very difficult task due to the inherent complexity of software. Evidence from industry shows that practical visualization techniques and tools are needed to help with the software understanding task. Broadly speaking, the significance of developing new visualization techniques goes well beyond the enhancement of the software understanding. Such techniques would provide substantial benefits in the following areas:

- **Training:** It is especially difficult for designers to understand the design, structure and behaviour of an existing large software system only by textual documents. Informative visual representation of the system can be very helpful in learning.
- **Maintenance:** System maintenance requires the maintainer to interact with various parts of a program. A visualization tool capable of displaying the system structure and behaviour can help them to get a clear picture of the whole system promptly.
- **High Level Debugging:** By comparing the visualized system behaviour with the expected one, developers can quickly locate the errors in the complex assembly of software components of a large program.
- **Documentation:** Usually it is difficult to describe the dynamic, complex behaviour of large programs. Visualization is particularly useful in assisting program documentation with its capability of visually animating the system behaviour.
- **Design:** Designers can use a visualization tool to explore existing systems to understand system organizations, to capture behaviour patterns from different programs, and to gain insight into system design.

Although it is generally recognized that the visualization of large software systems plays an important role during much of the software life cycle, little work has been reported in the literature. The lack of effective techniques for visualizing large software systems is

largely attributed to the following problems:

- *Abstraction Problem:* It is difficult to represent the targets at an appropriate level of abstraction. Large software systems are often composed of many hundreds or thousands of components. On the one hand, it is impossible to simply represent them at a low level of abstraction, as most program visualization systems do; on the other hand, it is difficult to decide upon a representation at a proper higher level of abstraction to meet the different needs of the users.
- *Space Problem:* It is difficult to accommodate the large amount of information in a limited screen space. Usually, the amount of information that must be displayed is too large to fit onto the screen. The existing techniques suffer because that they either hide too much information from the user or they display information in many places, both of which impose a cognitive burden on the user.
- *Context Problem:* It is difficult to preserve adequate contextual information. Due to the screen space limitation, contextual information is often lost when detailed information becomes the user's current interest.

The major effort of this study is to develop techniques to attack these problems. The work was initially motivated by the lack of adequate visualization techniques and the need for practical tools for the visualization of a large software Digital Multiplexed Switching (DMS) from Bell Northern Research Ltd. in Ottawa, Canada.

1.2 Objectives

The primary goal of this study is to *develop effective visualization techniques and to build a prototype system to explore practical ways for the visualization of large software systems*. Specific sub-goals in pursuit of this objective are:

- To represent the system structure and behaviour at a high level of abstraction.
- To develop a layered-fisheye-view technique, based on the fisheye view idea[18], to display large hierarchically structured information.
- To experiment with the technique to compare the effects of different distortions and transformations on the graphical display.

- To develop a prototype system as an experimental vehicle to visualize and explore examples of large software systems.

1.3 Overview of the thesis work

As part of the Carleton/BNR DOORS¹ project, this study was initially motivated by the need for practical visualization tools to help the design and understanding of the large DMS system. The visualization targets of this study are subsystems of DMS. Each subsystem is organized as a hierarchy of software components; the behaviour information is provided in trace files, which are lists of software components in the order of their execution sequence.

Started with the established visual notations for large, real-time software design, Buhr's MachineChart[9], and with the newly developed "timethread" notations from DOORS[10], this study represents the system structure and behaviour at a high level of abstraction. An existing CASE tool, TimeBench[23], provides a framework for the development of the prototype system SnakeTool.

The work in this thesis is centered around two issues: developing new visualization techniques and building a prototype to demonstrate the new ideas. As described earlier in the background, the following three subproblems are of our main concern in developing new techniques: (1) what to display, (2) how to accommodate the large amount of information in a limited screen space, (3) how to preserve contextual information. By using the box-and-timethread notations to build a graphical representation of large software systems, this study extends the "fisheye view" idea of George Furnas[18] and introduces a new technique, the layered fisheye view, to tackle these problems.

The layered fisheye view approach is developed and explored on a concrete application from industry: visualizing subsystems of DMS. The purposes are to help developers understand how a particular program works, how programs in general may be characterized by behaviour patterns. In spite of this specific application, the layered-fisheye-view technique is considered effective to the visualization of any large hierarchically structured systems with macroscopic execution threads.

1. The DOORS (Design of Object Oriented Real-time Systems) is a cooperate research project between Carleton University and Bell Northern Research Ltd. Ottawa, Canada.

One of the significant problems encountered in this work is that there are only very limited theories on the presentation and understanding of large software systems, and very little work with the same concern as ours in the literature. As a result, it is not yet possible to compare our results with others based on an established criterion. Chapter 2 provides a review of the related work we have been able to find, which can be used to evaluate the advantages and drawbacks of our technique.

A lot of the effort of this study is associated with the development of the prototype system SnakeTool. SnakeTool is developed to explore and illustrate the potentials of our technique. It automatically generates graphical layout to depict the structure and behaviour of the target systems. It supports interactive manipulation, smooth transition between views, and fast response time, and provides color-coding of objects to enhance the presentation of the system structure and behaviour.

SnakeTool uses TimeBench as its programming environment. Although it is developed with experiment in mind and used for the purpose of demonstration, its support of Object Oriented Programming (OOP) paradigm allows easy extensions and modifications. Developing the SnakeTool prototype into a practical tool for the visualization of large software systems is an ongoing project of DOORS.

1.4 Contributions

The prototype system SnakeTool presented in this thesis is an early work in the area of large software system visualization. It addresses some fundamental issues in this relatively new area and the experimental results are interesting and promising. This prototype provides a solid starting point for the development of a practical visualization tool.

The layered-fisheye-view approach introduces a new way of visualizing large systems. It provides a solution, to some extent, to the problem of how to display large hierarchically structured information to some detailed level without losing context.

In addition to the primary research, several other contributions have been made to the TimeBench project: 1) new moving operations which drag objects faster, 2) color-coding, and 3) updating the display smoothly. Also, the more detailed literature survey on program

visualization and information visualization is regarded as an contributions to the DOORS project.

At last, since SnakeTool uses visual notations of Buhr's MachineCharts[9]and the timethread notations[10], it can be readily extended to a useful design tool (environment) for software designers to experiment with their design ideas.

1.5 Thesis Outline

Chapter 2 gives an overview of related work in program visualization and information visualization, focusing on techniques of graphical representation and display, and their advantages and drawbacks.

Chapter 3 presents the SnakeTool prototype. It briefly describes the design considerations and the representation techniques used in the SnakeTool. It then presents the major features of the SnakeTool, and describes how it uses the layered fisheye view to illustrate the structure and behaviour of large programs.

Chapter 4 is devoted to a detailed description of the layered-fisheye-view technique. It begins with some interesting examples from Furnas's generalized fisheye view strategy. It then describes the layered fisheye view formation process and gives the algorithm. And it concludes with a comparison between the layered fisheye view and previous fisheye view research work.

Chapter 5 describes some issues related to the SnakeTool prototype implementation. It first briefly presents the TimeBench framework, and then focuses on issues of implementing timethread and applying the layered fisheye view technique to the box-and-timethread diagram. Issues associated with some human-computer interface concerns are also addressed.

Chapter 6 provides conclusions to the study and points out directions of future work.

Chapter 2 Related Work

Software system visualization plays an important role in system development. However, examination of the literature shows that, for visualizing large systems, a lot of work has yet to be done. The major challenge is the difficulty of representing and displaying the structure and behaviour of large systems. Fortunately, there are quite a number of interesting systems developed in some related areas, which provide a solid background for the work in this study. In particular, demonstration tools in the area of program visualization reveal how the structure and behaviour of programs are represented, and new techniques in the area of information visualization show how large amounts of information are displayed. Valuable ideas and techniques have also been found in other fields such as graphics and human-computer interface. All of these have helped us to get a better understanding of issues in large software system visualization.

This chapter provides an overview of the related work. Section 2.1 and Section 2.2 describe some representative systems in program and information visualization. Section 2.3 discusses different viewing techniques, and compares their advantages and drawbacks.

2.1 Program Visualization

Program visualization has almost the same long history as programming in high-level languages. As early as 1959, Haibt presented a demonstration system that could automatically create graphical flowcharts for Fortran language programs[19], and he proposed the goals of program visualization: to help the design, development, understanding, and description of the program; and to gain insight into the characteristics of computer programs. Since then, the term program visualization has been widely used in the literature and has become an active research field. In the literature, program visualization is defined as the use of computer graphics to enhance the art of presenting and communicating programs and thereby to facilitate the understanding and effective use of computer programs by people, where computer graphics encompasses the disciplines of typography, graphic design, animation, cinematography, and the technology of interactive computer graphics[4][31].

Myers[31] classified program visualization systems into four categories using the following two axes: 1) whether they illustrate the code or the data of a program; and 2) whether their displays are static or dynamic. As program details such as code and data are not of the main concern of this study, in the following discussion we focus on the static or dynamic aspects of the existing systems. Also we emphasize the graphical representations of the program structure and behaviour.

2.1.1 Static Aspects of Program Visualization

Many program visualization systems have static aspects, that is, the pictures generated are snapshots of the program at certain points. Although there are some pure static systems [4][19][28], recent systems usually have both static and dynamic features so that program structure and behaviour can be fully explored[5][6][8].

The earliest attempt to help the understanding of program by visualization is the Flowcharter written by Haibt[19]. In order to provide both a good general picture and detailed descriptions for a moderate size program, the Flowcharter automatically generated a series of flow charts on different levels of detail; each part of a chart is shown in more detail on a succeeding chart. Furthermore, a summary of the functionality of each part was presented on each box. The down side of this system, as of some similar systems later, is the use of the flowchart to represent the program. It has been argued that the flowchart is a very poor abstraction of the software structure and useless for any serious applications[7].

The PegaSys system of Moriconi and Hare[28] is another static system developed to facilitate software documentation and to help the design. In PegaSys, the program is illustrated at a high level of abstraction: entities such as subprograms, processes, modules, and data objects are represented by graphical icons; the program is described by a hierarchy of interrelated pictures. These pictures help the hierarchical decomposition of the application during the design phase. In addition, they are the formal documentations of the program. Contrary to systems that automatically generate graphical presentations of the program, pictures in PegaSys are drawn by users and are checked by the system to ensure that they are syntactically meaningful and consistent with the program.

Figure 1 depicts a snapshot during the development of a broadcast network using PegaSys. The first level in the network documentation hierarchy is shown in the upper

window, and the second level (after refinement) is displayed in the lower window. In the pictures, ellipses are used for representing process; dashed rectangles for modules; solid boxes for functions; and arcs and arrows for interactions.

Figure 1 PegaSys[28]: the first and second levels in a broadcast network documentation hierarchy.

The Rigi system of Muller and Klashinsky[29] was developed to represent and manipulate the building blocks of large software systems and their myriad dependencies. Its graph editor can generate the system description documents in aggregation or generalization hierarchies based on their component dependencies. It also allow designers to organize development information by manipulating system description documents and

by establishing links among those documents. The illustrated component documentation dependency can be used as a common thread and as a medium for communication and coordination by all members of the project, thereby aiding the designers and maintainers in manipulating and understanding the structure of large, integrated software systems.

Figure 2 shows a system structure presentation generated by the Rigi editor, which contains two subsystems of *searching* and *sorting* of a DBMS. The sys icon on the top level represents the entire DBMS. Its sorting and searching subsystems are presented by other two sys icons. The sorting subsystem contains two modules *array* and *list*, which implement sorting algorithms for arrays and linked lists. Each module consists of a definition and implementation part. The implementation part of the sorting module, which operates on array data structures, consists of three possible implementations: *smooth*, *quick*, and *heap*.

Figure 2 The structure presentation of two subsystems of a DBMS: searching and sorting.

Boecker and Nieper developed a system called KAESTLE, with both static and dynamic features, to visualize Lisp programs[6]. The system can automatically generate the

graphical representation of a list structure. As with PegaSys, KAESTLE is also integrated in a window system so that the structure hierarchy can be displayed in multiple views. Furthermore, all graphical objects can be manipulated by the user. Figure 3 shows the results of some list data structures where rowed boxes represent lists, and straight and curved lines with arrowheads represent links between lists.

Figure 3 Visualization of list data structures in KAESTLE[6]

KAESTLE can be used for designing, debugging and documenting Lisp programs. It is considered to be, to some extent, a useful tool for the Lisp programmers to understand certain aspects of the programming language and to explore data structures which are difficult to represent textually.

In FooScape[5], another program visualization system of Boecker and Nieper, a program is represented as a network of functions which call each other. The system can

automatically create a flat layout of the network of functions to help the user to get a first, rough overview of some part of the software. Figure 4 shows the graphical representation of such a network in which ellipses represent functions and arrows represent connections between functions.

Figure 4 : FooScape[5]: a program is described as a network of functions

Various program visualization tools have been used at each stage of the software life cycle. To integrate the separate phases of those tools within a unified framework, Brown et. al. [8] have developed the PV prototype. The major static aspects of the PV prototype are almost the same as those in the above systems, e.g. viewing structure diagrams and viewing data, but they are implemented differently.

Figure 5 is a structure diagram drawn by the user on the PV prototype. It is one of a set of diagrams that are linked hierarchically. The set of diagrams together represent a system architecture while each diagram represents one module of the system. In this static illustration, boxes represent modules and lines and arcs represent connections, and the user can zoom into a box to see its details. Two views are used to provide the detailed and the contextual information: details are displayed in the main window; the local context (one level above the current level of interest) is presented in the miniaturized view, where a “you are here” box indicates the correspondence.

Figure 5 A structure diagram in the PV prototype[8]: a main view for details and a miniaturized view for local context.

Figure 6 A visualization of a binary tree in PV prototype[8].

A different viewing technique is used in Figure 6 to visualize the data structure of a binary tree: the view in the upper window displays the entire tree structure while the detailed view in the lower window illustrates the highlighted portion of the upper window. The graphical representation is generated automatically. As the user selects different parts

for details, the display in the lower window is updated dynamically.

2.1.2 Dynamic Aspects of Program Visualization

“Dynamic” in program visualization refers to the animation of dynamic program execution. There are two kinds of program animation: run-time animation which happens during program execution, and animation by trace information which happens after program execution. Run-time animation is often used to depict the dynamic behaviour of small programs, while animation by trace information is mostly used to monitor large programs. This section describes the dynamic aspects of program visualization.

FooScape[5] produces run-time animation to depict the program behaviour. Figure 7 shows that the static rendering of the function network in Figure 4 is animated by highlighting functions when they become active. Users can tell from the pattern of highlights what the system is doing. Another interesting aspect is that sound is used to generate auditory effects: specific tones are played before and after a function is executed.

Figure 7 FooScape[5]: a snapshot during program execution, functions are highlighted when executed.

The animation in KEASTLE[6] focuses on the change of data rather than the execution thread. It generates a sequence of snapshots for the data structure of interest when the program proceeds. Figure 8 gives an example that illustrates the data update of a list.

Figure 8 KEASTLE[6]: generating a sequence of snapshots of a list

The PV prototype [8] supports animation of both control flow and data updates. The illustration of common data structures, such as arrays, linked lists, and trees, is updated in real-time to reflect changes in data; source code is highlighted line by line to indicate the progress of the program execution. To achieve a dynamic animation, the user can increase or decrease the speed of update by using the speed control facility.

A common characteristic of the above examples is that the user specifies program objects (the code or data structures) of interest to be visualized, then the objects are allocated some display area and automatically updated as the program runs, until the user shifts his focus and specifies some other objects. However, it is not always obvious to the user what objects to display for a large program. A different approach is taken by PROVIDE[27], which automatically updates display to illustrate the change in program objects at run-time. In PROVIDE, program depiction is constructed by binding program objects to predefined icons. When the values of program objects alter, or when new program objects become active, they "drive" their bound icons to be updated.

2.1.3 Program Representation

We have seen from above examples that different systems represent programs differently. To represent the program at a proper level of abstraction is one of the most important issues in the development of a program visualization system, and perhaps the first thing to consider in the design. This section summarizes the common strategies used for the representation of program structure and behaviour.

