INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

SOVIST-3D: Three-Dimensional Software Visualization in Smalltalk

By

Robert E. Tyson, B. Math

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment 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

September 1998

© Copyright
1998, Robert E. Tyson, B. Math

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

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

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

0-612-36881-5

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

SOVIST-3D: Three-Dimensional Software Visualization in Smalltalk

submitted by

Robert E. Tyson, B. Math

Dr. Evangelos Kranakis
(Director, School of Computer Science)

Prof. John Pugh
(Thesis Supervisor)

Carleton University
September 1998

Abstract

This thesis discusses the use of 3D software visualizations in Smalltalk. To explore the use of the 3D software visualizations, SOVIST-3D (Three-Dimensional software visualization in Smalltalk) was developed. SOVIST-3D is an extendable framework that creates the software visualizations without any user input, except for what they want to visualize. Each of the software visualizations created by SOVIST-3D is created by using a viewer. SOVIST-3D currently has three different viewers: hierarchy, which displays the object hierarchy; parts, which displays the parts of an object returned from the execution of a use case; and message tracer, which displays a trace of the interesting messages sent in the system. Each viewer can have the visualization presented in different ways. Each of these presentations are called a layout. SOVIST-3D currently utilizes four different layouts which results in the creation of seven different types of visualizations.

Acknowledgments

I would like to thank my supervisor, Prof. John Pugh, for his help in completing my thesis.

I would like to thank Mike Valenta for his work on the OpenGL implementation. Without it the 3D diagrams would not be possible.

I would like to thank Dorin Sandu for answering all my nagging little questions.

I would like to thank my Mother for proofreading this thesis.

I would like to thank Mike Valenta, Mike Just, Darcy Quesnel, Jason Morrison, Prof. Oommen, Prof. Deugo and Prof. Pugh who made it possible for me to finish this thesis well being in Switzerland.

In addition, I would like to thank all of the faculty and students who have been associated with Carleton University's Centre for Object-Oriented Programming. Their suggestions were quite helpful.

Trademarks

OpenGL is a trademark of Silicon Graphics, Inc.

Silicon Graphics is a trademark of Silicon Graphics, Inc.

Smalltalk/V is a registered trademark of ParcPlace/Digitalk Inc.

Visual Smalltalk is a registered trademark of ParcPlace/Digitalk Inc.

Window Builder Pro is a registered trademark of Object Share Systems Inc.

Windows 95 is a registered trademark of Microsoft Corporation.

Windows NT is a registered trademark of Microsoft Corporation.

Other products may be trademarks of their respective companies.

Contents

Abstract	iii
Acknowledgements	iv
Trademarks	v
1 Introduction	1
1.1 Intended User	2
1.2 Software Visualization	2
1.3 Goals of the thesis	5
1.3.1 Users goals	5
1.3.2 Design goals	6
1.4 Thesis Outline	6
2 Motivation	8
2.1 Why Visualize?	8
2.1.1 Assisting computer instruction	11
2.1.2 The Cost of Maintenance	13
2.1.3 Detractors	17
2.1.4 Summary – Why Visualize?	17
2.2 Moving From 2D to 3D	18

2.2.1	Disadvantages of 3D	18
2.2.2	Advantages of 3D	19
2.2.3	Empirical evidence	25
2.2.4	Summary – Moving from 2D to 3D	26
2.3	Summary	26
3	Related Work	27
3.1	Related Work in 2D	31
3.1.1	Algorithm Animation in 2D	31
3.1.2	Large Software Systems	35
3.1.3	Two Dimensional – Object-Oriented	37
3.1.4	2D other	41
3.2	Related Work in 3D	42
3.2.1	Algorithm Animation in 3D	42
3.2.2	Information Visualizer	44
3.2.3	Object-Oriented	46
3.2.4	Other systems	49
3.3	Summary	56
3.3.1	What was learned from the related work	56
3.3.2	How SOVIST-3D differs from the related work	57
4	SOVIST-3D	59
4.1	What is SOVIST-3D?	59
4.2	The Viewers	60
4.2.1	‘Look-and-Feel’	61
4.2.2	Hierarchy Viewer	62
4.2.3	Parts Viewer	64
4.2.4	Message Tracer Viewer	65

4.3	Layouts	69
4.3.1	Pickable Layouts	69
4.3.2	Non-Pickable Layouts	71
4.4	Understanding the Objects in the 3D Visualizations.	72
4.5	Navigation in a viewer	74
4.6	Summary	76
5	Design and Implementation	77
5.1	The Framework	77
5.1.1	Overview of the Architecture	78
5.1.2	Framework Design goals	79
5.1.3	Environment	80
5.1.4	Automatic Visualizations	82
5.1.5	Dynamic and Static Visualizations	82
5.1.6	Extendable framework	83
5.1.7	Viewers	83
5.1.8	The Layouts	88
5.2	Visualizations	92
5.2.1	The Individual Viewers	92
5.2.2	Layouts	97
5.3	Summary	99
6	Conclusions	100
6.1	Assessment of Work Done	100
6.1.1	Framework	101
6.1.2	The Visualizations within SOVIST-3D	103
6.1.3	Summary of Work Done	107
6.2	Limitations of SOVIST-3D	107

6.3	Future Work	107
6.3.1	3D Objects	108
6.3.2	Improving the Current Implementation	108
6.3.3	Adding Layouts	111
6.3.4	Adding new viewers	112
A	Definition of a Use Case	113
B	SOVIST-3D Users' Manual	115
B.1	Common Framework	116
B.1.1	3D Object Associated with a Class	116
B.1.2	Panes Contained in the Viewers	117
B.1.3	Menus	118
B.1.4	Using the Mouse	121
B.1.5	Transforming Objects	125
B.1.6	Translation Controls	127
B.1.7	Transforming the Scene	129
B.2	Message Tracer Viewer	130
B.2.1	Message Tracer Viewer – Viewer Panes	131
B.2.2	Message Tracer Viewer – Viewer Menu	133
B.2.3	Message Tracer Viewer Popup Menus	134
B.2.4	Message Tracer Filters	135
B.2.5	Message Tracer Layouts	136
B.2.6	Sequence Diagram Layout	137
B.3	Hierarchy Viewer	138
B.3.1	Hierarchy Viewer Menu	138
B.3.2	Hierarchy Viewer Popup Menus	138
B.3.3	Hierarchy Viewer Layouts	141

B.4	Parts Viewer	142
B.4.1	Parts Viewer – Viewer Menu	142
B.4.2	Parts Viewer Popup Menus	142
B.4.3	Parts Viewer Layouts	146
C	Colour Plates	147
D	TestCases’ Source Code	158
D.1	Bank Account Code	158
D.2	DepthCylinderExample Code	159
	Glossary	168
	Bibliography	171

List of Tables

2.1	Maintenance Costs	14
3.1	Related Work	29
3.2	Related Work	30
5.1	Class Methods for Viewers	84
5.2	Methods for Viewers	85
5.3	All Class Layout Methods	90
5.4	All Layout Methods	90
5.5	Class PickableLayout Methods	92
5.6	Layout Methods Pickable Layouts	93
5.7	Hierarchy Viewer Layout Methods	94
5.8	Parts Viewer Layout Methods	95
5.9	Message Tracer Class Layout Methods	95
5.10	Message Tracer Layout Methods	96
B.1	Help Keys	122
B.2	Mouse Buttons Usage	123
B.3	Filter example1	137
B.4	Filter example2	137

List of Figures

1.1	Software Visualization	4
2.1	Bubble-sort bar chart	20
2.2	3D Particle Simulation	21
2.3	Shortest Path	22
2.4	Multiple Views	24
3.1	Class Hierarchies	51
3.2	Version Control	53
3.3	Re-rooting	55
4.1	A 3D Viewer.	62
4.2	A hierarchy visualization	63
4.3	A parts visualization	64
4.4	Sequence Diagram	66
4.5	BankAccount 's ThreeDObject.	73
5.1	Viewer Classes	83
5.2	ThreeDViewer Aggregates	84
5.3	Layout Classes	89
5.4	Viewer Layout Aggregates	89
5.5	PickableLayout Aggregates	92

5.6	Hierarchy3DViewer Aggregates	93
5.7	Parts3DViewer Aggregates	94
5.8	MessageTracer3DViewer Aggregates	96
5.9	ThreeDDiagram Aggregates	97
5.10	SequenceDiagramLayout Aggregates	98
5.11	Trees Aggregates	98
5.12	Trees Factory Method	99
B.1	Viewer Design	117
B.2	Viewer Menus	118
B.3	Translation Object Dialog Box	126
B.4	Translation Controls Dialog Box	127
B.5	Transform Scene Dialog Box	130
B.6	Message Tracer Viewer Design	132
B.7	Filter Dialog Box	136
B.8	Add Class Dialog Box	139
B.9	Add Object Dialog Box	143
B.10	Show Parts Dialog Box	145
C.1	Cone Trees	147
C.2	History of QuickSort	148
C.3	BankAccount example2	149
C.4	BankAccount stepping	150
C.5	BankAccount example1	151
C.6	DepthCylinderExample 1	152
C.7	DepthCylinderExample 2	152
C.8	DepthCylinderExample 3	153
C.9	DepthCylinderExample 4	153

C.10 Parts viewer	154
C.11 Parts viewer2	155
C.12 Hierarchy viewer	156
C.13 Hierarchy viewer2	157

List of Program Listings

2.1	Polymorphism example	15
4.1	Bank Account Use Case 1	67
4.2	BankAccount threeDObject method	73
4.3	Magnitude threeDObject: method	74
5.1	The runLayoutViewerItem: Method	87
5.2	Initialize Layout Method for the Hierarchy Viewer	91
D.1	Bank Account Definition	159
D.2	Bank Account Class Methods	160
D.3	Bank Account Instance Methods	161
D.4	Transaction Classes' Definitions	162
D.5	Transaction Classes Methods	162
D.6	Transaction Instance Methods	163
D.7	CheckingTransaction Instance Methods	164
D.8	DepositTransaction Instance Methods	164
D.9	WithdrawalTransaction Instance Methods	164
D.10	DepthCylinderExample Classes' Definitions	165
D.11	DepthCylinderExample Classes' Class Methods	166
D.12	DepthCylinderExample Classes' Instance Methods	167

Chapter 1

Introduction

Software, especially object-oriented software, is complicated. Items such as inheritance, polymorphism and dynamic binding makes understanding Smalltalk programs difficult. Anything that aids in the understanding of such software is desirable.

Using graphical techniques to aid the understanding of the programs is what the rapidly growing field of software visualization encompasses¹. Most of the work in software visualization to date has been in 2D. Recently, with hardware being available relatively cheaply, software visualization has ventured into creating 3D visualizations. This is because the third dimension offers more degrees of freedom to convey information.

This thesis shows the design and implementation of an extendable framework for creating 3D software visualizations for Smalltalk called SOVIST-3D (**S**oftware **V**isualization in Smalltalk in 3D.)

Using SOVIST-3D, the visualizations can be created without any additional user input, i.e. the user does not have to annotate the code to create the visualizations. 3D models can be added to represent objects/classes in Smalltalk by implementing class methods for the object. The model can be made general for each class or specific for

¹Software visualization is discussed in detail in section 1.2.

each instance of the class. Normal inheritance rules apply to the methods creating the models.

1.1 Intended User

The visualizations that SOVIST-3D creates are primarily intended to be used to help people, (mostly programmers), understand existing code. This encompasses such things as:

- Showing how some code works to another person.
- Trying to figure out code so that it can be modified.
- Teaching someone object-oriented or programming principles.

SOVIST-3D can also be used to help a developer to debug his/her code. It is not as useful for finding syntax errors as it is for finding logic problems.

1.2 Software Visualization

Software is intangible, having no physical shape or size. After it is written, code ‘disappears’ into files kept on disks. *Software visualization* tools use graphical techniques to make software visible through the display of programs, program artifacts, and program behaviour.

Before any further discussions can be carried out the term *software visualization* must be defined. This section defines the term².

Throughout the literature, terms like *visual programming*, *program visualization* and *algorithm animation* are frequently mentioned but not always clearly defined. People

²This section is basically repeating the definitions given in [87, Section 2]. Unless otherwise mentioned all definitions given in this section are taken directly from the article.

often use one term (especially program visualization) for both the specific and the general case.

Visualization, because it contains the root word *visual*, is often considered to refer to visual images only. In fact, in the Oxford English Dictionary [105, page 700], its primary meaning is ‘the power or process of forming a mental picture or vision of something not actually present to the sight’. This means that a visualization can be a result of input from any of the five senses. *Program visualization* is the use of various techniques to enhance the human understanding of computer programs, while *visual programming* is the use of ‘visual’ techniques to *specify* a program. *Algorithm animation* (or *visualization*) is understood to be the visualization of a high-level description of a piece of software, which is in contrast to *code* or data visualization (which are collectively a kind of program visualization) where the actual implemented code is visualized.

Price *et al.* came up with the term *software visualization* to include all of the above terms and to remove the ambiguity of the terms. Thus, software visualization, covers all of the software design process from planning to implementation. A formal definition of *software visualization* is: the use of the crafts of typography, graphic design, animation and cinematography with modern human–computer interaction technology to facilitate both the human understanding and effective use of computer software. This thesis is not concerned with visual programming since this is an unintentional use of software visualization. Figure 1.1 shows how each of the terms in the literature fit together.

Put in another way there are three basic properties of software that can be visualized [5, page 33].

Software structure This shows the static nature of the code, an example being object hierarchies.

Runtime behaviour This covers the areas of algorithm visualization, code and data animation. Examples include: algorithm animations, sequence diagrams.

Figure 1.1: Venn diagram for each of the terms in the Software Visualization literature (the size of each area is not relevant and for simplicity the only intersection shown is that for visual programming) [87, Figure 1, page 213].

The code itself . such as pretty printers, which usually indent the code and use special fonts or colours to distinguish keywords and so forth, are a basic. widely used form of visualization [2].

1.3 Goals of the thesis

The main goal of this thesis is to move software visualization in Smalltalk into 3D. We want to do this because:

- software visualization can help users better understand existing code (see 2.1).
- using 3D we get an extra degree of freedom to create visualizations that can be better understood (see 2.2).

The way in which we intend to move software visualization in Smalltalk into 3D is via the creation of an extendable framework which easily allows the creation of various types of 3D visualizations. This framework (and the visualizations already created) are called SOVIST-3D.

There are two major types of goals of SOVIST-3D. The first dealing with using the system, the second, with the design of the system. These are discussed in more detail below.

1.3.1 Users goals

These are the main goals of the framework with respect to the users of SOVIST-3D. How these goals are met are discussed in more detail in Chapter 4.

The users goals are:

1. to create multiple types of visualizations (for example, the object hierarchy and the results of a message trace) that have the same ‘look-and-feel’.

2. to display the same information in multiple ways (for example, the object hierarchy as B-Trees or cone trees).
3. to allow different methods of identifying objects in the visualizations.
4. to allow the users to navigate the visualizations (in other words, allow the visualizations to be manipulated to increase understanding).

1.3.2 Design goals

These are the main goals of the design of SOVIST-3D. How these goals are met are discussed in more detail in Chapter 5.

The design goals are:

1. to create the visualizations in the same environment as the code to allow for interaction between the two.
2. to automatically create the visualizations (i.e. no annotation of the code is needed).
3. to visualize both the static and dynamic aspects of the code.
4. to make it easy for new viewers and layouts to be added to SOVIST-3D.

1.4 Thesis Outline

The remainder of this thesis is organized as follows: The next chapter presents our motivation for creating 3D software visualizations. Chapter 3 discusses related work. Chapter 4 introduces SOVIST-3D and explores the user goals. Chapter 5 contains details of the design and implementation of SOVIST-3D and explores the design goals. Finally, chapter 6 provides conclusions and suggestions for future work.

In addition, there are four appendices, a glossary and the bibliography. Appendix A contains the definition of use cases that is used throughout the thesis. Appendix B contains a small users' manual on how to use SOVIST-3D. Appendix C contains the colour figures referred to throughout the thesis. Appendix D contains the source code of the test cases used throughout the thesis.

Chapter 2

Motivation

This chapter provides the motivation behind creating 3D software visualizations in Smalltalk. It is broken down into two sections. The first describes why we want to visualize, the second, why we want to use 3D.

2.1 Why Visualize?

Everyone has heard the following saying: ‘A picture is worth a thousand words.’ The truth behind this saying is that the human mind is visually oriented. ‘People acquire information at a significantly higher rate by discovering graphical relationships in complex pictures than by reading text’ [90, page 12]. Throughout the ages designers have usually graphically sketch their design before proceeding to symbolic or language-oriented expressions.

There are several reasons that pictures are better than text¹:

Random vs. sequential access. The human eye can jump around to various parts of a picture, quickly taking in the picture as a whole. Text, on the other hand, is inherently sequential.

¹Points taken from [90].

Dimensions of expression. Text is a one-dimensional stream while pictures allow three-dimensions in which to lay out information.

Transfer rate. The human sensory system is ‘hard-wired’ for images. It can process the picture information at a far greater speed than text.

Concrete vs. abstract. Pictures allow the use of objects from the real-world to illustrate abstract ideas, since experience with the objects can be brought to bear in understanding the picture.

Pictures without names. Objects in pictures can be referred to by pointing. To refer to an object in text it has to be labelled with a name. The name gives an indirect reference to the object.

Real-world pictures. Pictures reflect the real world, whereas text can only point to the real world.

Animated pictures. Pictures can be animated to produce a sense of change over time. Text is static.

Metaphorically rich pictures. Pictures provide a large base of graphical metaphors that make them easier to understand.

This is essentially what software visualization is trying to do, to provide pictures that will enable code to be understood more easily than by just looking at the text. The field of scientific visualization has demonstrated repeatedly that the most effective way to present large volumes of data to users is in a continuous visual fashion [117, 83].

Bocker and Herczeg provide a good argument as to why software visualization is needed.

Today, constructing or analysing complex systems, as for example electronic devices, is only possible with the help of various tools and instruments, e.g. when troubleshooting an electronic device an engineer uses different measuring instruments, like oscilloscopes or frequency spectrum

analysers, which are sophisticated electronics systems themselves. These kinds of measuring instruments are most useful when they are connected to the system while it is in operation. They give the engineer insight into internal processes of the system which are normally invisible and may possibly reveal malfunctions of the device that cannot be derived from external symptoms of a system fault. So, these instruments give the engineer a feel of how a system works or why it does not work.

The problems of implementing and analysing computer programs are not so much different from the problems described above. Nevertheless, existing tools that help implementing, testing and debugging a program are usually not as handy and easy to use for a programmer as, for example, electronic measuring instruments for an engineer. The problem is not that a programmer does not know which internals of the program would be of interest for him. It is much more the problem of how the programmer specifies what he wants to see and how the internal processes of a program are presented to him, i.e. a problem of communication. The task is complicated by the fact that the programmer has to communicate with the program and the ‘measuring instruments’ via the same devices - a computer screen, a keyboard, and possibly a mouse. So how should the ‘software oscilloscope’ be hooked up with the program, how should it look like – how should it work? [8, page 991].

Chikofsky and Cross state that ‘the cost of understanding software, while rarely seen as a direct cost, is nonetheless very real. It is manifested in the time required to comprehend software, which includes the time lost to misunderstanding. By reducing the time required to grasp the essence of software artifacts ... [it] may greatly reduce the overall cost of software.’ [26, page 17, emphasis added].

As Lieberman and Fry indicated, what makes programming so difficult is that a programmer must imagine the dynamic behaviour of a program while he/she is writing the static description [69].

The following are the reasons that we might want to use software visualizations:

1. To assist program development.
2. To assist computer instruction.
3. To help the programmer debug his/her code.
4. To show others (e.g. managers) what the software is doing (or why it is taking so long).
5. To compare the program before and after changes.
6. To see how well the program matches earlier designs.
7. To help the programmer doing maintenance understand how the program works.

What all of the above have in common in each case is that someone is trying to understand how the software is actually working. Two of these, numbers two and seven, are discussed in more depth.

2.1.1 Assisting computer instruction

Visualizations are in constant use in computer instruction, especially in the form of diagrams. Diagrams are used for many things such as showing data structures, creating state machines, showing network topologies, etc. It is a rare book or course on computer science that contains no visualizations.

Diagrams are not the only form of visualization that are currently widely used in teaching. The film *Sorting Out Sorting* [3] is shown to many students when the topic of sorting is first introduced.

Ford [35, 36] suggests three particular difficulties in the teaching/learning situation:

1. It is difficult to teach ideas involving movement, for example, how a loop construct works.

2. When students execute programs that they wrote, the students are denied a visual dimension by which they can check their understanding of the language they are using and the logic of their programs.
3. Graders of programming assignments must resort to output of test sets or examine the program source in detail.

These difficulties could contribute to the well-documented problems of learners:

- learning dynamic concepts such as recursion, iteration, variable binding, flow of control, parameter passing.
- difficulty relating abstract concepts to concrete situations.

To see if software visualization would help the problem, in the second semester of first year, Ford *et al.* introduced the students to animation software² and had them design twelve animations based on imperative programming in C++. Some of their conclusions were:

- The animations that were created varied tremendously, even in how variables and loops were depicted, indicating that people visualized differently.
- Animations provide a complementary view to the programs' source code.
- Some of the animations produced showed how the students misunderstood some concepts, a common one being that the body of a while loop must run at least once. This helped the instructors to diagnose the misconception.
- After the animations were completed, the students seemed to learn how the language, C++, worked.

²The software is called Goofy and is based on TANGO (see 3.1.1.3).

Several other groups have studied the effects of software visualization in learning, two of which are discussed below.

The experience at Brown University [20] with Balsa (see 3.1.1.1), shows that graphical techniques for teaching algorithm analysis to undergraduates are superior to conventional methods.

Stasko *et al.* [111] did an empirical study on learning the pairing heap data structure [37] using animation. They took twenty graduate students who had not studied pairing heaps and divided them into two groups. The first group was given a textual description of the algorithm, the second, the same textual description and some time interacting with an animation of pairing heaps. After forty-five minutes the students were given a test. The group that used the animation finished the test faster and with better results.

2.1.2 The Cost of Maintenance

Maintenance³ today is a major part of significant software systems. It is virtually impossible to create a system that doesn't require maintenance. In fact, there have been several different estimates of the cost of maintenance a few of which are listed here:

- maintenance exceeds 60 percent of the total cost of software [40].
- programmers spend more than half of their time on maintenance [41].
- most programmers spend 50% of their time on maintenance, and some spend up to 80% of their time on the task [85].

The total maintenance cost can be broken down into three areas (see table 2.1).

Successful maintenance requires two things: the ability to make changes easily and an in-depth understanding of the software's structure and behaviour [128]. Other

³Maintenance being used in this context to refer to the modifications made to software systems after their first release

Type	Description	Percentage
Corrective	Correcting errors in the system	17
Adaptive	Make the system work in new hardware or software environment	18
Perfective	This includes adding new functions, modifying existing ones, improving user interfaces, etc.	65

Table 2.1: Maintenance Costs [70].

studies [67, 86] show that the comprehension process (understanding the original programmer's intent) takes more than 50% of the time spent on the maintenance task.