Many systems represent the program at the level of program statement and common data structures[1][4][6][19][27][30][33]. The advantage of this strategy is that the graphical representations are closely coupled with program units so the user can study the details of a program. Since any change or update of graphical display is derived from the change of data or the proceeding of code, the user can see animated pictures of the complex data structures which are difficult to explain otherwise (e.g. the reentrant structure in Lisp, the inheritance in Smalltalk); and he or she can visually follow the flow of control as the program is executed. However, this strategy has a number of significant shortcomings. First, it requires the user to work at a fixed, low level of abstraction with respect to the visual representation of basic program units. Second, the exploration or debugging of a program at such a low level can be tedious and time-consuming if the user doesn't specify his or her parts of interest, which are not always easy to specify in the first place. Third, visualization of program structure and behaviour usually requires different views to show different aspects of the program, since an in-depth understanding of a complex program is hard to achieve by a single low level representation. However, shifting back and forth between views would impose a cognitive task on the user. Finally, the lack of high-level abstraction leads to messy diagrams, making the overall structure and the general behaviour difficult to extract.

Some systems extend the "node-and-arc" techniques to represent the overall structure of large programs. For example, by representing large software at a high level, the subprogram-and-module level of abstraction, PegaSys[28] attempted to capture the salient aspects of the design of a program, rather than the details of particular algorithms or data structures. The disadvantage of this strategy is that it fails to provide support for programming-in-the-small. Additional mechanisms would be needed to facilitate the mapping between the representation and the program code or data. Besides, there is no representation of software behaviour at the higher level of abstraction. The animation

technique in FooScope (described in section 2.1.2) could be borrowed here to represent the run-time behaviour at a coarse granularity, but it would help little for the insight of large software design and development. Recently, a new notation called "timethreads" was developed by Buhr and Casselman[10] to facilitate large software system design. Conceptually, timethread expresses end-to-end behaviour that occurs in response to stimuli during the execution of large software systems, and it is visually represented as curves traced across pictures of the system structure. It has been argued that timethreads can help to capture the key patterns that are crucial to the correct and efficient behaviour of large software systems (more description in section 3.2.2).

Designers think about the system structure and data at many levels of abstraction during the design phase; program visualization should reflect this fact. Some systems provide multiple levels of abstraction[8][27], ranging from top-level pictures of the whole system to bottom-level depictions of language code. In PV prototype[8], the level of abstraction displayed can be chosen by the programmer. In PROVIDE[27], an interactive control over the program granularity is provided so that individual users can observe and control program execution at a level of abstraction tailored to their needs. Actually, the idea of providing pictures of multiple levels of abstraction can be found in the earliest systems. For example, The Flowcharter[19] outputs flow charts of different levels of detail, although they give the same view of the same program. The main concern with this strategy is that sophisticated viewing techniques are needed so that the user is unlikely to get lost during the transition between views of different granularity.

2.2 Information Visualization

Recently, some interesting visualization techniques have been developed to enhance the understanding of large information systems such as database, knowledge base, and file management systems[13][16][21][25][35]. This has helped to create a new area of research, information visualization. From the existing work, information visualization is regarded as the use of computer graphics to help people to understand large data collections, and to stimulate cognitive recognition of patterns in the data, but no formal definition has been established.

Because the volume of information is huge, to facilitate the management, large amounts

of information may be organized into some specific structural form, e.g. linear structure or hierarchical structure, and we say these kinds of information are well-structured. However, some information is too general and complex to fit into a specific structure, and we say it is ill-structured. Needless to say, it is more difficult to develop systems supporting the visualization of ill-structured information. However, no matter what kind of information they support, all information visualization systems share two significant problems: 1)the difficulty of presenting and accommodating the large amount of information that must be displayed on the limited screen space, and 2)the difficulty of preserving global context when examining local details. This section describes two representative systems and presents how the problems have been tackled.

A natural way to cope with the large amounts of information in visualization is to maximize the effective use of the available screen space. To pursue this objective, Card et. al. developed a prototype system, called the Information Visualizer[13]. The Information Visualizer extends the display area from the 2D screen to a 3D virtual space. By using recent technology in 3D graphics such as 3D animation and object manipulation, Card et. al.[25][35]developed two interesting viewing techniques, Perspective Walls and Cone Trees.

The Perspective Wall[25] is developed for the visualization of linearly structured information. It folds a wide 2D layout of linear information into three segments of “walls” in 3D space, with the center panel for detail and the perspective panels for context. The perspective panels recede and are shaded to enhance the perception of 3D. In addition, the vertical dimension of the wall can be used to visualize layering in an information structure. The idea behind the Perspective Wall is to support efficient use of display space by distorting wide 2D layout of linear information into 3D visualization.

Another interesting aspect of the Perspective Wall is that it uses hardware support to provide interactive animation: when an item is selected, it is moved to the center panel with a smooth animation. This animation helps the user perceive a clear structure change and ameliorate the distracting effect during the transition of views.

The intuitive distortion of the layout of the Perspective Wall provides efficient space utilization and allows smooth transitions of views. Besides, the perspective view has the further advantage of making the neighborhood of the detailed view larger than more distant

parts of the contextual view, that provides a fisheye view effect (more discussion in section 2.3.4 and section 4.1).

Figure 9 Perspective Wall[25]: Visualizing a file structure, the horizontal dimension for modification date, and the vertical dimension for file type.

Figure 10 Cone Trees[35]

Also using 3D graphical techniques, the Cone Tree[35] supports the visualization of hierarchically structured information. In the Cone Tree, hierarchical structures are represented as 3D trees. The nodes in the hierarchical structure are drawn in layers. Each node, except those on the bottom level of the structure, is the apex of a cone, and the children of the node are evenly placed around the rim of the cone. Cone base diameters for each level are reduced progressively, which insures that the bottom layer fits in the width of the screen. The body of each cone is shaded transparently, so that every cone can be perceived.

Interactive animation is used to help the smooth display transition. When a node is selected, the tree rotates so that the selected node and each node in the path from the selected node up to the top are brought to the front and highlighted. The rotations of each substructure are done in parallel, and are animated so the user sees the change at a rate the perceptual system can track. The animation helps the user to understand the information structure more completely. And by observing continuous rotations of the structure, the user gains insights into the relationships between substructures. Furthermore, tree operations are provided so that users can trim and reconstruct the Cone Tree.

The Cone Tree provides a way to see the whole structure of large hierarchical information. Using their prototype, the authors presented a whole UNIX file structure on the screen, which includes about 10,000 files. But for a clear presentation, there is a limit, about 1000 nodes, to the size of the hierarchy. It is also observed that the Cone Trees are not effective for balanced hierarchies.

The SemNet system developed by Fairchild et al[16] is an example of the attempts to visualize systems of ill-structured information. The target is a knowledge base of Prolog modules. In the system, the knowledge base is represented as a 3D directed graph, elements of the knowledge base are represented as labeled rectangles, and different kinds of relationships between the elements are represented as arcs with different color.

Compared with the Information Visualizer, SemNet has a more difficult task in presenting the knowledge base since it is ill-structured. Figure 11 shows that a direct display helps little to the understanding of the knowledge base. In an attempt to organize the ill-structured information, SemNet introduces new cluster-objects: every element is assigned to a unique set represented by a cluster-object, and each set can be assigned to a

superset represented by another kind of cluster-object, and so on. Thus, the result is a hierarchical structure, with knowledge-base elements at the bottom and with successive layers of cluster-objects above the objects.

Figure 11 SemNet[16]: the complete knowledge base of Prolog modules with nodes assigned to random positions (the original picture is in color).

By representing the knowledge base structure as a cluster hierarchy, SemNet is able to support a more sophisticated viewing strategy called the fisheye view[18]. Simply put, the fisheye view presents information of an interesting point in great detail and that of a less interesting point in less detail, which usually provides a useful balance of local details and surrounding context (more discussion in section 2.3.4 and section 4.1). Since successively higher layers in SemNet's structure representation have fewer and fewer objects in them, the fisheye strategy is implemented by displaying the elements of an interested area(focus) but only cluster-objects around the focus. As distance from the focus increases, the displayed cluster-objects correspond to higher levels of the hierarchy. Thus the user can examine knowledge elements in the interested area without losing the global context represented by cluster-objects.

Figure 12 SemNet[16]: A fisheye view of the complete knowledge base of Prolog modules

Figure 12 shows the resulting fisheye view of the knowledge base displayed in Figure 11. What elements of the knowledge base to display depends on the current viewpoint, which is controlled by the user. And the knowledge base is explored by traveling through the 3D space. The fisheye view technique of representing the more remote regions in correspondingly larger chunks essentially reduces the number of objects to be displayed on the screen, yet preserves a balance of local detail and global context.

SemNet is an early attempt to visualize large ill-structured information. Although it addresses the fundamental issue of the representation of ill-structured information, it suffers from some significant drawbacks. First, in SemNet, euclidean neighborhoods have been used to assign elements to set, which means the positions of the knowledge elements greatly affect comprehension of the knowledge base structure. In order for any spatial subdivisions to be meaningful, elements must first be placed in the space so that nearby elements are closely related. Otherwise the content of the spatially defined clusters will be essentially random, and the resulting metastructure will not capture the implicit structure. Although different subdivision schemes are proposed in SemNet, none of them is effective. Second, to explore graphical representations of the knowledge base, the user must be able

to move the viewpoint. But the resulting changes in orientation in the virtual 3D space are confusing and less meaningful. Consequently, users quickly become lost and disoriented. Finally, the abrupt transitions between different views force the user to mentally integrate the information structure, and the cognitive task of understanding the structure is still quite difficult.

2.3 On the Viewing Techniques

In the above representative systems, a number of viewing techniques have been proposed to tackle the issue of displaying large amounts of information on a small screen space. They can be roughly divided into seven categories: (1) flat layout, (2) windowing, (3) leveling, (4) distorted views, (5) 3D, (6) animation, and (7) sound. Since the displaying issue is of the major concern of this study, in this section we further discuss these viewing techniques, and their advantages and drawbacks.

2.3.1 Flat Layout

Flat layout of all information is a conventional viewing technique. Since it is easy to implement, it became the major technique of the early attempts in program visualization [19][34] [38]. Flat layout shows the target in one picture, which makes the cognitive task much easier for the user, therefore, it remains the most widely-used viewing technique.

The major problem of flat layout is that it is difficult or impossible to display all objects of large information on a small screen space because the objects would be too small to be seen. To cope with this problem, two variants of flat layout have been developed.

One variant, a common technique nowadays, is to layout all objects in a single large virtual workspace much larger than the screen, instead of forcing them to fit on the screen. The users can browse the display by scrolling. And the screen is simply treated as a movable viewport on this workspace. Sketchpad[38] is one of the earliest graphical programs that uses this technique.

Another variant is to present part of the information implicitly by using some conventional diagrams. A good example is the Tree-Map technique developed by Johnson

and Shneiderman[22]. This technique partitions the display space into a collection of rectangular bounding boxes representing the tree structure. The drawing of nodes within their bounding boxes depends on the context of the nodes, and can be interactively controlled. The Tree-Map maps hierarchical information onto a rectangular region in a space-filling manner, making 100% use of the available display space. The efficient use of space allows flat layout of very large hierarchies in their entirety.

Interactive control allows the user to determine how the tree is displayed. Sections of the hierarchy containing information of current interest can be allocated more display space while sections which are less interested can be allocated less space. Figure 13 shows a Tree-Map of 1000 files.

Figure 13 Tree-Map[10]: Tree-Map with 1000 files.

The Tree-Map is best suited to hierarchies in which the leaf nodes are of primary importance. However, since the view of nodes above the bottom level of a hierarchy are blocked by the leaf nodes, the user can't see an explicit structure and is forced to integrate the hierarchy mentally.

To summarize, flat layout of all information usually shows the global structure. It is suitable for displaying small systems and easy to implement. Its shortcomings are: 1) for large systems, details are too small to be seen, 2) an object can only have a limited number of neighboring objects because of the space constraint, 3) scrolling obscures the global structure, and 4) navigation by a big picture is hard and the user is easily lost.

2.3.2 Windowing

With the progress of hardware, windowing became a popular technique in the early 80's. The major advantage of a windowing system is that it extends the screen space virtually, and provides multiple views so that the user can see different parts of the information at the same time.

Typically there are two implementations of the windowing strategy, overlapping and tiled windows. Overlapping windows are allowed to cover each other. The user can create as many windows as needed and can manage a window's location and size in any way desired. Tiled windows are not overlapped. When a new window is created, other opened windows resize automatically to make space for the new one. Therefore, the user always sees the contents of all opened windows.

Windowing strategy is widely used in program visualization, in which different windows display different parts of a program, or provide different views. Rigi [29], KAESTLE[6] (see Figure 3 and Figure 8), PECAN[33], PROVIDE[27] are examples using overlapping windows; PegaSys[28](see Figure 1) and Cedar[39] are examples using tiled windows.

Window systems also provide some support for processing large amounts of information. For example, related information can be grouped in one window; contextual information can be provided by arranging the associated windows close together[20].

Windowing provides a solution to the issue of displaying both the local details and the global structure. However, in big applications, windows often obscure each other when there is a need to open many of them, e.g., when zooming down several levels for details or when viewing multiple parts of interest. This results in "window thrashing", in which the user has to move and reshape windows continuously to keep the desired views visible. Tiled windows don't obscure each other. But windows would show only little of their contents if there are many of them, which limits the number of windows that may be opened simultaneously.

A common technique (a special case of tiled windows) for integrating the detail and the context is to have two simultaneous views: an overview displaying the global structure or context, and a detailed view displaying the part of current interest, e.g., the PV prototype[8] (Figure 5 and Figure 6). The main advantage of using two views is that it is simple, and easy to understand and manipulate. The problem is that it doesn't convey enough information in the visualization of large programs.

2.3.3 Leveling

Leveling can be considered as a companion technique with windowing. When the user descends into an object, a new window is opened to display the information inside the object; as the user selects to get into deeper levels, more windows are opened and more detail is revealed. "One-level-down" always shows the local detail clearly and gives the user what he wants. However, for large amounts of information, the problems are: 1) the user may not know which objects to explore, which makes leveling and windowing inferior to flat layout, 2) it's difficult for the user to integrate an overall structure, 3) the user may easily get lost during the busy level-changing. All of these would impose a difficult cognitive task on the user.

2.3.4 Distorted Views

Generally speaking, a distorted view is a new presentation of the visualization target by distorting its original appearance. Since by distortion, some parts of a picture can be enlarged while other parts can be squeezed, a number of techniques[9][15][20][22][40][41] that provide distorted views have been developed to attack the issue of displaying and

integrating the global structure and the local details.

Perhaps the earliest technique of distorted views is the use of icons: a window shrinks to a small icon, which reminds the user of the original window and makes more space for other windows.

The Boxer system developed by DiSessa[15] uses spatial boxes to represent many semantic notions of containment. The boxes are nested in a hierarchy. Individual boxes can either appear expanded or shrunk to a symbol, depending on where the user is in the hierarchy.

The Bifocal Display of Spence and Apperley[37] creates a different kind of distorted-view. The technique was developed for visualizing linearly structured information. Their 2D display is a combination of a detailed view and two distorted views, where items on either side of the detailed view are distorted horizontally into narrow vertical strips. For example, the detailed view might contain a page from a journal and the distorted view might contain the years of publication for various issues of the journal. Based on this idea, Mackinlay et. al developed the Perspective Wall[25] (see Figure 9) by extending the 2D static display to 3D dynamic visualization.

Taking a different approach, George Furnas presented the fisheye view techniques in his pioneering work[18](more discussion in Section 4.1). The idea is to force all of the objects to fit on the screen space by allocating space to objects on the basis of their intrinsic importance and the user's current focus of attention. Objects which are more important and closer to the focus get more space whereas those far and less important get less space.

Some other works after Furnas' provide fisheye view effects[16][25][35][36]. Among them, an interesting graphical interpretation of fisheye view is given by Sarkar and Marc[36]. Figure 15 shows their fisheye view layout of a U.S. city map.

A fisheye view provides a balance of the interested details and the global structure. It has the advantage of presenting details without loss of context. It is more suitable for visualizing systems with large information. Also it can be used as a navigation map. The problems with fisheye views are: displays are perceived unnaturally after the transformation; the view transitions are not smooth unless animation is provided.