While object-oriented techniques may help with making changes, there are still difficulties with the understanding of the programs [128, 127, 66, 27, 49, 103, 45, 58]. This is mostly due to the following factors:

Complications from Inheritance. In an empirical study of the effect of inheritance on the maintainability of object-oriented software Daly *et al.* [27] found that with three levels of inheritance, subjects maintaining object-oriented software using inheritance performed the modification tasks, on average, approximately 20% quicker than those maintaining equivalent object-based software with no inheritance. This seems to be in conflict with the above statements that inheritance complicates matters, but in a survey done by the same group of people [28], of 275 practitioners, the majority of the people who responded, indicated that between four and six levels of inheritance depth is where difficulties began.

In another study, Hsia *et al.* [49] found data that suggested that maintainability of object-oriented systems is dependent on the characteristics of the inheritance trees. They studied two designs of a system with the same functionality, one composed of twenty-two classes, the other thirty classes. Each hierarchy had a depth of seven. The same functional modifications were made to each system. The study found that

the second system, the one with thirty classes was more difficult to maintain. They concluded that ‘a design with a higher broadness factor⁴ produced a system that was more maintainable’ [49, page 10].

Small Methods. The relatively large number and small size of methods in typical object-oriented programs increase the number of relationships that a maintainer must understand. In Smalltalk/V for Win32, LaLonde and Pugh found that there are roughly twenty-two methods per class [59]. Moreover, LaLonde and Pugh found that the average number of lines per method (discarding blank lines) is 7.34. If comment lines are also discarded, the average drops to 5.6 [59]. Haaland, who made measurements on thirteen research prototypes and industrial projects, found the mean across all measured projects was approximately eighteen methods per class [44].

Quite often chains of methods will have to be looked at before code is found that provides the functionality. Two reasons for the chaining are: inheritance and methods that provide redirections to components of the object. Because of this large number of small methods, many methods will often have to be looked at. These methods could be spread over a wide number of classes in order to understand what the one method accomplishes. This complicates the understanding of programs.

Listing 2.1 Polymorphism example

```
doSomething: aCollection  
  
    aCollection at: 1 put: 'stuff'.
```

Dynamic Binding and Polymorphism. These make understanding the program more difficult because the programmer may not be able to tell from looking at the code, what method is going to be executed. Take a look at listing 2.1. For the collection class that is passed in, it is unknown which **at:put:** method (from among

⁴Broadness factor is defined as the ratio of the depth of a hierarchy tree and the total number of objects in the hierarchy tree. This provides a measure similar to the average number of children for the system [25].

the collection hierarchy) will be executed. It may do what is desired or produce an error (that is assuming that the argument passed into the method is indeed a collection).

Cooperating Object Classes. Quite often several classes are tightly coupled. To understand what is happening in the one class you will have to be able to identify and decipher the other class(es) that is/are tightly coupled with it.

For example, in one object-oriented environment, windows are each built around three objects: aPane object that displays the data, aDispatcher object that handles user interactions and an application model object that provides the data. Several different sub-classes of Pane and Dispatcher handle different kinds of data and modes of interaction. The three objects communicate with each other in complicated ways that can be very time-consuming to unravel. For example, these are the first few steps in opening a window:

1. The application model sends a message to Pane to create Dispatcher.
2. The application model sends an 'open' message to Dispatcher.
3. Dispatcher gets the application model from Pane.
4. Dispatcher asks the application model for the initial size of the window.
5. Dispatcher tells Pane to reframe itself to fit.
6. Pane reframes itself and sends a 'reframed' message to the application model so that it can update itself.

Someone trying to understand the code obviously cannot make sense of any of the classes alone since the developers' plans are delocalized among them. To understand the methods of each class, he/she must understand the whole mechanism of the three cooperating classes⁵.

⁵Example from [128, page 79].

This complex relationship between classes makes it difficult to anticipate and identify the ripple effect⁶ [39].

Testing. The complex interactions make it difficult to prepare test cases and data that can adequately retest the system after a maintenance change [58].

2.1.3 Detractors

This is not to say that software visualization is the silver bullet that will cure all of the software problems. It does have its detractors, the best known being that mentioned in Frederick Brooks' "No Silver Bullet: Essence and Accidents of Software Engineering" where he wrote:

'A favourite subject for PhD dissertations in software engineering is graphical, or visual, programming—the application of computer graphics to software design. ... Nothing even convincing, much less exciting, has yet emerged from such efforts. I am persuaded that nothing will. In the first place, ... the flowchart is a very poor abstraction of software structure ... It has proved to be useless as a design tool. ... Second, the screens of today are too small, in pixels, to show both the scope and the resolution of any seriously detailed software diagram. ... More fundamentally, ... software is very difficult to visualize. Whether one visualizes control flow, variable-scope nesting, variable cross-references, data-flow, hierarchical data structure, or whatever, one feels one dimension of the intricately interlocked software elephant.' [14, page 15-16, emphasis added by [77]].

2.1.4 Summary – Why Visualize?

Over the last decade software has become more and more complex. The current tools (such as just looking at the code or reading out-of-date documentation) that are used

⁶The ripple effect refers to the phenomenon that changes made to one part of a software system ripple throughout the system.

to understand the existing code have been found lacking. Software visualization is one tool that can help alleviate this problem. It can distill large amounts of code into pictures that can be easily understood by the user.

In the previous section we have discussed in more detail two areas that can benefit from visualization:

Assisting Computer Instruction The use of various types of visualizations can greatly increase the understanding of the students.

Maintenance The current cost of maintenance is quite high. Quite a bit of that time is spent trying to understand the existing code. Anything that can reduce this time is beneficial.

2.2 Moving From 2D to 3D

This section will explain why moving from 2D to 3D is useful. It will start off by listing some of the disadvantages, then move on to the advantages and finally discuss some empirical evidence that 3D is better than 2D.

It is useful here to note that in 2D visualizations there are really three degrees of freedom, the two spatial dimensions and time⁷. Moving to 3D visualizations adds a third spatial dimension giving us an extra degree of freedom.

2.2.1 Disadvantages of 3D

These are the main disadvantages of 3D visualizations.

computationally more intensive . Rendering 3D visualizations are computationally much more complex than rendering similar 2D visualizations.

⁷There are actually more degrees of freedom than this such as colour, shape, luminance, volume and orientation.

requires better/more hardware . In order to render 3D visualizations at the same speed as 2D visualizations, faster hardware is needed (due to the fact that they are computationally more complex). In addition, better graphically displaying equipment, such as better monitors, specialized graphics' cards etc. are also needed.

not as transferable to paper . Being able to navigate around the visualization is one of the advantages of 3D (see 2.2.2.1). This gets lost upon printing the visualization.

harder to display text . Text is harder to display in 3D visualizations. This is because text needs to be 'flat' in order to be read. The 3D visualization can be moved around in such a way that the text either has to be repositioned so that it is 'flat' all the time, or it can't be read. Another problem with text is that it doesn't scale nicely⁸, so if the whole scene is moved in or out it makes the text not look correct.

navigation controls . Navigation controls are not as intuitive as those of 2D visualizations. This might change in the future as a more common and standard 3D navigation system emerges.

2.2.2 Advantages of 3D

This section will discuss some of the advantages of 3D visualizations. This is done by first looking at some properties of 3D visualizations and then looking at some categories of 3D visualizations and seeing how they are improvements over similar 2D visualizations.

2.2.2.1 Some properties of 3D Visualizations

Navigation. One of the main properties of 3D visualizations is the ability to navigate among/through the visualization to view it from many different perspectives.

⁸Unless something like true-type fonts are used.

This allows the user to look at the visualization from many angles and perspectives.

2.2.2.2 Categories of 3D visualizations

This looks at the categories of 3D visualizations and examines how these visualizations are improvements over the 2D versions.

Figure 2.1: An example of an augmented 2D view. A 2D bar chart view of a bubble-sort, in which constant depth is added to each element [110, Figure 4, page 13].

Augmented 2D views. These visualizations are those for which the display only requires two spatial dimensions. The third is added purely for aesthetic or presentation purposes, making the visualization more pleasant to look at, which empirical evidence seems to indicate helps people comprehend the data (see 2.2.3). An example can be seen in figure 2.1.

Inherent 3D application domain views. This category includes visualizations whose displays are inherently 3D such as visualization of a cube parallel architecture,

Figure 2.2: An example of an inherent 3D application domain view. This shows a frame from a 3D particle simulation showing where particles are in a 3D space at a given time [110, Figure 7, page14].

volume packing, no-planar graph algorithms. These visualizations capture the true state of what is happening far better than any equivalent 2D visualization could. An example can be seen in figure 2.2.

Figure 2.3: This shows an example of adapted 2D visualizations. Finding the shortest path in a weighed graph, (a) shows the original graph, (b) shows a second view of the graph on top of the original graph which is taken from part way through the algorithm. The second graph also features vertical bars over the nodes. The vertical bars represent the current weight of the edges going from the vertex [19, Figure 1a and 1c, page 4].

Adapted 2D views. These visualizations really only require two spatial dimensions. The third dimension encodes some other attribute. Typical uses of the third dimension are:

Value This is where the third dimension is used to encode values of computational elements such as variables. An example of this is a matrix of positive numbers. The third dimension is used to draw sticks at each cell, where the height of each stick is proportional to the value of the corresponding element.

Positional This is where the third dimension is used to encode the position or index within a structure. Examples of this are the processor number, array index, etc.

State This is where the third dimension is used to encode instantaneous state of a computation. Typically, the time dimension fits this role.

An example can be found in figure 2.3

Uniting multiple views of an object. In order to take advantage of the parallelism capabilities of human visual systems, it is desirable to display as many relations as possible in one visualization. However, there is often too much information to visualize. When all information is visualized in a two-dimensional visualization, the visualization becomes so complex that it does not help users and may even confuse them further. To avoid such a problem, each relation is represented as a different 2D visualization.

However, 3D visualizations are capable of showing both views. Figure 2.4, shows how a 3D visualization can supply different views from one visual representation. Created properly, users can rotate the object to focus on each individual view all the while seeing how it fits into the big picture.

There are times when, to understand what is happening, a single view of an object is not enough. Providing multiple views of the object is helpful. To understand the multiple views properly the user then has to form the relationship between the multiple views in his/her mind. With moving to 3D the multiple 2D views can be incorporated into a single image. This can almost be seen as a sub-category of the Adapted 2D views (see page 22), since, when you are merging multiple views, sometimes you are using the third dimension to just show the other views.

Capturing a history of a 2D view. Often the entire history can be helpful in understanding an algorithm. The history can give the context of how the algorithm has progressed each time the state is changed. It can expose patterns in the algorithm that are not otherwise observable. An example can be seen in figure C.2.

Figure 2.4: 3D representation offers different views [53, Figure 1, page 182].

Maximizing Screen Space. Screen space is limited. It is useful to be able to see as much as possible on the screen at one time, so that scrolling isn't needed. The additional dimension allows us to squeeze more information into the same screen real estate. An example is cone trees, in which a 600x600 pixel window 2D layout can typically display about 100 nodes. Using cone trees about 1000 nodes can be displayed [97, 65]. (See figure C.1 for an example of a cone tree.) Cone trees are discussed in more detail in section 4.3.2.1. Cone trees were originally developed by Robertson *et al.* (see 3.2.2).

Minimizing Arc Crossings. One of the major problems with laying out graphs in 2D is minimizing the edge crossings. With almost any non-trivial graph it is virtually impossible to accomplish this in 2D. By making use of the third dimension overlapping arcs can be eliminated. Since there are three dimensions in which to lay out the graph, the algorithms can be simpler, minimizing arc crossings aids in the understanding of the graph. In this way the eye isn't confused by the crossing arcs.

2.2.3 Empirical evidence

Some, such as Tufte [116], hold the opinion that when 2D is sufficient to portray the information then adding a third dimension can be harmful. Tufte offers no data to support this conclusion. Others have done experiments that seem to indicate that 3D viewing is better than 2D.

Spence [107] thinks that three-dimensional imagery can provide important cognitive cues for the human visual system. Spence conducted experiments in which two-dimensional information displayed was compared to their three-dimensional analogs (bar and pie charts to bar and cylinder solids). The study determined that people could process the information from the three-dimensional display quicker and just as accurately. He speculated that the attractiveness of the three-dimensional displays and their correspondence with our perceptions of the real world may be important influences in this result.

Ware and Franck [121, 120] conducted a study on user comprehension of graph structures. Two nodes in a graph were highlighted in a randomly laid out graph. The subject had to decide if there was a path connecting the two nodes. The results were that users can extract three times as much information from a graph that is displayed in 3D (using a stereo display and head coupled perspective) than from the same graph displayed in 2D.

2.2.4 Summary – Moving from 2D to 3D

In this section we have looked at why you would want to create visualizations in 3D. We have looked at various ways in which the third degree of freedom can be used to increase the ability of the visualization to provide information. Some studies were discussed that seem to indicate that 3D visualizations are better at providing cognitive cues than 2D visualizations.

2.3 Summary

This chapter has introduced two reasons for creating SOVIST-3D. The first is that visualizing software can help users understand how the software works. The second that using 3D can make the visualizations better.

Chapter 3

Related Work

This chapter discusses related work in the field of software visualization but not visual programming¹. This chapter aims to give both a general overview of software visualization and specifics of how some of the systems created relate to SOVIST-3D.

For a good survey and taxonomy of software visualization see Price *et al.* [87]. Other taxonomies and surveys include: Meyers's [77] well known survey updated from [75, 76], Roman and Cox [98], Stasko and Patterson [112], the earlier chapters of Shu's [104] book and several chapters in Chang's [24] book. These surveys are now several years out of date and do not provide the information on some of the later systems discussed in this chapter.

Each of the surveys seem to have developed their own taxonomy. Several of the main ones are:

- Shu [104] focuses on the increasing degrees of sophistication exhibited by software visualization systems. These range from pretty-printing to algorithm animations.
- Meyers [77] uses two axes to classify systems. The axes are: the program aspect (code, data, or algorithm) that is illustrated and the display style (static or dynamic).

¹See 1.2 for the definitions.

- Brown [16] uses three axes to classify algorithm animations. The three axes are: content (direct or synthetic representation of information about the program), transformation (discretely or smoothly changing images) and persistence (representations of the current state or of the entire execution history).
- Stasko and Patterson [112] use a four category system, the categories being aspect, abstractness, animation, and automation.
- Roman and Cox [98] use a five criteria system, the criteria being scope, abstractions, specification method, interface and presentation.
- Price *et al.* [87] utilize six categories to classify systems. The categories are: scope, content, form, method, interaction and effectiveness.

The last one (Price *et al.*) has come to be the standard taxonomy that is referred to in the literature.

While reading through the related systems the reader should keep in mind the key differences between, SOVIST-3D and the systems described below²:

1. The visualizations are in 3D.
2. The visualizations are in Smalltalk.
3. With more than one type of visualization, SOVIST-3D provides different types of visualizations as well as different ways of laying out the visualizations. In other words, it provides an extensible framework for creating software visualizations. The framework currently contains three different types of visualizations and four different layouts.
4. The visualizations use the language (especially the meta-facilities) to get the information used in the visualizations. No work needs to be done by the user to create the visualization.

These differences with the various systems are discussed both within the individual sections and at the end of the chapter (see 3.3.2).

²Not all reasons apply to all systems.

Name	Creator	Sec.	Description
An animation kit	London & Duisberg	3.1.1.4	An animation system for viewing algorithms in Smalltalk.
BALSA & BALSA-II	Brown & Sedgewick	3.1.1.1	An animation system designed for learning how algorithms work.
COOP	Carleton	3.1.3.4	Interaction, hierarchal and part diagrams created automatically in Smalltalk
Hyperbolic Browser	Lamping & Rao	3.1.2.2	Creates a fish-eye scheme for visualizing and manipulating large hierarchies.
Lens	Mukherjea & Stasko	3.1.1.3	A combination algorithm animation system and system debugger.
OO message Tracing	Pauw <i>et al.</i>	3.1.3.2	Uses matrices, graphs and histograms to track the dynamic behaviours of OO programs.
OO code changes	Kung <i>et al.</i>	3.1.3.3	Creates visualization showing the changes in the code from previous versions and the impact the changes have upon the system.
POLKA	Stasko <i>et al.</i>	3.1.1.3	A general purpose algorithm animation system especially good for parallel computations.
PROVIDE	Moher	3.1.4.2	Visualizations are created during the program execution by matching to pre-existing visualizations and prompted information.
PV	Kimelman <i>et al.</i>	3.1.4.1	Visualizations of different layers: the program itself, user-level libraries, the operating system and the hardware.
SeePs	Masnavi	3.1.1.5	Animates NeWS programs.
TRACK	Böcker & Herczeg	3.1.3.1	Creates program traces in Smalltalk resembling a 'jumping course' where 'hurdles' and 'fences' are built up around and between objects.
Visual Insights	Eick <i>et al.</i> at Bell Laboratories	3.1.2.1	Creates 2D visualizations of large amounts of data stored in databases.
XTANGO	Stasko <i>et al.</i>	3.1.1.3	An algorithm animation system using the path-transition paradigm for the animation design.
ZEUS	Brown	3.1.1.2	An animation system for viewing algorithms. An outgrowth of BALSA.

Table 3.1: A quick overview of the related work in 2D

Name	Creator	Sec.	Description
3D Graph Visualizer	Ware <i>et al.</i>	3.2.3.1	Uses a general graph layout tool to layout graphs of C++ code.
ANIM3D	Najork & Brown	3.2.1.2	An object-oriented 3D animation library targeted at visualizing combinatorial structures, and in particular at animating algorithms
COOP	Carleton	3.1.3.4	Interaction, hierarchal and parts diagrams. The information is created in Smalltalk, but Open Inventor is used to display the visualizations.
Information Visualizer	Robertson, Mackinlay & Card	3.2.2	A new desktop metaphor for 3D information visualizations.
Lisp program execution	Lieberman	3.2.4.3	Uses containment of objects, re-rooting and uniting multiple views to show the execution of Lisp programs.
Narcissus	Hendly <i>et al.</i>	3.2.4.1	A system that uses repelling and attractive forces between objects to create the visualization.
Obliq-3D	Najork and Brown	3.2.1.2	A high-level, fast turnaround system for building 3D animation.
Pavane	Roman <i>et al.</i>	3.2.4.4	A system for declarative visualization of concurrent computations.
PLUM	Reiss	3.2.3.2	A framework for producing 3D software visualizations.
Polka-3D	Stasko & Wehrli	3.2.1.1	A toolkit for creating 3D algorithm animations.
Virtual Images	Vion-Dury & Santana	3.2.3.3	Debugs 3D visualizations on a distributed object-oriented system called Guide.
VOGUE	Koike	3.2.4.2	3D visualizations of class libraries in object-oriented languages.
ZEUS-3D	Brown & Najork	3.2.1.2	A 3D version of ZEUS (see 3.1.1.2).

Table 3.2: A quick overview of the related work in 3D.

The rest of the chapter is divided into three sections. The first discusses software visualization in 2D, the second, software visualization in 3D. These sections are further sub-divided into categories. The third section gives a summary of what was learned and applied from these systems to SOVIST-3D. Tables 3.1 and 3.2 give a brief summary of the systems discussed.

3.1 Related Work in 2D

This section describes software visualization related work in 2D. It is broken down into sections describing algorithm animation (see 3.1.1), large scale systems (see 3.1.2), object-oriented systems (see 3.1.3) and then the rest of the systems (see 3.1.4).

Most of the systems described in this section are not directly related to SOVIST-3D, but they are briefly discussed for the following reasons:

- they help provide a complete picture of what has been done in software visualization.
- other systems mentioned in the rest of the system build upon the work discussed in this section.

The systems in this section are all 2D and thus suffer from the inherent drawbacks (see 2.2) that SOVIST-3D has overcome by using 3D.

3.1.1 Algorithm Animation in 2D

Algorithm animation is ‘the process of abstracting a program’s data, operations, and semantics, and creating dynamic graphical views of those abstractions’ [109, page 27]. In other words, algorithm animation tries to provide a changing visualization that demonstrates how the algorithm works. Most of the algorithm animation systems do this by making it possible for the animator to annotate the code to create the

animated visualization. This technique was pioneered by Balsa (see 3.1.1.1) and is still the most popular way to do algorithm animation. These animations created generally try and provide a high level overview of the algorithm.

SOVIST-3D has taken a different approach. It currently uses animation only in the message tracing viewer (see 4.2.4); the rest of the visualizations are static. In SOVIST-3D, the message trace is created automatically and the only animation is that the messages can be added one at a time (either by stepping or as an animation) rather than as a static diagram.

Annotating the software only works if the animator already has an understanding of how the software works and thus is not useful for some of the reasons we wish to visualize software (see 2.1).

3.1.1.1 Balsa and Balsa-II

Brown and Sedgewick at Brown University created Balsa [20] and its descendant Balsa-II [15]. They are among the first, and most likely the best known, algorithm animation systems. These systems allow the user to annotate an algorithm, setup the input to the algorithm and then to view the animation. Balsa-II has a more sophisticated interface, including multiple views of the algorithm.

Algorithms are animated in two-dimensions in black and white on Macintosh computers using Apollo workstations as a back-end. The systems contain extensive libraries of sophisticated animations that allow the annotations to be simple. The systems are tightly integrated, so much so that the algorithms must be run from within the algorithm animation system (i.e. the set up of the algorithms are all the same with separate subroutines needed to handle the algorithm code, the algorithm parameters and also initialization and termination).

3.1.1.2 ZEUS

Brown at the DEC Systems Research Center [17] developed ZEUS. ZEUS is an outgrowth of Balsa and the animations are created in a similar fashion (see 3.1.1.1).

ZEUS is ‘noteworthy for its object-oriented design, graphical specification of views and the fact that it is implemented in a multi-threaded, multi-processor environment, so it can easily animate parallel programs’ [87, page218].

ZEUS is written in a dialect of Modula-2 and was originally created to run on DEC stations but now also works on other UNIX platforms. Brown’s work with Hersberger [18] added colour and ‘algorithm auralization’ – using non-speech sound to convey the workings of algorithms. This work has since been extended to 3D (see 3.2.1.2).

3.1.1.3 Work done by Graphics, Visualization & Usability Center (GVU) at Georgia Institute of Technology

Stasko *et al.* at the GVU [124] have done quite a bit of work in software visualization. This section describes some of the work they have done in 2D.

XTANGO XTANGO [109, 108, 126] (Transition-based ANimation GeneratiOn) is a general purpose algorithm animation system. The X in XTANGO comes from the fact that it was extended to work in the X window System. The focus in the system is on ease-of-use in creating animations.

The animations make use of the path-transition paradigm. The program is annotated to produce a trace file of important events. These events then activate animation routines implemented in a separate file. Transitions on objects include movement, colour change, resizing and filling.

POLKA POLKA [125] is a general purpose animation system that is particularly well-suited to animating algorithms and computations, especially parallel computations. The focus on the system is on a balance of power and ease-of-use. POLKA is essentially a newer, more powerful XTANGO.

Lens Lens [79] was developed to straddle the position between carefully hand-crafted animation, such as those produced by POLKA, and the systems that automatically create low-level program structures for the purpose of debugging. Lens allows programmers to build rapidly (in minutes) algorithm animation-style program views without using textual coding and is integrated with a system debugger.

3.1.1.4 Animation Kit in Smalltalk

London and Duisberg at the Computer Research Laboratory at Tektronix, Inc. [71] have basically recreated Balsa (see 3.1.1.1) in Smalltalk. It allows users to annotate the algorithm with interesting events. London and Duisberg have taken advantage of the Model-View-Controller system in Smalltalk to allow multiple views of the animation to occur and be updated simply. Duisberg went on to create Animus [32].

This Animation Kit was developed using Smalltalk-80 upon a system comparable to a DEC Vax 11/750 workstation.

This is one of the few systems in which Smalltalk is used to create the visualizations. It, like SOVIST-3D, also creates the visualizations entirely within Smalltalk. The visualizations, however, are created by annotating the code.

3.1.1.5 SeePs

Masnavi at the University of London [74] developed SeePs. SeePs creates automatic animations for visualizing NeWS³ programs. It shows mostly text view panes with the current state of execution, event queues, the dictionary stack and operand stack, and some icons showing the flow of control among processes. It works as a tracer going through a NeWS procedure executing it one object at a time. If the object is an operator, NeWS has a rule which allows it to animate the object. If not, the execution views are updated.

³NeWS [42] can best be described as a superset of Adobe Postscript [115] with extensions for supporting the development of graphical user interfaces.

This is the only animation system that creates the animations automatically. It works much like the message tracing viewer (see 4.2.4) in that it creates the visualizations by using facilities of the language, except that it animates the objects based upon mapping operators to animations. It otherwise just displays textual information, like the Smalltalk debugger, in its views.

3.1.2 Large Software Systems

Most of the systems that have been developed for software visualizations don't scale up beyond looking at a few objects. The systems described in this section illustrate software visualizations that scale. These systems still restrict themselves to visualizing in 2D and have come up with various alternate ways to maximize the screen space (see page 25).

3.1.2.1 Visual Insights

Eick *et al.* at Bell Laboratories [122] developed Visual Insights. Visual Insights is a commercial project that is used to visualize data stored in large databases. Visual Insights is based on previous work by Eick *et al.* at Bell Laboratories [5]. They developed scalable techniques for visualizing program text, text properties and relationships involving program text. The techniques were developed in response to problems maintaining and enhancing legacy software systems in Bell Laboratories. Visual Insights contains components that do the following visualizations:

- code version history
- differences between releases
- static properties of code
- code profiling and execution hot spots
- dynamic program slices

The underlying goal of the visualizations is to try and provide both a global overview and fine-grained detail in the same display. This is done by attempting to use all of the screen real estate by having every pixel convey useful information⁴. The user interface then allows the users to filter and focus the display as desired. It displays the information in 2D using the relative size of rectangles and colours to display the information.

All of this work is implemented in C++ using Bell Laboratories Vz graphics library [43] which allows it to be run on workstations supporting X11 or Motif and OpenGL.

The next two paragraphs focus on a few of the different visualizations that Visual Insights creates.

3.1.2.1.1 SeeSoft. SeeSoft [33, 114] is used to visualize the source code in large software systems. The approach is to represent files in a directory in columns and the source code lines as rows of coloured pixels. Actual code is represented by the pixels, with each row of pixels representing the length and indentation of a row of code. Colours can be assigned based upon information retrieved from the change management system, i.e. age of code, programmer, etc. The impression is of a miniature picture of the source code. The user can interact with the system, adjust the display and look at the actual lines of code.

3.1.2.1.2 SeeSys. SeeSys [4] is used to visualize statistics associated with code that is divided hierarchically into subsystems, directories and files. It can display such information as the relative sizes of the components in the system, the relative stability of the components, the location of new functionality, and the location of error-prone code with many bug fixes. It uses animation to display the changes over time. SeeSys is essentially a system for displaying software metrics (as long as the metrics are a quantitative measure and are additive).

⁴This is an example of maximizing screen space (see page 25).

3.1.2.2 Hyperbolic Browser

Lamping and Rao at Xerox Palo Alto Research Center [64, 65] developed the hyperbolic browser. The hyperbolic browser presents a focus + context (or fish-eye) scheme for visualizing and manipulating large hierarchies. It works by laying out the hierarchy in a uniform way on a hyperbolic plane and then maps the plane onto the display region. This provides a fish-eyed distortion of the entire structure. The display can then be manipulated using the mouse to change the focus, using animation to show the change in focus.

The hyperbolic browser can display in a 600 pixel by 600 pixel window, 1000 nodes, of which the fifty nearest the focus can show from three to dozens of characters of text. This is opposed to the conventional 2D browser which can display about 100 nodes with three character strings.

The hyperbolic browser has a similar aim to the cone tree layout in SOVIST-3D (see 4.3.2.1), to maximize the screen space (see page 25) while displaying large hierarchies. This approach can display as many nodes as a cone tree with much lower computational needs, but is not as easy to navigate or to look at large sections of the hierarchy at a time.

The hyperbolic browser is created in a portable C++ implementation (which supports Unix/X, Windows 3.1, and Windows NT).

3.1.3 Two Dimensional – Object-Oriented

The systems discussed in this section, similar to SOVIST-3D, are created to display object-oriented systems.

3.1.3.1 TRACK

Böcker and Herczeg [8] at the Institut für Informatik, Universität Stuttgart, developed TRACK (Trace Construction Kit). TRACK creates a message tracer visualization system in Smalltalk.

TRACK works through a combination of textual notation and graphic images. Each object can be represented by a bitmap, with object connections shown by lines. The trace for a program resembles a 'jumping course' where 'hurdles' and 'fences' are built up around and between objects [8, page 993]. These obstacles allow the flow of the program to be followed including the setting of breakpoints. In addition, filters are used to determine which objects and messages will be shown. TRACK allows the system to be debugged in real-time, but does have the problem that none of the information is saved, so it can not be replayed.

TRACK performs a similar function to that of the message tracer viewer (see 4.2.4) in that it traces the execution of a piece of code. It differs in that it is similar to the debugger and displays information as it goes (with no replay) while SOVIST-3D runs through the code first and then creates the visualization. Both ways have their advantages. TRACK is more useful for debugging purposes, while SOVIST-3D can take advantage of the fact it knows all the objects to lay them out better and to allow the user to focus on certain areas.

In addition, the sequence diagram layout (see 4.3.1.2) allows the users to follow the flow of execution between many objects where TRACK becomes quite cluttered and difficult to follow if there are many objects.

3.1.3.2 Visualizing the Behaviour of Object-Oriented Systems

De Pauw *et al.* at IBM T.J. Watson Research Center [29, 30] have created a system for visualizing the behaviour of object-oriented systems. They annotate programs to generate events. The events that they are interested in are:

1. the construction of an instance of a class.
2. the destruction of an instance of a class.
3. the entry into a method.
4. the exit from a method.

This information is then used, in real-time, to provide the following visualizations:

Inter-class call cluster provides a dynamic overview of communication patterns between classes where the amount of communication between classes determines the distance between the labels in the visualization.

Inter-class call Matrix creates a matrix of instances of classes, with the elements showing the amount of calls of the two objects keyed by colour.

Histogram of instances shows all the current instances of various classes in the system. As instances are created, they are added to the histogram. When deleted, they change colour until the space is re-used by future creations.

This work is all done in C++, but the authors claim that techniques are language independent.

Similar to SOVIST-3D this system provides multiple kinds of visualization. However, most of the systems are quite simple relying on things like matrices and histograms.

3.1.3.3 Visualizing code changes in Object-Oriented systems

Kung *et al.* at the University of Texas at Arlington and Fujitsu Network Transmission Systems [58] have developed a system that automatically identifies different types of code changes and their impact upon the system. The system identifies four different types of code changes:

1. Data Changes. This includes global and local variables, instance variables, etc
2. Method Changes. This is further classified into three types: component changes, interface changes and control structure changes.
3. Class Change. This is further classified into three types: component changes, interface changes and relation changes.

4. Class library changes. This includes changing the defined members of a class, adding or deleting a class, and adding or deleting a relationship between two classes.

Each of these changes has a certain known impact on the rest of the system. This information is then presented using multigraphs.

This system is quite different in its aims from most of the ones mentioned here already; it is just interested in code changes. They have taken great pains to be able to automatically figure out the code changes and display the information in simple graphs.

3.1.3.4 The COOP Project

The Centre for Object-Oriented Programming (COOP) in the School of Computer Science at Carleton University worked on a project the aim of which was to develop a methodology and tools (environment) suitable to support re-analysis, re-design, re-implementation and reuse in object-oriented systems. Naturally, this entailed managing concurrent work in analysis, design and implementation. Specifically, the focus was on design capturing (i.e. the ability to capture both the static and dynamic characteristics of working software in both textual and pictorial form) and design regeneration (i.e. the ability to capture both the static and dynamic characteristics of working software in both textual and pictorial form after changes have been made) [63].

In part, to accomplish this, they created several different types of visualizations. The types created were interaction diagrams (created by Wang [119] and continued by Alguire [1] as their masters' theses), hierarchy diagrams and parts diagrams.

SOVIST-3D started out by extending these diagrams to 3D, before going on to create its current framework of 3D visualizations. The basic mechanisms to gather the data are the same in SOVIST-3D as the work done by the COOP group. SOVIST-3D's main advantages over the work done by the COOP group are:

- The visualizations are in 3D instead of 2D giving it the advantages of 3D (see 2.2).
- More information can be displayed, due to the fact that 3D can be used to maximize screen space (see 2.2.2.2)
- There are multiple ways to display the visualizations.
- It can allow models of objects to be used in place of the actual objects (see 4.4).

Its disadvantages are:

- Since no explanatory text is presented in the visualization, the mouse has to be moved frequently to see what everything stands for, if specific models of objects aren't used.
- More computer power is needed to create, view and navigate the visualizations in SOVIST-3D.

3.1.4 2D other

This section deals with the systems that don't easily fall into any of the previous categories.

3.1.4.1 PV

Kimelman *et al.* at the IBM Thomas J. Watson Research Center [52] developed PV. PV is a visualization system which provides concurrent visual presentation of behaviour from all layers, including the program itself, user-level libraries, the operating system, and the hardware, as this behaviour unfolds over time. PV was created to help both debug and tune software.

PV makes use of the trace output generated while running in AIX. The trace material is saved to files during the execution and then is used by PV afterwards. The views created by the system include:

- process scheduling and system activity.
- memory activity and application progress.
- hardware activity and source progress.

3.1.4.2 PROVIDE

Moher at the University of Illinois at Chicago [78] developed PROVIDE (a **P**rocess **V**isualization and **D**ebugging **E**nvironment). PROVIDE creates dynamic visualizations during program execution. The visualizations are created by the user asking for a certain type visualization (such as a pie chart), based upon examples shown. The user then enters into a ‘conversation’ with the type builder to supply the information to build the visualization. The information usually takes the form of mapping variables to parts of the visualization chosen. The visualization is then shown, or animated, as the program is executed.

PROVIDE works on programs written in C. The system is implemented on Macintosh workstations networked to a VAX station running 4.2 BSD UNIX.

3.2 Related Work in 3D

This section describes software visualization related work in 3D. It is broken down into sections describing algorithm animation (see 3.2.1), work done at Xerox Palo Alto Research (see 3.2.2), object-oriented systems (see 3.2.3) and then the rest of the systems (see 3.2.4).

3.2.1 Algorithm Animation in 3D

See the introduction to algorithm animation in 2D (see 3.1.1) for a discussion on algorithm animation.

3.2.1.1 Polka-3D

Stasko and Wehrli at the Georgia Institute of Technology [113, 110] have extended Polka (see page 33) to 3D. Polka-3D is implemented in C++ on SGI workstations using the GL graphics library. They have tried to abstract the 3D graphics knowledge into the toolkit, thus making it easier to create 3D algorithm animations.

3.2.1.2 Work done at Digital's System Research Center

Najork and Brown at Digital's System Research Center have created several 3D animation systems, each built on the previous system.

ZEUS-3D They [19] extended ZEUS to 3D (see 3.1.1.2).

They use three-dimensional graphics for three purposes:

- to express fundamental information about structures that are inherently two-dimensional (see page 22).
- to unite multiple views of an object (see page 23).
- to capture a history of a two-dimensional view (see page 23).

ANIM3D. They [80] created an object-oriented 3D animation library targeted at visualizing combinatorial structures and in particular at animating algorithms. The base library is written in Modula-3 [23] and supports several window systems (such as X) and several graphics systems (such as PEX and OpenGL). This is basically a rework of ZEUS-3D to speed it up and improve it.

Obliq-3D. They [81] have added an interpreted embedded language, Obliq [22], that drastically shortens turnaround time by eliminating the need for re-compilations to ANIM3D to get a new system called Obliq-3D.

Obliq-3D is founded on three basic concepts:

1. *graphical objects* for constructing scenes.
2. *properties* of the objects which include: colour, location, size, etc. Properties are time-variant.
3. *callbacks* for providing interactive behaviour.

3.2.2 Information Visualizer

Robertson, Mackinlay and Card at Xerox Palo Alto Research Center [95, 96, 97, 21, 73, 72] have created the Information Visualizer. The Information Visualizer has three major components:

1. *3D/Rooms*, a 3D version of Rooms [46]. Rooms adds the ability to share the same information objects in different workspaces, both individually and as groups.
2. The *Cognitive Co-Processor* [94], an animation-oriented user interface architecture. It includes mechanisms for 3D navigation and object manipulation.
3. *Information Visualizations*, which act as structure-oriented browsers into the sets of information.

The Information Visualizer was not developed for, and isn't really used for, software visualization. It was developed to view information in 3D. But, it has had a tremendous influence on the software visualization community by:

- developing many techniques for viewing information in 3D that can easily be used in software visualizations
- showing how smooth animation can be utilized to provide cognitive clues about the given information.

Some of the visualizations that they developed are discussed briefly.

Shadow Walls. While not a visualization by themselves, they help with depth cues for the user. Figure C.1 shows the use of a shadow wall with a cone tree (see next paragraph). The idea behind the shadow wall is that shadows are formed in the three different directions (behind, below and beside) of the object being visualized. This provides the users perception with clues by disambiguating the shapes and locations of the objects.

Cone Tree. A Cone tree [97] is a hierarchy that is laid out in 3D. Each parent and its children are laid out in the shape of a cone. The parent node is at the apex of the cone and the children spaced evenly about the base. Each of the children is, in turn, the apex of its own cone. The radius of the cones decreases as the level of the tree increases. An example of a cone tree with shadow walls (see the previous paragraph) can be seen in figure C.1.

Whenever a node in the cone tree is selected by the mouse, it is rotated to the front using smooth animation. Once this is finished, every parent node in turn is rotated to the front. This animation greatly enhances the ability of the user to understand the relations in the hierarchy.

The cone tree layout (see 4.3.2.1) used in SOVIST-3D is an adaptation of the cone tree described here. The smooth animation has not been added due to the speed difficulties encountered with SOVIST-3D.

Perspective Wall. The perspective wall [72] folds a 2D layout into a 3D wall. The wall smoothly integrates a central region for viewing the details with two perspective regions, one on each side, for viewing details. This is used to maximize space (see 2.2.2.2). The analysis of the wall indicates that it yields at least a three-fold increase over the 2D visualization.

Other visualizations. Other visualizations that were created include:

- Calendar visualizers [73] which create both a Spiral Calendar (designed for rapid access to an individual's daily schedule) and a Time Lattice visualization (used for analyzing the time relationships among the schedules of groups of people).
- Document Lens [96] which lets users grasp large documents. The document is arranged so that all of the pages can be seen, with some distinguishing features standing out. The user can then zoom in on any part keeping it in context.

3.2.3 Object-Oriented

The systems discussed in this section are the most similar to SOVIST-3D, in that they are trying to visualize object-oriented systems in 3D.

3.2.3.1 3D Graph Visualizer

Ware *et al.* at the University of New Brunswick [121] have created a 3D Graph Visualizer that takes a GDL (Graphical Description Language) and lays out a graph using seven different principles. The system runs on a Silicon Graphics Indigo2 Extreme workstation using stereo classes and head tracking.

The system creates software visualization of C++ code using the IBM compiler for the RISE System 6000. This compiler constructs an internal Prolog data base containing extensive information about the code structure [51]. From this database information is extracted using Prolog queries and converted to the GDL.

The 3D Graph Visualizer is just trying to view the static nature of the code, while SOVIST-3D is exploring both the static and dynamic natures of the code.

3.2.3.2 PLUM

Reiss at Brown University [91, 92] has developed a framework for creating 3D software visualizations. The backbone of the system is PLUM which offers a wide variety of

different strategies for presenting software visualizations. In this matter the system is much like SOVIST-3D. Two systems have been developed in addition to PLUM. PEACH is a package for hierarchically browsing data. TWIG is a package that maps data structures into the graphical structures required by PLUM.

PLUM takes graphical objects from an object-oriented database and then lays out the objects to form the visualization. The information has to be extracted from the software and put into the database. Most of this work can be done automatically (such as by TWIG) but some must be added by hand, in contrast to SOVIST-3D which is all automatic.

The graphical objects have properties (such as colour, text, etc.), components (such as children) and constraints. There are three basic types of graphical objects:

1. basic objects such as data objects, arc objects, light objects.
2. presentation objects that provide layout services, i.e. placement and sizing of their component objects such as tiled objects, timed sequences, and cone trees.
3. presentation styles that include styles that control both layout and presentation such as scatter plots and file objects.

Most of the layout work is done using level graph layout techniques [101] extended to 3D. The layouts include depth first, breadth first, level graph, value, and orthogonal layouts among the offered layouts.

3.2.3.3 Virtual Images

Voin-Dury and Santana at Bull-IMAG/Systèmes laboratory [118] have created Virtual Images. Virtual Images uses 3D interactive animations for representing large numbers of objects, complex relationships, and dynamic execution of concurrent activities in distributed object-oriented systems. Virtual Images is built on Guide [6], a distributed object-oriented system.

They use two main principles to create the visualizations:

1. *Workspaces use a 3D spatial model.* This maximizes the screen space (see page 25) that can be manipulated using the mouse.
2. *Objects are represented by polyhedrons having significant shapes, colours, volumes, and orientation.* They use eight degrees of freedom, the three spatial dimensions, shape, colour, luminance, volume and orientation.

With these principles they create three different types of visualizations:

Execution model where activities are shown using the polyhedrons described above arranged into pulsing spirals. Whenever a method call is done by the activity, the spiral grows/shrinks accordingly. Each method call is then represented by a connection to the calling object. This makes it easy to spot recursive calls and object sharing among activities.

Concurrency Each horizontal grid represents an activity and all objects linked in its context. Animated coloured lines represent the call graph.

Multiple stacks Activities are placed on a circle which symbolizes the application. The objects associated with each activity are stacked on the activity.

Virtual Images is similar to SOVIST-3D in that it provides multiple visualizations in 3D. Unlike SOVIST-3D, there is no way to interact between the visualizations and the code that created the visualizations. Virtual Images chose to use polyhedrals in different ways to represent different objects while SOVIST-3D allows models of the objects to be used in the visualizations.

3.2.3.4 3D work done in the COOP Project

Several members of the COOP group (see 3.1.3.4) experimented with creating 3D interaction and hierarchy diagrams. They created the diagrams by writing out object descriptions to files and then using that information to create 3D diagrams using Open Inventor.

This work is what started us thinking about 3D visualizations. SOVIST-3D's main advantages over the work done by the COOP group are:

- There are multiple ways to display the visualizations, i.e. SOVIST-3D contains layouts.
- SOVIST-3D has navigation tools that are specific for the visualizations created, rather than general ones like Open Inventor.
- It can allow models of objects to be used in place of the actual objects (see 4.4).
- In SOVIST-3D, the visualizations are created entirely within Smalltalk allowing the user to interact with the Smalltalk environment (including the code).

Its disadvantages are:

- More computer power is needed to create, view and navigate the visualizations in SOVIST-3D since the drawing of the visualizations in Open Inventor has been optimized, while the OpenGL code in SOVIST-3D has not been optimized.

3.2.4 Other systems

This section discusses the systems that don't easily fit into one of the above categories.

3.2.4.1 Narcissus: Visualizing Information

Hendly *et al.* at the University of Birmingham [48, 47, 31] developed Narcissus. Narcissus is a virtual reality system designed to convey information on large complex systems.

Narcissus uses self-organizing systems to generate the visualizations. Each of the objects is given a behaviour that determines its movement and eventually its final position. There are two forces that cause movement upon the objects:

- All objects in the system exert a repulsive force on all of the other objects.
- Active relationships between objects lead to attractive forces being exerted between related objects.

When the objects are first added to the space there is quite a bit of movement until the objects reach a stable state. The prototype for the system was written in SELF [106]. The current system is written in KQML ⁵.

Narcissus isn't really a software visualization system as it is an information visualization system. Its primary goal was to come up with a way to present large sets of data to users that 'will lead the users to form an intuitive understanding of the structure and behaviour' [48, abstract] of the data. The techniques described could be used, as future work, to form a layout.

3.2.4.2 VOGUE

Koike *et al.* at the University of Electro-Communications in Tokyo [53, 54] have created a prototype 3D visualization system called VOGUE. They have done (and are still doing) quite a bit of software visualization work using VOGUE.

VOGUE (Visualization Oriented Generic User-interface Environment) is composed of two main modules; an object-oriented database, and a 3D grapher module. These have been developed on UNIX stations using Lisp⁶. Several of the different types of visualizations are discussed in the following paragraphs.

VOGUE is similar to SOVIST-3D in that it provides multiple types of visualizations. Some of the ways of laying out the visualizations would be good additions to SOVIST-3D (see 6.3.3). However, unlike SOVIST-3D, the visualizations are drawn from information stored in a database and so the visualizations are unable to interact with the code.

⁵KQML stands for the Knowledge query and manipulation language [34].

⁶Actually the common lisp object system (CLOS).

Figure 3.1: Concept of a 3D representation of class libraries [53, Figure 3, page 184].

Class hierarchies. They have developed a layout to show the relationship between methods in classes and sub-classes [53]. This is done by uniting multiple views (see page 23), the two views being the class hierarchy and the methods in a class. Classes are drawn in a 2D tree structure in the xy-plane (like that of the 2D layout, see 5.2.2.3). Each method has the same xy-coordinates as the class to which it belongs, and all methods with the same name have the same z-coordinate. An example is shown in figure 3.1.

Fractal Approaches for visualizing huge hierarchies. A fractal-based approach is used to visualize huge hierarchies [57]. The geometrical characteristics of a fractal, self-similarity, make it possible to visually interact with trees in the same manner at every level. The similar view is obtained whenever a subtree, or part of a subtree is magnified, which is done through use of the mouse.

Version Control and Module Management. A layout to show the relationships between versions' controls and module management [56] has been developed. The system uses VOGUE and is called VRCS. This is again done by uniting multiple views (see page 23), the two views being the version history of files and the files needed for each release. Each version history is displayed as a 2D tree in the yz-plane. A cube is used to represent each version. Files (trees) in the same module are placed near each other in the z-plane. Releases are noted by spheres placed along the z-axis that are then connected to the files appropriate to the release. An example is shown in figure 3.2.

A bottom-up approach for visualizing program behaviour. They have developed a system that visualizes the behaviour of programs using a bottom-up approach [55]. It can be seen as a visualized tracing mechanism. As the program runs, it draws local pictures according to a set of rules. The combination of the local pictures gives the visualization. Local drawing rules are defined for specific conditions (usually for control flow) such as conditional branches, iterations, function calls etc. For example, the local drawing rule for a conditional branch is

Figure 3.2: A 3D framework for visualizing version/module information [56, Figure 1].