Figure 14 A City Map of the United States[36]

Figure 15 A graphical fisheye view of Figure 14. The focus is at St. Liouis[36]

2.3.5 Three Dimensional Views

In recent years, hardware development in 3D graphics has made 3D visualization a very popular topic. There are a number of practical 3D visualization tools that facilitate the discoveries of new molecular structures, the modelling of fluid field, and the design of new automobiles. The success of 3D visualization in other areas encourages research that uses 3D techniques to visualize large software and information systems.

3D space is a natural extension of the 2D screen. Some interesting systems have been developed in the attempts to layout flatly the large amount of information in 3D space (e.g the SemNet[35], the Perspective Wall[25], and the Cone Tree[16] described in section 2.2). At the extreme of this extension is the NASA helmet[17], which monitors the user's head and body movements to put the user in a simulated 3D space called virtual reality.

3D space provides a much larger room for displaying information. In addition, it provides a balance of the local details and the global structure, which arises automatically since in three dimensions a few nearby points loom larger and those further away appear smaller and smaller. "3D" seems to be the solution to the problem of visualizing large software and information systems, given that we simply can not put the huge amount of information into a very limited 2D screen. But how to represent software structure and behaviour in 3D and how to present software system in 3D are still open problems.

2.3.6 Animation

Animation provides an effective way to depict the software behaviour. Common techniques are to highlight the source code statement by statement to indicate the progress of program execution[8][33], or to update data or code at certain places[6][8]. These text-oriented animated display techniques are easier to implement, but they can represent only the lowest level of behaviour abstraction.

Complex techniques of graphical animation are more flexible at depicting the program behaviour at different levels of abstraction. One example is the FooScap[16](described in section 2.1.2), in which the function call paths are animated by highlighting the functions when they are active. Another example is the PROVIDE[27], in which programs are depicted by parameterized icons, and the icons are dynamically updated with the change of

their underlying variables.

In addition to illustrating the program behaviour, animation helps the smooth transition among views, which would otherwise force the user to integrate different views mentally.

2.3.7 Sound

Although viewing techniques generally refer to the display effect, it is observed that sound is more capable of representing the software behavior in some circumstances. For examples, in the FooScape[5](described in section 2.1.2), sounds played can familiarize the user to the behaviour patterns of the program; in the IC system, a program monitor of Cameron and Cohen[12], bells and whistles are used to alarm exceptional conditions.

Chapter 3 SnakeTool

The approach taken in this study for the visualization of large software systems is to develop an experimental system and to integrate new visualization techniques in the system. This chapter presents SnakeTool, our experimental vehicle for large program visualization. The first two sections briefly describe some design considerations and the representation techniques used in the SnakeTool. The remaining sections present the major features of the SnakeTool, and describe how it uses the layered fisheye view to illustrate the structure and behaviour of large programs.

3.1 Design Considerations

SnakeTool is built as an experimental system to explore practical techniques for the visualization of large software systems. The main design considerations are:

The representation: Each of our target systems usually comprises a large number of software components. To display such a system, much of the program detail such as code and data has to be hidden, because even the best display design would not be able to cope with the hundreds and thousands of individual program symbols. SnakeTool uses the box-and-timethread notations to represent the target system at a high level of abstraction, which helps to reduce the amount of information to be displayed while preserving the significant characteristics of the structure and behaviour of the target.

The programming environment: The current SnakeTool is developed for experimentation, with the primary goal of helping to explore viewing techniques and rapidly achieve good results. In this regard, a framework supporting window systems and advanced graphical operations such as direct object creation and manipulation is preferred to facilitate a fast prototyping. In addition, SnakeTool should be an experimental environment itself to allow future extension and modification. Thus, we chose TimeBench, which is an extensible object-oriented framework as the development environment.

The Main View: The primary view SnakeTool provides is the layered fisheye view.

which manages both the detailed and the contextual information in a single display. Target systems are explored mainly from this viewpoint. Another viewpoint it offers is the successive more detailed views, which is based on TimeBench leveling facility. Furthermore, these two features can be combined together to generate more effective results.

Interactivity: SnakeTool supports interactive control over the graphical display, which is one of the most important mechanisms for any visualization tools. This facilitates the display layout and the target system exploration. It also helps the rapid extraction of information with low perceptual and cognitive loads.

Response time: Due to the large number of graphical entities in the display, view transitions are normally slow, and achieving a fast response is a major challenge. SnakeTool tackles this problem by using a lock mechanism to localize the view update.

Multiple Views: A visualization tool should be able to present multiple views of the target so that different aspects of the program can be illustrated. This is also what SnakeTool should achieve. In principle, the SnakeTool design should allow flexible configuration so that new views could be added and different views could be combined.

3.2 System Representations

As discussed in section 2.3, we know that the major problem of representing large software systems is the difficulty of choosing or developing an abstraction at an appropriate level. This is because a high-level abstraction may sacrifice some information that is useful whereas a low-level one may cause the loss of the macroscopic features of the target systems in the large amount of details.

Most program visualization tools are primarily concerned with code and data. However, for a large system consisting of hundreds and thousands of lines and individual variables, it is difficult for the user to get an overview of the general patterns about the system structure and behaviour by monitoring the data change and code execution. Also it would be impossible for the user to extract the behaviour of large programs from a sea of graphical objects. In SnakeTool, we chose to represent the structure and behaviour of our

targets at a high level of abstraction, the level of component-and-timethread, which captures the significant characteristics of large programs and which suits our perceptual abilities.

3.2.1 Representation of The System Structure

Software is hierarchically structured in the sense that it is generally divided into a hierarchy of components (modules) in which each component has its sub-components. Thus a program structure is often depicted in a tree form, where each node of the tree represents a component, and the parent-child relationship in the tree corresponds the component-subcomponent relationship in the software structure hierarchy. Tree representation is a popular method of depicting large software systems, and a number of diagrams have been invented to provide graphical presentation of such trees[26]. The system structure illustration in SnakeTool is very similar to one of these diagrams, the nesting diagram.

Figure 16 A nesting-diagram-like depiction of a system structure, boxes represent components, buttons represent interface, and arcs represents possible call path; "nesting" implies the system hierarchy.

Figure 16 is a nesting-diagram-like structure depiction generated in SnakeTool. Each box represents a component or subsystem of the target. Some boxes have small boxes inside, which represent their sub-components. When the subcomponents are visible, we say these boxes are open. Other boxes present empty faces (except the names): some of them may not have further sub-components, some of them may have their children hidden; for the latter ones, we say they are closed.

The significant aspect of the above structure depiction is that “nesting” provides containment which agrees with the decomposition of the target structure, and that it implies the system hierarchy. More important, it matches the way designers perceive the system in operation as communicating black-boxes. Graphically, this kind of structure depictions offers such advantages that it makes better use of the screen space than direct tree illustration and that each component’s detail information is localized in it.

SnakeTool is able to present more structure information than containment. For example, buttons and fingers can be attached to the periphery of boxes to provide information about visiting paths between components. This is because the visual elements of SnakeTool are based on Buhr’s MachineChart notations[9], which are richer and more meaningful for program structure illustration than the nesting diagram.

3.2.2 Representation of System Behaviour

Since large software system often exhibits very rich and complex behaviour, it is almost impossible to illustrate such behaviour without information reduction. In SnakeTool, the system behaviour is extracted and represented by timethreads, which capture the salient behaviour patterns by omitting most of the detailed information of program execution.

3.2.2.1 Trace Information

During the running of our target system, the components being executed are recorded (by their names) as time proceeds. At the end of the program run, this information is saved out to a trace file, in which the component names are listed in the order of their activation sequence, and further called components are indented accordingly. Such information implies “timethreads”, the microscopic behaviour of the system, which exhibits the key

patterns of the system execution. SnakeTool uses such information to create its display of the system behaviour.

3.2.2.2 Timethreads

Timethreads are used in SnakeTool to illustrate the behavior of large programs. Conceptually, timethreads express cause-effect sequences during the system execution. A timethread is triggered at some point of stimulus, goes through components of the system in the order in which they are activated when the system runs, and ends at some completion of processing or delivery of response[10][11].

The appearance of a timethread is a curve winding its way through the structure of a large software system, like a snake (thus the name of our tool). Figure 17 gives an example in which timethreads are traced across the system structure shown in Figure 16. In the figure, the three timethreads came from three separate executions of the system, and they illustrate different sequences of component activation.

Timethreads represent the system behaviour at a higher level of abstraction. It depicts the course of program proceeding at the subsystem level. Timethreads omit those local behaviors such as the call and return paths between functions. Thus they are more abstract than threads of control, making them more effective for presenting and understanding macroscopic behaviour of large programs.

Timethreads exhibit system behaviours which help the designer to gain insight into system design. For example, the three different timethreads shown in Figure 17 present different behaviors under different circumstances. In addition, they all touch some common components, e.g. the biggest one Trunk_Starter, which could most likely be regarded as a "hot spot" of the system. For a more detailed description and discussion of timethreads the reader is referred to [10][11].

Timethread express causality flow at a high level of abstraction, this may impose a constraint on the behaviour illustration. But in its present incarnation, SnakeTool emphasizes how to depict the macroscopic behaviour patterns of a large software, not the fine details. We think that a combination of timethreads and other kinds of established low-level behaviour representation techniques would meet the different needs of SnakeTool

users. Adding other behaviour views is one of the objectives of the ongoing development of SnakeTool.

Figure 17 Timethreads, the system microscopic behaviour, “snakes” wind through boxes following the sequence of activation of components.

3.3 Layered Fisheye Views: What the User Sees

SnakeTool displays the system structure with nesting-diagram-like pictures and the system behaviour with timethreads. The major technique of viewing the structure picture and the timethreads is the layered fisheye view.

The layered fisheye view presents the structure information of the viewer’s interest in great detail and the surrounding context in less detail. It also provides more depiction on the behaviour of the viewer’s interested components than on that of the less-interested ones.

The layered fisheye view gives the user the control over what details to display. The internal representation of a target system in SnakeTool is a hierarchy of component objects(boxes). Normally, any box at an intermediate level of the hierarchy hides its

children. The layered fisheye view technique takes this advantage to reduce information to be displayed on the screen. The user controls which box to open to see details inside and which box to close to save screen space.

The layered fisheye view demonstrates a special (layered) fisheye view effect, which comes from its integration of a series of fisheye views generated at different levels. Before any integrating process, a flat layout is displayed, where the subsystems at the top-level of the target hierarchy are shown and everything underneath is hidden (see Figure 18). When the user selects a component, the immediate children of this component are added to the previous view, and the new view shows details one level down (see Figure 19 and Figure 20). This “drilling” process can continue until the components at the lowest level of the hierarchy are revealed. While seeing more and more about what is inside the components of his interest, the user also sees the timethreads going through more components, which illustrates more about the system behaviour. The most significant aspect during this process is that every updated view is a fisheye view, which establishes a useful balance of local detail and global structure by presenting the information of the viewer’s interest at the level of detail as desired and the contextual information in less detail (smaller size or hidden).

Figure 18, 19 and 20 are screen dumps from SnakeTool (original pictures are in color), illustrating the above process. In Figure 18, a visualization target consisting of 19 components at the top level of its hierarchy are displayed, with a single timethread going through all of them. Figure 19 shows a fisheye view that exposes component c_1 , c_2 , and c_3 , the immediate children of b_6 , when the component b_6 is selected as the viewer’s current interest. Figure 20 shows a further (one more layer down) fisheye view after the selection of c_2 as the immediate interested component.

To enhance the presentation, color coding is used when generating the fisheye views. Boxes at different levels of the structure hierarchy are colored differently; closed box has a solid color face whereas an open one has a thick color outline.

Figure 18 The top-level view of a hierarchical structured system.

Figure 19 A layered fisheye view with b_6 as the current interested component, adding components c_1 , c_2 , and c_3 , the immediate children of b_6 , into the display.

Figure 20 A layered fisheye view, one more layer down than the previous figure, with c_2 as the current interested component, adding d_1 , d_2 , d_3 , and d_4 , the immediate children of c_2 in the display.

If we call the component (box) selected by the user, the *focus*, the view integration is actually a process of recursively applying the same view transition technique to previously generated fisheye views whenever new focus is selected. For example, in Figure 19, when a focus b_6 is selected, it opens and occupies more screen space so that its children can be displayed. At the same time, other neighboring boxes get smaller and are pushed away from b_6 to make room for displaying detailed information. In Figure 20, when c_2 , a child of b_6 , is selected, it expands to make space for its children, which in turn makes b_6 even larger and b_6 's neighbors even smaller. The size and position of a box in the display depend on the current focus, the focus selection history, and at what level it is in the system hierarchy.

Basically, the more detail is revealed, the smaller the boxes at higher levels become. Therefore what we see are layered fisheye views. Chapter 4 contains a more complete discussion of the layer-fisheye-view technique and an evaluation of its advantages and drawbacks.

In addition to the layered fisheye views, other viewpoints can be added to meet different needs of different users. Another viewpoint supported in SnakeTool is the detailed views at different levels, which is based on the TimeBench leveling facility. The detailed views make it possible to visualize the system structure and behaviour at a certain level lower than that of software components and timethreads. Moreover, by combining the layered fisheye view and zooming, the user has the control over where to begin seeing layered fisheye views rather than right from the top level, which is sometimes unnecessary and which may make the display clustered when there are too many graphical objects to fit onto the screen.

Other views can be readily supported in SnakeTool since the system is designed to allow easy extension. For example, one view could be a display of the system structure tree, or a subtree, which would explicitly depict the tree structure of a target. And another view could be a documentation inspection view that would display the textual documents corresponding to the visual focus.

3.4 Interactivity

One of the user's principal goals of using a program visualization tool is to explore the program by exploring its display. SnakeTool provides the user the full power of interactive control over the display.

SnakeTool offers some important features with its interactive control of views. First, the user can control where to "dig in" and what to explore, can explore multiple components and can go back and forth interactively. Second, the user can combine the two viewing mechanisms so that he can first go down to a certain subsystem of his interest and then use fisheye views to explore it.

SnakeTool provides interactive manipulation of objects. The user has a complete control over graphical objects. Common object manipulation operations such as creating,

deleting, resizing, moving are embedded in TimeBench framework. Color coding and interactively coloring objects are added in SnakeTool. All these operations can be invoked by selecting the context sensitive menus, or by directly clicking the mouse button.

View transitions in some program visualization systems are conducted automatically, which imposes a heavy cognitive task on the user since the user usually can't expect when and where the transition will happen. SnakeTool provides the user an interactive control over view transitions which helps to reduce the cognitive task. Furthermore, such interaction helps a smooth view transition.

In SnakeTool, the user can not only control the view transition but also the response time of the view transition. Lock and unlock facilities are provided so that the user can choose to update only part of the display during the view transition. This facilitates a fast response time, because only those objects that need to be redrawn are updated.

Figure 21 shows a box-specific menu, from which the Lock and Unlock functions can be invoked. When the box b_6 is "locked", its size will be fixed in future view transitions until it is "unlocked". Figure 22 shows a subsequent picture that when the box c_3 is selected, the opening of c_3 only affects those objects inside b_6 , the locked box, because locking localizes the view transition impact on display update.

Figure 21 Invoking the Lock function from a box specific menu: b6 is locked.

Figure 22 Localizing the view transition: Because its parent b_6 is locked, the opening of c_3 only affects c_1 and c_2 . C_2 with its children shrinks and is pushed to the boundary of b_6 , and c_1 becomes even smaller and hides its name. Thus a new local fisheye view is generated in the box b_6 .

3.5 Presentation of Software Systems: An Example

We have discussed different viewing techniques in section 2.3 and described the layered fisheye view in section 3.3. Our observation strongly suggests that the layered fisheye view should be the centerpiece of our graphical display. In this section, we give an example of using SnakeTool to explore a software system.

Figure 23-28 show the process of visualizing a software system which is hierarchically structured. In addition to the layered fisheye views generated during the exploration process, these figures also show how the view transition affects buttons, fingers, and arcs, and how the layered fisheye view looks when the user explores multiple interested components simultaneously. All these figures are screen dumps from SnakeTool. The windowing dress and user controls are cropped to save space.

Figure 23 The top-level flat view: buttons and figures represent communication paths.

Figure 24 A layered fisheye view: **B₄** is the focus, it opens and its children are displayed. This affects not only its surrounding boxes, but also buttons and fingers. Fingers **f₁₋₅** are distorted and pushed away from **b₄**; buttons are moved with their boxes and become smaller.

Figure 25 A layered fisheye view (one more layer down than Figure 24): Two foci have been selected, first c_2 then c_1 , both at the third level of the structure hierarchy and in the same box b_4 ; after these two view transitions, c_3 becomes much smaller and hides its name.

Figure 25 is a layered fisheye view after two view transitions on Figure 24, first c_2 is selected as the focus and then c_1 . Since both c_1 and c_2 are in a locked box b_4 , the view transitions are localized inside b_4 . Other parts of the display are the same as they are in Figure 24. Note the significant changes with c_1 , c_2 , c_3 and c_1 's button.

Figure 26 Fisheye views with multiple foci.

From Figure 25 to Figure 26, two new focuses b_3 and c_4 have been selected. When b_3 opens, the whole display is affected: all other objects outside b_3 are pushed and distorted (boxes getting smaller, arcs and fingers being squeezed), except b_4 whose size is maintained by locking. Then b_3 is locked and c_4 is selected; generating another local fisheye view in b_3 .

Figure 27 Layered fisheye views

Figure 27 shows the final display generated by integrating multiple fisheye views. Three interesting features can be perceived from this figure. First, the whole display provides a fisheye view effect: the user's two main interested components b_3 and b_4 are depicted in detail and other components are displayed for the only purpose of preserving context. Second, two small fisheye views are presented locally inside b_3 and b_4 , and a even smaller fisheye view (on a deeper layer) inside c_4 . Finally, the integration of these fisheye views of different layers enhances the overall fisheye view effect.

Figure 28 is a sequence of snapshots, which illustrates the above exploration process by putting the figures together.

(1)

(2)

(3)

(4)

(5)

(6)

Figure 28 The process of exploring a target system.

3.6 Summary

SnakeTool is a prototype system. It automatically generates layered fisheye views of the target systems, and uses color to enhance the presentation. It supports interactive manipulation, smooth transition between views, and fast response time. Integrated in an OOP framework, the system design is flexible so that new views can be explored and different views can be combined.

SnakeTool represents large software systems at a high level of abstraction, the level of component and timethread. It presents nesting-diagram-like pictures to illustrate the system structure, and uses timethreads to depict the system behaviour.

The primary view SnakeTool provided is the layered fisheye view. The layered fisheye view shows the information of the user's interest at the level of detail the user desires, and presents the contextual information in less detail. Furthermore, the layered fisheye view allows multiple focuses so that the user can explore multiple components of his interest simultaneously.

The current SnakeTool is experimental. In this chapter, it is introduced to demonstrate the idea and illustrate the potentials of the layered fisheye view. SnakeTool is being developed at the DOORS group into a production tool.

Chapter 4 Layered Fisheye Views

In the development of SnakeTool, our primary task is to develop new techniques to visualize the target systems on a two dimensional screen space. Following this goal, the layered-fisheye-view technique is introduced. Due to its significance in this thesis work and its generality, this chapter is devoted to a full description of this new technique. The chapter starts with a rather detailed description of Furnas' generalized fisheye views. It follows Furnas' idea and introduces the layered fisheye views for the visualization of hierarchical structured information. The view formation process is described, and the algorithm and its implementation are presented. Finally, a comparison of the layered fisheye view and the previous work on fisheye view is given.

4.1 Generalized Fisheye Views

Our layered fisheye view technique follows from Furnas' pioneering work on the naturalistic study and the formalism of fisheye views[18]. For a better understanding of the fisheye view strategy, we give a more detailed description of Furnas' work in this section.

In his study, Furnas first presented many convincing examples of fisheye views occurring naturally in life, arguing that fisheye views help to the organization and understanding of large information systems. One interesting example is the so-called "New Yorker's View of the United States", which is a poster by Steinberg showing a distorted map of the States. The "map" shows midtown Manhattan in great detail -- street by street. The neighboring states of New York are just patches of different colors while other major parts of the country are reduced to a few landmarks. From this map, the New Yorker can get information related to his daily life as well as a geographical overview of his country. This distorted map is a typical example of how people use fisheye views to present large amount of information. Similar observations are made in other examples such as the fisheye editorial strategy of news editors and the employee's fisheye view of large cooperations[18]. Furnas proposed that many naturally occurring views of the world exhibit a fisheye character, and he suggested that fisheye views might be a common strategy for humans to represent large structures cognitively.


```

1  #define DIG 40
2  #include <stdio.h>
3
4  main()
5  {
6      int c, i, x[DIG/4], t[DIG/4], k = DIG/4, noprint = 0;
7
8      while ( ( c = getchar() ) != EOF ) {
9          if ( c >= '0' && c <= '9' ) {
10             x[0] = 10*x[0] + (x - '0');
11             for( i=1; i<k; i++){
12                 x[i] = 10*x[i] + x[i-1]/1000;
13                 x[i-1] %= 10000;
14             }
15             } else {
16                 switch (c) {
17                     case '+':
18                         t[0] = t[0] + x[0];
19                         for(i=1; i<k; i++){
20                             t[i] = t[i] + x[i] + t[i-1]/10000;
21                             t[i-1] %= 10000;
22                         }
23                         t[k-1] %= 10000;
24                         break;
25                     case '-':
26                         t[0] = (t[0] + 10000) - x[0];
27                         for( i=1; i<k; i++){
28                             t[i] = (t[i] + 10000) - x[i]
29                                 -(1 - t[i-1]/10000);
30                             t[i-1] %= 10000;
31                         }
32                         t[k-1] %= 10000;
33                         break;
34                     case 'e':
35                         for ( i=0; i<k; i++) t[i] = x[i];
36                         break;
37                     case 'q':
38                         exit(0);
39                     default:
40                         noprint = 1;
41                         break;
42                 }
43                 if( !noprint ) {
44                     for( i=k-1; t[i] <= 0 && i > 0; i--);
45                     printf("%d", t[i]);
46                     if( i > 0 ) {
47                         for( i--; i>=0; i--) {
48                             printf("%04d", t[i]);
49                         }
50                     }
51                     putchar('\n');
52                     for( i=0; i>k; i++) x[i] = 0;
53                 }
54             }
55             noprint = 0;
56         }
57     }

```

Figure 29 A flat view of a C program[18].

```

1  #define DIG 40
2  #include <stdio.h>
4  main()
5  {
6      int c, i, x[DIG/4], t[DIG/4], k = DIG/4, noprint = 0;
8      while ( ( c = getchart()) != EOF) {
9          if ( c >= '0' && c <= '9') {
16             } else {
17                 switch (c) {
18                     case '+':
25                     case '-':
34                     case 'e':
35*                     for ( i=0; i<k; i++) t[i] = x[i];
36                     break;
37                     case 'q':
39                     default:
42                 }
43                 if( !noprint) {
53                     }
54             }
55             noprint = 0;
56         }
57     }

```

Figure 30 A fisheye view of the C program in Figure 23[18].

Another interesting example of Furnas' is a fisheye view presentation of a computer program. Figure 29 (adapted from [18]) is a conventional flat view of a C program (a calculator program that does integer addition and subtraction), where the box indicates the viewport bounded by a standard window. Suppose the user's focus is the *for* loop under *case 'e'* statement (line 35), we see that some of the details inside the box helps little to the understanding of the focused code. The user has to scroll up and down to refer to variables and the program structure. Figure 30 shows a fisheye view of the same program: the underlined code in Figure 29 is displayed on a screen space with the same size of the viewport, the *for* loop is shown as the current interest, the *switch-case* statement as the local context, and the *main()*, *define* as the global context. We can see that the main difference between the fisheye view and the flat one is that some superfluous detail of the flat view has been traded for some more remote but higher-level, contextual information in the fisheye view.

Furnas found that the fundamental principal behind the fisheye view strategy is to provide a balance between local detail and global context. And he continued to formalize this balance by the "Degree of Interest" (DOI) function. In his study, different target systems under investigation have different DOI functions. For a specific target, its DOI

function maps each of its objects to a number that tells how interested the user is in seeing the object, given the current focus. Therefore, a successful fisheye view presentation depends on discovering a good DOI function. Furnas proposed that, in general, distance and *a priori importance* are two major factors of DOI functions. A generalized formula presented by Furnas is

$$DOI_{\text{fisheye}}(x | y) = API(x) - D(x,y)$$

where $DOI(x | y)$ defines the user's degree of interest in seeing object x , given that object y is the focus; $D(x,y)$ is the distance between x and y , and API is the *a priori importance* of x . This simple formula tells that the user's interest in seeing object x increases with x 's API and decreases with the distance between x and y .

Based on the generalized fisheye view strategy, Sarkar and Brown[36] introduced layout considerations into Furnas's formalism and interpreted the DOI function as the following two equations:

$$(I) P_{\text{feye}}(v | f) = \mathcal{F}_1(P_{\text{orig}}(v), P_{\text{orig}}(f))$$

where $P_{\text{feye}}(v | f)$ is the position of the object v in the fisheye view with f as the focal object, and $P_{\text{orig}}(v)$ and $P_{\text{orig}}(f)$ are the positions of objects v and f in normal coordinates, respectively.

$$(II) S_{\text{feye}}(v | f) = \mathcal{F}_2(S_{\text{orig}}(v), P_{\text{orig}}(f), P_{\text{orig}}(v), API(v))$$

where $S_{\text{feye}}(v | f)$ is the size of the object v in the fisheye view, $S_{\text{orig}}(v)$ is the size of the object v in the original view, and $API(v)$ is the *a priori importance* of object v .

The above two are general relations, specific and sensible function $\mathcal{F}_1$ and $\mathcal{F}_2$ have to be chosen to generate useful fisheye views. For example, it is reasonable to assume that a more interesting object has a larger size, thus, a simple relation to decide the size of an object v in a fisheye view can be: $S_{\text{feye}}(v | f) = K * S_{\text{orig}}(v) * API(v) / lvfl$, where K is a constant, and $lvfl$ is the distance between v and f . This tells that the interest of seeing a certain object v increases with its importance and decreases with its distance to the focus.

In spite of the widespread applicability of the generalized fisheye view strategy, we are particularly interested in using it to present software system structure and behaviour on a 2D screen space (two dimensional Euclidean space). Thus, from here on, we will discuss our layered fisheye views in the narrow scope of the graphical fisheye layout of our diagram illustration of software systems --- the nesting-diagram-like pictures.

4.2 Layered Fisheye Views

The approaches of both Furnas and Sarker et al are most suitable for presenting flat 2D graphs such as maps. With their techniques, hierarchically structured information is usually represented as a tree and its fisheye view a distorted tree. However, since our graphical representation of software systems is nesting-diagram-like pictures whose nesting level may be controlled by the user during the display, their techniques can not apply. Another problem is that it is difficult to illustrate the system behaviour with trees. To cope with these problems, we introduce the layered fisheye view technique.

4.2.1 Terminology

We first explain some terms which are important to a clear description of our layered fisheye view technique. For the illustrative purpose, we use the simple example presented in Figures 19 to 24 in Chapter 3, of which Figure 22 is repeated in Figure 32. Figure 31 is a conventional tree presentation of the same system. In Figure 31, we see that three successive *layers* are displayed. The top layer is represented as layer 1, and the layer number increases successively while going down the hierarchy, for example, Figure 31 shows that layer 1 has 19 modules, b1-b19; layer 2 has 3 modules, c1-c3; and layer 3 has 7 modules, d1-d4, k1-k3. In Figure 32, an open box with other boxes displayed inside is a *focus*, the most recently opened box is called the *current focus*; and boxes which have the same parent as the current focus are called *neighbors* of the current focus. Each box has a *position*, specified by the coordinates of its center in the polar coordinate system while the center of current focus is the origin. The *size* of a box is defined by its width and length. The *area* of a box is defined as its width times its length. The *distance* between two boxes is the euclidean distance between their positions.

Since SnakeTool uses the graphical representation model in TimeBench, graphical

objects are initially created and laid out by the parent-child relationship between software models. Figure 33 shows an initial layout of sub-modules of c3, which is a part of the graphical presentation of the layer 3 in Figure 31. We say that the area of the bounding box of all the children of a focus is the *world area* of the focus, e.g. the bounding box of k1, k2 and k3 is the *world area* of c3, and the area of the focus itself is its *window area*.

Figure 31 A conventional tree presentation of the system illustrated in Figures 19 to 24

Figure 32 Layered fisheye view presentation of Figure 31

Figure 33 Initial layout sub-modules of c3, the outlined box area is the world area.

4.2.2 Non-linear Magnification

The creation of a layered fisheye view involves two major geometric transformations: non-linear magnification, denoted as T_m , and perspective projection, denoted as T_p . During the non-linear magnification, the current focus is magnified and its neighbors are pushed away from it so that its children have enough screen space to be projected and displayed on its magnified space. The overall effect is that the whole picture is magnified.

In Figure 34, two boxes b_6 and b_{11} are taken from the display in Figure 18 to demonstrate the T_m transformation, where b_6 is the current focus and b_{11} is a neighbor. v_1 and v are the positions of b_6 and b_{11} , respectively. Suppose, after the magnification, b_6 is magnified (presented as the bigger box b'_6), and b_{11} is pushed away to the new position v' (presented as b'_{11}). We see that the T_m 's main tasks are to recompute the size of b'_6 and the position of b'_{11} .

In a polar coordinate system with v_1 as the origin, we shoot a ray from v_1 to v which intersects the boundary of b_6 at p_1 and boundary of b'_6 at p_2 . The size of b'_6 will depend on a magnification factor, K_m ,

$$d' = d * K_m \quad (1)$$

where $d = |v_1 p_1|$, $d' = |v_1 p_2|$, and $K_m \geq 1$.

And the new position of b_{11} , i.e. v' , will be somewhere on this ray,

$$|v_1 v'| = r + d * (K_m - 1) + d_1 * f(r) \quad (II)$$

where $r = |v_1 v_2|$, $d = |v_1 p_1|$, $d_1 = |p_1 p_2|$, and $\mathcal{F}(r) = K_{rr} K_r / r$, the function $\mathcal{F}(r)$ is called the repulsive function, and K_r ($K_r \geq 0$) is called the repulsive factor.

Figure 34 Non-linear magnification

Equation (1) tells that the magnification of the current focus depends on its original size and the magnification factor K_m . Since d is the distance from the position of the focus to its boundary, it is easy to get the magnified size of the focus. K_m can be either interactively controlled by the user (the user controls how big he wants the focus to be), or automatically adapted (by a function of the number of the focus' sub-modules, the more the sub-modules, the bigger K_m).

Equation (II) has three components: r is the distance between the current focus and the neighbor in the original view; the second component is the expanded length of the focus; and the third component is a contribution from the repulsive function, whose magnitude is inversely proportional to r . The effect of the repulsive function is the magnification of the close neighboring area of the focus. When applying Equation 2, however, we can choose K_r to be zero, in this case such effect is omitted. The repulsive function is analogous to a magnetic field function, imagine that the opening focus is a magnetic source which puts repulsive force on unhorsing objects.

The above two equations also indicate that the current focus is always magnified by a linear scale, and its close neighboring areas may be magnified nonlinearly if $\eta(r) > 1$. Note that the non-linearity of such transformation will change the linear structure of the initial layout of the target model. The magnified display is nonlinear now in the sense that the size of the focus is enlarged and the distances of the neighbors from the focus are lengthened nonlinearly. Mathematically, this raises the possibility of object overlapping, fortunately, an acceptable solution to this problem can be reached by choosing an appropriate repulsive factor K_r .

The picture generated by the non-linear magnification is an intermediate result, and has to be projected onto the view plane. Therefore, further transformation is needed.

4.2.3 Perspective Projection

The transformation T_p can be accomplished in two steps. The first step is to project the children of the focus, which were supposed to be laid out on a deeper layer plane initially, onto the magnified focus area. The second step is to further project the intermediate result on the current layer plane onto the view plane (the screen). This two-step transformation is illustrated in Figure 35.

In the first step, there are two cases:

Case 1: If the world area of a focus is smaller than its window area, the children of the focus are projected by a simple parallel projection. In this case, a simple translation is actually used, by which the coordinates of all the children of the focus are translated so that the center of its world area is moved to the position of the focus.

Case 2: If the world area of a focus is bigger than its window area, a perspective projection is applied, which calculates a new set of coordinates on the current layer plane, by the ratios of the lengths and width of the world area vs. those of the window area, respectively.