1. draw a cube
2. evaluate the control expression, and draw a blue sphere
3. if the expression is true; turn the direction by -30 degrees, otherwise turn the direction by 30 degrees.

3.2.4.3 3D representation for program execution in Lisp

Lieberman at MIT [68] has developed a system that allows an abstract 3D representation for computer programs in the context of an interactive dynamic debugger. It was designed to show the program execution of Lisp programs. This is done using coloured polyhedra with text labels to represent program elements. A large icon represents a function with graphically enclosed smaller icons representing arguments.

Execution is shown through the use of a ‘move the context’ focus shifting. Lieberman calls the technique re-rooting. The concept is that as the user moves down the tree to its branches he/she can view the result by a new tree that has the target node as its root. This keeps the current target node as the largest displayed object. Colour is used to show whether the links are downward (cool colours) or upward (warm colours). Figure 3.3 shows an example.

Evaluation of a program object is shown through the use of rotation. There are two different points of view of the visualization. The ‘heads on’ view is that of the program execution. The second is that of the stack which is shown in the third dimension. This is another example of uniting multiple views (see page 23).

3.2.4.4 Pavane

Roman *et al.* at Washington University [99] have created Pavane, a system for declarative visualizations of concurrent computations. Pavane creates visualizations of Swarm [100] programs, a concurrent programming language.

Figure 3.3: This shows the re-rooting of a tree. The root is moved from node 1 to node 2. To the right of the tree, the rectangles show how the containment is changed. [68, Page 114].

Pavane is declarative in the sense that visualization is treated as a mapping from program states to a 3D world. The mappings are specified by a rule-based notation: rules may be added, deleted and modified at any time during the visualization.

3.3 Summary

This section gives a brief summary of what was learned from previous systems and applied to SOVIST-3D, as well as, how SOVIST-3D differs from the related work.

3.3.1 What was learned from the related work

The following summarizes what has been learned from the related work and subsequently applied to SOVIST-3D:

- Moving to 3D can help in presenting the visualizations in ways that 2D can not. Almost all of the papers present just a few ways in which it can help. An earlier section (see 2.2) has a more complete explanation.
- That having more than one kind of visualization can help explain the code better than any one single visualization could. This led to SOVIST-3D providing multiple viewers (see 4.2).
- That even though visualizations can be trying to visualize the same type of information there are many ways of accomplishing this. This led to the idea of layouts (see 4.3) that allows multiple ways of displaying information in the same type of visualization. The cone tree layout (see 4.3.2.1) is one such layout used in SOVIST-3D that is similar to one found in the above related work.
- That both the static and dynamic aspects of the code need to be visualized to fully understand the code.

3.3.2 How SOVIST-3D differs from the related work

SOVIST-3D differs from the related work in many ways, which of course are different for the different systems. The main ways it can differ from any of the systems are:

- Many of the visualizations are in 2D. SOVIST-3D is in 3D which has the advantages discussed in an earlier chapter (see 2.2).
- Many of the visualizations are not created within the same environment as the program that is being visualized. SOVIST-3D is created completely from within the Smalltalk environment.
- Few of the visualization systems are for object-oriented environments. Object-oriented systems provide concrete items (objects) to be visualized. In SOVIST-3D the user can create specialized 3D objects (i.e. a safe to represent a bank account object (see 4.4)). The specialized 3D object will then be used as the object in the visualization⁷.
- Many of the systems discussed depend on user intervention to create the visualizations. In other words, the user has to do something, such as annotating the code, to create the visualization. SOVIST-3D makes use of the meta-facilities in Smalltalk to automatically create the visualizations.
- Many of the systems create only one type of visualization. SOVIST-3D has created an extensible framework in which many different visualizations can be produced. SOVIST-3D goes even further by allowing different layouts to be created for each of the different types of visualizations. SOVIST-3D currently consists of three different types of visualizations and four different types of layouts which create seven different visualizations.
- Most of the systems are done with the program once the visualization has been created. SOVIST-3D, because the visualizations are created completely within

⁷In the sequence diagram layout (see 4.3.1.2), the objects are used as markers above the cylinders rather than replacing the cylinders themselves.

the Smalltalk environment, allows the user to interact between the visualization and the program.

- Unlike most of the systems, SOVIST-3D offers visualization of both the static and dynamic aspects of the code.
- SOVIST-3D offers a navigation system to allow the visualization to be viewed in different ways, allowing more knowledge about the code being visualized.

Chapter 4

SOVIST-3D

This chapter introduces SOVIST-3D and some of its concepts. This chapter isn't meant as a complete description on how to use the system. That is done in the users manual (see appendix B). Frequent references will be made to the appropriate section in the users manual which gives more details on the topic in question. Chapter 5 deals with the design and implementation of SOVIST-3D.

The chapter starts out with a general overview of SOVIST-3D, and then goes into more details describing the important aspects of the system.

4.1 What is SOVIST-3D?

The simple answer to the question is: SOVIST-3D is a framework for creating 3D visualizations. The framework gives programmers the ability to both to create different types of 3D visualizations and to display the information generated in a variety of ways. SOVIST-3D currently can display seven different types of visualizations.

The users goals of the framework are (restating the goals in the introduction (see 1.3.1) to:

1. create multiple types of visualizations (for example, the object hierarchy and the results of a message trace) that have the same ‘look-and-feel’. This is what we refer to as a viewer.
2. display the same information in multiple ways (for example, the object hierarchy as B-Trees or cone trees). This is what we refer to as a layout.
3. allow different methods of identifying objects in the visualizations.
4. allow the users to navigate the visualizations (in other words, allow the visualizations to be manipulated to increase understanding).

The other goals that deal more with how the system is designed are discussed in chapter 5. The goals listed above form the remaining sections of this chapter.

The key point to remember is that in order to create a visualization you need both to gather information to display (a viewer) and to have a way of displaying the information (a layout). Once the visualization has been created and displayed you need to be able to identify the objects in the visualization and navigate through the visualization.

4.2 The Viewers

The viewer is responsible for creating a particular type of visualization. Examples include: the object hierarchy, the results of a message trace, parts of an object, the state machine, the collaborations of objects, etc. The first three of the examples are viewers in SOVIST-3D. These three were chosen since there are 2D equivalents created by the COOP group (see 3.1.3.4). In addition, they demonstrate many of the ideas behind the framework and make use of some of the reasons for using 3D discussed in section 2.2.

In this section we will take a look at the general ‘look-and-feel’ of the viewers and then a closer look at the three viewers developed as part of SOVIST-3D.

4.2.1 ‘Look-and-Feel’

The framework results that all of the viewers have the same ‘look-and-feel’. This includes the following:

- The screens of the viewers have the same layout. This is described in more detail below (see 4.2.1.1).
- The viewers have a common menu system, except for the viewer and layout menus that are unique to each viewer. In addition, each layout can add items to the end of the viewer menu that are layout specific. The menus perform functions such as: changing the layout, transforming objects or the scene, setting basic options for the viewer, zooming out on the visualization, etc. Section B.1.3 contains details.
- The resulting viewers can be navigated the same way. This is described in more detail below (see 4.5).

4.2.1.1 The Parts of a Viewer

Figure 4.1 shows a 3D viewer and its parts. The parts are mostly self explanatory, with the possible exception of the object display pane which is described in more detail in section 4.4. Section B.1.2 contains details.

Throughout the rest of the thesis certain terminology is used to refer to various types of objects in the OpenGL pane. The terms are:

Object This is a three-dimensional object that represents either a class or an instance of a class within a visualization.

Marker These are used within the sequence diagram layout to help describe the objects, since all objects used are cylinders (see B.2.5).

Connector This is something that connects two objects together. Connectors are usually lines or arrows.

Figure 4.1: A 3D Viewer.

Background This is part of the OpenGL pane where no object, marker or connector appears.

4.2.2 Hierarchy Viewer

A hierarchy visualization shows the inheritance among classes, that is, it shows the taxonomic relationship between a superclass and its subclasses. In UML notation to show the inheritance relationship, a solid line is drawn from subclass to superclass with a large closed, unfilled triangular arrowhead at the superclass end. Figure 4.2 shows the classes used in the bank account example used throughout this thesis. The same visualization in SOVIST-3D can be seen in figure C.12. The definitions of the classes are in appendix D.1.

The basic idea behind the viewer is that classes are added to the visualization, either by using the popup background menu or the viewer menu. Once a class has been

Figure 4.2: A hierarchy visualization of the classes associated with the **BankAccount** class.

added, its superclasses (immediate or all) and its subclasses (some of the immediate, all of the immediate or all) can be added by using the popup menu for the object. Objects can be removed in the same manner. This quickly allows the user to view the object hierarchy of the classes desired.

There are three layouts that can be used with the hierarchy viewer:

1. A default layout (see 4.3.1.1).
2. A 2D B-tree layout (see 4.3.2.2).
3. A cone tree layout (see 4.3.2.1).

Section B.3 contains details on using this viewer.

Figure 4.3: A parts visualization of a **BankAccount** object.

4.2.3 Parts Viewer

A parts visualization shows the parts of a particular instance of an object. In the case of this software, that is the value returned from a particular use case¹. Figure 4.3 shows the result of the running of **BankAccount example1** and its parts (owner, balance and transactions). This is done using UML notation. The part is an instance of a class with a solid line under the name of the object and the name of the part is optionally attached to the connector between the parts². The same example using SOVIST-3D is given in figures C.10 and C.11.

The basic idea behind the parts viewer is that a use case is selected. This can be done via the background popup menu or the viewer's menu. The use case is run and the object returned when the use case is ended is added to the visualization. The parts of that object (some or all) can then be added to the visualization, and their parts can in turn be added, giving a picture of the type of object returned from the use case.

There are three layouts that can be used with the parts viewer:

¹The definition used for a use case in this thesis can be seen in Appendix A

²This is a simplification of the UML diagrams omitting such things as the roles and multiplicity. The explanation given is sufficient for this thesis (see [11, 13] for further details).

1. A default layout (see 4.3.1.1).
2. A 2D B-tree layout (see 4.3.2.2).
3. A cone tree layout (see 4.3.2.1).

Section B.4 contains details on using this viewer.

4.2.4 Message Tracer Viewer

A message tracer visualization shows the messages sent during the running of a use case³, after a filter has been applied. One of the most popular means of viewing this information is a sequence diagram⁴. We will use a sequence diagram example to explain the concept of a message tracer visualization.

A sequence diagram describes how each use case is satisfied by showing how the objects communicate. The sequence diagram shows the relevant⁵ message traffic and relevant objects involved in the use case. Figure 4.4 shows a typical sequence diagram. The sequence diagram shows the use case for the code in listing 4.1.

The vertical lines in the sequence diagram represent each of the interesting objects⁶ in this use case. In this case they are the `BankAccount` class and `aBankAccount` (an instance of the bank account class). Above the vertical bar is the name of the object.

Also represented, as a vertical bar with the slashes attached, is the system border. This shows the inputs into the system. In our case the system border is `useCase1` as shown in figure 4.4.

³The definition used for a use case in this thesis can be seen in Appendix A

⁴Sequence diagram is the term used by UML [11, 13]. They are more commonly known as interaction diagrams from Jacobson's OOSE [50]. The new term is used throughout this thesis instead of the term interaction diagram.

⁵Relevant is being used in this case to mean the classes/methods that have been selected by the use of filters.

⁶There are usually many more objects involved in a use case that we are really not interested in. Such objects would be things such as collections, strings, magnitudes, etc. These classes have been filtered out.

Figure 4.4: Sequence Diagram

Listing 4.1 Bank Account Use Case 1

```
class:           BankAccount
superclass:      Object
instanceVariableNames: 'owner balance accountNumber'
poolDictionaries: ""

class methods:

new
    super new initialize.

useCase1
    BankAccount new
        owner: 'Rob Tyson';
        accountNumber: 123456;
        balance: 1000.

instance methods:

owner: aString
    owner := aString.

accountNumber: aNumber
    accountNumber := aNumber.

balance: aNumber
    balance := aNumber.
```

The messages that are sent from one object to another are horizontal arrows. The arrow goes in the direction of the message sent. For example, the BankAccount sends the initialize method to aBankAccount. Messages that are sent to the same object are shown as three sides of a square.

How long the method that was activated lasts can be seen by the box around the vertical line. Thus it can be seen that within the new message sent to BankAccount the **initialize** message was sent to aBankAccount. Time proceeds vertically, so the timing of messages can easily be seen.

3D visualizations of sequence diagrams for the bank account use cases (example1 and example2 in the BankAccount class that can be seen in listing D.1) can be seen in the appendix in figures C.5 and C.3.

The basic idea behind the message tracer viewer is that a use case is selected (which in this case means selecting a class and then a class method), a filter is defined (see B.2.4) and then the use case is executed. The message tracer visualization is then created. There are three different ways that the visualization can be executed:

1. The whole use case appears.
2. The messages sent are stepped through, being added to the diagram one at a time. The next message is added whenever the user presses the step button.
3. The messages sent are animated. They are added one at a time with a second or so between each message being added.

There is currently only one layout that can be used with the message tracer viewer:

1. A sequence diagram layout (see 4.3.1.2).

Currently the message tracer viewer can not be used with any use case that involves the user interface⁷.

Section B.2 contains details on using this viewer.

⁷Also, if the date class is used, it is highly suggested that it be filtered out using the message filters B.2.4, since it often causes the Smalltalk Environment to hang.

4.3 Layouts

Layouts are used to display the information gathered by the viewer. There are currently two different types of layouts (each of which in turn has two different layouts), pickable and non-pickable. They are described in the following sections.

4.3.1 Pickable Layouts

Pickable layouts are those that allow the objects/markers in the visualization to be selected⁸ using the left mouse button. Once objects are selected they can be moved around. Section B.1.5 contains details.

4.3.1.1 Default Layout

This layout does basically nothing in terms of creating an organized visualization. Whenever an object is added to the visualization, it places the item at the origin⁹, allowing the user to place the objects where desired.

This layout is used for both the parts and the hierarchy viewer. Examples of this layout can be seen in figures C.12 and C.10.

4.3.1.2 Sequence Diagram Layout

This layout provides a sequence diagram-like visualization for the message tracer viewer. Examples of this layout can be seen from figures C.3 to C.9 in appendix C.

All the classes and instances of classes are represented by cylinders, the messages being sent by arrows¹⁰. By default all the cylinders are set up so that they are in the xy plane. The system border is the cylinder with the smallest x co-ordinate. All

⁸Or picked, hence the term pickable.

⁹The origin, before the scene transformations are applied.

¹⁰The arrowheads are cones on the arrows, which slows down the rendering of the message tracer.

the cylinders are then placed with increasing x co-ordinates depending on when a message was first sent to the object. The cylinders can then be moved around since this is a pickable layout.

To help identify which object the cylinders represent, each of the cylinders can have two markers. The markers appear over top of the cylinder (i.e. the same x and z co-ordinate but a higher y co-ordinate). The one closest to the cylinder is the threeDObject associated with the object (see 4.4). The one above that is the threeDObject associated with the Class object. This marker indicates if the cylinder represents a class. Currently this is set to being a simple circle, since we couldn't think of a threeDObject that immediately screamed 'this is a class object'. The markers can be toggled on/off by using the *Viewer→Show Marker* menu item (see B.2.2).

There are two ways of indicating information about the messages for the object. There is the 'length of method' colours. All the cylinders start off with a certain colour¹¹. When a message is sent to an object from another object, the cylinder is coloured the next colour in the series for the receiving cylinder. The cylinder colour changes for the length of the method. If during the method, the object sends a message to itself, it then colours the cylinder the next in the series of colours for the cylinder. The colouring of the cylinder shows the duration of each message. The length of methods can be toggled on/off by using the *Viewer→Show Length of Method* menu item (see B.2.2).

The second way of showing messages is what we call the 'depth of the method', which is an example of an adapted 2D view (see page 22). Quite often it isn't the class that implements a method but one of its superclasses. We use the size of the cylinder to indicate this. The cylinders all start off at the same width. Each time a method is implemented by the superclass the cylinder size increases. If the method is not implemented by the class or its superclass but by its superclass superclass, then it increases in width again. The further up the hierarchy before a message is sent the 'fatter' the cylinder. The depth of methods can be toggled on/off by using the

¹¹There are currently five colours that are cycled through. They are: red, blue, green, yellow and brown.

Viewer→*Show Depth of Method* menu item (see B.2.2).

4.3.2 Non-Pickable Layouts

Non-pickable layouts are those that don't allow the individual objects to be moved around. This is because the objects in question have been placed in a specific location and moving them to another location would confuse the visualization.

Both of the layouts below create a series of visualizations for each set of connected objects, that is, if two objects such as the collection hierarchy and the magnitude hierarchy have been created but they haven't been joined by the Object class, there will be two separate trees¹². Each tree is then placed further down the x-axis; so that each of the full trees can be observed in each case.

4.3.2.1 Cone Tree Layouts

This layout is based upon the cone tree developed by Robertson, Mackinlay and Card [97] (see 3.2.2). It was developed to help maximize screen space (see 2.2.2.2).

The idea is that you place the objects so that they are in a cone-like structure. The root object is at the apex of the cone and all of the children are equally spaced along the base. Each child node is in turn the apex of a new cone. The height of each level in the tree is the same, but the radius diminishes at each level. This way there is little or no overlap between the objects.

This layout is used for both the parts and the hierarchy viewer. An example can be seen in figure C.13.

4.3.2.2 2D B-Tree Layout

This layout creates B-Tree like structures in the x-y plane. The tree is displayed so that objects at each level are evenly spaced, that is, objects won't be centred under

¹²Actually just adding the class to the diagram is not enough, it has to be explicitly told about the connection.

their parent. It is useful mainly for visualizing a small number of objects.

This layout is used for both the parts and the hierarchy viewer. An example can be seen in figure C.11.

4.4 Understanding the Objects in the 3D Visualizations.

The visualizations created by the viewers described in section 4.2 are only useful if the objects in them have to be easily understood by the user. Since using text is difficult in 3D visualizations (see 2.2.1), we have come up with other methods to figure out what the objects in the visualization represent. In SOVIST-3D, this is done in two ways:

1. by having a description of the object shown in the object display pane (see figure 4.1) whenever the cursor is over an object, connector or marker. The user can then just move the mouse to be overtop of any item that he/she wants to identify.
2. by using a specific 3D object to represent the system object. For example, a 3D automobile could represent a Car class.

Each of the layouts has a default type of object. The default type is different for each layout and is defined by the class method `default3DObject`. To help distinguish the objects, as described in the second point above, each class can define its own type of `ThreeDObject`. To define a `ThreeDObject` for every instance of the class or for when there is no instance of the class, the class method `threeDObject` must be implemented to return the `ThreeDObject`. The `ThreeDObject` should be a subclass of `Solid3DObject` and be centred on the origin. Listing 4.2 shows an example of a class specifying its `threeDObject`. In this case, the `BankAccount` will be represented by a cube with the bitmap of a bag of money on each side of the cube. The resulting

object can be seen in figure 4.5. In this case a textured bitmap is used but it could just as easily be an object such as a piggy bank or a bank vault.

Listing 4.2 BankAccount threeDObject method

```

three D object

threeDObject
    "Return the threeD object to use"

    ^ (Cube width: 1 height: 1 depth: 1
        coating: (TexttrueCoating fromFile: 'money.bmp');
        yourself.

```

Figure 4.5: BankAccount's ThreeDObject.

Another way of representing the object is by having an object change depending on the instance of the object. To do this the class must implement the method `threeDObject:`. The method can then create a `ThreeDObject` using the specific instance of the class. If the method is not implemented or there is no instance of the class (as is the case in the hierarchy viewer), then the class attempts to use the `threeDObject` method of the class and then the layout's default class. Listing 4.3, shows the method for the magnitude class. This will produce an `OpenGLText` object with the string being the first 11 characters of the `printOn:` method of the

object¹³.

Listing 4.3 Magnitude threeDObject: method

```

three D object

threeDObject: aMagnitude
    "Return the threeD object to use"
    | aString |

    aString := aMagnitude asString.
    aString size > 11
        ifTrue:[aString := (aString copyFrom:1 to: 11)].

    ~(OpenGLText fromString: aString)
        coating: Color yellow.

```

Examples of the objects being used can be seen in various colour plates in Appendix C.

4.5 Navigation in a viewer

One of the benefits of 3D is that you can manipulate the visualizations to view them in various ways. This is done through the navigation system. Currently, there is no common 3D navigation system. The navigation system used in this thesis was developed to try and maximize the ability to manipulate the visualizations but at the same time not be too complicated.

The main parts of the navigation system are:

- The mouse. A brief overview is given here. Section B.1.4 contains details.
 - When the mouse cursor is over an object, connector or marker when no buttons have been pressed it causes a text description of the object to be shown in the object display pane (see 4.4 for details).

¹³This does produce text which has been noted as being difficult to use in 3D diagrams. These objects though are much bigger than normal text could be and are not prevalent in the diagrams.

- In pickable layouts (see 4.3.1), the left mouse button allows objects/markers to be selected. Multiple objects/markers can be selected via the shift key. If multiple objects are under the cursor, the CTRL key brings up a multiple selection list box with a list of the objects/markers.
- The right mouse button brings up a viewer specific popup menu depending on whether the mouse cursor is over:
 1. the background.
 2. a connector.
 3. an object or marker.

If multiple objects/connectors/markers are under the cursor, then a single selection list box shows the items, allowing one of them to be selected.

- In pickable layouts (see 4.3.1), the individual objects can be moved by:
 1. selecting a single object, and then using the *Transform→Translate 3D Object* menu item to select a new location for the object. Section B.1.5.1 contains details.
 2. selecting one or more objects and then moving them by holding down the left mouse button and dragging the objects. The direction they are moved and how far each pixel movement is relative to OpenGL co-ordinates can be set using the *Transform→Translation Controls* menu item (see B.1.6). Section B.1.5.2 contains details.
- The entire scene can be translated, rotated and scaled using the *Transform→Transform Scene* menu item. Section B.1.7.2 contains details.
- The camera position can be translated and rotated using the keyboard keys (page up; page down; right, left, up and down arrow keys; and the CTRL key). How big the movements are can be set using the *Transform→Translation Controls* menu item (see B.1.6). Section B.1.5.2 contains details.

4.6 Summary

This chapter has shown the basic ideas behind how SOVIST-3D creates visualizations. There are **viewers** that gather information and **layouts** that decide how the information is going to be displayed. Also discussed were how the users can identify objects in the visualizations and navigate through the visualization.

Chapter 5

Design and Implementation

This chapter contains details on the design and implementation of the 3D viewers. The previous chapter introduced the software. A small users' manual on how to use the software can be found in appendix B. In this chapter, we will be giving a general overview of the software and only going into more detail for some of the more interesting design/implementation features.

Throughout this section design patterns [38] are used to help explain how the software is designed. In addition to using the design patterns, we tried to follow the coding patterns referred to by Beck [7]. The diagrams we use follow the UML [11] notation.

There are really two parts to SOVIST-3D that are being addressed in this chapter. The first is the framework that was created and the second is the visualizations that make use of the framework. These two sections comprise the bulk of the chapter.

5.1 The Framework

This section discusses the design and implementation of the framework. It will start out giving an overview of the framework. This will be followed by the goals in the design of the framework and then go into some detail about how the goals were accomplished.