Figure 35 Two steps of perspective projection

These two cases can also be applied to the second step. The difference is that the first projection only applies to the children of the current focus whereas the second projection applies to both the children (their transformed coordinates resulting from the first projection) and the neighbors of the current focus.

Note that the term "perspective projection" is used here mainly for the purpose of clarifying the concept. Because we only deal with transformations for 2D pictures, the implementation of this "perspective projection" is easier than that of 3D perspective projection in the conventional pipeline approach in computer graphics.

4.2.4 Locking and the Layered Fisheye View Formation

T_m and T_p are two basic transformations. The layered fisheye view formation process consists of a sequence of T_m s and T_p s which are applied to the focus and its parents. When a box B_i at a certain layer i ($i \geq 1$) is selected as the focus, its magnification will affect its parents B_{i-1} , B_{i-2} , ..., B_1 in turn, and force them to be magnified. The overall effect looks like this: B_1 is selected, magnified, and projected with its neighbors onto the view plane, B_2 is selected, magnified, and projected with its neighbors onto the window area of B_1 , then B_3 , ..., until B_i . This process is illustrated in Figure 36.

Figure 36 Transformation sequence of the layered fisheye view formation

A problem with the above process is: Each time a new focus is selected, the whole picture is enlarged by a certain scale, however, the picture must be presented on the limited screen space so that the user can perceive it. Our solution for controlling the display size is “locking”.

The size of a layered fisheye view is affected differently in two cases: 1)Nonlocking: The virtual screen space (which is defined as 3000 by 3000 pixels in our SnakeTool prototype) is the bounded area. The formation of the layered fisheye view is restricted in the virtual screen space, that is, the whole view is generated on it. The user can browse the whole picture by scrolling. 2)Locking: Some open boxes are locked and a view transition will only affect the objects inside the locked box (as we saw in the examples in Chapter 3). The user can control the size of the layered fisheye view and maximize the usage of the screen space. In both cases, the layered fisheye view is always generated on a specified limited area. We called the boundary of such area *the view transition boundary*.

The view transition boundary sets the stop condition for the transformation sequence in Figure 36. When the T_m transformation process hits this boundary, it stops and the T_p process begins. When locking is applied as in most cases, the view transition boundary is decided by the locked box and the layered fisheye view is formed inside it.

The overall effect of the transformation sequence is that each box B_j ($1 \leq j \leq i$) in Figure 36 is magnified by a certain scale. Therefore, the layered fisheye view formation is a process to decide and apply the real magnification scale of each B_j . Figure 36 shows that the current focus B_i is magnified first and projected last onto the window area of B_{i-1} , its immediate parent, during the view formation process. Thus, the real magnification scale of B_i is determined by both the magnification factor K'_m , which is used in $T_m(B_i)$, and the

window area of B_{i-1} (the latter of which is indirectly determined by the view transition boundary). Following we give the relations between the real magnification scale and these determinative factors.

Suppose $T_m(B_i) \rightarrow B_i'$, $T_m(B_{i-1}) \rightarrow B_{i-1}'$, $T_m(Wld) \rightarrow Wld'$ (illustrated in Figure 37), where Wld is the bounding box of all neighbors of B_i , i.e. the world area of B_{i-1} in the original view.

Figure 37 After transformation T_m : $B_i \rightarrow B_i'$, $B_{i-1} \rightarrow B_{i-1}'$, $Wld \rightarrow Wld'$

Also as illustrated in Figure 37, the size of B_i is denoted as w by h , the size of B_i' as w' by h' , the size of Wld as W_wld by H_wld , and the size of Wld' as W_wld' by H_wld' . To simplify the discussion, we omit the effect of the repulsive function, letting $K_r = 0$. If we denote the initial magnification factor of the focus B_i as K_m^i , we can obtain the real magnification scale K^j for each B_j through the following two steps:

Step 1: In the T_m transformation sequence in Figure 36, every B_j ($1 \leq j \leq i$) is magnified without consideration of the view transition boundary. Thus we have:

$$w' = K_m^i w, \quad h' = K_m^i h,$$

$$W_{wld'} = W_{wld} + (w' - w) = W_{wld} + (K_m^i - 1)w$$

$$H_{wld'} = H_{wld} + (h' - h) = H_{wld} + (K_m^i - 1)h$$

The magnification factor of B_{i-1} is

$$K_m^{i-1} = \min \left(1 + \frac{(K_m^i - 1)w}{W_{wld}}, 1 + \frac{(K_m^i - 1)h}{H_{wld}} \right) \quad (III)$$

and this relation can also be applied to any B_j ($1 \leq j \leq i-1$) in the sequence by replacing w and h with the width and height of the magnified B_{j+1} , W_{wld} and H_{wld} with the width and height of B_j .

Step 2: In the T_p transformation sequence in Figure 36, T_p is applied to B_j ($2 \leq j \leq i$) after B_{j-1} completes its transformation. K_j^j , the real magnification scale of B_j , will depend on the window area of B_{j-1} , which has been decided and become the available window area for the display of B_j and its neighbors. For the current focus B_i , if we denote the size of the window area as W_{win} by H_{win} , we have three cases:

Case 1: $W_{ld'}$ is smaller than the window area, i.e. $W_{wld} \leq W_{win}$ && $H_{wld} \leq H_{win}$. This means that there is enough space for displaying $W_{ld'}$. Thus

$$K^i = K_m^i \quad (IV.1)$$

Case 2: $W_{ld'}$ is bigger than the window area, but B_i' is smaller than the window area, i.e. $(W_{wld} \geq W_{win} \parallel H_{wld} \geq H_{win})$ && $(w' \leq W_{win} \parallel h' \leq H_{win})$. In this case, we can still magnify B_i by the factor of K_m^i , but neighbors of B_i has to become smaller. Relations from perspective projection are

$$\frac{K^l w}{K_m^l w} = \frac{W_{wld}}{W_{win}} = \frac{W_{win} + (K^l - 1)w}{W_{win}}$$

and

$$\frac{K^l h}{K_m^l h} = \frac{H_{wld}}{H_{win}} = \frac{H_{win} + (K^l - 1)h}{H_{win}}$$

From the above relations, we have

$$K^i = \min \left(\frac{K_m^l (W_{wld} - w)}{W_{win} - K_m^l w}, \frac{K_m^l (H_{wld} - h)}{H_{win} - K_m^l h} \right) \quad (IV.2)$$

Case 3: B_i' is bigger than the window area, i.e. $w' \geq W_{win}$ & $h' \geq H_{win}$. Similarly as in case 2, we get

$$K^i = \min \left(\frac{C_i (W_{wld} - w)}{(1 - C_i) w}, \frac{C_i (H_{wld} - h)}{(1 - C_i) h} \right) \quad (IV.3)$$

where C_i is a logarithmic function of B_i 's layer number

$$C_i = \begin{cases} \log_{(\alpha+1)}(\alpha - i + 1), & \text{if } C_i \geq \max(w/W_{wld}, h/H_{wld}) \\ \max(w/W_{wld}, h/H_{wld}), & \text{otherwise} \end{cases} \quad (V)$$

C_i is called the layer factor, and α is the layer number of the current focus. In the case presented in Figure 36, $\alpha = i$, thus $C_i = \max(w/W_{wld}, h/H_{wld})$; and $C_l = \log_{(\alpha+1)} \alpha$.

The same as in step 1, without loss of generality, the above relations regarding the computation of K^i and C_i can also be applied to any B_j ($1 \leq j \leq i-1$) in the transformation sequence depicted in Figure 36.

Theoretically, the three cases in step 2 have covered all possibilities during the perspective projection. However, when the current focus is at quite a deep layer in a hierarchical structure, and its parent box at higher layer is locked, things may happen such as the window area of the focus will be too small to display all of its children clearly. The problem in this special case can be solved by limiting the layers in a layered fisheye view. For example, suppose the maximum number of layers which can be presented is 10, when the user selects a focus at the eleventh layer, the display of objects of the first layered would collapse and only a dashed outline would exist to present the first layer. In this way, more space can be allocated for the presentation of objects at deeper layers. Such a solution would be acceptable because users are always more interested in the focuses and their context which are presented most recently.

4.2.5 The Algorithm

The layered fisheye view formation process is implemented by a recursive algorithm that applies the T_m and T_p transformations to the initial model and the intermediate results. When the user selects a focus and double-clicks a mouse button, the event (for magnification) is passed from the focus upwards to its parents at higher level, until the parent which is "locked" in the view transition boundary, then the magnification scale for each parent along this sequence is computed, and accordingly, the coordinate transformations is applied to them downwards.

Following is the two main procedures of the algorithm:

```
-handleEvent: aEvent
{
    if ( doubleClickInView ) then {
        currentFocus := getFocus(aEvent)
        if (currentFocus hasChildren) then {
             $K_m^t$  := getInitialMagFator()
            currentFocus magnifyBy:  $K_m^t$ 
            currentFocus projectChildren
        }
    }
    else {
        process other event cases
    }
}
```

```

-magnifyBy:  $K_m^t$ 
{
    if ( myParentIsRoot ) then {
        windowArea := VirtualScreenSpace
    }
    else if ( myParentIsLocked ) then {
        windowArea := MyParentSize
    }
    else {
        compute my parent magnification factor  $K_m^{t-1}$  by equation (I)
        myParent magnifyBy:  $K_m^{t-1}$ 
        windowArea := MyParentSize
    }
    if (  $W_{wld} \leq W_{win} \ \&\& \ H_{wld} \leq H_{win}$  ) then { /* case 1 */
        compute the magnification scale  $K^t$  by equation (IV.1)
    } else if ( ( $W_{wld} \geq W_{win} \ \&\& \ H_{wld} \geq H_{win}$ ) && ( $w' \leq W_{win} \ \&\& \ h' \leq H_{win}$ ) ) then { /* case 2 */
        compute the magnification scale  $K^t$  by equation (IV.2)
    } else { /* case 3 */
        compute the layer factor  $C_t$  by equation (V)
        compute the magnification scale  $K^t$  by equation (IV.3)
    }
    magnify self (the current focus) by scale  $K^t$  using equation (I)
    foreach neighbour of self
        push the neighbour away from self using equation (II)
    end
    project self onto the windowArea by a perspective projection
    foreach neighbour of self
        project the neighbour onto the windowArea by a perspective projection
    end
}

```

Note that the above two procedures give the activity sequence and the functional descriptions of each activities of the algorithm. Since SnakeTool was implemented in an OOP style, the implementation of the layered fisheye view algorithm would be consistent with the system structure and programming style. Actually, the functionalities of the first procedure were implemented in classes: *View* and *TBox*¹, *View* handles the mouse event detection and processing whereas the *TBox* deals with the fisheye view formation. The functionalities of the second procedure were implemented in classe groups of Box, Button, Tail, Arc, and Thread. Moreover, to provide efficiency, a unified transformation combining T_m and T_p was actually used in the implementation for coordinate transformation, instead of the two separate transformations in the algorithm description. More implementation issues regarding the layered fisheye view will be discussed in section 5.3.

1. Appendix A provides a diagram of part of TimeBench class hierarchy. For a complete description of TimeBench class defination, readers are refered to [40]

4.2.6 Transformation on Arcs

As described in Chapter 3, there are two kinds of arcs in our presentation of software system: the finger and the timethread. Conceptually, they are different, however, they have the same graphical representation, the Arc. Since the representation of our targets may have fingers and timethreads, some similar transformations as above must be applied to their graphical representations so that they can be part of the layered fisheye view.

Basically, the structure of an arc contains a collection of points, which controls the size and shape of the arc. In actuality, these controlling points can be regarded as separate objects, like boxes of very small size. Thus, T_m and T_p are also applicable to the controlling points of arcs during the view formation. Therefore, arcs can be appropriately magnified or demagnified because the controlling points of each arc are fit into the view. However, instead of directly applying such transformation, some important adjustments are necessary in the implementation, because of the connection between the finger and its end components, and the association between the timethread and the components coupled with it. Issues related to the implementation will be discussed in Chapter 5.

One problem aroused during the transformation of arcs is that the magnified focus may overlap with some segments of the arc. This problem can be solved by transforming a very large number of intermediate points on the arc, or by modifying some ray tracing algorithm.

4.3 Discussion

The *layered fisheye view* is a new technique of fisheye viewing strategies, following Fumas' *generalized fisheye view*[18] and the *graphical fisheye view* of Sarkar and Brown [36]. On the one hand, these three techniques have the same advantage over other viewing techniques, that is, they all provide some kinds of balance between local detail and global context while presenting large amount of information. On the other, their effectiveness and applicability varies with different applications. In this section we will discuss some application-related issues of these techniques.

4.3.1 Fisheye Views of Hierarchical Structures

Furnas' generalized fisheye view has a wide applicability, and his DOI function is a general formula that must be defined before used to any specific application. Sarkar and Brown applied this formula to 2D planar graphs by defining the "distance" in the DOI function as euclidean. Specifically, both the generalized fisheye view and the graphical fisheye view can be used to illustrate hierarchical structure, given that our visualization targets are hierarchically structured software systems.

Figure 38 shows a conventional tree presentation of a hierarchical structure (adapted from [36]). By using Furnas' technique, each node is first assigned an API (In this example, the root has the highest API of 8; node 99, the leaf node, has the lowest API of 1. The API of each node is simply related to its level in the tree, the higher level it sits, the more important it is), the distance between two nodes is defined as the length of the shortest path between the nodes in the tree. A node is displayed in the fisheye view when its DOI value is greater than a given threshold. Thus, each node is either displayed or undisplayed based on its DOI value. The final view (a generalized fisheye view) is shown in Figure 39, with node 48 as the current focus. Based on the same API assignments but a different definition of the distance, a graphical fisheye view of Figure 38 is shown in Figure 40. We see that the generalized fisheye view and the graphical fisheye view of the hierarchical structure are both distorted trees. Their difference is: details in the former view are those objects close to the focus in terms of object's connections in the tree whereas those in the latter view are objects close to the focus in terms of their euclidean distances. Also noticed is that the importance of an object is illustrated by varying its size in the graphical fisheye view.

Figure 38 A hierarchical structure with 100 nodes

Figure 39 A generalized fisheye view of Figure 38, the distance between two nodes are defined as the path length in the tree.

Figure 40 A graphical fisheye view of Figure 38, the distance between two nodes are defined as their euclidean distance.

Both the generalized fisheye view and the graphical fisheye view are suitable for the hierarchical structure represented by trees. However, they are not suitable for the depiction of software systems because: First, it's difficult to define a notation (or DOI function) of differential API of components for a software system. The user usually has one or more components of his interest before exploring the system, but for other components, it is difficult to identify those which may be more interesting. Another reason is that, for the visualization of software systems, the technique we need should allow the user to control what he wants to see interactively rather than just to present a static view. Thus, predefining a API value to each component is not of much use. Secondly, designers usually don't represent software systems as trees. Thirdly, it is difficult to present behaviour information such as timethreads along with a tree structure.

Figure 41 A layered fisheye view of Figure 38.

Developed for the visualization of large software systems, the layered fisheye view technique is a sensible solution to the above problems. It is interactive; it presents the system by nesting-like-diagram pictures, which are closer to the depictions designers use; and timethreads fit well into the structure diagram. Therefore, it is more meaningful and effective in terms of presenting hierarchically structured software systems. Figure 41 shows a layered fisheye view of Figure 38.

4.3.2 Multiple Focuses

The generalized fisheye view has only one focus at a time. Furnas raised the issue of fisheye views with multiple foci, which were not consistent with his fisheye model, and he left the problem unsolved. Sarkar and Brown suggested that their technique was able to provide fisheye views with multiple foci by some screen space partition strategy, but provided no details. The multi-focus issue has not been addressed further in previous research on fisheye view because the related applications can be made up by fisheye views with one focus. To explore software systems, the user often need to see details inside

multiple components simultaneously, and the fisheye view with one focus is inadequate. In this regard, the layered fisheye view provides the capability that interactively integrates multiple focuses of the user's interest during the view formation.

Chapter 5 Implementation

This chapter describes issues in the SnakeTool prototype implementation. Section 5.1 describes the aspects of TimeBench which are most directly related to the SnakeTool implementation. Section 5.2 shows how the timethread is implemented and incorporated into the TimeBench framework. Implementation of the layered fisheye view for the box-and-timethread diagram are provided in section 5.3. At last, issues regarding some human computer interface concerns are discussed in section 5.4.

5.1 The TimeBench Framework

TimeBench is an existing prototype CASE tool which is developed to facilitate the design and development of real-time systems[23][40][41]. The primary features of TimeBench are visual design capture and code generation for real-time systems. Using it as a framework to prototype a visualization system, however, we are interested in its system structure representation and its graphical presentation capabilities. Two aspects of TimeBench are considered particularly useful from the perspective of fast prototyping SnakeTool: 1) its internal representation of the system under design, and 2) its view component.

5.1.1 System Structure in TimeBench

In TimeBench, the cooperation of three major components, logical, graphical and toolcomponent, provides the framework of representing the structure of a software system. Each of the components corresponds to a class hierarchy, and the system under design is built up by objects of classes of these hierarchies. In this section, we briefly present the TimeBench logical, graphical, and toolcomponent class groups, and indicate how they are related to the SnakeTool implementation. A class hierarchy diagram of these classes is provided in Appendix A, and for a complete description of the TimeBench class specifications and definitions, readers are referred to [40].

The TimeBench logical classes define the components of the MachineChart visual notation[10]. Their main responsibility is to describe and enforce the relationships between the notations in the MachineChart context. SnakeTool uses this property and extends it to timethread, for example, a timethread moves a certain segment with the corresponding box it attaches. The logical classes are also responsible for managing the system structure. An object of the MachineChart components knows its internal children and parent by responding to the message *-internals* and *-parent*, and an operation on one object may affect those connected with it logically, e.g. deleting an object will lead to the deletion of all its children. This “built-in” structure management mechanism is particularly useful for the layered fisheye view transformations in SnakeTool, where operations of moving and projecting a group of objects, e.g a box and all its children, need to be implemented.

Classes in the TimeBench graphical class hierarchy concern the shape of graphical icons displayed on the screen. Their main responsibilities are basic graphical operations such as drawing, erasing and moving. In TimeBench, most icons belong to either arcs or nodes, as illustrated in Figure 42. Arcs are curved lines whose shape is determined by a collection of points, and nodes are two-dimensional geometric shapes bounded by rectangles. Arc and node classes provide the most common methods such as *-moveBy:* and *-moveTo:*. These methods are used to implement the more complex methods for the fisheye view transformation in SnakeTool.

Figure 42 Class hierarchy of the graphical component: all graphical icons are derived from two major classes, <- indicates inheritance. GThread is the graphical representation of timethread.

The TimeBench toolcomponent class hierarchy contains classes describing the way objects responding to user events. Their main responsibility is to handle keyboard and mouse events and to separate them into logical and graphical events for further processing.

A software system in TimeBench is represented by a tree consisting of logical, toolcomponent and graphical objects, and each component of the system has such three objects to represent itself. Since classes in the logical class hierarchy are responsible for setting up the relations of different components, the structure of the tree is determined by the logical objects alone. Figure 43 shows a TimeBench internal representation of a simple system, which consists of three components, where components c2 and c3 are subcomponents of c1.

Figure 43 The TimeBench internal representation of a simple system. Circles represent logical object; ovals are for graphical objects; and boxes for toolcomponent objects. The system contains three components, the c2 and c3 subcomponents of the c1.

The logical, graphical, and toolcomponent objects of a system are closely related to each other. For example, adding or deleting an icon on the display will lead to the restructuring of the program's logical tree; and resizing the display will change the geometrical properties of the affected graphical objects.

To summarize, a system in TimeBench is represented internally as a logical tree, which contains the logical, the graphical and the toolcomponent objects. The logical objects interact each other to build and manage the system structure. The graphical objects deal with the appearance of each component on the screen, the user does not see the internal

logical tree directly, but instead sees views of graphical representations of it. The toolcomponent objects connect logical and graphical objects, and handle all the input events from the user.

5.1.2 The TimeBench Viewing Mechanism

The TimeBench view component is another important component related to our SnakeTool implementation. The view component handles all input from the user by merging input from the mouse or the keyboard into one entity called an “event”, and routing the event to a corresponding object (e.g. a toolcomponent) for processing. It also handles display and view update requests from toolcomponents, and draws on the view canvas[54].

Because the view component provides all basic facilities for event handling and view management, the implementation task of controlling the SnakeTool display becomes quite simple. We only need to modify the TimeBench view generation and update process in order to display objects from multiple levels of the system hierarchy rather than from only one level. In the following discussion, we will focus on the TimeBench viewing mechanism.

When TimeBench starts, a view object is created which controls the display on a SunView canvas. Of the instance variables of the view object, *newClnOfIcons*, *oldClnOfIcons* and *aLogicalObject* are used for the view generation and update. *NewClnOfIcons* keeps a collection of icons which will be displayed for next screen update; *oldClnOfIcons* keeps a collection of icons which are currently displayed; and *aLogicalObject* is the object whose contents are being displayed. When a display of a target system is first generated, a message called **-buildDisplayList:** is sent to the view object, with *newClnOfIcons* as the argument. This method asks *aLogicalObject* for its **-internals** and **-externals**, which return all children of *aLogicalObject*. Then for each child, a **-buildDisplayList: from:** method is invoked with the *newClnOfIcons* and the child as arguments, which actually adds the child and its affiliated icons (annotations in MachineCharts terminology) to the display list (*newClnOfIcons*). Finally, a method called **-updateDisplay** is called to draw the icons in the display list on the screen.

When there is a request for updating the display, which can come either from the above process, or from user events such as deleting, or moving icons, the **-updateDisplay** message

is sent to the view object. This method scans over the *newClnOfIcons* and *oldClnOfIcons* collections. If an icon in *newClnOfIcons* is marked as changed, a message *-erase:* is sent to its correspondent in *oldClnOfIcons* so that its old appearance can be erased from the screen. Then, the new one is drawn on the screen, and *oldClnOfIcons* is updated by replacing the old icon with a copy of the changed one.

TimeBench only displays icons from the same level of the system hierarchy, when the user wants to see components of next deeper level, the *newClnOfIcons* and *oldClnOfIcons* display lists will be cleaned up and rebuilt for the new level. However, this doesn't satisfy the display needs of SnakeTool, since layered fisheye views are presentations of components from different layers of the system hierarchy. Modifications on the TimeBench viewing mechanism will be discussed in section 5.3.4

5.2 Implementing Timethread

We have briefly presented the concept and significance of timethreads in section 3.2.2. In terms of its implementation, timethreads fit well in the arc group since it has the same graphical appearance as arcs and it needs a similar interface to handle events. Thus, we can incorporate the timethread into TimeBench framework by simply inheriting the basic behaviour of TimeBench arc classes. The difference between timethreads and TimeBench arcs is that 1) unlike arcs, the starting and ending points of a timethread need specific representations because these points have their own conceptual meanings, and 2) a timethread needs to be logically associated with the representative icons of its activated components, whereas an abstract arc is logically linked with its source node and destination node.

To meet the above requirements, the timethread is designed to be composed of three components: the thread start, the thread and the thread end. Correspondingly, each component has its logical, graphical, and toolcomponent representations.

5.2.1 Logical

The implementation of timethread logical part is simple. Three classes, *LThread*, *LThreadStart* and *LThreadEnd*, are defined in the logical hierarchy. Both *LThreadStart*

and *LThreadEnd* are derived from *LNode*, and *LThread* from *LArc*.

Classes *LThreadStart* and *LThreadEnd* have no new instance variables, they differ in behaviors from their superclass by overriding the superclass methods *-checkSrcArc:* and *-checkDstArc:*. The two methods are used to check whether the argument is a valid link to be attached to the end nodes. Thus, they enforce that only timethreads can attach them. Also a thread start logical object answers its timethread by responding to the message *-thread*, so does the thread end.

The class *LThread* has a new instance variable *aClnOfAnchoredBox*. This variable points to an ordered collection which contains the logical objects of the timethread activated components, in the order of their activation (since components are represented by boxes, we call the box representing the activated component an anchored box of timethreads). The main responsibility of *LThread* is to take care of the relationship of the timethread and the boxes it anchors. For this purpose, a group of methods are implemented, for example, *-addBox:* or *-removeBox:* adds a box to or deletes a box from the collection *aClnOfAnchoredBox*; *-associateWith:* or *-deassociateWith:* sets up or destroys the relationship between the timethread and an anchored box.

5.2.2 Graphical

In the graphical class group, a new class *GThread* is defined as a subclass of *GArc*. This new class has its own instance variable *aClnOfTags*, which points to a collection of tags of boolean type. Each of the tags corresponds to a point of the thread and indicates if this point is visible or not. A *GThread* object behaves differently from a *GArc* in the drawing operation. The *-draw:* method of *GArc* only draws line segments from one visible point to the next visible point by skipping over the invisible points in between. This is necessary because a timethread may go through many components, but not all of these components are visible in a layered fisheye view. A method *-visible: pointNum:* is implemented so that a point on a timethread can be set to visible by its anchored box when the box becomes visible.

A *GThread* object also behaves differently from a *GArc* during movement, because of its logical association with its anchored boxes. Some methods are developed especially for timethread moving operations. For example, the method *moveSegmentBy: follow:* moves

the affected segment of the receiver following the movement of an anchored box. A non-trivial issue concerning the movement is that moving a timethread may affect all of its anchored boxes, which may cause a severe performance problem when there are many objects affected and needed to be redrawn on the screen (section 5.4.1). In this case, a more sophisticated algorithm would be needed. Other methods specific to *GThread* are developed for applying the fisheye view transformation on timethreads. This will be discussed in section 5.3.3

The class *GThreadEnd* is derived from *GArcTail* and uses the shape of *GArcTail*. Basically, a *GThreadEnd* differs from a *GArcTail* in the sense that the former knows how to adjust its direction when responding to *-draw:*.

Currently, the *GThreadStart* class is derived from but the same as the *GCircle*. Like other classes, new responsibilities may be given to *GThreadStart*.

5.2.3 Toolcomponent

Three toolcomponent classes, *TThread*, *TThreadStart* and *TThreadEnd* are developed for the timethread. *TThread* is defined as a subclass of *TStructureArc*, *TThreadStart* and *TThreadEnd* are subclasses of *TStructureNode*.

TThread has different behaviour from *TStructureArc* when handling creation and deletion. First, during its creation, a timethread needs to set up the associations with its anchored boxes. This is implemented in two *TThread* methods *-selectStartingIcon: forView:* and *-selectEndingIcon: forView:*. Second, the *TThreadStart* and *TThreadEnd* objects are dependents of the *TThread*: they are created only when a timethread is created, and deleted when their timethread is deleted.

Since a major responsibility of toolcomponent is to provide an interface between the user and the system, most methods of the timethread toolcomponent classes are about interactive manipulation of the timethread. For example, *TThreadStart* overrides its superclass method *-moveTo: forView:* (the first argument is an mouse event), so that the user can move the starting point of a timethread to a new anchored box. In this case, if the thread start has been anchored in a box before, the message *-deassociateWith:* will be sent to the thread to break the association, and *-associateWith:* will be sent to set up a new

association with the new anchored box. Much in the same sense, *TThread* overrides methods *-pickSegOrgArc: forView:* and *-shapeLine: forView:* of *TStructureArc* so that the user can interactively set up or break the association between a timethread and its anchored boxes.

Methods described above are about dynamic manipulation of the timethread. This may not help our visualization purpose, since timethreads, as the behaviour representation of software systems, need to be generated and updated automatically when display changes. The reasons that we provide these methods for interactive manipulation are: 1) besides visualization, SnakeTool may be used for the designer to play with their design ideas, where interactive manipulation on timethreads is essential; 2) Allowing the user to interact with timethreads is consistent with the TimeBench framework, where every icon can be manipulated.

Other methods implemented in this class group are specific to the fisheye view generation, which we will describe in section 5.3.3.

5.3 Applying the Layered Fisheye View Technique to the Box-and-Timethread Diagram

In this section, we address the implementation of layered fisheye views on our box-and-timethread diagram, based on the algorithm we presented in section 4.2.5. To help our discussion, we give the pseudo codes of some important methods when they are useful.

5.3.1 Initial Layout

Currently the first layout of the target system is taken directly from the system hierarchical structure and the execution tracefile. As an initial implementation, this decision is a sound one since it offers the desired experimental environment at minimal effort. The principal rule for the initial layout is to put related components in close proximity to ease the task of laying out timethreads, where component relations are from the system semantic information. The second rule is to arrange all immediate children of a component in such a way that the display area they occupy has a length vs. width ratio

approximating 4/3. The reason for this is to maximize the use of available display space. An automated layout algorithm conforming to these rules is under implementation.

The layout of the layered fisheye view is generated automatically by a sophisticated graphical transformation technique. However, the appearance of the results depends on the initial layout. An initial layout that doesn't satisfy the above rules would lead to a less meaningful layered fisheye view.

The most significant problem in the display layout is the difficulty in automatically drawing the timethreads. Two requirements have to be met: the timethread should be complete, which means it should touch all of its components that are visible in the current display; and it should have as little cross-over with unrelated components as possible. A fully automated layout planning algorithm would be very time consuming and insufficient. Therefore, the solution in SnakeTool is resorted to interaction: When a fisheye view is generated, some segments of timethreads are stretched and some are squeezed by applying the layered fisheye view algorithm. After this "first draft" of the layout is generated, the user can modify the appearance of timethreads by reshaping them to arrive at a "nice" output.

5.3.2 Handling the Event

The TimeBench main program is implemented in an event-driven style. To incorporate the layered fisheye view formation in the event-capture-then-process loop, a double click of the right mouse button on a box is designated to the user's intention of generating a layered fisheye view with the box as his focus. The view object captures this event and sends the message *-handleEvent: forView:* to the focused box. Based on the event type, the focus will further call the method *-doSelection: forView:*, which is overridden in *TBox* to incorporate the invocation of the layered fisheye view processing. The pseudo code implementing this function is as follows (note that we borrowed the scope operator of C++ language to clarify the description):

```
TBox :: -doSelection: event forView: aView {  
    if (event is mouse double click && event is MS_RIGHT){  
        if ([self canOpen] ) {  
            compute the magnifying scale,  
            apply the recursion algorithm of section 4.2.5, with the above scale,  
            project the internal children of the receiver,  
        }  
    }  
}
```

```

        update the view,
    }
} else {
    [super doSelection: event forView: aView].
}
}

```

The above method is the interface between the user event and the fisheye view processing and determines the processing sequence for the view generation. First, a simple method *-canOpen* checks that the focus should have internals and be closed so that it can be opened. If the condition is satisfied, a magnification scale is computed based on the number of the children of the focus. Second, the layered fisheye view transformation algorithm is applied by invoking a method *-magnifyBy:*. Then, the children of the focus are projected into the area it newly occupies, and all timethreads are updated. This is implemented by a method called *-projectInternals*. These two methods *-magnifyBy:* *-projectInternals* complete coordinate transformations for all affected graphical objects. Finally, the display list is rebuilt and the layered fisheye view is shown, this is fulfilled by a method *-fisheyeFocusAt:* of the *View* class.

5.3.3 Transforming the Coordinates of Nodes and Arcs

The layered fisheye view transformation algorithm described in section 4.2.5 is applicable to graphs organized in hierarchical structures, with boxes and curved lines as the building blocks. Our box-and-timethread diagram has boxes, buttons, push arcs, arc tails, threads, thread starts and thread ends as its basic objects. Of these different objects, push arcs and threads are in the arc group, and others in the node group. Although TimeBench treats arcs as curved lines and nodes as boxes (rectangles), regarding our diagram as simple graphs grouped hierarchically and simply applying the transformation algorithm, however, may lead to non-sensible fisheye views. This is because the our generic algorithm is graphic-oriented, but a presentation of a system must conform with the logical constraints between system components, i.e, buttons must keep attached with their boxes, and threads must keep attached with their anchored boxes. In this section, we address how to apply our generic transformation algorithm to the box-and-timethread diagram.