5.1.1 Overview of the Architecture

This section gives a brief overview of the architecture of SOVIST-3D. The framework of SOVIST-3D is composed of three different parts. It is somewhat similar to MVC¹ in how the pieces fit together. The parts are:

CommandLists are used to keep track of the information to be visualized. The command lists are based on command list created for the COOP group (see 3.1.3.4). The command lists are used to know what objects are currently in the visualization. In some cases, like for the message tracer command list, the **CommandList** generates the information. Each viewer has one **CommandList** associated with it. This is like the model in MVC.

Viewers provide the user-interface for the visualization. It allows the users to create the visualizations (add or remove information from the visualizations). It has one **CommandList** and one or more layouts. This is like the controller in MVC. Further information on the design of the viewers can be found in section 5.1.7.

Layouts define how the information is to be displayed, in other words the lay out of the information. There can be many layouts for any one viewer. A layout can be used by more than one viewer. This is like the views in MVC. Further information on the design of the layouts can be found in section 5.1.8. The layouts get all of the information about which objects to add/delete from the layout from the viewer. At all times, all of the layouts for a given viewer are in sync. This is done by something much like the Observer pattern [38, Pattern is found on page 293].

Some background on the design process follows. We knew when we started development that we wanted to create a framework, and we had certain goals in mind (see 5.1.2), but were unsure of what abstractions were needed to create the framework. We decided to start off by building several visualizations and then abstracting

¹MVC stands for Model-View-Controller. It is a concept introduced originally in Smalltalk to encapsulate some data together with its processing (the model) and to isolate it from the manipulation (the controller) and presentation (the view).

out the information that could create the framework. We chose three visualizations to start (the default layout of the Hierarchy and Parts viewers and the sequence diagram layout of the Message Tracer viewer). This method of developing a framework we found out afterwards is one advocated by Roberts and Johnson [93].

The initial three visualizations led to the use of CommandLists and Viewers. When we tried to add the ability to display the Hierarchy as a cone tree, it led to a refactoring of the responsibilities of the Viewers into Viewers and Layouts.

5.1.2 Framework Design goals

These are the main goals with which the framework was designed (some of the goals are repeated from section 4.1):

1. to create the visualizations in the same environment as the code to allow for interaction between the two.
2. to create the visualizations automatically (i.e. no annotation of the code is needed).
3. to visualize both the static and dynamic aspects of the code.
4. to create multiple types of visualizations (for example, the object hierarchy and the results of a message trace) that have the same 'look-and-feel'. This is what we refer to as a viewer.
5. to display the same information in multiple ways (for example, the object hierarchy as B-Trees or cone trees). This is what we refer to as a layout.
6. to allow different methods of identifying objects in the visualizations.
7. to allow the users to navigate the visualizations (in other words, allow the visualizations to be manipulated to increase understanding).
8. to make it easy for new viewers and layouts to be added to SOVIST-3D.

Some of these goals were examined in more depth in chapter 4. How the remaining goals are achieved form the discussion for the rest of this section.

5.1.3 Environment

The target environment that was chosen for SOVIST-3D was Visual Smalltalk V3.1.1 using Window Builder Pro V3.1.1. It runs on both the Microsoft NT and Windows 95 platforms.

In order to fulfil our goal of having the code and visualizations created within the same environment we had to develop a way to create 3D diagrams in general from within Visual Smalltalk, since this is not currently available. OpenGL was chosen for this.

5.1.3.1 OpenGL

OpenGL (Open Graphics Library) is a software standard interface for producing interactive 3D graphics.

OpenGL, originally developed by Silicon Graphics Inc., is currently a software standard that runs on many platforms, including Microsoft's NT and Microsoft's Windows 95. OpenGL is currently defined by an industry consortium known as the Architecture Review Board (ARB). The OpenGL ARB is composed of such industry leaders as Digital Equipment Corporation, IBM, Intel, Microsoft, and Silicon Graphics. OpenGL version 1.1 is the current release.

OpenGL [82, 84] consists of about 120 functions that allow users to build up 3D models from a small set of geometric primitives such as points, lines and polygons. From these primitives, sophisticated objects can be rendered. In addition to building geometric models, these OpenGL functions allow the programmer to interact with the objects, control colour and lighting, manipulate pixels, and perform such tasks as alpha blending, anti-aliasing, creating atmospheric effects, and texture mapping.

Since OpenGL is a hardware-independent interface, it contains no facilities for providing window, mouse, or keyboard commands. These must all be setup by the user. The work done in this thesis uses Microsoft's OpenGL implementation [88, 89] in Windows NT and Windows 95. The OpenGL functions are stored in DLLs.

The OpenGL DLLs provided by Microsoft still had to be interfaced with Visual Smalltalk. A basic interface between Visual Smalltalk and the OpenGL DLLs was done for two articles in the Smalltalk column in JOOP [61, 62]. This work not only provides an interface to the DLLs, but it also provides an object-oriented framework in which to build 3D graphics' applications. This work has been greatly expanded upon². The main additions/changes to the OpenGL implementation since the articles are: picking and selecting have been added, lighting has been changed, the OpenGL pane has been improved and the structure of how 'real world' objects (e.g. faces, etc) has been changed a great deal, with many additional objects being added.

5.1.3.2 Problems with the OpenGL implementation

The current OpenGL implementation is quite slow while running on the machines in the COOP group (see 3.1.3.4). This is mainly due to the following:

- The machines are slow by today's standards and are not using any specialized graphic cards.
- The current implementation contains a transformation matrix for each object. At render time the matrix is added to the stack using the OpenGL matrix operations. OpenGL matrix operations are slow³. The OpenGL Programming Guide [82] suggests instead to use the transformation functions, since these are heavily optimized.
- Composite objects, which are objects composed of other objects, have a transformation matrix for each of the objects in their composition. For example, a

²Work on expanding the OpenGL interface was done in conjunction with Mike Valenta.

³This information was found after the OpenGL implementation was completed. It was tucked away in an appendix of the OpenGL Programming Guide [82].

cube is made up of six faces, the matrix for each of the faces is added to the stack at render time using matrix operations. A better solution would be to have just one matrix for each ‘real world’ object, i.e. one for the cube instead of six, reducing both the number of matrices and the number of matrix operations performed.

To summarize, the OpenGL implementation, like OpenGL itself, is not optimized for performance.

5.1.4 Automatic Visualizations

SOVIST-3D can create automatic visualizations because:

- Smalltalk has meta-level facilities that allow Smalltalk programs to get information about programs as they run. This allows the visualizations to be created without annotating the code.
- There are layouts that allow the information to be displayed without the user deciding each time where the objects should go.

5.1.5 Dynamic and Static Visualizations

SOVIST-3D can create both dynamic and static visualizations because:

- Smalltalk has meta-level facilities that allow Smalltalk programs to get information about programs as they run. This allows the visualizations to be created without annotating the code.
- Use cases (see appendix A) are the focal point in running the system to get runtime information. This allows a straightforward interface into the code and a definite endpoint with which to create dynamic visualizations.

5.1.6 Extendable framework

We believe that it is easy to add viewers and layouts to SOVIST-3D. This is because a large part of the functionality has been abstracted into the framework's abstract classes. Creating new viewers/layouts is just a matter of creating new subclasses and then implementing a few well-defined methods.

5.1.7 Viewers

The viewers form the heart of SOVIST-3D. They allow the user to create different visualizations that will have the same 'look-and-feel'.

To have a consistent 'look-and-feel', all of the viewers are subclasses of an abstract class **ThreeDViewer**. The viewers' class hierarchy can be seen in figure 5.1. The aggregations of the abstract class **ThreeDViewer** can be seen in figure 5.2.

Figure 5.1: The viewers' class hierarchy.

Most of the functionality that is common to all of the viewers is encapsulated in the **ThreeDViewer** class. The exceptions to this are the class methods (table 5.1) and instance methods (table 5.2) that must be implemented for each of the viewers. The viewers also need all of the layouts to understand certain messages (see 5.1.8).

The class methods contain information that is common to all the viewers, such as the

Figure 5.2: The **ThreeDViewer** aggregates.

Method	Description
<code>buildView:forModel</code>	Creates the view used for the viewer.
<code>buildMenus:forModel</code>	Creates the menus used for the viewer.

Table 5.1: Class methods that viewers must implement.

events the OpenGLPane handles, but because we are using Window Builder Pro it is much simpler to have the panes/menus defined again for each viewer⁴.

Method	Description
initializeLayouts	Creates the layouts used for the viewer.
rebuildDiagram	Recreates the <code>Composite3DObject</code> that actually shows the visualization.
popupBackgroundMenu	the popup menu when the mouse is right clicked on the background of the <code>OpenGL Pane</code> .
popupMenuForConnector:	the popup menu when the mouse is right clicked on a connector.
popupMenuForObject:	the popup menu when the mouse is right clicked on an object.

Table 5.2: Methods that viewers must implement.

The rest of this section describes a few of the more interesting design features of the viewers. Each individual viewer is discussed in section 5.2.1.

5.1.7.1 Display Pane

The implementation of the display of the name of the object whenever the mouse is over the object (see 4.4) is based on an article [60] that created a bubble type help system in Smalltalk similar to the standard one in Windows. Information is shown in the display pane instead of in a bubble because the OpenGL Pane was being redrawn whenever the bubble was closed. SOVIST-3Dis slow to do this.

5.1.7.2 The Menus

As mentioned earlier, all of the menus except the viewer and layout menu are common to all of the viewers. The implementation of most of the items is quite straightforward

⁴Because of some of the functionality (i.e. menu selectors and events on the panes) it is recommended that to create a new viewer, one of the current viewers is called up in WindowBuilder and then copied to create the new viewer. This will make sure that everything expected is included in the creation methods.

except for the *Scene→Fit To Window* item.

In order to fit the scene⁵ to the window, the boundingBox of the items in the scene must first be established. Currently this is not done the best way. The best solution would be to ask the **Composite3DObject** for its bounding box and have it return it. This isn't done since the objects don't know their positions explicitly; they would have to calculate them from their matrix. In addition, they would have to know not only where their origin is, but also their size. Instead, since each of the layouts needs to know the positions of the objects to create them, the information used to calculate the bounding box is stored. Unfortunately this still leads to the problem of not knowing the size of each object.

It should be noted that the bounding box of the visualization contains the co-ordinates of the scene before the scene is translated using the global translate scene option. In addition, one of the problems with the algorithm is that the rotation of the scene is not taken into account, only making it really fit the scene if there are no current rotations. Once the bounding box of the scene is determined, it is fit to the window by the following algorithm:

- Calculate the mid-point of the bounding box.
- Find the distance from the centre to a corner of the bounding box.
- Until $15 < \text{field of view of the camera (fovy)} < 50$ degrees
 - Calculate the distance from the camera to the transform scene translation point.
 - Calculate the field of view (fovy) of the camera.
 - If $\text{fovy} > 50$ degrees then move the scene back one world co-ordinate unit.
 - else $\text{fovy} < 15$ degrees then move the scene forward one world co-ordinate unit.
- Set the camera's fovy to the one calculated.

⁵Scene is used here as a synonym for visualization.

5.1.7.3 The Viewer Menu

The viewer menu is built up in two ways. The first part is the items that are common to all layouts. This is built in the standard way in the `buildMenus:forModel:` method.

The second part is the layout specific items which are changed each time the layout changes. The items that were created for the previous layout are removed and then the layout is asked to give back a dictionary containing menu items for the layout. The menu items are then added to the bottom of the menu with the selector set to `runLayoutViewItem:`. This method can be seen in listing 5.1. The reason the message is redirected through the viewers is so that the viewer can then rebuild the visualization and update the display.

Listing 5.1 The `runLayoutViewItem:` Method

menu-handles

```
runLayoutViewItem: aMessage
    "Run the message sent with the message item"

    aMessage evaluate.
    self
        rebuildDiagram;
        update.
```

5.1.7.4 Dragging Objects

If the current layout is a pickable layout, (see 4.3.1) the objects/markers in the visualization can be selected. Once an object is selected it is added to the selected object collection. Its layout is asked to highlight it, the visualization is rebuilt and the viewer updated. After the object(s) has/have been selected, it/they can be dragged.

The dragging works as follows:

- whenever the left mouse button pressed event is received, the current location of the mouse is stored.
- whenever the mouse move with the left mouse button pressed event is received:
 - the OpenGLPane is set so it will only draw wire-framed objects. (This is done to help speed up the dragging).
 - the selected objects are translated, according to the information in the translationControls object (see B.1.6).
 - the new co-ordinates of the object are sent to the layout.
 - the layout is asked to redraw the visualization⁶.
 - the dragging flag is set to true.
- whenever a left mouse button up event is received and the dragging flag is true:
 - the OpenGLPane is set back drawing filled polygons.
 - the objects that were selected are unselected and unhighlighted.
 - the dragging flag is set to false.

5.1.8 The Layouts

Layouts are what actually determine how the visualization will be displayed. Since one of the goals is to be able to have multiple ways of displaying the same information, layouts were created. The idea was to abstract away from the viewer how the information was to be displayed. Whenever the OpenGL pane needs to display the information, it asks its current layout for the information. Since each viewer expects that each of its layouts will respond to the same messages, to aid in this, all of the layouts have been created as subclasses of **ViewerLayout**. The layouts' class hierarchy can be seen in figure 5.3. The aggregations of the abstract class **ViewerLayout** can be seen in figure 5.4.

⁶The dragging could be sped up by not redrawing the whole scene but only the objects that are selected. This was not done since it seems that time used to create the visualization is short compared to the amount of time it takes to render it.

Figure 5.3: The layout class hierarchy.

Figure 5.4: The **ViewerLayout** aggregates.

All of the layouts must implement certain methods that the viewers expect to be implemented. The class methods can be seen in table 5.3 and the instance methods in table 5.4.

Method	Description
default3DObject	returns the default ThreeDObject used in the layout
defaultConnector	returns the class of the default ThreeDObject used as the connector in the layout

Table 5.3: Class Methods that all layouts must implement.

Method	Description
createComposite3DObject	returns a Composite3DObject containing the layout
boundingBox	returns an OpenGLBoundingBox specifying the bounding box of the layout.
classFromAnInspectedObject:	returns the class of an InspectedObject
inspectedObjectFrom3DObject:	returns an InspectedObject that contains information about the 3DObject.
menuSelectorFor:selector:	returns a message that contains the menu name for the layout along with the selector and receiver passed in.
menuItemstoAddToViewerMenu	returns a dictionary of items to be added to the viewer menu specific for the layout. The dictionary should contain the label as the key and has as the value, a selector message that should be sent.
updateViewerMenu:	updates the layout specific menu items (i.e. grays them out, check them etc.) as needed.

Table 5.4: Methods that all layouts must implement.

5.1.8.1 Adding Layouts to a Viewer

Layouts are added to a viewer via the **initializeLayouts** method (see listing 5.2). The method is called during the initialize method of the class. The first item in the

Listing 5.2 Initialize Layout Method for the Hierarchy Viewer

*initializing***inititalizeLayouts**

“Returns an ordered collection of ViewerLayouts.
The first item is used as the default layout”

~OrderedCollection new

```
add: (ThreeDDiagram menuName: '&Default');  
add: (ConeTrees menuName: '&Cone Trees');  
add: (TwoDTrees menuName: '&TwoD Trees');  
yourself.
```

ordered collection is set as the default layout for the viewer.

Thus to add a new layout to a viewer two things have to happen. A layout corresponding to the interface needed for the viewer needs to be created (see 5.2.1) and the layout must be added to the **inititalizeLayouts** method of the viewer.

The layout menu is created by asking each of the layouts for its menu name and selector. The current layout is then checked. The user can change between layouts by selecting another item from the layout menu.

Currently in SOVIST-3D all of the layouts for a particular viewer are always present. This creates a large storage overhead for large visualizations, but improves the speed of switching from one layout to another.

5.1.8.2 Types of Layouts

It was discovered while creating the layouts that there are some layouts which become less effective in terms of conveying information if the user can move the objects around. Hence we created two types of layouts: pickable and non-pickable (see 4.3). An abstract class **PickableLayout** was created to contain the information needed for a layout to be pickable.

The aggregates of **PickableLayout** class can be seen in figure 5.5.

Figure 5.5: The **PickableLayout** aggregates.

Method	Description
defaultHighlightColor	returns the default colour to highlight objects with

Table 5.5: Class Methods that all Pickable Layouts must implement.

In addition to the methods all layouts must implement, all of the pickable layouts must implement certain additional methods. The class methods can be seen in table 5.5 and the instance methods in table 5.6.

5.2 Visualizations

This section describes some of the design issues behind the visualizations created in SOVIST-3D. The section is divided into two subsections: the first describing the viewers and the second the layouts.

5.2.1 The Individual Viewers

This section takes a look at each of the viewers. The following information is given for each viewer:

- the aggregates of the viewer.

Method	Description
highLightObject:	highlight the given object to denote it has been selected
unHighLightObject:	unhighlight the given object to denote that it is no longer selected
unHighLightObjects:	unhighlight all the objects in the collection
getOpenGL3DPointFrom:	returns an OpenGL3DPoint where the object is currently located
translateObject:by:	translate the object by an OpenGL3DPoint
randomObjects	randomly places the object in the layout

Table 5.6: Methods that all pickable layouts must implement.

- the methods its layouts must implement.
- any other information that may be of interest.

5.2.1.1 Hierarchy Viewer

```

graph TD
    H3DViewer[Hierarchy3DViewer]
  
```

Figure 5.6: The **Hierarchy3DViewer** aggregates.

The aggregates of the hierarchy viewer can be seen in figure 5.6. The hierarchy viewer uses an **HierarchyList** as its command list. The rest of the design is quite straightforward. The methods each of its layouts⁷ must implement can be seen in table 5.7.

Method	Description
existingInspectedObjectFrom:	returns the existing InspectedObject that is the same as the object passed in, or nil if there is no such object.
existingInspectedObjectFromASymbol:	returns the existing InspectedObject that is the same as the symbol passed in, or nil if there is no such object.
add:	adds an InspectedObject to the layout.
connect:to:label	connects two InspectedObjects with a connector and sets the label of the connector.
disconnectAllTo:	disconnects all the connectors going to an InspectedObject .
remove:	removes an InspectedObject from the layout.

Table 5.7: Methods that hierarchy viewer layouts must implement.

Figure 5.7: The **Parts3DViewer** aggregates.

5.2.1.2 Parts Viewer

The aggregates of the parts viewer can be seen in figure 5.7. The parts viewer uses a **BasicCommandList** as its command list. The rest of the design is quite straightforward. The methods each of its layouts⁸ must implement can be seen in table 5.8.

Method	Description
add:	adds an InspectedObject to the layout.
connect:to:label	connects two InspectedObjects with a connector and sets the label of the connector.
disconnectAllFrom:	disconnects all the connectors going from an InspectedObject .
remove:	removes an InspectedObject from the layout.

Table 5.8: Methods that parts viewer layouts must implement.

5.2.1.3 Message Tracer Viewer

The aggregates of the message tracer viewer can be seen in figure 5.8. The message tracer viewer uses a **MessageTracerList** as its command list. The methods each of its layouts⁹ must implement can be seen in tables 5.9 and 5.10.

Method	Description
defaultDiagram	the class that implements the diagram.

Table 5.9: Class methods that message tracer viewer layouts must implement.

The message tracer is based upon the interaction diagram generator (IDG) created by Wang [119] and continued by Alguire [1] as their masters' theses. The IDG was created for Smalltalk/V32. The IDG was changed somewhat and ported to

⁷in addition to the ones all layouts must implement (see 5.1.8).

⁸In addition to the ones all layouts must implement (see 5.1.8)

⁹In addition to the ones all layouts must implement (see 5.1.8).

Figure 5.8: The MessageTracer3DViewer aggregates.

Method	Description
messagesSize	the number of messages in the message trace.
diagram:	sets the diagram that creates the layout.
inspectedObjectWithTitle:	the object by the title
createComposite3DObjectStart	just shows the objects and not the messages in the layout.
createComposite3DObjectStart-ToCurrentOperation	creates the diagram up to the current point.
step:	adds the next message to the composite3DObject passed in as the argument.

Table 5.10: Methods that message tracer viewer layouts must implement.

Smalltalk/V2.0 by the COOP group (see 3.1.3.4). We ported the IDG to Visual Smalltalk 3.11 and modified the area that creates the actual visualization but none of the other structures that are actually used to gather the message trace information, except for the class names, which were changed to reflect the message tracer and sequence diagram terms used (instead of the old terms like interaction diagram). The message tracer still suffers from the problem that it can not trace user interfaces¹⁰.

5.2.2 Layouts

This section briefly describes how the four layouts in SOVIST-3D are implemented.

5.2.2.1 Default Layout

The default layout is a pickable layout (see 4.3). It is implemented by the **ThreeDDiagram** class. The aggregates of the class can be seen in figure 5.9.

Figure 5.9: The **ThreeDDiagram** aggregates.

5.2.2.2 Sequence Diagram Layout

The sequence diagram layout is a pickable layout (see 4.3). The aggregates of the **SequenceDiagramLayout** can be seen in figure 5.10.

¹⁰It also seems to have problems in the Visual Smalltalk version with the date class for some reason that we can't discover. This means that the date class should always be excluded by using the filters or the message tracer won't work.

Figure 5.10: The **SequenceDiagramLayout** aggregates.

Since this layout makes use of the earlier work, (see section 5.2.1.3) the Adapter pattern [38, Pattern is found on page 139] is used to fit the sequence diagram layout into the layout hierarchy. The `sequence3DDiagram` instance variable points to the class that actually creates the visualization.

5.2.2.3 The Tree Layouts

The tree layouts are non-pickable layouts (see 4.3). Both the 2D B-tree and cone tree layouts are subclasses of the abstract class **Trees**. The aggregates of **Trees** are shown in figure 5.11. All of the work is done in this class. The subclasses just contain a Factory Method [38, Pattern is found on page 107] to create the correct type of **OpenGLTreeNode** (see figure 5.12).

Figure 5.11: The **Trees** aggregates.

Figure 5.12: The classes created using the factory method in the trees hierarchy.

Each of the **OpenGLTreeNodes** is a separate tree structure and knows how to create the appropriate tree structure of itself and its children.

5.3 Summary

This section has looked at the design and implementation of the two main areas of SOVIST-3D, the first being the framework. We discuss the goals of the framework and how we think we meet the goals. Along the way we discussed the architecture of the system and got into some details on how specific aspects of the system work.

Secondly, we look at how the viewers and layouts that are currently part of SOVIST-3D have been designed and implemented.

Chapter 6

Conclusions

This chapter starts with an assessment of the work done in the thesis. This is followed by the limitation of the SOVIST-3D. Finally, the chapter discusses possible future work.

6.1 Assessment of Work Done

In creating SOVIST-3D we accomplished our two major goals:

1. We moved software visualization in Smalltalk into 3D through the creation of the framework.
2. We created a number of visualizations that both utilize the framework and show various ways in which 3D visualizations are improvements over 2D visualizations.

Each of these are dealt with in detail in the sections below.

SOVIST-3D differs from other software visualization systems in the following ways¹ (see 3.3.2 for details):

¹Not all differences are for each system.