5.3.3.1 The main method *-magnifyBy:*

Following is the implementation of the method *-magnifyBy:* in pseudo code form.

```

TBox :: -magnifyBy: (double)scale {
1   local variable: worldArea, windowArea, recursionScale, magnificationFactor,
2
3   /** STEP 1, recursively go up the hierarchy to find the boundary of this view transition */
4   if ( self parent isOpen ) {
5       if (self parent is unlocked ) {
6           compute the recursionScale for my parent by formula III,
7           self parent perform magnifyBy: recursionScale,
8       } else {
9           self parent adjust the size of its buttons
10      }
11      compute the windowArea of the focus;
12  } else /* self parent is the folder at the top level */
13      assign the default window area to the windowArea,
14  }
15  compute the worldArea of the focus,
16  increment self display level;
17
18  /** STEP 2, compare worldArea and windowArea, compute the magnification factor by */
19  /** a corresponding equation described in section 4.2.5 */
20  switch (compare worldArea and windowArea ) {
21      case 1: there is enough display space for the desired magnification
22          magnificationFactor := scale; /* Equation IV.1 */
23      case 2: there is enough display space only for self magnification
24          compute the magnificationFactor by Equation IV.2
25      case 3: there is no enough display space for self magnification
26          compute the magnificationFactor by formula IV.3
27  }
28
29  /** STEP 3, actual coordinate transformations */
30  self perform
31      -magnifyBy: magnificationFactor projectFrom: worldArea into: windowArea,
32  each of the self neighbors perform
33      -pushAwayBy: magnificationFactor center: self projectFrom: worldArea into: windowArea,
34  }

```

The *-magnifyBy:* method consists of three parts of operations. The first part (line 4-16) recursively goes up the structure hierarchy and sends the same message to the receiver's parent (line 7) until the display boundary of this view transition. The boundary is reached when the parent is either a locked box (line 8) or the top level folder (line 12). Here self parent always refers to the toolcomponent object of the receiver's logical parent. The purpose of this program segment is to find the maximum area (the windowArea) for the following fisheye view transformation and view display. If the recursion stops at a locked box, this area is determined by the size of the box, its display level and the size of its buttons; otherwise, the area is given a default value, which can be the display screen space, or can be as large as the display canvas.

The second part (line 20-27) computes the magnification factor which will be used in the coordinate transformation. The computations are quite straightforward by applying the

equations we described in section 4.2.4, except that coordinate adjustments are needed when the focus (the receiver) has buttons on it.

The third part is the actual coordinate transformations. The two transformation T_m and T_p discussed in Chapter 4 are combined into two methods: **-magnifyBy: projectFrom: into:**, which is applied for the receiver, and **-pushAwayBy: center: projectFrom: into:** for neighbors of the receiver.

5.3.3.2 Transformation on the focus

The method **-magnifyBy: projectFrom: into:** is only implemented in **TBox** and applied to the focuses during the recursion. The first argument of the method is used for magnifying the receiver's graphical size (as the magnification factor) and pushing the receiver's buttons (as the repulsive factor), the second and third arguments are rectangles whose sizes are used to compute the projection ratio. The method first sends message **GBox::-magnifyBy:** and **GBox::-projectFrom: into:** to the receiver's graphical object to let it transform its coordinates. These two methods are basic operations of **GBox**, the first one is a simple implementation using the Equation(1) given in section 4.2.2, and the second using perspective projection described in section 4.2.3. After transforming the receiver's graphical object, the "window" area of the receiver which other objects will be projected into may be moved by some distance, thus all dependents of the receiver should be moved by the same distance. This is implemented by the method **moveAllChildrenBy:**, which recursively move all box and arc children. Finally, a message **-pushAwayBy: center: projectFrom: into:** is sent to every buttons of the receiver so that they can transform their coordinates.

```
TBox :: -magnifyBy: aFactor projectFrom: wldArea into: winArea {
    /* local variable: oldCenter, newCenter, moved */
    oldCenter = self center;

    /* magnify self graphical size and then project it into the winArea */
    self graphical object perform -magnifyBy: aFactor;
    self graphical object perform -projectFrom: worldArea into: windowArea;

    /* reposition self children */
    newCenter = self center;
    moved = newCenter - oldCenter;
    self moveAllChildrenBy: moved;

    /* reposition self buttons */
    each of the self buttons perform
```

```

        -pushAwayBy: aFactor center: self projectFrom: worldArea to: windowArea.
    }

TBox :: -moveAllChildrenBy: aRelativePoint {
    if (self is open) {
        each of self children of TBox perform -moveBy: aRelativePoint.
        -moveAllChildrenBy: aRelativePoint.
        each of self children of TArc perform -moveBy: aRelativePoint.
    }
}

```

5.3.3.3 Transformation on other affected icons

The transformation method **-pushAwayBy: center: projectFrom: into:** is common to all nodes and arcs, each of which has its own implementation based on its graphical characteristics and logical associations.

Boxes: A box object handles this message by first invoking the *GBox* method **-pushAwayBy: center:**, which uses the Equation(II) (section 4.2.2) to compute and return a push distance. After moving itself and all its children by this distance, it executes another method **-projectFrom: into:** which recursively invokes the projection operation for all of its children.

```

TBox :: -pushAwayBy: aFactor center: aBox projectFrom: wldArea into: winArea {
    /** local variable: pushDistance */
    pushDistance = self graphical object perform -pushAwayBy: aFactor center: aBox,
    self perform -moveBy: pushDistance
    self perform -moveAllChildrenBy: pushDistance
    self perform -projectFrom: wldArea into: winArea,
}

TBox :: -projectFrom: wldArea into: winArea {
    self graphical object perform: -projectFrom: wldArea into: winArea,
    each of self graphical button perform -projectFrom: wldArea into: winArea,
    if (self is open) {
        newWorldArea := area of self children,
        newWindowArea := self screen area,
        each of self box child perform -projectFrom: wldArea into: winArea,
        each of self arc child perform -projectFrom: wldArea into: winArea,
    }
}

```

Arcs: *TArc* divides the implementation task of **-pushAwayBy: center: projectFrom: into:** into in two methods, **-pushAwayBy: center:** and **projectFrom: into:**. Each of them implements a simpler task, and both simply pass the same message to their graphical correspondents to complete the coordinate transformation.

2 OF/DE 2

PM-1 3 1/2"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

```

TArc: -pushAwayBy: aFactor center: aBox projectFrom: worldArea to: windowArea {
    self -pushAwayBy: aFactor center: aBox,
    self -projectForm: worldArea into: windowArea;
}

GArc: -pushAwayBy: aFactor center: aBox {
    for each point of the self except the two end points do {
        if (the point is not located in aBox) {
            compute the push distance by Equation(1) (section 4.2.2),
            move this point by the distance;
        }
    }
}

GArc :: -projectFrom: worldArea into: windowArea {
    for each point of self except the two end points do {
        compute the projection image of this point as described in section 4.2.3
        move this point to new image point
    }
}

```

Buttons: *TPushButton* needs a special treatment in its *-pushAwayBy: center: projectFrom: into:* method. Buttons must be attached to their boxes, therefore, after the normal “pushing” and projection transformations performed by its *-pushAwayBy: center:* and *projectFrom: into:*, the coordinates and size of the button need to be adjusted, which can be accomplished by a TimeBench method *GExternalButton::forceToEdge:*.

Timethreads: Logically, a timethread can be regarded as child of some box (or the top folder), if it anchors at its immediate child box(s). But it is impossible to put a timethread in a certain layer to fit into our implementation strategy. A timethread often goes through boxes of different layers and each step of the recursion will affect only a segment of the timethread. Letting a thread to find the affected segments and transform its coordinates many times would be inefficient. Therefore, we don’t require timethreads to handle its coordinate transformation, it overrides the method *-pushAwayBy: center: projectFrom: into:* by just returning. Instead, we let *TBox* to handle all transformations on timethreads. The implementation is simple: Every box object keeps a table storing its timethreads and the anchoring points (index) of the timethread. Before a focus box returns from *-magnifyBy: projectFrom: into:* or a neighboring box returns from *-pushAwayBy: center: projectFrom: into:*, they will send a message *-moveTo: pointNum:* to each timethread to move its anchoring position to a new point.

Others: *TThreadStart* and *TThreadEnd* objects handle the method *-pushAwayBy: center: projectFrom: into:* in a way similar to *TBox* objects except that they don’t have

any children. *TArcTail* objects transform their coordinates like *TPushButton* objects do.

5.3.3.4 The method *-projectInternals*

The method *-projectInternals* completes the coordinate transformation by projecting the children of the current focus onto the window area of the focus. These children will be added to the display list and shown on the screen during the next view update. To transform the coordinates of these children, the method implements a "perspective projection" process, which computes the projection ratio based on the receiver's graphical size and the union size of its children. In spite of this, this method also asks each of its box children to send two messages *-moveTo: pointNum:* and *-visible: pointNum:* to its timethread, so that the associations between the timethreads and these child boxes will also be shown in the next view updated. We conclude this section with the pseudo code of this method.

```
TBox :: -projectInternals {  
    /* compute the position and size of the windowArea */  
    if ( self has more than two children )  
        windowArea = self display area;  
    else windowArea = self display area adjustment;  
    /* compute the worldArea */  
    worldArea = the union of the areas of all self children;  
  
    for each of the self children {  
        perform -projectFrom: worldArea to: windowArea  
        if ( child is a TBox ) {  
            for each timethread of child perform  
                -moveTo: aDestPoint pointNum: indexNumber  
                -visible: true pointNum: indexNumber  
        }  
    }  
}
```

5.3.4 Updating the View

After the coordinate transformation, a message called *-fisheyeFocusAt:* is sent to the view object with the focus as the argument. This method is implemented as:

```
View :: -fisheyeFocusAt: aFocus {  
    add internal children of aFocus to the display list.  
    adjust display list  
    animate the opening of the focus  
    update display  
}
```

We know that TimeBench only displays icons belonging to the same level of the system hierarchy. For displaying icons of multiple levels as needed in the fisheye view, the viewing mechanism of TimeBench is modified. First, for presenting the fisheye view, the method *-fisheyeFocusAt:* adds all internal children of the current focus into the display list. If one of the children is selected as the next focus, its internal children will then be added to the display list. In this way, icons of multiple levels can be displayed if the user descends deeply into the system hierarchy. Second, in case of the whole display update, which will cause rebuilding of the display list, the method *View::-buildDisplayList:from:* is modified to recursively incorporate internal children of all opened boxes into the display list.

The second step of *-fisheyeFocusAt:* is to adjust the visibility and position of the timethread. Since icons in the display list are drawn by their storing order (first-in-first-drawn), newly added icons may be drawn on top of timethreads. To make the whole timethread visible, the display list is adjusted before updating the view. This is implemented in a method *View::-adjustDisplayList*, which scans over the display list and moves timethreads to the end of the list.

Another method *View::-animateFocusOpen* is also called before the view update. This will be discussed in section 5.4.3.

5.4 Some User Interface Issues

In this section, we describe some implementation efforts for the enhancement of our prototype user interface. Issues involved are: 1) response time to user events, 2) animation, and 3) color coding.

5.4.1 Response Time

Our experiments show that the SnakeTool prototype can manage real time response to a layered fisheye view transition which affects about 100 hundred icons (boxes, buttons and arcs). The response time is largely contributed by erasing and redrawing icons rather than computing the fisheye view transformations. Since only icons whose coordinates are changed will be redrawn in the next view update, the prototype response time to larger diagrams (of several hundreds of icons) is acceptable, assuming that the locking facility

will be used often for the sake of saving the screen space. When locking is applied, only icons in the locked box will be affected by the fisheye view transformation, while icons out of the locked box will neither be affected by the transformation nor be updated on the screen. Therefore, the performance of the prototype suffers not much when more icons are displayed.

However, the locking mechanism helps little to enhance performance when the timethread is involved. A timethread may wind around the whole display and go through icons both in and out of the locked box. When a focus is selected for the fisheye view formation, all affected boxes will send message to their timethreads informing them to change. Timethreads in turn will send the same message to all of their anchored boxes, even though these boxes may not be affected by the fisheye view transition. And further on, these boxes will pass the message to new boxes and timethreads connected to them. In some cases, the chain reaction will spread to all icons displayed on the screen.

One possible solution to this problem would be achieved by modifying the structure of *TThread*. In the current implementation, each *TBox* maintains a table storing the information of the timethreads anchoring at it and the corresponding anchoring points (by index). However, a timethread has to scan over its logical association list to search for an anchored box. A similar table could be added to *TThread* so that each timethread knows the specific anchored box corresponding to each of its points. Thus, when a box is changed, it sends messages to a timethread for updating the coordinates of the anchoring point. The timethread can check if the two adjacent points have any anchored boxes. If they do, the changed message will be passed only to the two “adjacent” boxes rather than to all anchored boxes. This would prevent the proliferation of marking all icons as changed.

5.4.2 Animating the Focus Opening

Early experience with SnakeTool shows that the display change caused by the fisheye view formation was so dramatic that the viewer often lost track of what happens, for example, which box was magnified to where and which boxes were squeezed to where. To help capturing the viewer’s visual focus and to keep track of the major changes in the display, a animation process is implemented, which starts from the focused object the viewer clicked and animates the opening of the box.

The animation is implemented in a method **-animateFocusOpenAt:** which is called after the coordinate transformations but before the view update. The simple implementation has produced a satisfactory smooth view transition. Following is the pseudo code of this method:

```

View :: -animateFocusOpenAt: aFocusBox {
    /* temp variables: oldBox newBox */
    if ( targetBox is NULL ) return;
    assign the graphical size of aFocusBox to oldBox;
    draw oldBox;
    while (oldBox is less than targetBox ) {
        newBox := oldBox increment
        draw newBox;
        erase oldBox
        oldBox := newBox
    }
}

```

5.4.3 Color Coding

Boxes and timethreads in SnakeTool are color coded. This serves for different purposes. First, all child boxes (subcomponents) of the same box are assigned a same color. The purpose is to help the grouping of objects so that it will be easier for the viewer to find the closely related visual targets[34]. This is based on an assumption that the viewer would also be interested in things close to his focus than things remote. Currently, color is automatically assigned to the boxes when they become visible and a box keeps its color throughout its life span. When a box opens, it draws a thick boundary of its color but leaves a blank face so that its internal children can be displayed.

Second, color is used to distinguish timethreads. Timethreads are colored differently, which helps the viewer to trace along his interested timethreads. Due to the importance of the timethread in capturing design patterns, more exploration with color needs to be conducted for enhancing the presentation of timethreads. For example, timethreads may change their color during display to reflect its significance in the system, and they may also flash to reflect that they have passed some important component.

Third, color is used to help animations. Our simple animation of the focus opening has already shown the power of animation with color. And this is an area whose vast potential has yet to be explored in SnakeTool.

Chapter 6 Conclusion

6.1 Summary

Large software systems are complex and difficult to comprehend. To facilitate the learning and understanding tasks of large programs, recent research has been directed to visualization, which uses computer graphics to present the structure and behaviour of large software systems and thereby to enhance their understanding. The objective of this thesis work is to develop new techniques for the visualization of large software systems, and to build a prototype system to experiment with the new techniques.

Related work in a wide range of fields has been examined during this research. Of particular interests are a number of representative tools in the areas of program visualization and information visualization. While program visualization tools focus on the depiction of small program at the detailed level of statement and data structure, information visualization tools concern the illustration of the structural patterns of large data collections at a high level of abstraction. However, for the visualization of large software systems, the techniques developed with these tools are far too inadequate. This is largely attributed to three prominent problems: the abstraction problem, the context problem and the space problem.

Motivated by the lack of effective visualization techniques and the practical need of visualization tools to enhance the presentation and understanding of the large DMS system, we have introduced the layered fisheye view technique, and have developed the SnakeTool prototype to demonstrate the potentials of this new technique in visualizing large software systems.

SnakeTool represents software systems at a high level of abstraction, the level of components and timethreads. It presents nesting-diagram-like pictures to illustrate the system structure, and uses timethreads to depict the system behaviour. We think that this abstraction is an appropriate one because it helps to preserve conspicuous characteristics of large software systems by filtering out detailed information and because it is consistent to

the natural way by which designers think and understand large software systems.

To tackle the context and space problems, the layered fisheye view technique has been developed. In layered fisheye views, components of interest are expanded and exposed to the level of detail the user desires, and the less interesting components are shrunk and shown for the only purpose of preserving context. Furthermore, the layered fisheye view technique and the box-and-timethread diagram work together to ameliorate the screen space contention. The high level abstraction is an implicit filter for information reduction so that only those salient aspects of the target system may be displayed. The layered fisheye view technique gives the user the control over what part and how detail of these aspects to be displayed.