- visualizations are in 3D.
- visualizations are of Smalltalk code.
- visualizations are created automatically (no annotation of the code is needed).
- visualizations are created in the same environment as the code allowing the user to interact with the visualization and the environment.
- different types of visualizations are created (and the framework exists in which more can easily be added) showing both the static and dynamic aspects of the code.
- visualizations can show models of the objects to aid in understanding the visualizations.
- there is a navigation system to manipulate the visualization once created.

6.1.1 Framework

The framework of SOVIST-3D allows developers to easily create 3D visualizations in Smalltalk. In SOVIST-3D a visualization is composed of three parts (see 5.1.1 for details):

A CommandList That keeps track of the information to be visualized.

A Viewer That provides the user-interface for the visualization.

One or more Layouts That specifies how the information is to be displayed in the visualization.

In order to create a new visualization the developer simply has to subclass the abstract classes in the framework and write a few well-defined methods (see 5 for details).

The creation of the framework has allowed us to meet the goals set out in the introduction (see 1.3). The framework meets these goals by taking care of the following details of creating the visualizations:

- Setting up the GUI, including setting up the OpenGLPane and the menus.
- Handling all interaction with the user (for example, mouse and keyboard events).
- Providing a navigation system that has the following functionality:
 - The ability to transform either the entire scene, the camera or both.
 - The mouse provides different functionality depending upon the context within the diagram (whether over an object, connector or background).
 - The creation of visualizations in which either the objects can be individually moved about or where the objects are fixed in place depending on which type of layout (pickable or non-pickable), the layout has been subclassed from.
 - Providing a mechanism to display a text description of whatever object is currently under the mouse cursor.
 - Allowing for the selection of a single object when multiple objects are underneath the mouse cursor. That object can then be used to perform further actions.
 - Providing a mechanism for the user to interact with the objects in the visualizations.
- Controlling the basic interaction between the three different parts of a visualization.
- Providing a mechanism for using models to represent the objects in the visualizations.
- Creating the visualizations in the same environment as the code through the use of an OpenGL implementation in Smalltalk (see 5.1.3.1). This allows interaction between the visualization and the code to facilitate greater understanding of the program being visualized.
- Allowing visualizations of both static and dynamic aspects of the code.

In addition, the framework allows:

- Many visualizations to be created without duplicating the work that is already encompassed in the framework.
- A consistent ‘look-and-feel’ among the different visualizations.
- Through the separation of the visualization into three parts, visualizations can display the same information in multiple ways (multiple layouts for the same viewer) and can display different information in the same way (the same layout being used for different viewers).
- Visualizations can be created without the need to annotate the code. The information can be gleaned directly from Smalltalk making use of the meta-facilities available in the language.

The framework has accomplished our main goal of moving software visualizations in Smalltalk to 3D.

6.1.2 The Visualizations within SOVIST-3D

In addition to the framework, SOVIST-3D, consists of a number of visualizations that demonstrate various aspects of the framework. The visualizations also demonstrate some of the ways in which the third dimension can be used.

SOVIST-3D can create a total of seven different types of visualizations, that are comprised of three different types of information being visualized, each of which corresponds to a viewer in the implementation. The viewers are:

Hierarchal Viewer Creates visualizations of the class hierarchy.

Parts Viewer Creates visualizations of the parts of an object returned from the run of a use case.

Message Tracer Viewer Creates visualizations of a trace of the interesting messages sent during the run of a use case.

The viewers make use of layouts to design the display of the visualization. There are currently four different layouts:

Default layout Adds the objects to the visualization allowing the user to layout the objects as desired.

2D B-Tree layout Creates a series of 2D B-Trees of the connected objects in the visualization.

Cone tree layout Creates a series of cone trees of the connected objects in the visualization.

Sequence diagram layout creates a sequence diagram-like layout of the visualization.

The first three layouts are used with both the hierarchy and parts viewer. The last one is used in the message tracer viewer, creating a total of seven different types of visualizations. The viewers and layouts are discussed in more detail in the sections below.

6.1.2.1 Viewers

The types of viewers were chosen to help demonstrate SOVIST-3D framework's ability to deal with different types of visualizations. We have:

- A visualization of some static code (the hierarchy viewer).
- A visualization of the result of running code (the parts viewer).
- A visualization of the dynamic aspects of the code (the message tracer viewer).

In addition, the viewers demonstrate the ability of the framework to:

- Create the GUI and handle the interaction with the users.
- Provide a navigation system.
- Create visualization in the same environment as the code.
- Create visualizations without the need to annotate the code.
- Provide a mechanism to use models to represent the objects in the visualization.
- Create a variety of visualizations that all have the same ‘look-and-feel’.

6.1.2.2 Layouts

The layouts were chosen both to demonstrate various ways in which the third dimension can be used and to demonstrate the framework.

The combination of all of the layouts demonstrates the following ability of the framework to:

- Create multiple layouts for the same type of information (both the parts and hierarchy viewer have three different types of layouts available).
- Create layouts that can be used to display multiple types of information (all three of the default, B-Tree and cone tree layouts are used by both the parts and hierarchy viewers).
- Create both pickable (default and sequence diagram) and non-pickable (B-Tree and cone tree) layouts.
- Make use of the navigation system to better understand the visualizations.

In addition, each layout is discussed in detail below.

Default layout

- An example of maximizing the screen space visualization (see page 25) since the objects can be moved to any position in 3D space.
- A means to let the user create his/her own visualization through the use of the navigation system.

2D B-Tree layout

- An example of an augmented 2D visualization (see page 20).
- It is only useful for a small number of objects and was created mainly to show a contrast between how many objects can be fit on the screen with the cone tree layout.

Cone Tree layout

- An example of maximizing screen space (see page 25). Compared to the 2D B-Tree layout, many more objects can be displayed in one visualization.

Sequence diagram layout

- An example of maximizing screen space (see page 25). The cylinders can be rearranged on the screen to maximize the screen space.
- An example of adapting the 2D view (see page 22). The depth of the method is displayed using depth cylinders.
- In addition, the markers (both to mark the class and the individual objects), make use of SOVIST-3D's ability to represent objects with a model of the object, making it easier to identify the individual objects in the visualization.

6.1.3 Summary of Work Done

The framework of SOVIST-3D demonstrates that meaningful 3D software visualizations can be created in Smalltalk. The framework makes it easy to create various different types of visualizations that have the same ‘look-and-feel’. To demonstrate this seven different types of visualizations (a combination of the three different viewers and four different layouts) were created. These visualizations also show some of the ways in which the 3D visualizations are improvements over their 2D counterparts.

6.2 Limitations of SOVIST-3D

SOVIST-3D provides a framework for exploring 3D software visualizations but at this time is not useful other than for academic purposes due to the following reasons:

- SOVIST-3D is currently far too slow to work with more than a few objects at a time. Most likely it can be sped up using the suggestions noted in section 6.3.2.1.
- SOVIST-3D has very few 3D objects that represent domain objects (see 6.3.1).
- The visualizations are not nearly as useful on paper as they are on the screen where they can be manipulated. Since there is no current way of storing the visualizations, other than by creating a bitmap, there is no way of referring to visualizations of the past. Since the viewers make use of the command lists that were developed by the COOP group (see 3.1.3.4), it should not be too difficult to make the additional changes needed to save the visualizations.

6.3 Future Work

There is significant work that can be done in the future. We have divided the suggestions for future work into the following areas:

- Creating more 3D objects to match classes in the system.
- Improving the current implementation.
- Adding layouts to SOVIST-3D.
- Adding new viewers to SOVIST-3D.

6.3.1 3D Objects

For SOVIST-3D to really be effective it needs to have many, many more 3D objects to represent the classes in the hierarchy. Being able to see at a glance what the different objects in the visualizations are greatly improves the ability to understand the visualizations. There are two problems to the adding of more 3D objects:

- Finding 3D models that convey information on the objects to the user.
- Creating the 3D models decided upon. This is a non-trivial task².

6.3.2 Improving the Current Implementation

This section discusses a number of improvements that could be made to the existing software. Many of these have been mentioned earlier in the thesis and are mentioned here for completeness. The rest of this section further subdivides the area that could be improved.

6.3.2.1 Improving the Speed of the Implementation

The greatest problem with the current implementation is its speed. The following will probably increase the speed:

²The work that Mike Valenta has done on reading in VRML scripts and creating 3D objects based on the script could be invaluable for doing this.

- Speeding up the OpenGL implementation (see 5.1.3.2).
- Currently, if a viewer has multiple layouts, then all of the layouts that are used are all created whenever the visualization is created. It would be faster to just create the current selected layout and whenever the layout is changed, to then create the new layout using the information stored in the command list (see 5.1.7).
- Using faster machines with hardware support for 3D graphics.

6.3.2.2 Improving the User Interface of the Implementation

- Add an undo function to undo the latest change made to either the items added to the visualization or to objects being moved in the visualization.
- Allow individual objects to be also rotated/scaled instead of just translated.
- Give options on the defaults of the viewers that can be saved for the next time one of the viewers start, for example, the background colour, the startup transformations, etc.
- Improve the current navigation and object movement systems or implement a new one, such as a navigation system that uses a device such as a 3D mouse.
- Use virtual relative, stereo view and head coupled perspective [120].
- Have the ability to zoom in and out on a particular region, something similar to how the zoom tool works in many drawing programs.
- Be able to bring an object to the front as one of the popup menu objects. This could be done in a manner similar to that of the cone tree as done at Xerox Palo Alto Research (see 3.2.2) where you get an animated view of the diagram turning until the requested object is in front.

6.3.2.3 Improvements to the Message Tracer Implementation

- Improve the filtering system, so that after the diagram has been executed all of the classes and methods in the current use case can be shown, allowing the user to select which ones he/she wants.
- Do the message tracing in real time. Currently the system runs through the whole use case, storing the results. It would be nice while stepping through the use case to be able to interact with the objects in their current state (i.e. call up their inspector, open a parts viewer on the object, etc.). This way the message tracer facility could be more useful for debugging purposes.
- Add the ability to set breakpoints as in TRACK (see 3.1.3.1) in the message tracer viewer instead of having to step to the required location.
- Implement the items mentioned in the future work sections of Wang [119] and Alguire [1] in their masters' theses. The one most relevant to this thesis deals with improving the information gathering mechanism to handle cases where the user interface is used.

6.3.2.4 Other improvements to the implementation

- Make the connectors in viewers not just simple lines/arrows so that they could also be arcs.
- Be able to match a class with a specific 3D object on the fly.
- Improve the *Scene*→*Fit to Window* function so it works better and correctly (see 5.1.7.2).
- Be able to fold objects in to create 3D Rooms like Robertson *et al.* [95] describe.
- Save the diagrams. This could be accomplished using the command lists (see 5.1.7).
- Add smooth animations. To do this the OpenGL implementations need to be much faster.

- Add 3D text fonts like that of the OpenGL screen saver that comes with the Windows 95 OpenGL DLLs.

6.3.3 Adding Layouts

This section looks at a few of the layouts that could be added to the viewers.

6.3.3.1 Hierarchy Layouts

These are a few of the many layouts that could be constructed:

- Use fractal-based trees (see 3.2.4.2) for displaying in the hierarchy or parts viewers.
- Display classes in the xy plane with the methods lined up and trailing behind them as in *VOGUE* (see 3.2.4.2).
- Use the graphing techniques used in *Narcissus* (see 3.2.4.1).

6.3.3.2 Message Tracer Layouts

- Draw the trace of the methods using local drawing rules such as a bottom-up approach for visualizing program behaviour done with *VOGUE* (see 3.2.4.2).
- Place the methods on a spiral and show the objects radiating out from the spiral like in *Virtual Images* (see 3.2.3.3).
- Use the graphing techniques used in *Narcissus* (see 3.2.4.1) to layout the cylinders so that objects that interact a lot are close to each other.

6.3.4 Adding new viewers

This section describes some new visualizations that could be added as viewers. Most of these are based on some of the diagrams described by UML [11].

Collaboration Visualizations show object interactions organized around the objects and their links to each other. Unlike message tracer visualizations, a collaboration visualization shows the relationship among the objects and not the messages being sent among the objects.

State Visualizations show the change of state machine, the sequence of states that an object goes through during its life in response to events.

Use Case Visualizations show the relationship among actors and use cases within a system.

Code Change Visualizations show the types of code changes made to certain components. These could be developed like the system described in section 3.1.3.3.

Appendix A

Definition of a Use Case

This appendix contains the definition of a use case that is used throughout this thesis.

Use cases are a term originally used by Jacobson in his book Object-Oriented Software Engineering (OOSE) [50]. They have now been incorporated into the new UML ¹ notation [12, 11, 13].

According to Jacobson, a *use case* is a specific way of exercising a system by using some part of the functionality. Each use case constitutes a complete course of events initiated by a user; the use case specifies the interaction that takes place between a user and the system. For example, in a bank account system, opening an account, making a withdrawal transaction, or making a deposit transaction are all use cases. Use cases can be used during the analysis to specify the requirements of a system.

There are two possible ways of looking at a use case. One is to view the system as an object and the use cases as operations that get invoked on the system. The other way is to view use cases as classes and instances. Jacobson chooses the second view which of course maps easily onto the basic notions of object-orientation.

In this approach, each use case is a specific way of using the system and every execution of the use case may be viewed as an instance of the use case. When a user

¹UML stands for the Unified Modelling Language.

inputs a stimulus (or sends a message), the use case instance executes and starts a transaction belonging to the use case. This transaction consists of different actions to be performed. The use case instance exists as long as the use case is operating. These use case instances follow a specific class like all instances in an object-oriented system. When a use case is performed, we view this as instantiating an instance from the use case class. A use case class is a description which specifies the transactions of the use case. The set of all use case descriptions specifies the complete functionality of the system [50, page 128].

In the case of SOVIST-3D use cases are set up as class methods (in the example category) of the classes. The **BankAccount** `example1` and `example2` methods found in code listings in appendix D.1 are examples of use cases.

Appendix B

SOVIST-3D Users' Manual

This appendix contains SOVIST-3D users' manual. It is not meant to describe what SOVIST-3D does or how it does it but how it can be used. This is meant to be a more or less stand alone document, but there are references from here to the rest of the thesis, to avoid having to duplicate information or to point to areas where general concepts are discussed. For general information on SOVIST-3D see chapter 4; for design and implementation see chapter 5.

SOVIST-3D is a prototype. It is not meant to be a commercially viable product. Thus there are some rough spots in using it. Suggestions on improving the implementation can be found in section 6.3.2.

SOVIST-3D is built upon a common framework to create all the visualizations. This framework is discussed first, followed by a discussion of each of the message tracer, hierarchy and parts viewers.

Throughout this section the terms object, marker, connector and background are used frequently when referring to items in the OpenGL pane. The terms are defined as follows:

Object This is a three-dimensional object that represents either a class or an instance of a class within a scene.

Marker These are used within sequence diagrams to help describe the objects since all objects in message tracer are cylinders (see B.2.5).

Connector Something that connects two objects together. Connectors are usually lines or arrows.

Background This is part of the OpenGL pane where no object, marker or connector appears.

B.1 Common Framework

Each of the visualizations is built upon a common framework so that it will have the same look and feel. This will enable the user to be able to use all the viewers in a similar manner. The common framework includes the following items, each of which is discussed in more detail below:

- a 3D object associated with a class.
- the panes contained in each viewer.
- the menus.
- the use of the mouse.
- transforming objects.
- transforming the whole scene.

B.1.1 3D Object Associated with a Class

Each class can have a `ThreeDObject` associated with it to help identify it in the diagrams. This can be set by having the object implement, as a class method, either `threeDObject` or `threeDObject:` method. Section 4.4 contains details.

B.1.2 Panes Contained in the Viewers

Figure B.1: The general design of a three dimensional viewer.

Each of the viewers has the same look as in figure B.1. They are composed of the following parts:

- A title bar which displays the type of viewer.
- A menu (see B.1.3).
- A section that will have special information for that particular viewer. At present only the message tracer viewer (see B.2.1) uses this section.
- The object display pane that shows information about the items underneath the mouse (see B.1.4).
- An OpenGL pane that displays the three dimensional diagrams.

B.1.3 Menus

Figure B.2: The common parts of the menus.

This section describes the menus that are in common among the viewers. The menu parts that are specific to a particular viewer are discussed within the section describing the viewer (see B.2, B.3 and B.4). The menu can be seen in figure B.2.

The menus are composed of the following items each of which is described in the following sections:

- a file menu.
- a viewer menu.
- a transform menu.
- a layout menu.
- a scene menu.
- an options menu.
- an inspect menu.
- a help menu.

B.1.3.1 File Menu

The *file* menu contains the following menu items:

New Message Tracer 3DViewer launches a new message tracer viewer (see B.2).

New Hierarchy 3DViewer launches a new hierarchy viewer (see B.3).

New Parts 3DViewer launches a new parts viewer (see B.4).

Save as Bitmap saves the current contents of the OpenGL pane as a bitmap after prompting for a file name.

Exit This closes the viewer.

B.1.3.2 Viewer Menu

The *viewer* menu contains information that is specific for each viewer. For a description of the items see the section for the appropriate viewer (see B.2.2, B.3.1 and B.4.1). The menu is not only specific for each viewer, but also, possibly, for each layout. All of the layouts have the ability to add items to the end of the viewer menu. The only layout that does so currently is the sequence diagram layout (see B.2.6).

B.1.3.3 Transform Menu

The *transform* menu is disabled until the OpenGL pane contains a visualization. It contains the following menu items:

Transform Scene brings up the *Transform Screen* modeless dialog box¹ which allows the user to translate, rotate or scale the entire scene (see B.1.7).

Translate 3DObject allows the user to translate a single 3D object to a new location. This menu option is disabled except when a single object has been selected. Details can be found below on how to select an object (see B.1.4) and how to translate objects (see B.1.5).

Translation Controls brings up the translation Control modeless dialog box with the translation controls (see B.1.6).

¹a modeless dialog box is a dialog box that doesn't have to be closed to keep working on the main application (in this case the viewer). In other words, the dialog box can stay open all the time.

B.1.3.4 Layout Menu

The *layout* menu is disabled until the OpenGL pane contains a visualization. The menu contains all the layout types that the viewer can display. The current layout is checked. Any changes, with respect to adding/removing/connecting etc. objects changed in one layout will also change in all the other layouts. Information on the types of layouts for each viewer can be found in the respective sections below (see B.2.5, B.3.3 and B.4.3). Information on layouts in general can be found in section 4.3.

B.1.3.5 Scene Menu

The *scene* menu is commonly disabled until the OpenGL pane contains a visualization. The menu contains the following items:

Fit To Window resizes the diagram, by changing the camera location and viewing angle until the entire diagram can be seen on the screen. This function doesn't work perfectly. It occasionally leaves some objects just outside the viewing volume. See the discussion in section 5.1.7.2 for details.

Random Objects places the objects randomly in the viewer. The objects may be placed outside of the current viewing area. This is often useful to get a different perspective on the scene.

B.1.3.6 Options' Menu

The *options'* menu contains the following menu items:

Wire-frame Only shows the objects on the screen as wire-frame objects only. This increases the speed in which the visualization can be rendered. It is not initially checked.

Use Textures allows any textures in the scene to be rendered. This decreases the speed at which the scene in the viewer can be rendered. It is initially checked.

Lights On turns on lighting. The lighting doesn't work in Windows 95, so this option will have no effect in Windows 95. If lighting is turned on in Windows NT, there is a point light that is attached to the camera. This will show shading. Having the lights on decreases the speed in which the scene in the viewer can be rendered. It is not initially checked.

Background Color brings up the standard colour choosing dialog box. A new colour for the background can then be selected. The default background colour is black.

B.1.3.7 Inspect Menu

The *inspect* menu contains the following menu item:

Inspect opens up the inspector on the viewer.

B.1.3.8 Help Menu

The *help* menu contains the following menu item:

Keys brings up a message box that contains the information on the keys used to move the camera around. The scene, as opposed to the camera, can also be moved (see B.1.7). The information in the message box is repeated in table B.1.

B.1.4 Using the Mouse

The mouse is used in the OpenGL pane. Table B.2 contains a summary of how the mouse works. Details can be found below.

It is useful to note here that selecting a marker selects the object that it is referring to.

Key	Description
Translations	
Up Arrow key	move forward
Down Arrow key	move backward
Left Arrow key	move left
Right Arrow key	move right
Page Up key	move up
Page Down key	move down
Rotations	
Ctrl + Up Arrow key	look up
Ctrl + Down Arrow key	look down
Ctrl + Left Arrow key	look left
Ctrl + Right Arrow key	look right
Ctrl + Page Up key	rotate clockwise
Ctrl + Page Down key	rotate counterclockwise

Table B.1: Keys used to move the camera.

B.1.4.1 The Mouse with no Buttons Pressed

If the mouse stays stationary over an object, marker or connector in the OpenGL pane for a little while (approximately a second) then the description of the item is displayed in the object display pane. If more than one object is under the cursor, then each of the descriptions is shown in the object display pane, separated by a space. The message tracer displays are a little more complicated (see B.2.1.1 for details).

This works like the bubble help that is common in Windows applications, except that, instead of popping up a bubble, the information is displayed in the display pane. For details on how it works see section 5.1.7.1.

B.1.4.2 The Mouse with the Left Button Pressed

Clicking once with the left mouse button allows an object to be selected. Once an object is selected, it can be translated using the mouse (see B.1.5) or via the menu item *Transform→Translate 3D Object*. (see B.1.3.3). The selected objects are

Button	Meta Key	Description	Section for more information
None	None	When over an object, marker or connector it displays a description of the object in the object display pane.	B.1.4.1
Left	None	If the mouse cursor is over an object, the object is selected.	B.1.4.2
Left	Shift	If an object is already selected, it adds the object under the mouse to the selected objects. This allows multiple objects to be selected.	B.1.4.2
Left	CTRL	If there are multiple objects under the mouse, it brings up a multiple list box that shows the description associated with each object. One or more objects can then be selected in the standard way using the mouse and shift key.	B.1.4.2
Left	Moving the mouse	Translates the selected objects.	B.1.4.2, B.1.5
Right	None	Brings up one of three popup menus depending on whether the mouse is over an object (or marker), a connector or the background.	B.1.4.3

Table B.2: The mouse button combinations and their usage when in the OpenGL pane.

highlighted² to denote the fact that they are selected. You can not select objects when the selected layout is non-pickable (see 4.3).

The following lists the possibilities for selecting an object:

- If the mouse is over an object in the OpenGL pane that can be selected, the object is selected.
- If the object is already selected, it deselects it.
- If another object is selected, it deselects the other object and selects the object under the mouse.
- If there is more than one object under the mouse, the one with the smallest z-value is selected.

Multiple objects can be selected by using the shift key. Holding the shift key down when selecting additional objects adds the objects to the list of selected objects. Multiple selected objects can be translated using the mouse, but not the transform 3D object menu item.

If there are multiple objects under the mouse and the CTRL key is held down when the left mouse button is clicked, then this opens a multiple list box with a description of all the objects under the mouse. One or more of these objects can then be selected.

B.1.4.3 The Mouse with the Right Button Pressed

When the right mouse button is clicked, it opens a popup menu depending on what is under the cursor at the time it was clicked. If multiple objects are under the cursor, then a list box with the description of the objects is opened. One object can then be selected and it is to this object that any action on the popup menu is directed.

There are three types of popup menus each of which are different depending upon the type of viewer.

²Currently a highlighted object changes to a gray or bluish colour.

1. Object or marker popup menus.
2. Connector popup menus.
3. Background popup menus.

For more information see the section on the viewer in question (see B.2.3, B.3.2 and B.4.2):

B.1.5 Transforming Objects

Currently the only transformations that can take place are translations. They can only take place if the layout is a pickable layout 4.3.1.

Whenever an object is translated, its associated connector and marker are moved with it. Individual objects must first be selected before they can be transformed. Objects are selected using the mouse (see B.1.4.2). Multiple objects can be selected.

Once selected, objects can be translated using the mouse or by using the menu item *Transform→Translate 3D Object*.

The menu item and associated dialog box allow only one item to be translated at a time but allow placement of the object to an exact location. With translation using the mouse, more than one object can be translated at a time but it is difficult to move objects to an exact location.

B.1.5.1 Translating an Object with the Dialog Box

Once the menu item *Transform→Translate 3D Object* is selected ³ the *Translate Object* dialog box appears, as shown in figure B.3. The dialog box shows the current position of the object. The coordinates can then be changed to the new location. Once the **OK** button is pressed, the object is moved to the new co-ordinates and deselected.

³the menu item is disabled if no objects or more than one object is selected.

Figure B.3: The translations object dialog box.

Notes:

- The new co-ordinates can make the object move so it cannot be seen in the viewing volume.
- The co-ordinates shown are those relative to the scene. In other words, the transformations of the scene (see B.1.7) are done after this translation and so the values don't reflect the scene transformations.

B.1.5.2 Translating Objects Using the Mouse

Translating objects using the mouse is done by first selecting objects (see B.1.4.2). Whenever some objects are selected and the mouse is dragged with the left mouse button down, the objects will be moved according to the information in the transformation controls' dialog box (see B.1.6).

Notes:

- If the mouse left button is pressed while on an object (or marker) and then dragged before being let up, that object is the one that is selected.
- If there are many items on the screen, this can be excruciatingly slow. To help speed up the process, drag the objects in wire frame mode only.

B.1.6 Translation Controls

Figure B.4: The translation controls dialog box.

This dialog box, seen in figure B.4, controls how the transformations done by the mouse and keyboard work. It will appear as the result of picking the *transform→translations controls* menu item (see B.1.3.3). The dialog box is a modeless dialog box.

Whenever the **OK** button is pressed, the changes will take place in the viewer.

The mouse section controls how the mouse drags objects (see B.1.5.2). The mouse part of the dialog box is further divided into two sections. One controls each of the horizontal and vertical motions of the mouse. The mouse only has two degrees of freedom and three are needed to move objects in 3D. The radio buttons allow the choice of what axis that the mouse movement will effect. For example, if the idea is to move the objects in the xy plane, the horizontal/vertical settings can be set to x and y respectively. If you want to move in the xz plane, the horizontal/vertical settings can be set to x and z respectively. This means that any movement of the mouse vertically is changed into moving the objects along the z-plane instead.

The scaling factor changes show how many OpenGL world co-ordinate units to move for every pixel the mouse moves. Each of the horizontal and vertical movements can be set separately.

The keyboard section of the dialog box controls the size of the translations and rotations are done when using the keyboard to move the camera (see B.1.7.1) (in other words, the magnitude of each mouse pixel movement). The translation box shows how many OpenGL world co-ordinate units the camera will move every time one of the translation movements for the camera is done. The rotation box shows how many degrees the camera is rotated when a camera rotation movement is done.

The defaults for the dialog box are:

- horizontal movement is mapped to the x-axis.
- vertical movement is mapped to the y-axis.
- horizontal and vertical scaling factors are 0.05 OpenGL world co-ordinates.
- keyboard translations are set to 1.0 OpenGL world co-ordinates.
- keyboard rotations are set to 10 degrees.

NOTE TO USERS

Page(s) not included in the original manuscript are unavailable from the author or university. The manuscript was microfilmed as received.

129

UMI

Figure B.5: The transform scene dialog box.

Notes:

- The order in which the transformations are done is: translations, rotations, then scaling.
- A rotation is done for each axis.
- Scaling the image is computationally expensive for a large visualization. It is usually better to just move the scene farther from the camera making the z value in the translation a larger negative number.
- The camera is initially at the origin facing down the negative z axis.

B.2 Message Tracer Viewer

Message tracers show a particular series of interactions among objects in a single execution of a use case (see A). Section 4.2.4 contains general information about the

message tracer viewer.

The basic idea behind the message tracer viewer is that it selects a class, then a class method for that class (this is the use case that will be run). It creates a filter to describe which classes/methods that are or aren't interesting (see B.2.4) and then execute the use case to create the visualization.

There are three ways in which the message tracer can be executed (completely through, by stepping, or by animating), each of which can be selected from the viewer menu (see B.2.2).

Currently the message tracer viewer can not be used with any use case that involves the user interface. Also, if the date class is used, it is highly suggested that it be filtered out using the message filters B.2.4, since it often hangs the Smalltalk Environment.

B.2.1 Message Tracer Viewer – Viewer Panes

The additional panes that are in the message tracer viewer (see figure B.6) that are not in all the viewers (see B.1.2) are:

Class ListBox is a listbox used to select the class that the sequence diagram will be generated from. The classes are listed in the same order as in the Hierarchy Browser. The class can also be selected using the *Viewer→Find Class* menu item (see B.2.2).

Method ListBox is a listbox used to select the class method for the selected class. This is the use case that will be run to create the message tracer. If no class has been selected, there are no items in the listbox.

Stepping Pane When the message tracer is being stepped through or animated (see B.2.2), the message that has just been added to the message tracer visualization is displayed.

Figure B.6: The message tracer viewer design.

Stepping Button When stepping through a message tracer visualization, the button is pressed to go onto the next step. The button is disabled when not in stepping mode.

B.2.1.1 Object Display Pane

When the mouse cursor is over objects, markers or connectors, but with no buttons pressed (see B.1.4.1), a description of the object is shown in the object display pane.

When the mouse is over an object or marker, it shows the **printOn:** method of the object. If the object is an instance, then it also shows after the **printOn:** method. It is followed by the # sign and then the number of the occurrence in the scene (for example, if is the second instance of a **BankAccount** class it would read #2 at the end.) When the mouse is over a connector, it shows the method name and to which object the method is being sent.

B.2.2 Message Tracer Viewer – Viewer Menu

The viewer menu is composed of the following items:

Execute shows the message tracer visualization after a complete run through of the use case selected, after applying the filters. It is disabled if no class and method have been selected (see B.2.1). If the diagram is already on the screen, it re-executes it.

Animate Execution shows the message tracer visualization, showing the messages added to the visualization one right after another. This is stepping with no pauses between each step. The item is disabled if neither class nor method has been selected.

Execute By Step runs through the execution of the messages being sent pausing after each message is added to the visualization until the step button (see B.2.1)

has been pressed. The method that has just been added to the visualization is displayed in the step pane. The item is disabled if neither a class nor a method has been selected.

Find Class brings up a dialog box that allows a class to be found by name. Wildcards are allowed in the search. The class selected is then shown in the class listbox (see B.2.1).

Reload Classes reloads in the list of classes from the system, since the last time the list was updated was when the menu item was last called or when the viewer opened.

Modify Filters brings up the filters' definition box (see B.2.4). The item is disabled if neither a class nor a method has been selected.

Items can be added to the bottom depending on the layout. Currently the sequence diagram layout adds items (see B.2.6).

B.2.3 Message Tracer Viewer Popup Menus

This section describes the popup menus activated using the right mouse button on objects, connectors and the background (see B.1.4.3).

B.2.3.1 Message Tracer Viewer Background Popup Menu

There is no current background popup menu.

B.2.3.2 Message Tracer Viewer Object Popup Menu

The object popup menu is composed of the following items:

Add Class to filter adds the class and all its messages to the bottom of the filter as items to be excluded (see B.2.4).

Open Browser on opens the *Class Hierarchy Browser* on the objects class.

Open Hierarchy3DViewer on opens the *Hierarchy Viewer* on the objects class (see B.3).

B.2.3.3 Message Tracer Viewer Connector Popup Menu

The connector popup menu is composed of the following items:

Add Method to Filter for this Class adds the method for the class it is being sent to, to the bottom of the filter as items to be excluded (see B.2.4).

Add Method to Filter for all Classes adds the method for all classes, to the bottom of the filter as items to be excluded (see B.2.4).

Open Browser on opens the *Method Browser* on the method for the class and all of its super classes.

B.2.4 Message Tracer Filters

Filters can be added by using the modify filter menu item (*Viewers*→*Modify Filters*) or by using the popup menus to add items to the bottom of the filter list (see B.2.3). Figure B.7 shows the filter dialog box.

There must be a filter list before the diagram can be executed. If one of the execute commands (see B.2.2) is run before a filter list has been created, it automatically opens up the filter definition box. The filter list can be empty.

The filters take the form of [type][class]>>[method].

Type There are two types of filters:

1. filters that exclude classes and methods, the type is a: -.
2. filters that include classes and methods, the type is a: +.

Figure B.7: Shows the filter dialog box.

Class The class is the name of a class. A class effects not only the class itself but also all of its subclasses. A * means all classes.

Method The name of a method. A * means all methods.

The filters work from the bottom of the list up. So if a filter at the top excludes something, a filter later in the list can re-include it. Filters must, as a minimum, include the class>>method name that is being used as the use case and the MethodTracer class.

The easiest way to understand the filters is to give a few examples. See tables B.3 and B.4.

B.2.5 Message Tracer Layouts

There is currently only one layout, a sequence diagram layout.

Filter	Description
-Object>>*	Exclude all the methods of the class object and all of its subclasses
+MethodTracer>>*	MethodTracer class (needed to work).
+BankAccount>>*	Include all the methods of the BankAccount class.

Table B.3: Filter example 1. The net result is that any methods sent to the BankAccount class will be shown in the diagram.

Filter	Description
-*>>new	Excludes the new method on all classes
+BankAccount>>new	Includes the new method for the BankAccount class

Table B.4: Filter example 2. The net result is all of the new methods except that of the BankAccount class will be excluded.

B.2.6 Sequence Diagram Layout

These are the items added to the bottom of the viewer menu for the sequence diagram layout.

Show Markers shows the markers if checked. Every time an execution is done it is reset to being checked.

Show Length of Method shows the length of methods if checked. It is initially checked. Every time an execution is done it is reset to being checked.

Show Markers shows the depth of methods if checked. It is initially checked. Every time an execution is done it is reset to being checked.

Section 4.3.1.2 contains details.

B.3 Hierarchy Viewer

The hierarchy viewer allows the display of visualizations which show the inheritance between objects in the systems. Section 4.2.2 contains details about the hierarchy viewer.

The basic idea is that classes can be added to the visualization, then their super and subclasses can be added using the popup menu on any object in the visualization to create the hierarchy.

B.3.1 Hierarchy Viewer Menu

The viewer menu is composed of the following items:

Add Class performs the same function as the popup background menu *add class* (see B.3.2).

Items can be added to the bottom of the viewer menu depending on the layout. Currently none of the hierarchy layouts adds any items.

B.3.2 Hierarchy Viewer Popup Menus

This section describes the popup menus activated using the right mouse button on objects, connectors and the background (see B.1.4.3).

B.3.2.1 Hierarchy Viewer Background Popup Menu

The background popup menu is composed of the following item:

Add Class creates the *add class* dialog box (see figure B.8). The dialog box shows only the classes not currently in the diagram. One or more classes can then be added to the diagram in the viewer as desired.

Figure B.8: The add a class dialog box.

B.3.2.2 Hierarchy Viewer Object Popup Menu

The object popup menu is composed of the following items:

SuperClass is composed of the following submenu:

Add adds, if there is one, the superclass of the current object. A connector is created between the superclass and the current object. The connector is labeled superclass.

Add All adds, if there are any, all of the superclasses of the current object. Connectors are added between each of the objects added and each of their superclasses. The connectors are labeled superclass.

SubClass is composed of the following submenu:

Add brings up a dialog box similar to the *add class* dialog box (see figure B.8). That shows all of the subclasses that haven't yet been added to the visualization and allows zero or more of them to be chosen. A connector is created between each of the subclasses and the current object. The connectors are labeled subclass.

Add All Immediate adds, if there are any, all of the immediate subclasses of the current object. A connector is created between each of the subclasses and the current object. The connectors are labeled subclass.

Add All adds, if there are any, all of the subclasses of the current object recursively. A connector is created between each of the subclasses and the current object. The connectors are labeled subclass.

Remove is composed of the following submenu:

Class removes the current object and any connectors associated with it from the visualization.

SuperClass removes from the visualization the superclass of the current object, if it is in the visualization, the and any connectors associated with it.

All SuperClasses removes all of the superclasses of the current object and any connectors associated with them from the visualization.

Immediate SubClasses removes all of the immediate subclasses of the current object and any connectors associated with them from the current visualization.

All SubClasses removes recursively all of the subclasses of the current object and any connectors associated with them from the visualization.

Open Browser on Opens the *Class Hierarchy Browser* on the objects' class.

B.3.3 Hierarchy Viewer Layouts

There are three layouts currently associated with the Hierarchy Viewer, the default layout, cone tree layout and the B-Tree layout.

B.3.3.1 Hierarchy Viewer Default Layout

The default layout adds each object at the origin in the scene co-ordinates (in other words, before the scene transformation). The objects can then be moved using the mouse and menu items (see B.1.5). Section 4.3.1.1 contains details on the layout.

B.3.3.2 Hierarchy Viewer Cone Tree Layouts

The cone tree layout creates a cone tree out of each set of independent objects⁴. The cone tree layout is a non-pickable layout. The whole scene can still be moved (see B.1.7).

Section 4.3.2.1 contains details on the layout.

⁴Independent objects are objects that are currently not connected via a connector to any other objects.

B.3.3.3 Hierarchy Viewer 2D B-Tree Layouts

The 2D layout creates a 2D B-tree out of each set of independent objects. The layout is a non-pickable layout. The whole scene can still be moved (see B.1.7).

Section 4.3.2.2 contains details on the layout.

B.4 Parts Viewer

The parts viewer shows the parts of a particular instance of an object returned from a run of a use case (see A). Section 4.2.3 contains details about the parts viewer.

The basic idea is that objects are added to the system by running use cases (a class method of a particular class in this case). The value returned is then added to the visualization. The parts of the object can then be added to the visualization.

B.4.1 Parts Viewer – Viewer Menu

The viewer menu is composed of the following items:

Add Object performs the same function as the popup background menu *add object* (see B.4.2.2).

Items can be added to the bottom depending on the layout. Currently none of the parts layouts adds any items.

B.4.2 Parts Viewer Popup Menus

This section describes the popup menus activated using the right mouse button on objects, connectors and the background (see B.1.4.3).

Figure B.9: The add an object dialog box.

B.4.2.1 Parts Viewer Background Popup Menu

The background popup menu is composed of the following item:

Add Object creates the *add object* dialog box (see figure B.9). The dialog box shows the classes in the system. One class can then be selected. All of the class methods of the selected class are then shown. One of the class methods can then be selected. Once the **OK** button is selected the class method is run. The object returned from the class method appears in the diagram.

B.4.2.2 Parts Viewer Object Popup Menu

The object popup menu is composed of the following items:

Inspect opens the inspector on the object.

Show Parts opens up the *show parts* dialog box (see figure B.10). Zero or more parts can then be selected to appear in the diagram. A connector is added between the object and each of the parts added. The connectors are labeled the name of the instance variable associated with the part.

Show All Parts adds all of the parts of the object to the diagram. A connector is added between each object and each of the parts added. The connectors are labeled the name of the instance variable associated with the part.

Hide All Parts removes all of the parts (but not the parts parts) and the associated connectors.

Remove removes the object and any connectors associated with it from the current diagram.

Open Browser on opens the *Class Hierarchy Browser* on the objects' class.

Open Hierarchy3DViewer on opens the *Hierarchy Viewer* (see B.3) and then adds the current object's class to the diagram.

Figure B.10: The show parts dialog box.

B.4.3 Parts Viewer Layouts

The layouts for the parts viewer are identical to that of the hierarchy viewer (see B.3.3) except that the connectors are labeled the part or instance variable name.

Appendix C

Colour Plates

Figure C.1: An example of maximizing screen space [123, Figure from the WWW].

Figure C.2: An example of capturing a history of a 2D view. A scatter plot view of quick sort. The dots to the right indicate the array values and the planes to the left detail a history of the exchanges performed [124, Figure from the WWW].

Figure C.3: A message tracer visualization with the sequence diagram layout, shows use case **BankAccount example2**. This is what the layout looks like by default.

Figure C.4: A message tracer visualization with the sequence diagram layout, shows use case **BankAccount example2**. This shows the same visualization as figure C.3 except that it is being stepped through.

Figure C.5: A message tracer visualization with the sequence diagram layout, shows use case **BankAccount example1** with the filters set to show only **BankAccount** and **Transactions** classes without the **new** messages. The cylinders have been moved around by use of the *Scene*→*Random Objects* menu item and rotated. Only the OpenGL pane is shown.

Figure C.6: A message tracer visualization with the sequence diagram layout with all of the options turned off, shows use case **DepthCylinderExampleSubSub example1**. Only the OpenGL pane is shown.

Figure C.7: A message tracer visualization with the sequence diagram layout with markers, shows use case **DepthCylinderExampleSubSub example1**. Only the OpenGL pane is shown.

Figure C.8: A message tracer visualization with the sequence diagram layout with markers and length cylinders, shows use case **DepthCylinderExampleSubSub example1**. Only the OpenGL pane is shown.

Figure C.9: A message tracer visualization with the sequence diagram layout with markers, length and depth cylinders, shows use case **DepthCylinderExampleSubSub example1**. Only the OpenGL pane is shown.

Figure C.10: A parts viewer using the default layout with the objects having been moved around, shows the value returned from use case **BankAccount example1**. The green spheres represent collections. The top one is an **OrderedCollection**, the second one an **Array** which contains the contents of the ordered collection.

Figure C.11: A parts viewer using the 2D B-tree layout, shows the value returned from the use case **BankAccount example1**. The green spheres represent **Collections**. The top one is an **OrderedCollection**, the second one an array which contains the contents of the **OrderedCollection**. The red spheres are **Transactions** (deposit and withdrawal). The white box to the left of the red spheres is a cheque for the **CheckingTransactions**.

Figure C.12: A hierarchy viewer showing the classes: **Object**, **Transaction** (and subclasses) and **BankAccount** using the default layout.

Figure C.13: A hierarchy viewer showing all the subclasses of the **Object**. The green spheres are **Collection** and its subclasses. The blue disks are **ThreeDObject** and its subclasses. The grey wire-frame boxes are **Window** and its subclasses. The rest of the classes are red spheres.

Appendix D

TestCases' Source Code

This appendix contains the code of the classes used in the diagrams that appear in this thesis. The first section contains code for the **BankAccount** classes, the second for the **DepthCylinderExample** classes.

D.1 Bank Account Code

This section contains the code for the **BankAccount** classes. It is composed of **BankAccount** class, a **Transaction** abstract class; a **CheckingTransaction**, a **DepositTransaction** and a **WithdrawalTransaction** class. They create a simple bank account system.

The **BankAccount** class definition is in listing D.1, the class methods are located in listing D.2 and the instance methods are located in listing D.3.

The **Transaction** class and subclasses definitions are in listing D.4, the class methods are located in listing D.5, the **Transaction** class instance methods are in listing D.6, the **CheckingTransaction** instance methods are in listing D.7, the **DepositTransaction** instance methods are in listing D.8 and the **WithdrawalTransaction** instance methods are in listing D.9.

D.2 DepthCylinderExample Code

This section contains the code for the `DepthCylinderExample` class. There are four classes `DepthCylinderExample`, `DepthCylinderExampleSub`, `DepthCylinderExampleSubSub` and `DepthCylinderExampleSubSubSub`. They were created to demonstrate the use of depth of method markers (see 4.3.1.2).

The class definitions are in listing D.10, the class methods in D.11 and the instance methods in D.12.

Listing D.1 Bank Account Definition

```
class:          BankAccount
superclass:     Object
instanceVariableNames: 'owner balance transactions'
poolDictionaries: "
```

Listing D.2 Bank Account Class Methods

*examples***example1**`"BankAccount example1"``|bankAccount1|``bankAccount1 := BankAccount new owner: 'Jim Smith'.``bankAccount1 post: (DepositTransaction new amount: 100).``bankAccount1 post: (WithdrawalTransaction new amount: 20).``bankAccount1 post: (CheckingTransaction new amount: 30).``^bankAccount1`**example2**`"BankAccount example2"``|bankAccount1 bankAccount2|``bankAccount1 := BankAccount new owner: 'Jim Smith'.``bankAccount2 := BankAccount new owner: 'Sally Smith'.``bankAccount1 post: (DepositTransaction new amount: 100).``bankAccount1 post: (WithdrawalTransaction new amount: 50).``bankAccount2 post: (DepositTransaction new amount: 50).``bankAccount2 post: (CheckingTransaction new amount: 20; checkNumber: 25).``^Array with: bankAccount1 with: bankAccount2`*three D object***threeDObject**`"Return the threeD object to use"``^(Cube width: 1 height: 1 depth: 1``coating: (TextureCoating fromFile: 'money.bmp');``yourself.`*instance creation***new**`super new initialize.`

Listing D.3 Bank Account Instance Methods

*accessing***balance** $\sim$ balance**balance:** amount

balance := amount

owner: aString

owner := aString

owner $\sim$ owner*money operations***post:** aTransaction

self record: aTransaction.

self process: aTransaction

canCover: amount $\sim$ self balance >= amount**record:** aTransaction

transactions add: aTransaction

credit: amount

self balance: self balance + amount

process: aTransaction

aTransaction updateAccount: self

debit: amount

(self canCover: amount) ifFalse: [self error: 'insufficient funds'].

self balance: self balance - amount

*initializing***initialize**

self owner: 'Unknown';

balance: 0.

transactions := OrderedCollection new

*printing***printOn:** aStream

aStream nextPutAll: self owner;

nextPutAll: ''s \$'.

self balance printOn: aStream.

aStream space.

self class printOn: aStream.

Listing D.4 Transaction Classes' Definitions

```

class:          Transaction
superclass:     Object
instanceVariableNames: 'date time amount'
poolDictionaries: "

class:          CheckingTransaction
superclass:     Transaction
instanceVariableNames: 'checkNumber'
poolDictionaries: "

class:          DepositTransaction
superclass:     Transaction
instanceVariableNames: "
poolDictionaries: "

class:          WithdrawalTransaction
superclass:     Transaction
instanceVariableNames: "
poolDictionaries: "

```

Listing D.5 Transaction Classes Methods

Transaction*instance creation*

```

new
    super new initialize.

```

CheckingTransaction*three D object*

```

threeDObject
    "Return the threeD object to use"

    ^(Cube width: 1 height: 0.5 depth: 0.5
        coating: (TextureCoating fromFile: 'cheque.bmp');
        yourself.