As the primary view provided in SnakeTool, the layered fisheye view becomes the most significant components of this thesis. This technique is novel in terms of its graphical layout and its interesting application domain. Also comparing with previous fisheye view work, the layered fisheye view allows multiple focuses so that the user can explore multiple components of his interest simultaneously. Furthermore, examples of using SnakeTool to explore software systems show that the layered fisheye view technique is effective and promising.

The two significant transformations for the layered fisheye view formation are: non-linear magnification and perspective projection. Non linear magnification applies to the focus first and expands the focus by an appropriate scale factor, it then applies to the surrounding neighbors to push them away. The neighbors are repulsed as they were in a magnetic field. This results in an overall enlarged picture. The perspective projection is applied to make the enlarged picture fit into the available display area on the screen. The layered fisheye view at a user specified level is an integration of a series of fisheye view transformations at different levels.

Locking is an significant mechanism of SnakeTool. Intuitively, it is necessary to control the display space because of the small screen space and the necessity of preserving context, and locking satisfies these needs. Locking also facilitates performance, since only those objects in the locked box need to be redrawn for the display update.

To incorporate SnakeTool functionalities into TimeBench framework, our main

implementation efforts are directed to the defining and coding of timethread and node related classes or methods. Implementation techniques are also excised for applying the generic layering fisheye view transformation algorithm to the box-and-timethread diagram. Also, some user interface issues about the prototype performance, color and animation are discussed, and some simple but effective implementations are worked out.

6.2 Conclusions and Future Work

The prototype system SnakeTool presented in this thesis is developed as an experimental vehicle to explore practical techniques for the visualization of large software systems. It presents the hierarchically structured large software system and the system timethread behaviour in a novel graphical layout, the layered fisheye view. And the results are interesting and promising. In addition, since flexibility and extensibility are the central issue during design, new viewing techniques can be readily integrated. The SnakeTool provides a solid starting point for the development of a practical visualization tool.

The layered-fisheye-view is a new technique of visualizing large systems. It presents the structure information of the viewer's interest in great detail and the surrounding context in less detail. It also provides more depiction on the behaviour of the viewer's interested components than that of the less interested ones. Comparing with the viewing techniques summarized in Chapter 2, its advantages are considered as follows:

- Information reduction is automatically achieved because only those detailed information of the user's interest is displayed while those less interesting details are hidden.
- It provides a solution, to some extent, to the context problem. The interesting components can remain fully expanded to allow detailed observation and exploration, whereas the less interesting components can become smaller and smaller but still preserve the context.
- It allows multiple focuses so that the user can explore multiple components simultaneously.

- It allows interactive manipulation. The user controls what and how to display by manipulating both the individual objects and the fisheye views generated at different levels.
- It supports fast view transition by the locking facility.

In this thesis work, the SnakeTool prototype is developed to explore and demonstrate new ideas and to illustrate the potentials of the layered fisheye view, rather than to be a production tool. For the ongoing SnakeTool project and future research on large software system visualization, a number of issues need further investigation. These issues are summarized below:

Multiple Views: To help to fully explore a large software system, other views should be added to the SnakeTool so that different aspects of the program can be illustrated. Furthermore, a sensible combination of different views would generate more effective results. For example, a 3D tree structure view such as the Cone Tree would help the user get a picture of the whole system structure and would make the navigation task easier; a text display view could be added so that the user could see the documents corresponding to the current interested component(module). This would put the textual documents in graphical context and therefore enhance the understanding.

3D: 3D provides much more space than the 2D flat screen and is more natural to human visual perception, thus will produce the most promising techniques for large software system visualization. Given the advanced graphic station nowadays, many interested ideas can be explored using 3D. For example, a 3D fisheye view may not need to hide subcomponents, but instead, put them deep "inside" the screen, or when the user selected a focus, he could rotate it to the front, and inside components could be pulled out; a 3D bird's eye view could provide an impressive structure outline of the system.

Behaviour Abstraction: The timethread is a high level of abstraction, which is designed to illustrate the macroscopic behaviour patterns of a large software, not the fine details. To meet the different needs of the SnakeTool users, other kinds of established low-level behaviour representation techniques could be integrated so that the low level behaviors of large software systems would be depicted.

Scaling: The current prototype has been used for demonstration. It is essential that the

production SnakeTool tool be able to cope with targets containing many hundreds or even thousands of components. This may result in overly cluttered layered fisheye views. Mechanisms interacting with the fisheye view would help to tackle this scaling problem. For example, a grouping facility which allows the user to create a meta component out of those less interested components would save more screen space; for the same purpose, when the user digs into some very low level for detail, the boxes opened at the higher levels could be allowed to merge into a simple outline.

Screen Layout: As we saw in section 5.3.1, the initial screen layout of boxes and timethreads is influenced by a number of different factors. To avoid excessive screen clutter, related components should be put in close proximity, but other aspects such as clearly presented timethreads should also be taken into consideration. The ways in which these factors interact and possibilities for efficient implementations together present a number of difficult problems for an automated initial layout algorithm design.

Animation: Since the timethread presents the dynamic behaviour, it would be best received if animation is used.

Fisheye View Restore: Because the layered fisheye view is aimed to display large amount of graphical icons, one view transition may affect too many objects for an undo stack to handle. The situation will get even worse if the user interactively changed the display to explore something and then asked for going back to one level up. Both a better algorithm and a skillful implementation are needed to meet restore layered fisheye views.

References

- [1] R.M. Baecker. "Experiments in On-Line Graphical Debugging: The Interrogation of Complex Data Structures," First Hawaii International Conference on the System Sciences, Jan. 1968, pp. 128-129.
- [2] R.M. Baecker. "Two Systems which Produce Animated Representations of the Execution of Computer Programs," SIGCSE Bulletin, 7(1) Feb. 1975, pp. 158-167.
- [3] Ronald Baecker and Aaron Marcus. "Design Principles for the Enhanced Presentation of Computer Program Source Text," Human Factors in Computing Systems: Proceedings SIGCHI'86, Boston, MA, Apr. 1986, pp 51- 58.
- [4] R. M. Baecker and Aaron Marcus. "Human Factors and Typography for More Readable Programs", ACM press, Addison-Wesley, 1990.
- [5] H.D. Boecker and H. Nieper, "The Enhancement of Understanding through Visual Representations," Proceedings of ACM CHI'86, Conference on Human Factors in Computing Systems, 1986, pp 44-50.
- [6] H.D. Boecker, H. Nieper, "Making the Invisible visible: Tools for Exploratory Programming," Proceedings of the First Pan Pacific Computer Conference, Melbourne, Australia, Sep. 1985.
- [7] F. P. Brooks, Jr. "No Silver Bullet: Essence and Accidents of Software Engineering," IEEE Computer, 20(4), April, 1987, pp. 10-19.
- [8] Gretchen P. Brown, Richard T. Carling, Christopher F. Herot, David A. Kramlich and Paul Souza. "Program Visualization: Graphical Support for Software Development," IEEE Computer, 18(8) Aug. 1985, pp. 27-35.
- [9] R.J.A. Buhr. "Practical Visual Techniques in System Design (with Applications to Ada)," Englewood Cliffs, NJ: Prentice-Hall, 1990.
- [10] R.J.A. Buhr and R.S. Casselman. "Architectures with Pictures," OOPSLA'92, 1992
- [11] R.J.A Buhr. "Pictures That Play: Design Notations for Real-time & Distributed Systems," Technical Report SCE 93-04, Department of System and Computer Engineering, Carleton University.
- [12] E. J. Cameron and D.M. Cohen. "The IC* System for Debugging Parallel Programs via Interactive Monitoring and Control," In Proc. of Workshop on Parallel and Distributed Debugging, Jan 1989, pp. 261-270.
- [13] Stuart K. Card, George G. Robertson, Jock D. Mackinlay. "The Information Visualizer, An Information Workspace, " Proceedings of ACM CHI'91, Conference on Human Factors in Computing Systems, Information Visualization, 1991, pp 181-188.
- [14] W. Cunningham and K. Beck. "A Diagram for Object-Oriented Programs," OOPSLA'86 Proceedings, pp 361-367

- [15] A. Disessa, "A principled design for an integrated computational environment," *Human-Computer Interaction* 1, 1, January, 1985, pp1-47.
- [16] K.M. Fairchild, S.E. Poltrock and G.W. Furnas. "SemNet: three-dimensional graphic representations of large knowledge bases," In *Cognitive science and its applications for human-computer interaction*, R. Guindon(ed), Lawrence Erlbaum, 1988.
- [17] S.S. Fisher, M. McGreevy, J. Humphries, and W. Robinett. "Virtual environment display system," *Proceedings of the 1986 Workshop on Interactive 3D Graphics*.
- [18] George W. Furnas. "Generalized fisheye views," *Proceedings of ACM CHI'86, Conference on Human Factors in Computing Systems*, 1986. pp 16-23.
- [19] L. M. Haibt. "A Program to Draw Multi-Level Flow Charts," *Proceedings of the Western Joint Computer Conference*. San Francisco, CA. 15 Mar. 1959. pp 131-137
- [20] D.A. Henderson and S.K. Card. Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical interface. *ACM Transactions on Graphics*, 5, 3, 211-243, July, 1986.
- [21] C. F. Herot, R. Carling, M. Friedell, and D. Kramlich. "Spatial management of data," *ACM Trans. Database Syst.* December, 1980. pp 493-514.
- [22] Brian Johnson and Ben Shneiderman. "Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures," *Proceedings of IEEE Visualization' 91*. San Diego, CA. Oct. 1991. pp 284-291
- [23] G.M. Karam, R.J.A. Buhr, C.M. Woodside, and R. Casselman. "TimeBench: A Next Generation CASE Tool," TR CU-SCE-89-18, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1989.
- [24] Linsay M. MacDonald. "Using Color effectively in displays for computer-human interface," *DISPLAYS*, pp129-142, July 1990.
- [25] Jock D. Mackinlay, George G. Robertson, and Stuart K. Card. "The perspective Wall: Detail and Context Smoothly Integrated," *Proceedings of ACM CHI'91, Conference on Human Factors in Computing Systems, Information Visualization*, 1991. pp 173-179.
- [26] J. Martin and C. MacClure. "Diagramming Techniques for Analysts and Programmer," Englewood Cliffs, N.J.: Prentice-Hall, 1985.
- [27] T.G. Moher. "PROVIDE: A Process Visualization and Debugging Environment," *IEEE Transactions on Software Engineering*. Vol. 14, No. 6. June, 1988. pp. 849-857.
- [28] M. Moriconi and D. F. Hare. "Visualizing Program Designs Through PegaSys," *IEEE Computer*, 18(8) August, 1985. pp. 72-85.
- [29] H.A. Muller and K. Klashinsky. "Rigi - A System for Programming-in-the large, " *Proceedings of the 10th International Conference on Software Engineering(ICSE)*, Raffles City, Singapore, April 1988. pp. 80-86.

- [30] B.A. Myers. "Incense. A System for Displaying Data Structures." Computer Graphics: SIGGRAPH' 83 Conference Proceedings. 17(3) July 1983. pp 115-125.
- [31] Brad A. Myers. "The State of the Art in Visual Programming and Program Visualization," Graphics Tools for Software Engineers, Cambridge University Press, 1989. pp 3-26
- [32] G. Raeder. "A Survey of Current Graphical Programming Techniques," IEEE Computer, 18(8). August, 1985. pp. 11-23
- [33] S.T. Reiss. "PECAN: Program Development Systems that Support Multiple Views," IEEE Transactions on Software Engineering, Vol. SE-11, No.3, March 1985. pp. 276-285.
- [34] John F. Rice. "Ten rules for color coding," Information Display, 7(3): 12-14, March 1991.
- [35] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. "Cone trees: Animated 3d visualizations of hierarchical information," Proceedings of ACM CHI'91, Conference on Human Factors in Computing Systems, Information Visualization, 1991. pp 189-194.
- [36] Manojit Sarkar and Marc H. Brown. "Graphical Fisheye Views of graphs," Proceedings of ACM CHI'92, Conference on Human Factors in Computing Systems, 1992. pp 83-91.
- [37] R. Spence and M. Apperley. "Data base navigation: An office environment for the professional," Behaviour and Information Technology 1(1). 1982. pp 43-54.
- [38] I.E. Sutherland. "Sketchpad: A man-machine graphical communication system," AFIPS Spring Joint Computer Conference, vol. 23, 1963, pp. 329-346.
- [39] W. Teitelman. "A Tour Through Cedar," IEEE Software, Vol. 1, No. 2, April 1984. pp. 44-73.
- [40] Real-Time and Distributed System Group, Carleton University, Ottawa, Canada. "User's Guide for TimeBench, Oct. 1990.
- [41] Real-Time and Distributed System Group, Carleton University, Ottawa, Canada. "An Overview of TimeBench Software", Jan. 1990.

Appendix A:

This appendix gives the TimeBench Class Hierarchy with the Extended Classes of SnakeTool.

A1: Class Hierarchy of ToolComponent Classes

```

-Object
  -ToolComponent
    -TArc
      -TStructureArc
        -TACMArc
        -TChoiceFrame
        -TGenericArc
        -TInstallArc
        -TJabArc
        -TRemoveArc
        -TThread
      -TComment
      -TNode
        -TDeskTop
        -TModelNode
          -TClipboard
            -TGarbageBag
          -TFileCabinet
          -TSystem
            -TFolder
            -TLibrary
            -TSpecFolder
          -TTrashCan
        -TStructureNode
          -TACM
          -TACMState ... (Collapsed)
          -TAssociateNode ... (Collapsed)
          -TThreadStart
          -TThreadEnd
        -TBox
        -TConnectionPoint
        -TDelay
        -TExtIntStructureNode
          -TArcTail
          -TGate
          -TPushButton
            -TExternalPushButton
            -TInternalPushButton
        -TStore ... (Collapsed)
          -TWaitingPlace
        -TRobot ... (Collapsed)
        -TStaticScroll
    
```

A2: Class Hierarchy of Logical Classes

- Object
 - LogicalObject
 - LMachineChart
 - LArc
 - LACMArc
 - LACMPushArc
 - LChoiceFrame
 - LDelayArc
 - LInstallArc
 - LJabArc
 - LPullArc
 - LRemoveArc
 - LRendezvousArc
 - LSimplePushArc
 - LThread
 - LComment
 - LDataFlow
 - LFileCabinet
 - LNode
 - LACMState ... (Collapsed)
 - LArcTail
 - LConnectionPoint
 - LMCLCodeNode
 - LACMCode
 - LEngine ... (Collapsed)
 - LStore ... (Collapsed)
 - LNodeWithExterns
 - LNodeWithInterns
 - LClipboard
 - LDeskTop
 - LNodeExtInts
 - LBox
 - LButton
 - LGate
 - LPushButton
 - LExtPushButton
 - LIntPushButton
 - LRobot ... (Collapsed)
 - LSystemRootNode ... (Collapsed)
 - LTrashcan
 - LScroll
 - LTimeOut ... (Collapsed)
 - LThreadEnd
 - LThreadStart
 - LWaitingPlace

A3: Class Hierarchy of Graphical Classes

- Object
 - Arc
 - GArc
 - GGenericArc
 - GInstallArc ... (Collapsed)
 - GJabArc
 - GPullArc
 - GThread
 - GNil
 - Point
 - ExternalForm
 - Rectangle
 - Form
 - BlockingForm
 - CommentForm
 - SMArcForm
 - SRVNForm
 - GNode
 - GACM
 - GACMStateEnd
 - GACMStateStart
 - GBox
 - GDateFlow
 - ExternalButton
 - GExternalGate
 - GExternalPushButton
 - GInternalButton ... (Collapsed)
 - GModelNode
 - GClipboard
 - GDesktop
 - GFileCabinet
 - GFolder ... (Collapsed)
 - GGarbageBag
 - GTrashCan
 - GReactiveRobot
 - GRobot ... (Collapsed)
 - GScroll
 - GSquare
 - GArcTail
 - GExternalArcTail
 - GInternalArcTail
 - GThreadEnd
 - GCircle
 - GConnectionPoint
 - GThreadStart
 - GTimeOut
 - GWaitingPlace
 - GEngine
 - GStore ... (Collapsed)

END

14-12-93

FIN