```

Listing D.6 Transaction Instance Methods

```
accessing

amount
    ^amount

amount: aNumber
    amount := aNumber

date
    ^date

date: aDate
    date := aDate

time
    ^time

time: aTime
    time := aTime

initializing

initialize
    self date: Date today;
    time: Time now;
    amount: 0.

printing

printOn: aStream
    aStream nextPutAll: 'a $'.
    self amount printOn: aStream.
    aStream space.
    self class printOn: aStream.
```

Listing D.7 CheckingTransaction Instance Methods

*accessing***checkNumber***^checkNumber***checkNumber: anInteger***checkNumber := anInteger**initializing***initialize***super initialize. self checkNumber: 0.**printing***printOn: aStream***super printOn: aStream.**aStream nextPutAll: ' for check number '.**self checkNumber printOn: aStream.**aStream space.**updating***updateAccount: anAccount***anAccount debit: self amount*

Listing D.8 DepositTransaction Instance Methods

*updating***updateAccount: anAccount***anAccount credit: self amount*

Listing D.9 WithdrawalTransaction Instance Methods

*updating***updateAccount: anAccount***anAccount debit: self amount*

Listing D.10 DepthCylinderExample Classes' Definitions

```
class:          DepthCylinderExample
superclass:     Object
instanceVariableNames: 'a b c'
poolDictionaries: ""

class:          DepthCylinderExampleSub
superclass:     DepthCylinderExample
instanceVariableNames: ""
poolDictionaries: ""

class:          DepthCylinderExampleSubSub
superclass:     DepthCylinderExampleSub
instanceVariableNames: ""
poolDictionaries: ""

class:          DepthCylinderExampleSubSubSub
superclass:     DepthCylinderExample
instanceVariableNames: ""
poolDictionaries: ""
```

Listing D.11 DepthCylinderExample Classes' Class Methods

DepthCylinderExample*examples***example1**

"DepthCylinderExample example1"

self new

a: 10;

b: 20;

c: 30;

yourself

DepthCylinderExampleSub*examples***example1**

super example 1;

DepthCylinderExampleSubSub*examples***example1**

self new

a: 10;

b: 20;

c: 30;

yourself

DepthCylinderExampleSubSubSub*examples***example1**

super example 1;

Listing D.12 DepthCylinderExample Classes' Instance Methods

DepthCylinderExample*accessing***a** $\sim$ a**a: anInteger**

a := anInteger

b $\sim$ b**b: anInteger**

b := anInteger

self a: self b / 2. self c: self b * 2.

c $\sim$ c**c: anInteger**

c := anInteger

DepthCylinderExampleSub*accessing***a: anInteger**

a := anInteger

DepthCylinderExampleSubSub*accessing***c: anInteger**c := anInteger

Glossary

Algorithm Animation	Also referred to as algorithm visualization, is understood to be the visualization of a high-level description of a piece of software [87, page212].
Background	The part of the OpenGL pane where no connector, marker or object appears.
Broadness Factor	This is defined as the ratio of the depth of a hierarchy tree and the total number of objects in the hierarchy tree. This provides a measure similar to the average number of children for the system [25].
Code or Data Visualization	This is where the actual implemented code is visualized [87, page 213].
Connector	Something that connects two objects together in the OpenGL pane. Connectors are usually lines or arrows.
COOP	Centre for Object Oriented Programming at Carleton University. Section 3.1.3.4 contains details.
Interaction Diagram	The term used by Jacobson <i>et al.</i> [50] to refer to what UML now refers to as sequence diagrams.
JOOP	Journal of Object Oriented Programming published by SIGS Publications.
Layout	A layout describing how a diagram will appear.
Marker	These are used within the sequence diagram layout to help describe the objects in an OpenGL pane since all objects in sequence diagrams are cylinders (see B.2.5).

Modeless Dialog Box	A dialog box that doesn't have to be closed to keep working on the main application (in this case the viewer). In other words, the dialog box can stay open all the time.
Object	A 3D object that represents either a class or an instance of a class in an OpenGL pane.
Object-Oriented Software Engineering	<i>Object-Oriented Software Engineering</i> is an object-oriented design methodology which is based on use cases, developed by Jacobson <i>et al.</i> [50].
OpenGL	Open Graphics library is a powerful software interface for graphics hardware that allows graphics programmers to produce high-quality colour images of 3D objects. The functions in the OpenGL library enable programmers to build geometric models, view models interactively in 3D space, control colour and lighting, manipulate pixels, and perform such tasks as alpha blending, antialiasing, creating atmospheric effects, and texture mapping (see 5.1.3.1).
Program Visualization	The use of various techniques to enhance the human understanding of computer programs [87, page212].
Ripple Effect	The ripple effect refers to the phenomenon that changes made to one part of a software system ripple throughout the system.
Sequence Diagram	A diagram that shows object interactions arranged in time sequence. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged. A sequence diagram includes time sequences but does not include object relationships. A sequence diagram can exist in a generic form (describes all possible scenarios) and in an instance form (describes one actual scenario) [10].
Software Visualization	The use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software [87, page 213].
3D Viewer	A <i>3D viewer</i> allows the display and manipulation of a particular type of visualization. The visualization is displayed in three dimensions.

Unified Modeling Language	The <i>Unified Modeling Language</i> is a purposed open standard. It is a language for specifying, visualizing, and documenting the artifacts of an object-oriented system under development. It is a unification of the Booch [9], OMT [102] and OOSE [50] methods. It was (and still is) being developed by Booch, Jacobson and Rumbaugh [12, 11, 13].
Use Case	A sequence of actions a system performs that yields an observable result of value to a particular actor. Usually scenarios illustrate prototypical use case instances [10].
Visual Smalltalk 3.1.1	<i>Visual Smalltalk</i> is a smalltalk development environment sold by ParcPlace-Digitalk, Inc.
Visual Programming	The use of 'visual' techniques to <i>specify</i> a program [87, page 212].
Window Builder Pro	Is a package created by Object Share Systems Inc. for use in Visual Smalltalk 3.1 that allows the user to easily design user interfaces.

Bibliography

- [1] Jeffery C. Alguire. On-line dynamic views of applications written in Smalltalk. Master's thesis, Carleton University, 1995.
- [2] Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for more Readable Programs*. Addison-Wesley, Reading, Massachusetts, 1990.
- [3] Ronald M. Baecker and David Sherman. Sorting out sorting. Shown at SIGGRAPH '81, 1981. 16mm colour sound film.
- [4] Marla J. Baker and Stephen G. Eick. Space-filling software visualization. *Journal of Visual Languages and Computing*, 6(1):119–133, 1995.
- [5] Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.
- [6] Roland Balter, Sacha Krakowiak, and M. Riveill. The guide language: Design and experience. *Computer Journal*, 37:519–530, December 1994.
- [7] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.
- [8] Heinz-Dieter Böcker and Jürgen Herczeg. Browsing through program execution. In *Human Factors in Computing Systems INTERCHI'90*, pages 991–996, 1990.
- [9] Grady Booch. *Object-Oriented Design with Application*. Benjamin/Cummings, RedWood City, California, 1991.
- [10] Grady Booch, Ivar Jacobson, and James Rumbaugh. UML glossary. Published by Rational Software Corporation, January 1997.
- [11] Grady Booch, Ivar Jacobson, and James Rumbaugh. UML notation guide. Published by Rational Software Corporation, January 1997.

- [12] Grady Booch, Ivar Jacobson, and James Rumbaugh. UML overview. Published by Rational Software Corporation, January 1997.
- [13] Grady Booch, Ivar Jacobson, and James Rumbaugh. UML semantics. Published by Rational Software Corporation, January 1997.
- [14] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.
- [15] Marc H. Brown. Exploring algorithms using balsa-II. *IEEE Computer*, 21(5):14–36, 1988.
- [16] Marc H. Brown. Perspectives on algorithm animation. In *Proceedings Computer-Human Interactions '88: Human Factors Computing Systems*, pages 33–38, 1988.
- [17] Marc H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *Proceedings of the 1991 IEEE Workshop on Visual Factors Computing Systems*, pages 4–9, 1991.
- [18] Marc H. Brown and John Hershberger. Color and sound in algorithm animation. *IEEE Computer*, 25(12):52–63, 1992.
- [19] Marc H. Brown and Marc A. Najork. Algorithm animation using 3D interactive graphics. In *ACM symposium on User Interface Software and Technology*, pages 93–100, 1993.
- [20] Marc H. Brown and Robert Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28–39, 1985.
- [21] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Human Factors in Computer Systems HCI'91 Conference Proceedings*, pages 181–188, 1991.
- [22] Luca Cardelli. A language with distribute scope. In *ACM symposium on the Principles of Programming Languages*, pages 286–297, 1995.
- [23] Luca Cardelli, James Donahue, Lucille Glassman, Mike Jordan, William Kalsow, and Greg Nelson. Modula-3 language definition. *Sigplan Notices*, 27(8):15–42, 1992.
- [24] Shi-Kuo Chang, editor. *Principles of Visual Programming Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

- [25] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
- [26] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–19, 1990.
- [27] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. The effect of inheritance on the maintainability of object-oriented software: An empirical study. In *IEEE Conference on Software Maintenance*, pages 20–29, 1995.
- [28] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. Issues on the object-oriented paradigm: A questionnaire survey. Research Report EROCS-8-95, Department of Computer Science, University of Strathclyde, Glasgow, 1995.
- [29] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *OOPSLA*, pages 326–337, 1993. Published in ACM Sigplan Notices 28(10).
- [30] Wim De Pauw, Doug Kimelman, and John Vlissides. Modeling object-oriented program execution. In *ECOOP'94 Conference Proceedings*, 1994.
- [31] Nick S. Drew and Robert J. Hendley. Visualising complex interacting systems. Found on the WWW at the School of Computer Science for the University of Birmingham at <http://www.cs.bham.ac.uk/nsd/Research/>.
- [32] Robert A. Duisberg. Animated graphical interfaces using temporal constraints. In *Human Factors in Computer Systems SIGCHI'86 Conference Proceedings*, pages 131–136, 1986.
- [33] Stephen G. Eick, Joseph L. Steffen, and Eric E. Summer, Jr. Seesoft—a tool for visualizing line-oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.
- [34] T. Finin, R. Fritzson, D. McKay, and et. al. A knowledge query and manipulation language. Technical report, Department of Computer Science, University of Maryland, 1992.
- [35] Lindsey Ford. How programmers visualize programs. Research Report 271, Department of Computer Science, University of Exeter, Exeter, U.K., 1993.
- [36] Lindsey Ford. Interactive learning and researching with visualization. Research Report 274, Department of Computer Science, University of Exeter, Exeter, U.K., 1993.

- [37] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Targan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1:111–129, 1986.
- [38] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.
- [39] T. Gane and C. Sarson. *Structured System Analysis*. McDonnell Douglas, 1982.
- [40] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli, editors. *Fundamentals of Software Engineering*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.
- [41] Virginia R. Gibson and James A. Senn. System structure and software maintenance performance. *Communications of ACM*, 32(3):347–358, 1989.
- [42] J. Gosling, D. S. H. Rosenthal, and M. Arden, editors. *The NeWS Book*. Springer-Verlag, 1989.
- [43] Visualization Group. *Vz Visualization Library: User's Guide Version 1.0*. AT&T Bell Laboratories, Naperville, Illinois.
- [44] Kevin Glenn Halland. Towards metrics for object oriented languages. Master's thesis, Carleton University, 1992.
- [45] Warren Harrison and Curtis Cook. Insights on improving the maintenance process through software measurement. In *IEEE Conference on Software Maintenance*, pages 37–44, 1990.
- [46] D. A. Henderson, Jr and Stuart K. Card. Rooms: The use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Transactions on Graphics*, 5(3):211–243, 1986.
- [47] Robert J. Hendley and Nick S. Drew. Visualisation of complex systems. Found on the WWW at the School of Computer Science for the University of Birmingham at <http://www.cs.bham.ac.uk/nsd/Research/>.
- [48] Robert J. Hendley, Nick S. Drew, Andrew M. Wood, and Russell Beale. Narcissus: Visualising information. In *Proceedings of IEEE symposium on Information Visualisation*, 1995.
- [49] P. Hsia, A. Gupta, C. Kung, J. Peng, and S. Lui. A study on the effect of architecture on maintainability of object-oriented systems. In *IEEE Conference on Software Maintenance*, pages 4–11, 1995.

- [50] Ivar Jacobson. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, Wokingham, England, 1992.
- [51] S. Jarvey, H. Mitsui, and H. Nazkamura. Architecture of the xl C++ browser. In *CASCON '92 (IBM Centre for Advanced Studies)*, 1992.
- [52] Doug Kimelman, Bryan Rosenburg, and Rova Roth. Strata-various: Multi-layer visualization of dynamics in software system behavior. In *1994 IEEE Visualization Conference*, pages 172–178, 1994.
- [53] Hideki Koike. An application of three-dimensional visualization to object-oriented programming. In *Proceedings of the International Workshop on Advanced Visual Interfaces '92*, pages 180–192, 1992.
- [54] Hideki Koike. The role of another spatial dimension in software visualization. *ACM Transactions on Information Systems*, 11(3):266–286, 1993.
- [55] Hideki Koike and Manabu Aida. A bottom-up approach for visualizing program behavior. In *Proceedings of the 1995 IEEE Symposium on Visual Languages (VL '95)*, pages 91–98, 1995.
- [56] Hideki Koike and Hui-Chu Chu. VRCS: Intergrating version control and module management using interactive three-dimensional visualization. In *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL '97)*, page :to appear. 1997.
- [57] Hideki Koike and Hirotaka Yoshihara. Fractal approaches for visualizing huge hierarchies. In *Proceedings of the 1993 IEEE Symposium on Visual Languages (VL '93)*, pages 55–60, 1993.
- [58] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object oriented software maintenance. In *IEEE Conference on Software Maintenance*, pages 202–211, 1994.
- [59] Wilf LaLonde and John Pugh. Gathering metric information using metalevel facilities. *Journal of Object Oriented Programming*, 7(1):33–37, March-April 1994.
- [60] Wilf LaLonde and John Pugh. Providing a dynamic bubble help facility: Part2: The help facility. *Journal of Object Oriented Programming*, 7(4):74–76, July-August 1994.
- [61] Wilf LaLonde and John Pugh. Preparing for 3D graphics: Interfacing to OpenGL. *Journal of Object Oriented Programming*, 8(2):77–84, May 1995.

- [62] Wilf LaLonde and John Pugh. Rendering 3-D graphics in OpenGL. *Journal of Object Oriented Programming*, 8(3):63–72, June 1995.
- [63] Wilf Lalonde, John Pugh, Paul White, and Jean-Pierre Corriveau. Towards unifying analysis and design, and implementation in object-oriented environments. In *Proceedings of East EWOOP'93*, Bratislava, Slovakia, 1993.
- [64] John Lamping and Ramana Rao. Laying out and visualizing large trees using a hyperbolic space. In *Proceedings of the User Interface Software and Technology'94*, pages 13–14, 1994.
- [65] John Lamping and Ramana Rao. The hyperbolic browser: A focus + context technique for visualizing large hierarchies. *Journal of Visual Languages and Computing*, 7(1):33–35, 1996.
- [66] Moises Lejter, Scott Meyers, and Steven P. Reiss. Support for maintaining object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1045–1052, 1992.
- [67] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(5):41–49, 1986.
- [68] Henry Lieberman. A three-dimensional representation for program execution. In *Proceedings of the 1989 IEEE Workshop on Visual Languages*, pages 111–116, 1989.
- [69] Henry Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'95)*, pages 480–486, 1995.
- [70] Bennet P. Lientz and E. Burton Swanson. *Software Maintenance Management; a study of the maintenace...* Addison-Wesley, Reading, Massachusetts, 1980.
- [71] Ralph L. London and Robert A. Duisberg. Animating programs using Smalltalk. *IEEE Computer*, 18(8):61–71, 1985.
- [72] Jock D. Mackinlay, George G. Robertson, and Stuart K. Card. The perspective wall: Detail and context smoothly integrated. In *Human Factors in Computer Systems HCI'91 Conference Proceedings*, pages 173–179, 1991.
- [73] Jock D. Mackinlay, George G. Robertson, and Robert DeLine. Developing calendar visualizers for the information visualizer. In *Proceedings of the User Interface Software and Technology'94*, pages 109–118, 1994.

- [74] Siamak Masnavi. Automatic visualization of the dynamic behaviour of programs by animation of the language interpreter. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 16–21, 1990.
- [75] Brad A. Meyers. Visual programming, programming by example, and program visualization: A taxonomy. In *Proceedings SIGCHI 1986: Human Factors in Computing Systems*, pages 59–66, 1986.
- [76] Brad A. Meyers. The state of the art in visual programming and program visualization. Technical Report CMU-CS-88-114, Graphics, Visualization, and Usability Center, Carnegie Mellon University, Pittsburg, Pennsylvania, 1988.
- [77] Brad A. Meyers. Program visualization. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 877–892. John Wiley & Sons, Inc., New York, 1994.
- [78] Thomas G. Moher. PROVIDE: a process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, 1988.
- [79] Sougata Mukherjea and John T. Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source-level debugger. *ACM Transactions on Computer-Human Interaction*, 1(3):215–244, 1994.
- [80] Marc A. Najork and Marc H. Brown. A library for visualizing combinatorial structures. In *1994 IEEE Visualization Conference*, pages 164–171, 1994.
- [81] Marc A. Najork and Marc H. Brown. Obliq-3D: A high-level, fast-turnaround 3D animation system. *Journal of Visual Languages and Computing*, 7(2):219–242, 1996.
- [82] Jackie Neider, Tom Davis, and Mason Woo; OpenGL Architecture Review Board. *OpenGL Programming Guide: The official guide to learning OpenGL, Release 1*. Addison-Wesley, Reading, Massachusetts, 1993.
- [83] G.M Nielson, B.D Shriver, and J. Rosenblum. *Visualization in Scientific Computing*. IEEE Computer Society Press, Washington, 1990.
- [84] OpenGL Architecture Review Board. *OpenGL Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1992.
- [85] G. Parikh. *Techniques of Program and System Maintenance*. QED Information Sciences, Inc., second edition, 1988.
- [86] G. Parikh and N. Zvegintzov. Tutorial on software maintenance. *IEEE Computer*, 16:61–62, 1983.

- [87] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.
- [88] Jeff Prossie. Advanced 3-D graphics for windows NT 3.5: Introducing the OpenGL interface, part 1. *Microsoft Systems Journal*, 9(10), 1994.
- [89] Jeff Prossie. Advanced 3-D graphics for windows NT 3.5: Introducing the OpenGL interface, part 2. *Microsoft Systems Journal*, 9(11), 1994.
- [90] Georg Raeder. A survey of current graphical programming techniques. *IEEE Computer*, 18(8):11–25, 1985.
- [91] Steven P. Reiss. A framework for abstract 3D visualization. In *1993 IEEE Symposium on Visual Languages*, 1993.
- [92] Steven P. Reiss. An engine for the 3D visualization of program information. *Journal of Visual Languages and Computing*, 6(3):299–323, 1995.
- [93] Don Roberts and Ralph Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. <http://st-www.cs.uiuc.edu/users/droberts/evolve.html>. Web Pages.
- [94] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay. The cognitive co-processor architecture for interactive user interfaces. In *ACM SIGGRAPH Conference on User Interface Software Technology*, pages 10–18, 1989.
- [95] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay. Information visualization using 3D interactive animation. *Communications of ACM*, 36(4):57–71, 1993.
- [96] George G. Robertson and Jock D. Mackinlay. The document lens. In *Proceedings of the User Interface Software and Technology'93*, pages 101–108, 1993.
- [97] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone trees: Animated 3D visualizations of hierarchical information. In *Human Factors in Computer Systems HCI'91 Conference Proceedings*, pages 189–194, 1991.
- [98] Gruia-Catalin Roman and Kenneth C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, 26(12):11–24, 1993.
- [99] Gruia-Catalin Roman, Kenneth C. Cox, C. Donald Wilcox, and Jerome Y. Plun. Pavane: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3(2):161–193, 1992.

- [100] Gruia-Catalin Roman and H. C. Cunningham. Mixed programming metaphors in a shared dataspace model of concurrency. *IEEE Transactions on Software Engineering*, 16(12):1361–1373, 1990.
- [101] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan. A browser for directed graphs. *Software Practice and Experience*, 17(1):61–76, 1997.
- [102] James Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.
- [103] Johannes Sametinger. A tool for the maintenance of C++ programs. In *IEEE Conference on Software Maintenance*, pages 54–59, 1990.
- [104] Nan C. Shu. *Visual Programming*. Van Nostrand Reinhold, New York, 1988.
- [105] J.A. Simpson and E.S.C. Weiner, editors. *The Oxford English Dictionary*. Oxford University Press, Oxford, second edition, 1989.
- [106] Randall B. Smith and David Ungar. SELF: The power of simplicity. In *OOP-SLA*, pages 227–241, 1987. Published in ACM Sigplan Notices 22(12).
- [107] David Spence. Visual psychophysics of simple graphical elements. *Journal of Experimental Psychology: Human Perception and Performance*, 16(4):683–692, 1990.
- [108] John T. Stasko. Simplifying algorithm animation with TANGO. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 1–6, 1990.
- [109] John T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, 1990.
- [110] John T. Stasko. Three-dimensional computation visualization. Technical Report GIT-GVU-92-20, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 1992.
- [111] John T. Stasko, Albert Badre, and Clayton Lewis. Do algorithm animations assist learning? an empirical study and analysis. In *Human Factors in Computing Systems INTERCHI'93*, pages 61–66, 1993.
- [112] John T. Stasko and Charles Patterson. Understanding and characterizing software visualization systems. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 3–10, 1992.

- [113] John T. Stasko and Joseph F. Wehrli. Three-dimensional computation visualization. In *Proceedings 1993 Symposium on Visual Languages*, pages 100–107, 1993.
- [114] Joseph L. Steffen and Stephen G. Eick. High interaction data visualization using seesoft to visualize program change history. In *Human Factors in Computing Systems INTERCHI'93*, page 517, 1993.
- [115] Adobe Systems. *PostScript Language Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1985.
- [116] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Connecticut, 1983.
- [117] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics & Applications*, 9(4):30–42, 1989.
- [118] Jean-Yves Vion-Dury and Miguel Santana. Virtual images: Interactive visualization of distributed object-oriented systems. In *OOPSLA*, pages 65–84, 1994. Published in ACM Sigplan Notices 29(10).
- [119] Yun Wang. Interaction diagram generator - A tool for visualizing use cases in Smalltalk. Master's thesis, Carleton University, 1994.
- [120] Colin Ware and Glenn Franck. Viewing a graph in a virtual reality display is three times as good as a 2d diagram. In *1994 IEEE Conference on Visual Languages*, pages 182–183, 1994.
- [121] Colin Ware, David Hui, and Glenn Franck. Visualizing object oriented software in three dimensions. In *CASCON '93 (IBM Centre for Advanced Studies)*, pages 612–620, 1993.
- [122] Visual insights. <http://www.bell-labs.com/project/visualinsights/>. Web Pages.
- [123] Digital 3d animation. <http://www.research.digital.com/SRC/3D-animate>. Web pages.
- [124] Gvu: Software visualization. <http://www.cc.gatech.edu/gvu/softviz>. Web pages.
- [125] Gvu: POLKA animation system. <http://www.cc.gatech.edu/gvu/softviz/parviz/polka.html>. Web pages.

- [126] GVU: XTANGO algorithm animation system. <http://www.cc.gatech.edu/gvu/-softviz/algoanin/xtango.html>. Web pages.
- [127] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, 1992.
- [128] Norman Wilde, Paul Matthews, and Ross Huitt. Maintaining object-oriented software. *IEEE Software*, 10(1):75–80, 1993.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved