METAVIZ – ISSUES IN SOFTWARE VISUALIZING BEYOND 3D

Jian Qun Wang

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

August 2003

Your file Votre référence
ISBN: 0-612-83920-6
Our file Notre référence
ISBN: 0-612-83920-6

ABSTRACT

Metaviz – Issues in Software Visualization beyond 3D

Jianqun Wang

Software visualization can play a significant role in program comprehension. A large number of visualization tools have been developed to support program comprehension. Traditionally, these tools are 2D representations. In recent years, 3D software visualization techniques have been introduced to support program comprehension. These techniques provide new approaches to visualizing and comprehending software system structures and their internal relationships. At the same time, they introduce new research challenges. The software metaphors, layout algorithms, and readability criteria generally applicable in 2D software visualization cannot directly be applied in 3D visualizations. In this thesis, we present our research on the use of a new metaphor based on energy fields using the Metaballs 3D modeling and visualization technique. We also present grouping and layout algorithms, specially designed for 3D Metaballs based software visualization. These are built into Metaviz, a software visualization tool, which we have designed and implemented as part of our larger program comprehension environment, CONCEPT. Using Metaviz, we also show examples that illustrate how these visualization techniques, when combined with program slicing and metric based analysis, provide guidance during software comprehension during the testing and maintenance phrase.

To my wife Susan, my son Walter, who offer me unconditional love...

ACKNOWLEDGEMENT

I wish to express my gratitude to my advisor, Professor Juergen Rilling, for his supervision. Without his kindness and patience, this work would not have been feasible.

Special thanks go to my co-advisor, Professor Sudhir P. Mudur, for the hours he dedicated to my training. He was also my dependable guide through the risks of my graduate studies.

Finally, a sincere thank you to my wife Susan and son Walter, who had been my motivation to finish my studies.

Jian Qun Wang

TABLE OF CONTENTS

LIST OF TABLES

PAGE

LIST OF FIGURES

PAGE

CHAPTER I. INTRODUCTION

1.1 Introduction to Program Comprehension

Program comprehension is the process in which programmers try to understand what an existing program does and how the program achieves the intended goal. An existing program may be an incomplete program written by colleagues or in most cases a program that is undergoing some types of maintenance (Foltz 2001; Rajlich 2001). In theory, programmers can understand a system by studying its requirement and specification documentation. However, there are several problems with this approach. Often documentation might not be available or it may be out of date. The documentation does not provide enough detail to support the comprehension process for performing a certain task, e.g. debugging, maintenance tasks, etc. For most of the maintenance tasks, a programmer needs to have a detailed understanding of the source code and its internal relationships. More specifically it is necessary to have a real snapshot of the system and its actual implementation, rather than a specified/design view, which might have never been implemented in that fashion. Achieving such a detailed level of program comprehension is much more challenging because the program may be poorly documented or the documentation may be out of date. Considerable researches has been done from different perspectives to ease the difficulties of program comprehension (Sarita 2001; Tip 1995).

Cognitive aspects

From a psychological perspective, researchers have distinguished different cognitive models that can be applied during the comprehension process (Rilling 2002). The bottom-up approach reconstructs a high level of abstraction that can be derived through the reverse engineering of the source code. The top-down approach applies a goal-oriented method by utilizing domain/application specific knowledge to identify parts of the program that are necessary for identifying the relevant source code artifacts. This diversity in cognitive models is required to accommodate different skills of users and domain levels of the people comprehending a software system, and the different types of comprehension tasks. For example, during software maintenance, programmers might prefer an "over view first, zoom and filter, then details on demand" approach (Favre 2001; Harel 1992; Mayrhauser 1998; Rilling 2001; Sanlaville 2001; Storey 1999). On the other hand, when programmers debug a system, they might be only interested by the components related to the work.

Program slicing

As Mayhauser (1998) points out, for a typical comprehension task, a programmer might not be required to comprehend the whole system. Rather a partial understanding of the system is sufficient in many situations. One approach to support the creation of such a partial mental model is to focus a programmer's attention only on those parts of the system that are relevant with respect to a particular variable or function of interest. Program slicing (Korel 1998, Rilling 2003), a program reduction technique is one approach to help programmers to focus on specific parts (*cf* Figure 1). Through program slicing, programmers can concentrate on the relevant parts of a software system,

with respect to a particular slicing criterion. Static slicing is based on the analysis of source code, while dynamic slicing reduces the program through execution information.

1 scanf("%d",&n); 2 s=0; 3 p=1; 4 while(n>0) 5 { s=s+n; 6 p=p*n; 7 n=n-1; 8 printf("%d", s); 9 printf("%d", p);	1 scanf("%d",&n); 2 s=0; 4 while(n>0) 5 { s=s+n; 7 n=n-1; }
Original program fragment	**Slice whith respect to ({s}, 9)**

Figure 1. A program fragment and its slice

Software Visualization

Software visualization is one of the major approaches used in practice to support program comprehension. Software visualization provides a high-level abstraction of the detailed information found in the source code. From the perspective of visualization researchers, the logic of grouping elements should match programmers' mental grouping, and the layout algorithm used to display the elements should result in layouts that help improve readability.

Knight provides the following definition for software visualization: "Software visualization is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration (Knight1999)."

Text based software visualization provides detailed information, while graph-based software visualization provides structural and overall information about the software system, which can significantly help program comprehension. Graphic

visualization is used in many other disciplines, because people abstract information from good pictures faster than from texts (Herman 2000).

Clearly, software visualization can play a significant role in the program comprehension process (Knight2000). Good visuals at different levels of abstraction can often benefit the comprehension process much more than when one is looking through just a large textual representation. It is a well-known fact that the difficulty of large program comprehension forms to a major obstacle during software maintenance. When programmers have to maintain large legacy software, which may be poorly documented, they have to rely on the source code to construct a mental model of the system's structure and the internal relationships. Visualizations that represent this structure and internal relationships do greatly help in the task of program comprehension. Therefore, it is essential to develop tools and techniques that support software visualization at different levels of abstraction.

Graphic visualization techniques are widely used in program comprehension and software documentation (Knight 2000). Standard notations for graphic visualization are developed in documentations. UML is today's most popular notation and is accepted by most organizations. Among UML notations, different diagrams are used for different purposes. Use-case diagrams are for viewing the overall structure of software systems or of large subsystems; Class-diagrams allow programmers to concentrate on important concepts by hiding detailed information; collaboration-diagrams reveal internal relationships of software systems; sequence-diagrams is for dynamic behaviors visualizations.

Tool support

Several tools have to be developed that address the different program comprehension issues. One approach is a text-based documentation generator, for example Java Doc. The text-based documentation generators allow the user to select the level of documentation detail, while also providing an accurate picture (documentation) of the system under investigation. Another type of tool, which forms the main focus of this thesis, is a graphic-based software visualization tool. When the size of software becomes large, software visualization faces new challenges. Rilling el at (Rilling 2002) mentions several reasons for these difficulties: "(1) the diagram complexity is increased because of the large amount of information to be displayed, (2) the awkward layout techniques provided by the visualization approach, (3) their non-intuitive navigation, and (4) often their very specialized scope in depicting only certain program artifacts and their relationships."

1.2 Overview of CONCEPT project

The CONCEPT project is an software comprehension framework that integrates program slicing, reverse engineering, and software visualization techniques. Figure 2 shows the CONCEPT environment architecture. The CONCEPT project basically consists of three parts: the parsers, the program analyzers, and the visualization tools.

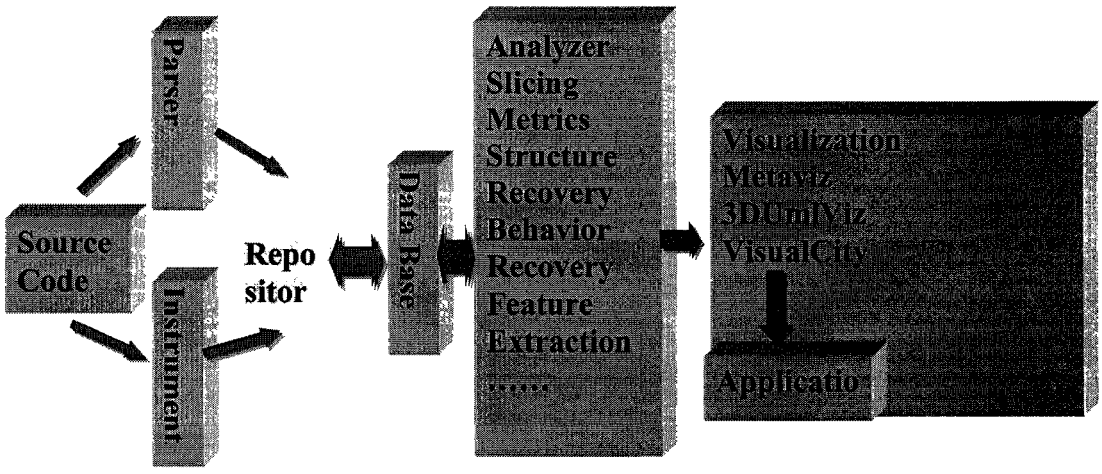


Figure 2. Architecture of CONCEPT project

The parser part is responsible for extracting information from programs. Currently, we use two parsers. Jike, developed by IBM, parses the static source codes and stores the parsed output into XML files. In order to quickly search the data, we convert the XML files into MySQL database. Another parser, developed by Yonggun, can extract execution information from running programs through Java Virtual Machine and generate an AST, abstract syntactic tree (Yonggun 2003). We also convert the AST tree into a Postgres database for quick searching.

Once the data is stored in the databases, further analysis can be performed on it. The analyzer part contains a variety of source code analysis techniques. More specifically at the current development stage, the CONCEPT project supports static program slicing, dynamic program slicing, hybrid slicing, feature analysis and software metrics analysis. The analyzer part has the following functions: software maintenance, testing, and software architecture recovery. For the analysis results to be useful, the results have to be further abstracted and visualized to enhance the comprehension process.

The visualization front-end is the top layer of our framework, providing the necessary visual support for program comprehension, by creating abstract views of the underlying parsed and analyzed source code. The visualization part contains three visualization tools, which are under development. 3D UML is an extension of traditional 2D UML visualization techniques, by taking advantages of the third dimension (Xiaohua 2003). The VirtualCity uses a city representation to abstract and visualize software artifacts (Shenghua 2003). Metaviz, implemented as part of this thesis research, is another approach to visualize software artifacts. It maps software components to the properties of Metaballs. The combination of Metaviz with source code analysis allows programmers to visualize information that is required to perform specific comprehension tasks. As we shall see later, Metaviz in combination with grouping and layout algorithms supports an "overview whole structure, zoom in" approach (Rilling 2002).

1.3 Overview of Metaviz

Metaviz was developed using Java 3D, a graphic package, as an independent, but reusable 3D visualization tool to investigate the various application domains for the Metaballs metaphor. As part of this research, we integrated Metaviz as a plug-in into our CONCEPT project to investigate the use of Metaballs for software visualization and software comprehension. The Metaviz tool provides a programming interface that allows for an easy extension and further reuse of the tool. Along with Metaviz, two Java API packages for Metaballs and 3D grid layout are developed as reusable components for further research.

Depending on the input data, Metaviz can visualize the class level of programs either statically or dynamically. For static visualization, Metaviz takes input data from MySql database generated by Jike, and displays the class-level pictures. For dynamic visualization, the input data is generated from execution trees, and the output pictures generate animations. We compare the static software visualization and the dynamic software visualization in Metaviz in Figure 3.

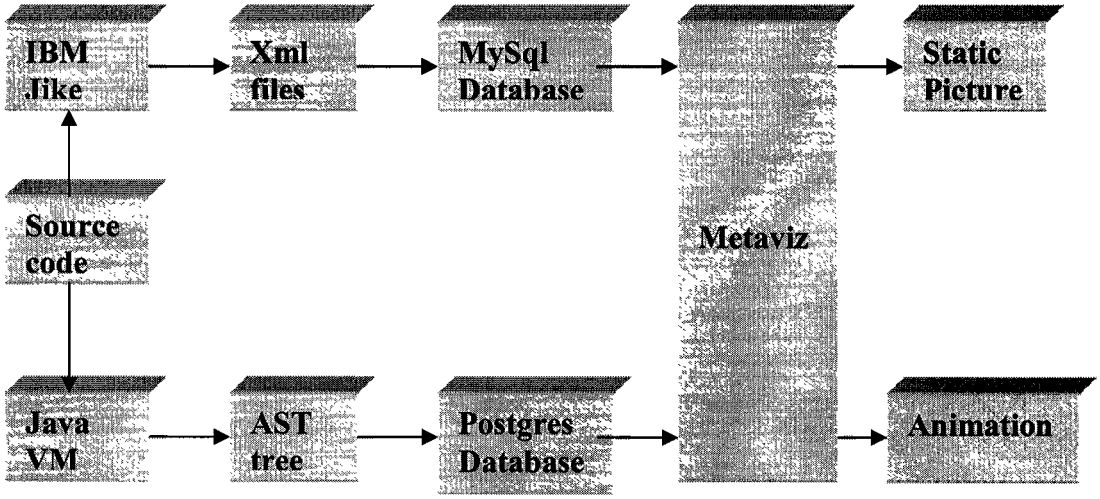


Figure 3. Comparison of static and dynamique visualization in Metaviz

Metaviz is designed from the perspective of graph drawing, instead of the requirements for program comprehension. In other words, the Metaballs metaphor, and the clustering and layout algorithms are not developed only for a systematic program comprehension. Rather they are designed to support different research areas within the context of program comprehension and the CONCEPT project (e.g. software metrics, architecture recovery, etc.).

1.4 Significant Contributions

In this research, we have explored the use of a new 3D metaphor, the Metaballs, which is based on an energy function defined as every point in 3D space, for software visualization. We developed a 3D visualization tool, Metaviz. The tool has been integrated into the CONCEPT project. We use the Metaviz tool to investigate the use of Metaballs metaphor in software visualization (Rilling2002). The motivation was to use our implementation of Metaviz and the Metaballs, to explore possible application areas for the Metaballs metaphor, as well as their applicability and usability for different purposes.

As part of this research, we developed two java packages, which are the Metaballs package itself and a 3D-grid-layout algorithm package. The Metaballs package is the first public domain implementation of a Metaball modeling visualization program written in Java3D. The package can be used to view any model that can be represented by a formula from a propeller to a horse-seat. The 3D-grid-layout algorithm package can be used for improving the readability of any 3D graph that represents internal-relationships as lines between entities. This algorithm is already used by Cubical software visualization for 3D layout the CONCEPT project (Xiaohua 2003).

1.5 Thesis organization

This thesis consists of six chapters. In this chapter, we briefly discussed the use of graphical visualization techniques in program comprehension, and introduced both the CONCEPT project and the Metaviz tool.

In the second chapter, we present an overview of related literature, and compare software visualization techniques in 2D and 3D. .

In the third chapter, we introduce the Metaballs metaphor in detail and its visualization. We also discuss the motivation for using Metaballs in software visualization.

In the fourth chapter, we discuss different algorithms that are implemented to visualize Metaballs within Metaviz. We also discuss optimization of the Marching-cube algorithm that improves Metaballs rendering speed. Additionally, we discuss how a 3D-grid-layout algorithm improves readability of a 3D based visual.

Chapter five explains the design and implementation of Metaviz, including the architecture of Metaviz, the choice of implementation environment and some experimental results.

Finally, in chapter six, we provide our conclusions and propose some future research directions.

CHAPTER II. RELATED WORK

2.1. Taxonomies of Software Visualization

"A well founded taxonomy can further serious investigation in any field of study" (Price1992). Taxonomies of software visualization serve as a common language or terminology. Taxonomies provide this common language and allow new discoveries to be identified and catalogued. Software visualization is classified in different ways from different perspectives. In this section, we discuss Myer's classification, which is considered as one of the best known taxonomies (Price, 1992; Knight, 2001). We also discuss the taxonomy based on representations, which is related to our research.

2.1.1. Myer's taxonomy

Myer introduced a systemic taxonomy for software visualization in 1986 (Myer 1986), and updated it twice (Myer 1988; Myer 1990). He classified software visualization in the following six categories (Myer1990).

Static code visualization

Dynamic code visualization

Static data visualization

Dynamic data visualization

Static algorithm visualization

Dynamic algorithm visualization

Myer's taxonomy clearly distinguishes software visualization from visual programming. He points out that software visualization is concerned with the use of graphs to visualize some parts of the program after it has been written. Knight (Knight1998) summarized Myers classification into a two dimensional table as shown in Figure 4. In our research, we only investigate dynamic and static code visualization, related to cell 1 and 4 in the table of Figure 4.

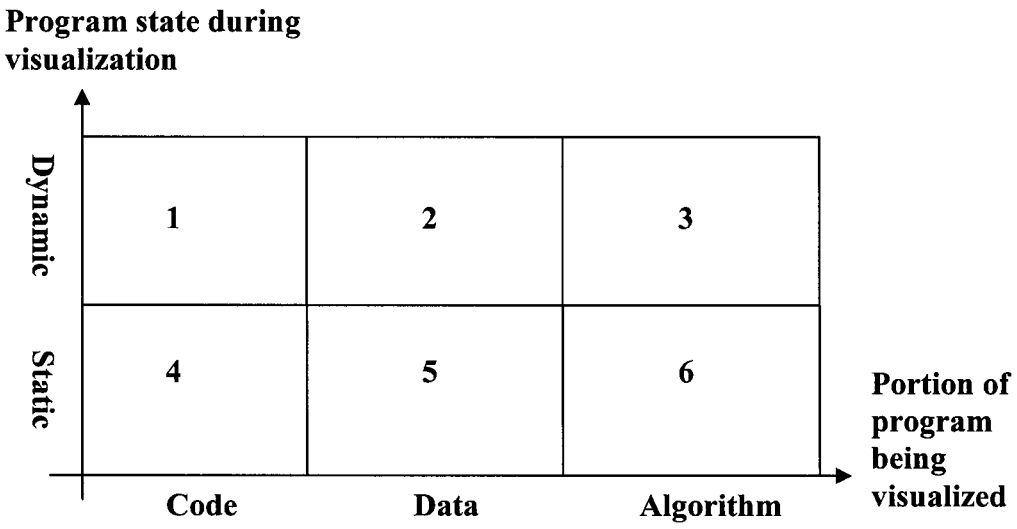


Figure 4 - Visual Representation of Mayer's Taxonomy

2.1.2. Taxonomy by representation

Other taxonomies are produced from different perspectives (Price, 1992; Rom, 1993). Price tries to create a road-map for software visulization, "and provides a common language and allows new discoveries to be identified and catalogued" (Price, 1992). Since this thesis is about how to visualize software, discussing all the different taxonomies is beyond the topic of this thesis. In what follows, we discuss the taxonomy of software visualization according to the visual representation, which was inspired by

Knight (Knight2001). The importance of visual representation of program comprehension is well documented (Knight, 1999; Price, 1992). Knight C. el at (Knight 2001) distinguish "Nodes and Arcs" as early representations, and three-dimensional software visualization as newer representations.

2.1.2.1. "Nodes and arcs"

A visualization survey by Ivan (Herman 2000) showed that 90% of the available software visualization tools are based on "nodes and arcs" or "nodes and edges", with nodes typically representing entities and arcs/edges representing the relationships. The node and arc representation raises some general questions. What kind of entities and internal relationships should be chosen to be visualized for a specific application? Also, what type of metaphor can help programmers build the right mental model? How does one group the entities in order to help software comprehension? Which layout algorithms are suitable for improving readability in a defined application?

Figure 5 shows the applications that use "nodes and arcs" as representation. Although (1) balloon view and (Melançon 1998); (2) h-tree view are 2D visualizations (Eades 1999); (3) hyperbolic view (Munzner 2000); (4) radial view (Sarkar 1992, 1994) are 3D visualizations, all of them represent software as "nodes and arcs".

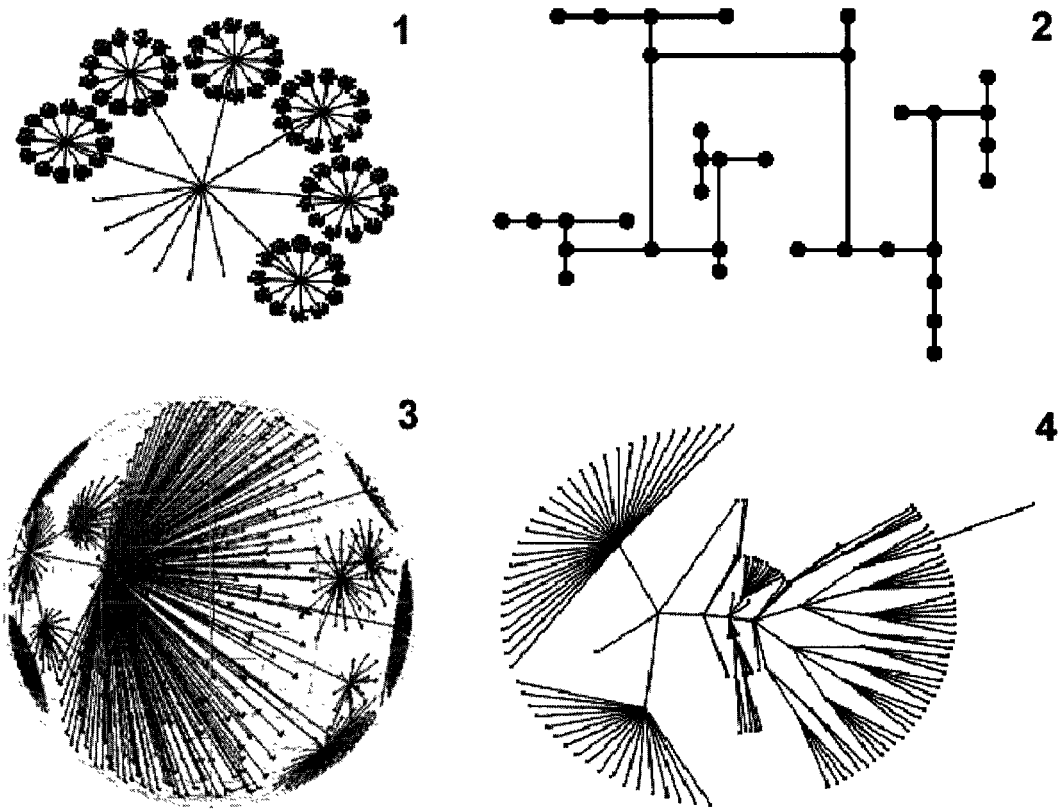


Figure 5 Representations of Nodes and Arcs (1) Balloon view (2) H-tree view (3) Hyperbolic view (4) Radial view

2.1.2.2. Other representation approaches

Recent researches in software visualization have advocated the use of 3D graphic techniques. The artifacts of software are represented by color, shading, and the extra third dimension. Some of these representations use geometric objects, which is a kind of extension of "Nodes and Arcs". Others use virtual reality, in which the artifacts of software are mapped to the real objects world that humans are already familiar with.

CallStax is an example of using geometric objects for nodes, as seen in Figure 6 (Yong1999). The visualization of CallStax attempts to move away from the standard

visualizations of call-graph structures, i.e. a network consisting of nodes and arcs in 2D. "CallStax makes full use of the extra dimension afforded by VR to maximize the amount of information available and the flexibility for displaying and interacting with that information" (Young 1997).

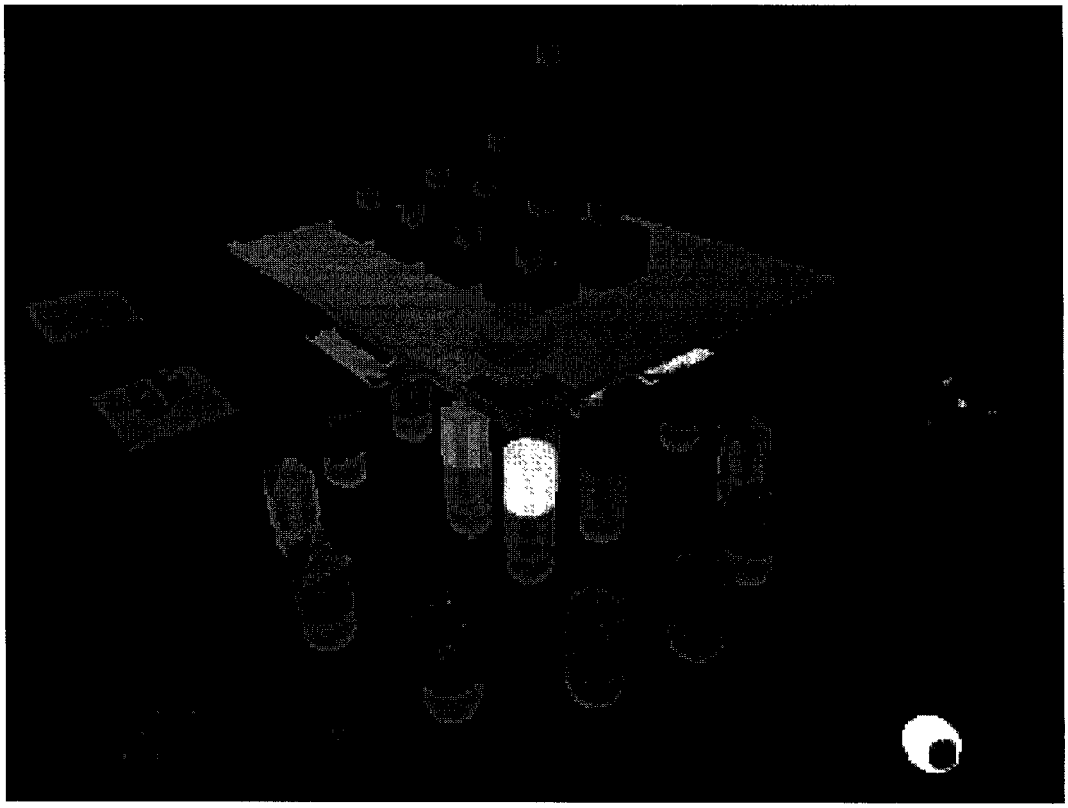


Figure 6 CallStar visualizations underneath part of a FileVis display

Others map software systems to the real world objects, such as Software World, which maps software to buildings in a city (Yong1998). Using virtual reality to visualize software is quite a different method compared to the conventional representation of "Nodes and Arcs". Figure 7 is a visualization of standard Java API. The mapping between virtual reality and Java APIs is listed in Table 1. The height of buildings represents the lines of the codes that the corresponding method contains. Mapping

software to cities can help people build a mental map and ease the difficulties of navigation and interaction within such a virtual reality.

Table 1. Actual mapping from Java code to graphics (Yong1998)

Visualization Level	Code Element
World	**The software system as a whole**
Country	**Directory structure, which maps to the packages in Java**
City	**A file from the software system**
District	**Class (contained within the specific file and hence city in the visualization).**
Building	**Methods**

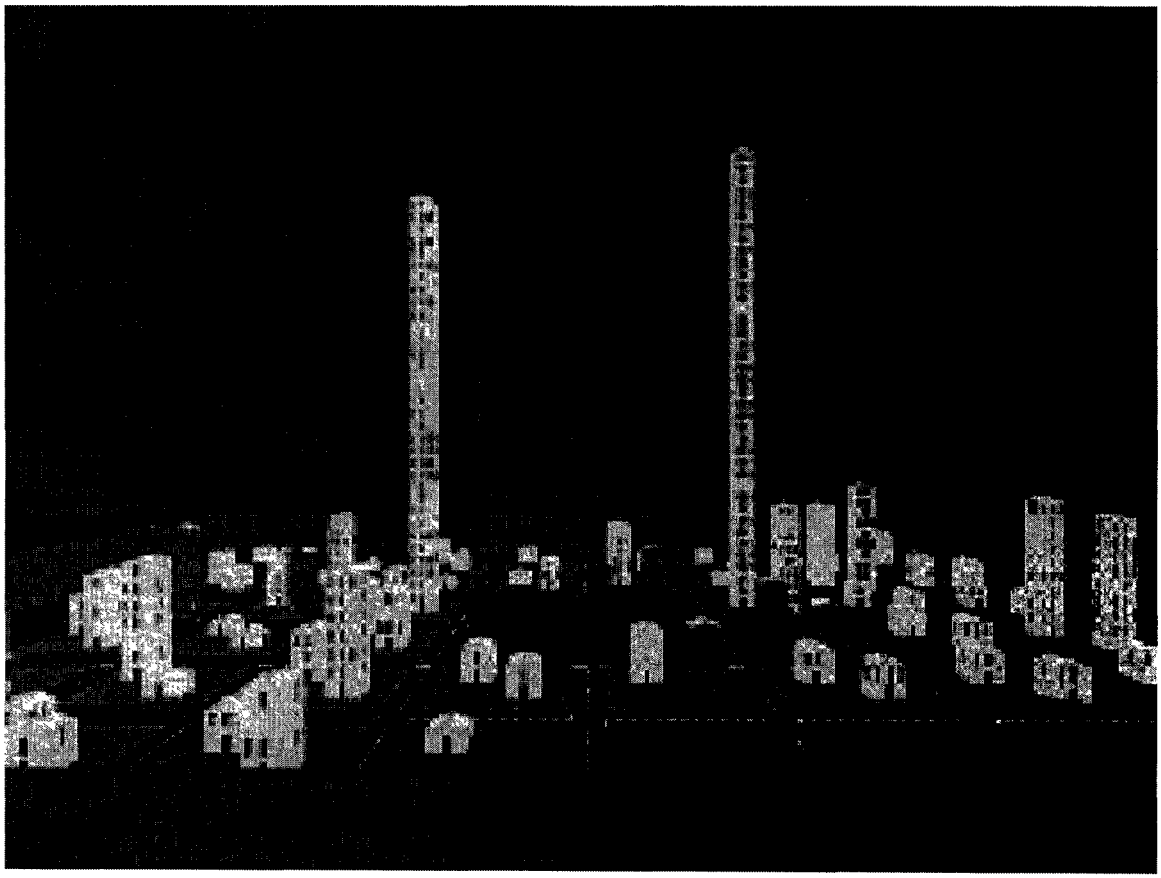


Figure 7. A longer range view of part of Software World

2.2. 2D software visualization techniques

2D software diagramming techniques are widely used in the program development process. They are either used as documentation during the software specification/ design phase or generated by reverse engineering tools to support the comprehension process. From flow diagrams used in structured programs to UML used in Object Oriented Programs, 2D graphic software visualization techniques greatly help in understanding the complexity of software system, by providing a high-level view of the underlying system.

2.2.1. Flow chart

Since 1947 Goldstein and von Neumann (Neumann 1947) demonstrated the usefulness of flowcharts. They are widely used for visualizing data structures and control flows. Price reviews in (Price 1992) the development of flow charts. The first application that could automatically generate flowchart from FORTRAN or assembly language programs was developed in 1959 by Haibt. However, early experiments of flowcharts were most used for design instead of being used as an aid to comprehension. Scanlan designed an experiment to find out if real differences in comprehension exist between structured flowcharts and pseudo code. The results strongly indicate that structured flowcharts do indeed aid algorithm comprehension. A large difference was found even for the simplest algorithm (Scanlan 1989).

Figure 8 shows a comparison between a flowchart and control structural diagram.

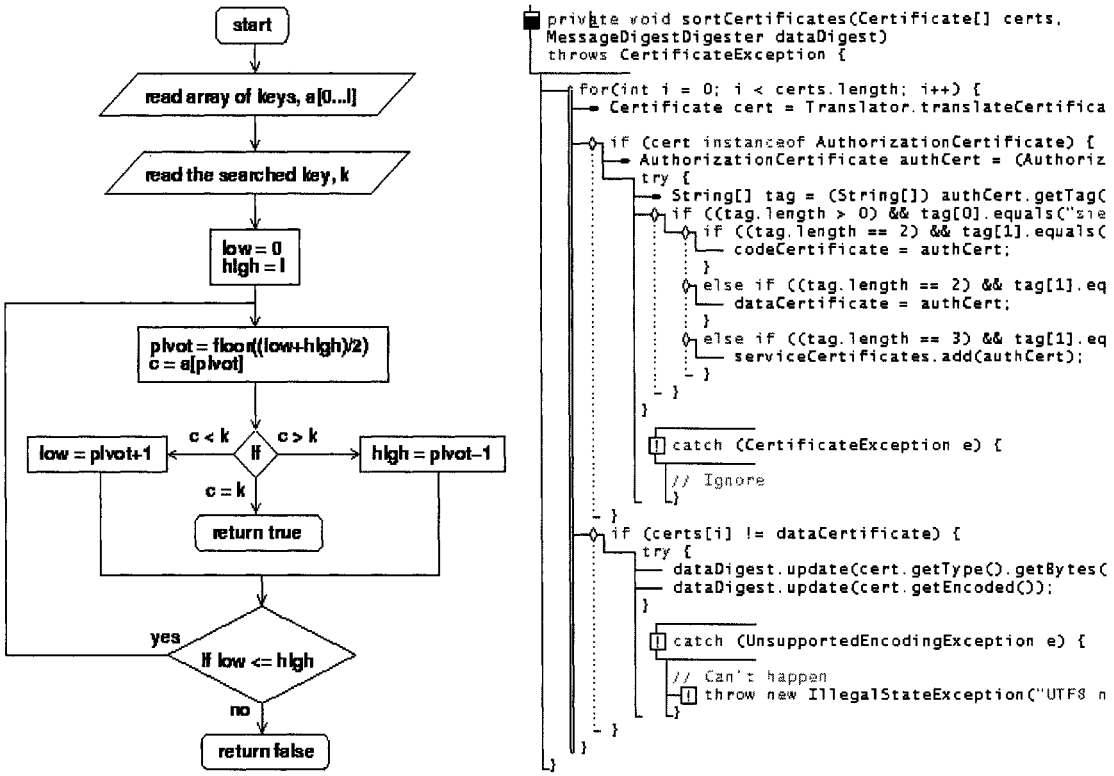


Figure 8. A simple flowchart Vs. Control Structure Diagram

2.2.2. Call graph

The pointers to functions can be depicted by call graphs. The call graphs can help discover the internal relationships of software. A 2D call graph cannot avoid line crossings, and too many line crossings will result in a massy, unreadable graph. aiCall (AbsInt 2003) automatically calculates a GDL (graphic description language) representation of the call graph and the control flow graph of an application code. The graph can then be visualized, interactively explored and printed with aiSee (AbsInt 2003), which is included in the aiCall distribution package.

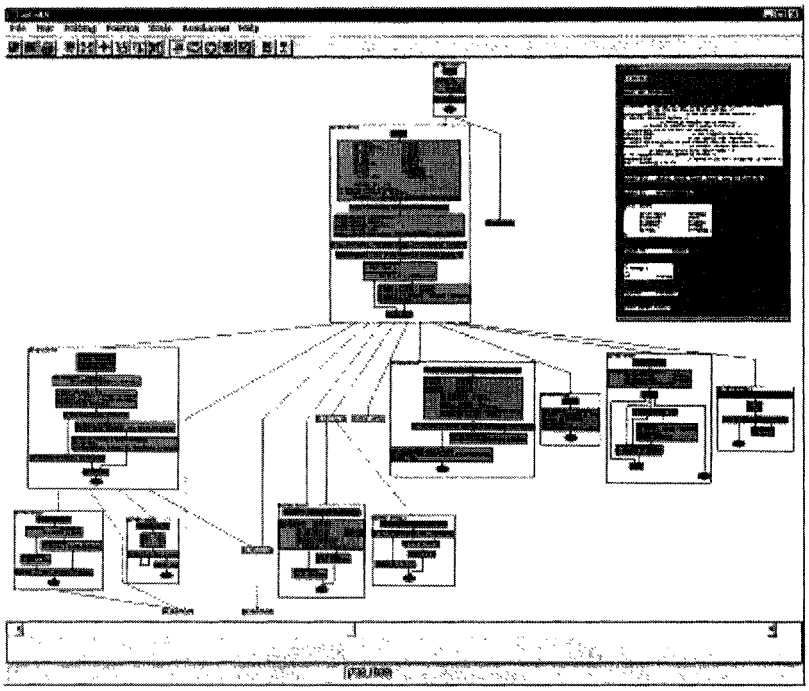


Figure 9. Call graph a software application Generated by aiCall for C.

2.2.3. UML diagram and Object Oriented Program

"The Unified Modeling Language (UML) is a visual language for modeling software designs and is currently the most widely accepted standard for software diagrams in the software engineering field" (Dwyer 2001). The notations of UML are mostly tools of design, although these can be generated by reversing engineer source code for the purpose of supporting program comprehension.

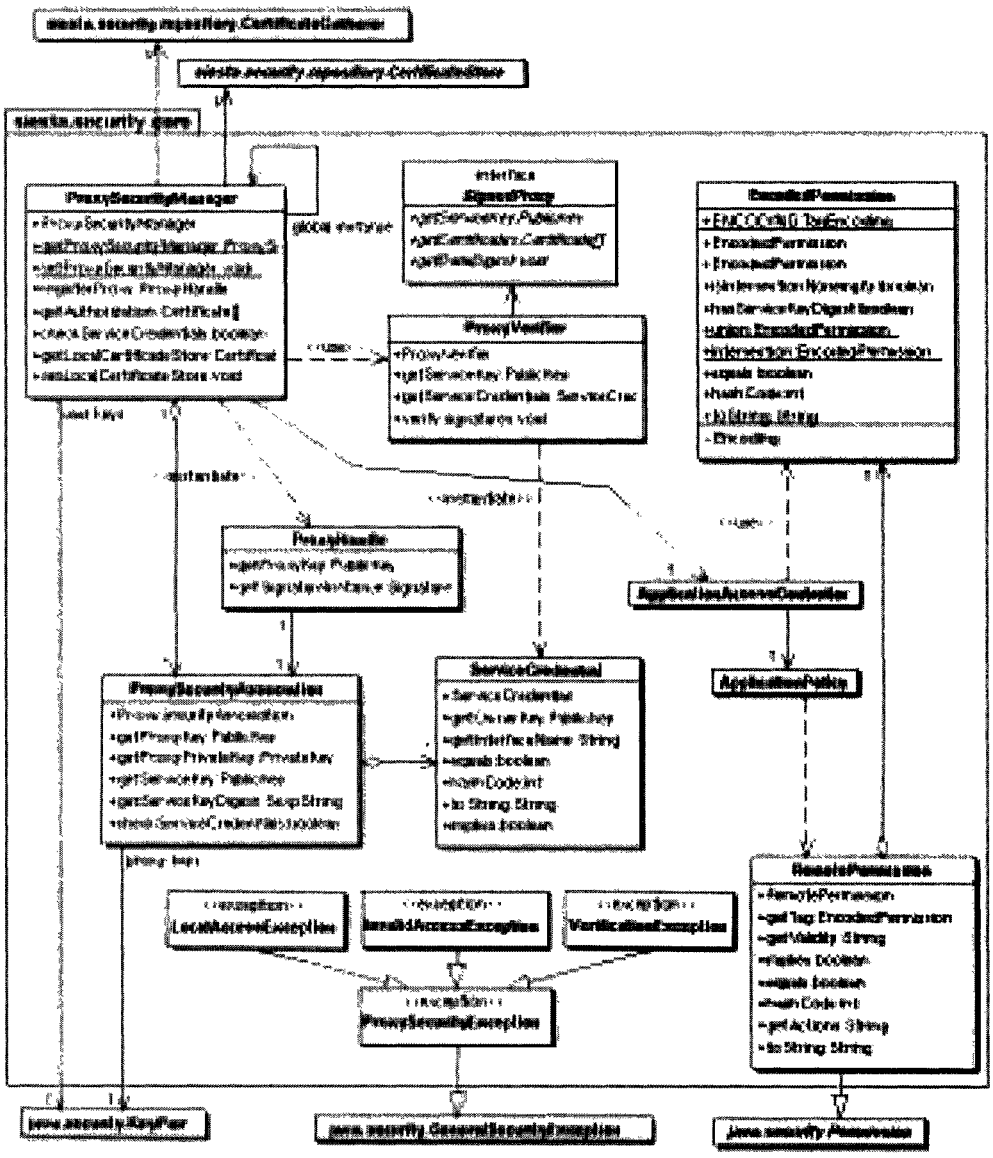


Figure 10. UML Class Diagram

2.2.4. Tree map

A graph of nodes and edges can represent the parent-children relationships, but attributes, such as node size, are ignored in this conventional graphical representation for tree structures. Shneiderman (Wesley 1991) presents tree maps that fall in the category of space-filling diagrams, shown in Figure 11. Within the tree maps, each node is depicted

as an area proportional to a selected attribute. A rectangle representing the entire tree is split vertically into rectangles whose areas are proportional to the sizes of the corresponding sub trees. The algorithm is recursive, splitting rectangles vertically for even levels and horizontally for odd levels. Shneiderman suggests that appropriate color coding can be used to identify other properties such as category.

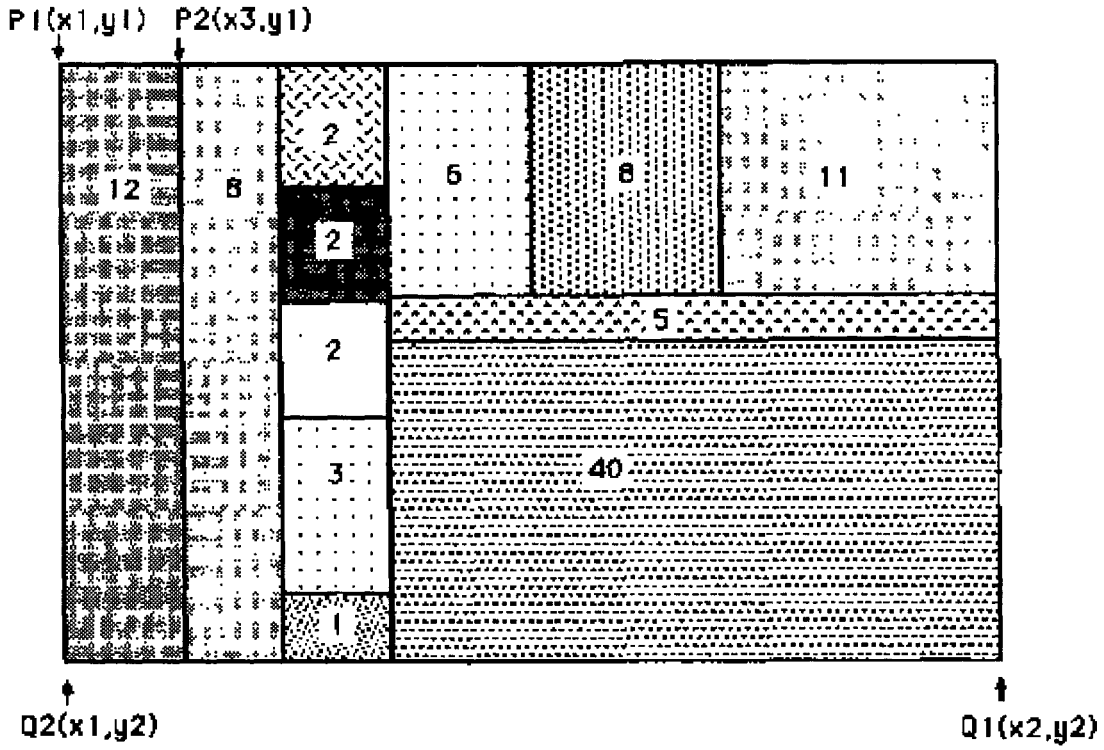


Figure 11 Examples of 2D software visualization (Wesley 1991)

In summary, the collection of UML diagrams is well developed, and is a dominant tool for software design and visualization of object-oriented programs. Other 2D software visualization techniques are designated for different purposes. ER diagrams are suitable for visualizing structural aspects of relational database; tree maps can be used for visualizing the amount of data.

2.3 3D software visualization techniques

It is a well-known fact that 3D software visualization has potential advantages over 2D, not only because the extra dimension provides more space, but also because the real world is three-dimensional (Knight 2001, 2002). This added realism of the visualization technique makes 3D representations often more likely to be accepted by programmers as a mental map. By mapping source code structures and program executions to 3D space, 3D software visualization techniques are considered as one approach to reduce the limitations of the visual medium used to visualize the data. Compared with 2D software visualization, the 3D visualizations are based on mathematical models that use shading, light, shape, and animation to ease the complexity of software system. The strength and usefulness of 3D software visualization is already documented in several different research projects (Knight 2001). Below, we will review some 3D software visualization techniques to illustrate their advantages over 2D software visualization techniques.

2.3.1. The extension of 2D software visualization

The current research in 3D software visualization techniques is expected to provide significant benefits over the widely used 2D software visualization techniques, by utilizing the extra dimension available. In what follows we investigate some of the current approaches to 3D visualization, which, as we shall see, are attempts to map traditional 2D techniques into the 3D space.

2.3.1.1 Call graph in three-dimensional space

Gogolla proposes a representation and animation of UML diagrams in a three-dimensional diagram style (Gogolla 1999). He tries to improve the comprehension of complex diagrams through three-dimensional diagram layout and animation. Figure 11 shows the comparison between 2D sequence diagram and its counterpart in 3D space.

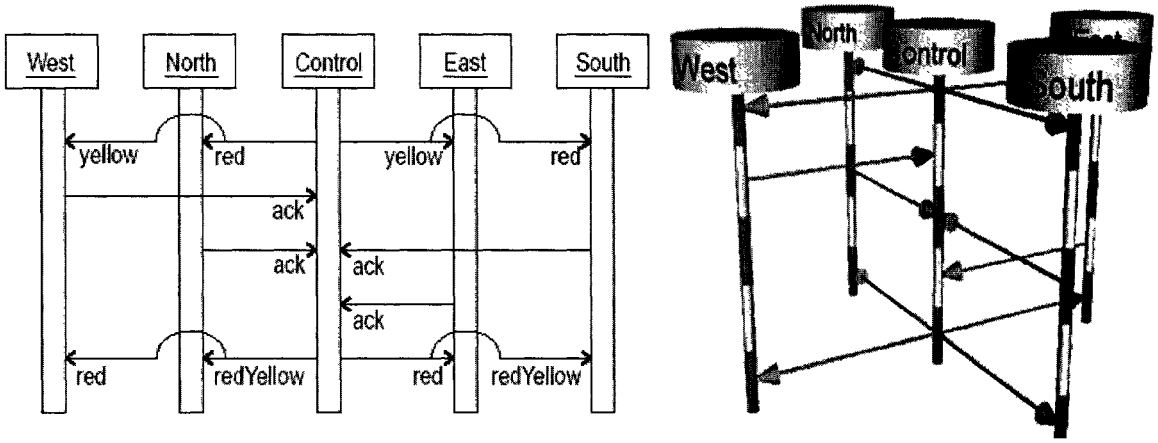


Figure 12. 3D sequence diagram (Gogolla 1999)

2.3.1.2. 3D tree map

The nested tree map (Figure 13) utilizes the third-dimensional space by combining "nodes and arcs" with 2D tree maps. Churcher el at (Churcher 1999) shows that "this combination is particularly useful for situations where a tree map is appropriate at lower levels and would be too complex for the whole of a large tree."

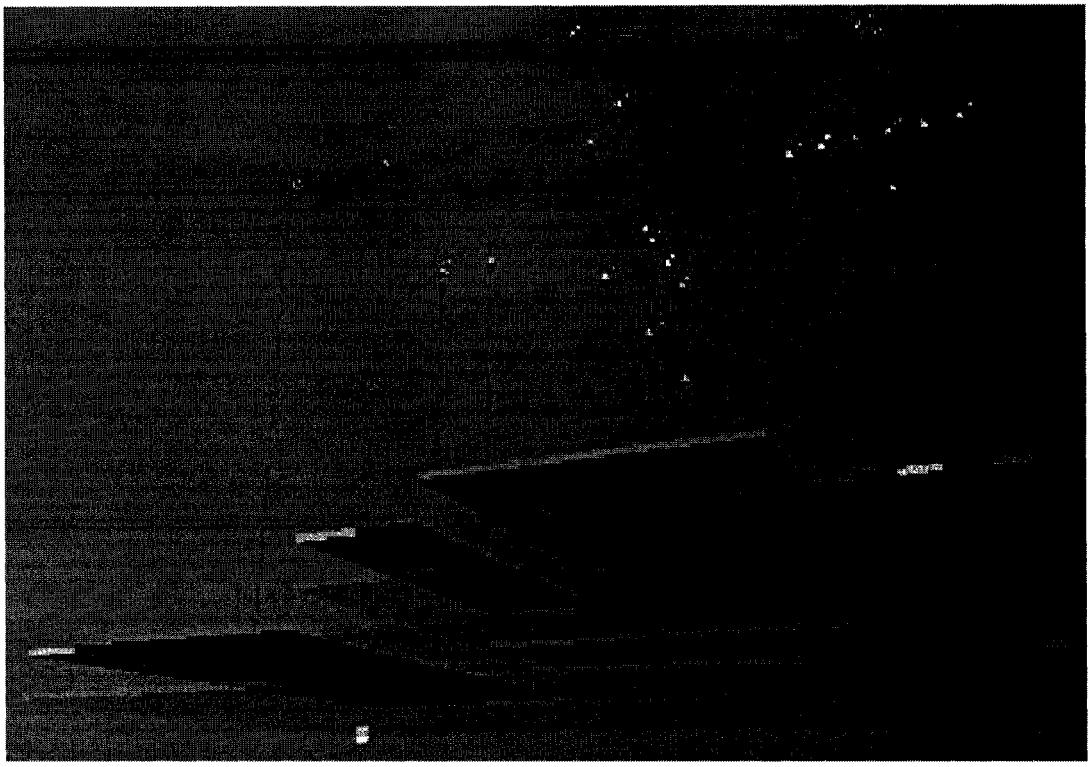


Figure 13 Variations on tree maps implemented in VRML (Johnson 1991)

2.3.2. Semi-transparency in Information cube and Cone Tree

The uses of transparency and the depth of the third dimension also make 3D visualization more powerful than traditional 2D visualization approaches. For example, the information cube, as shown in the left part of Figure 14 (Rekmoto1993) illustrates how to visualize hierarchical data by using semi-transparency. Similarly, the Cone Tree (see the right part of Figure 14) (Robertson1993) shows the depth of sub-trees glued by semi-transparency. Displaying multiple transparent objects is not supported by current graph renders due to performance issues. The semi-transparency is a practical technique to avoid the complexity of rendering multiple transparent objects.

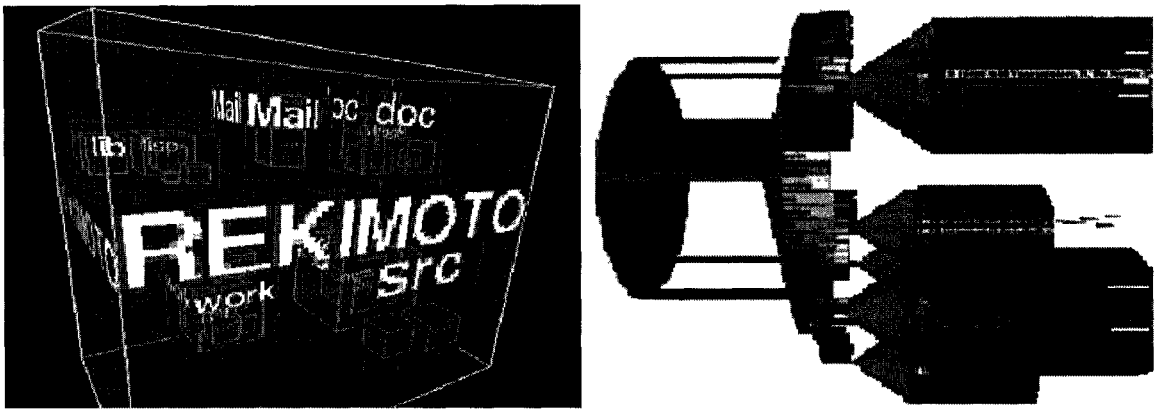


Figure 14. The use of semi-transparency

2.3.3. Virtual reality

With three-dimensional visualization, it is possible to actually build a VR (virtual reality) environment that maps program artifacts and their relationships to real world objects. The static aspects of the virtual environment help represent the structure and internal relationships of software. In combination with animation, the same virtual environment can also be used to comprehend behavioral aspects (e.g. program executions, dynamic binding, etc.). Further, as it has been pointed out by L. Feijs el at. (Feijs1998), the consistency with objects in a 3D layout along with new options of multiple viewpoints makes 3D visualization techniques worthwhile to investigate. From the perspective of program comprehension, the power of 3D software visualization is that it closes the conceptual gap between 2D representations as mostly used and the real world objects that are all in 3D space. The human brain is already trained and wired to comprehend existing 3D real world objects, thus making the mapping of software artifacts into 3D space a "natural" extension from a human perspective. Knight C. *et al.* (Knight2000) define 2D software visualization as merely a representation, while 3D

software visualization is considered as a representation plus a metaphor. The metaphor plays an important role in 3D software visualization because a non-intuitive mapping from a user perspective might limit the usability of the visualization technique.

2.4 Open problems in software visualization

Knight *et al.* (Knight 2001) summarize eight software visualization issues: "evolution, scalability, navigation and interaction, automation, correlation, visual complexity, and finally metaphor." In this section, we discuss some of these issues, which are also addressed by our research.

2.4.1 Visual complexity

When people use software visualization tools, the comprehension of large and complex programs is restricted by the resolution limits of the visual medium (2D computer screen) and the limits of user's cognitive and perceptual capacities. With 2D computer monitors, the current available visual equipment, our task is how to reduce the limitation of visual complexity. In other words, we have to investigate all possible issues related to visualizing software on computer screens.

2.4.3 Information overload

Information overloading is another problem in software visualization. From the perspective of psychology, the amount of information that people are able to obtain from one screen is very limited (Herman 200). When this limitation is exceeded, people cannot get any more useful information. This situation is also referred to as information

overloading. Even, if the visual medium is large enough to display all information, we may reach the bound of information overloading. In Figure 15 (Lanza 2002) the only information we can observe is a tree structure. However, we are not able to identify and comprehend any details from the current view, because of the large amount of information displayed.

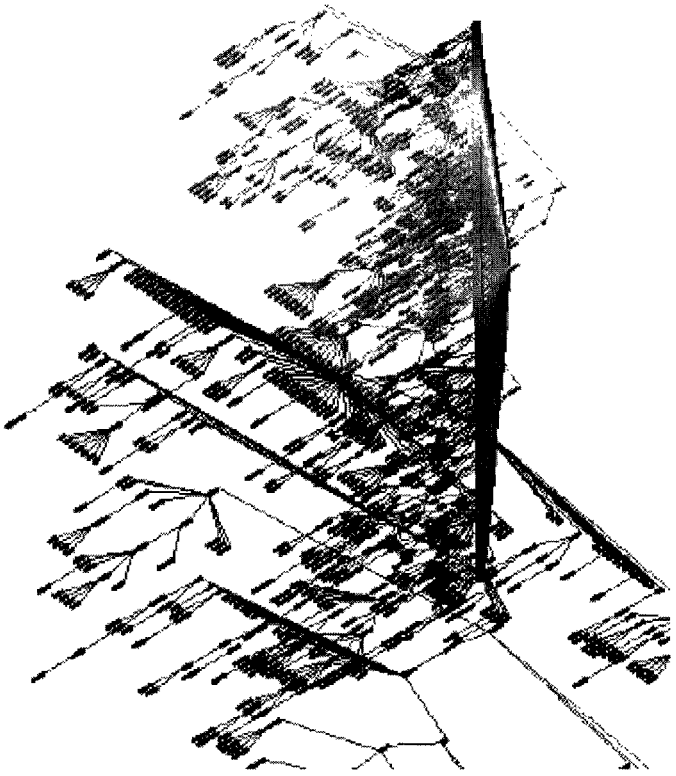


Figure 15. The Smalltalk class hierarchy by Jun/OpenGL

2.4.2 Scalability

Visualizing large software is the main challenge for all software visualization techniques. Most software visualization tools work well for small software systems, but encounter difficulties when visualizing larger software systems. Large legacy software and modern software demand visualization tools to be scalable, thus scalability is

becoming one of the most important criteria for testing software visualization tools and their usability.

2.4.3 Navigation and user interaction

Navigation and user interaction are other aspects that can improve scalability and usability of software visualization tools and their ability to cope with large amount of information in an intuitive way. Intuitive navigation and interaction is important. Intuitive and flexible navigation techniques also enhance the comprehension process and determine the acceptance of visualization tools (Knight 2001). Young even creates guidelines for navigation and orientation from city planning textbooks when he developed his Software World (Yong1999).

2.4.4 Metaphors for software visualization

A metaphor is an expression of the understanding of one concept in terms of another concept, where there is some similarity or correlation between the two. With the introduction of 3D software visualization techniques, many new metaphors are being investigated to help in the understanding of large programs (Knight2000). Mapping program artifacts into the 3D space allows users to identify common shapes or common configurations that may become apparent, and which could then be related directly to design features in the code. Soft City (Knight2001), Cone Tree (Robertson1993), hyperbolic tree (Munzner 2000), and Information Cube (Rekmoto 1993) are some examples of software visualization techniques that are based on different 3D metaphors. Additional examples for 3D metaphors can be found in (Herman2000). Each of these

metaphors has its own pros and cons, and is often only suitable for a specific comprehension purpose. Moreover, a metaphor that is easily understood by one programmer may be confusing to another programmer. Therefore, one of the challenges of current research is to investigate and explore new metaphors.

2.4.5 Existing problems in 3D software visualization

"A badly designed three-dimensional visualization is worse than none at all. The extra dimension can open a whole new world of possibilities but at the same time also new challenges."(Knight2000)

While the benefits of adding a third dimension seem to be obvious, these benefits will become distinct only if the visualization techniques explicitly take the advantages of the added dimension. However, as we have seen earlier, most of the current approaches simply transform established 2D visualizing techniques into a 3D space (Herman2000). Simply extending "nodes and arc" technique into the 3D space does not necessarily harness the power of 3D software visualization.

Figure 17 (Herman2000) shows an idealized graph with a nearly perfect layout. Note that the readability of this graph is diminished because the level of abstraction is not high enough. The visualization has to deal with a large quantity of information, but the graph does not scale up to effectively visualize this large quantity of data. Scalability is a well-known barrier that exists both in 2D and 3D software visualization. The large number of elements and their internal relationships pose several difficult problems. Figure 16 shows such problems in using Call graph in 3D.

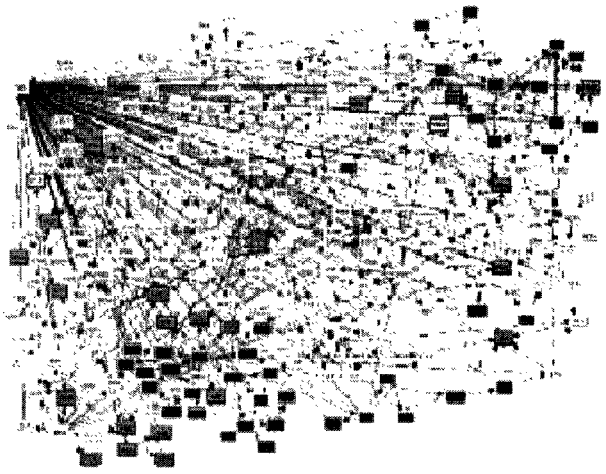


Figure 16. Call graph in 3D

Many of these 3D visualization techniques lack a good layout algorithm, resulting in large number of interference (line crossings) in the visual image. Providing good layout management is expensive and often very difficult. The interrelationships among elements are actually multi-dimensional data, which cannot be mapped onto 2D or 3D objects in a natural manner. It has been proven that line crossings are unavoidable in the 2D graph theory (seven bridges of königsberg) (Euler 1736), but a good layout algorithm can reduce line crossings and improve readabilities tremendously.

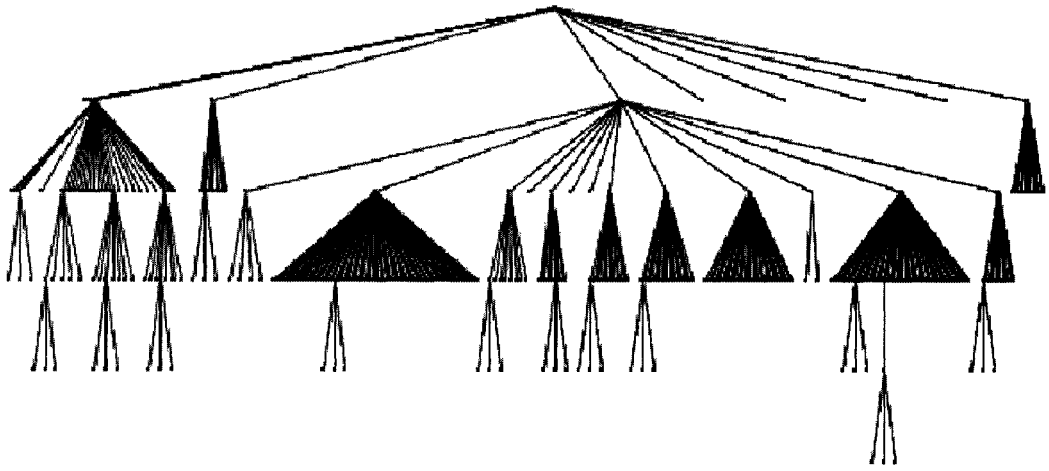


Figure 17. A tree layout for a moderately large graph

Improving and optimizing layout, particularly for large 3D graphs, is expensive in terms of time and memory requirements. A more detailed discussion of layout management and its complexity is presented in Section 4. Even if one improves the layout, readability and usability remain a concern, particularly when the size of system to be visualized becomes too large. Ivan H. *et al.* (Herman2000) mention that "large number of elements can compromise performance or even reach the limits of the view platform".

In our research, we have investigated some of these readability issues of visual representations in 3D space, and present improvements from three different aspects. First, we will introduce the use of Metaballs as a metaphor to visualize software systems rather than the more traditional representation of "nodes and arcs". Second, we will use hierarchic grouping of entities to abstract higher level entities and improve the usability. This will lead to an "overview first, zoom and filter, then details on demand" approach. Finally, we will address issues related to gird based 3D layout algorithms as some of the techniques to improve readability of the 3D visuals created.

CHAPTER III. THE METABALLS METAPHOR

3.1 Introduction to Metaballs modeling and visualization

Metaballs is an implicit surface based on a equation like $1/(x^2+y^2+z^2)=r$. The Metaballs visualization technique models particles in 3D space, which have energy (strength) and a well defined, parametrically controlled influence over the surrounding and neighboring particles (Rilling 2003a). The potential or energy function is commonly used to model the Metaballs iso-surface. Figure 18 is an analogy from physics (the blending of two light sources) that shows how an iso-surface is formed. Each candle has several circles representing different level of energy (illumination). When two candles are close to each other, the energies add up. Points in 3D space with the same value of energy a smooth surface.

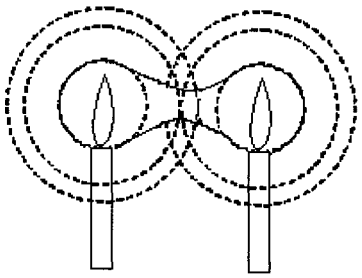


Figure 18. Iso-surface of equal temperature around two head source (Watt2000)

3.1.1. Implicit modeling vs. explicit modeling

The mathematical description of three-dimensional surfaces usually can be classified as parametric and implicit. An implicit surface is a set of all points which satisfy some equation $F(x, y, z) = 0$. Figure 19 is an implicit surface implemented by James (James 1982). The pixel coordinates can be calculated by first- and second-order polynomial functions (James 1982), or Marching cubes algorithm. On the other hand, an explicit surface is all points that satisfy equations with the following form.

$$x = f_1(t)$$
$$y = f_2(t)$$
$$z = f_3(t)$$

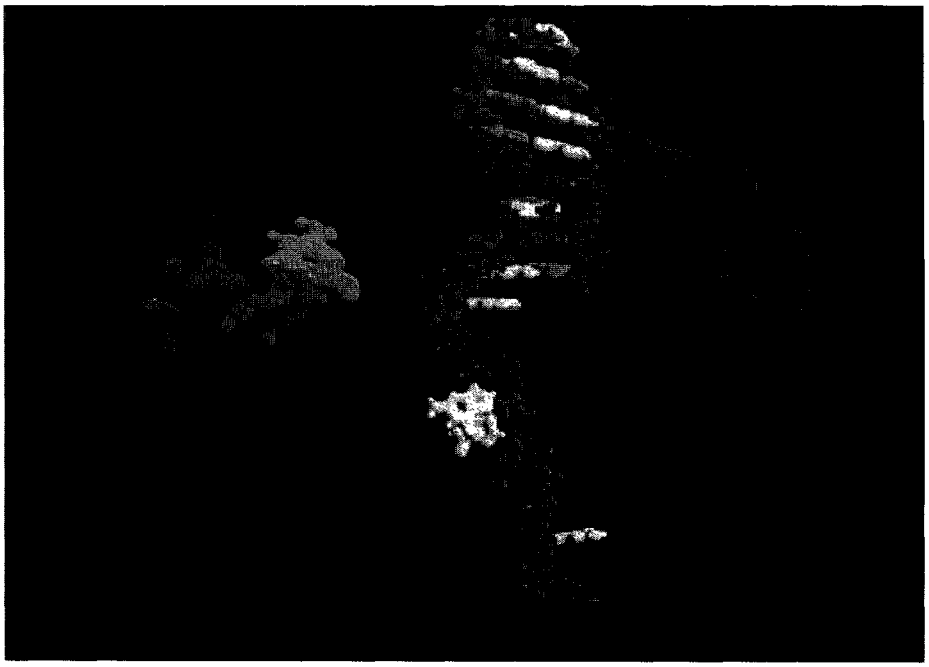


Figure 19. Implicit surface description. (James 1982)

3.1.2 Applications of visualization using Metaballs

People have used Metaballs to describe a surface that is not easily modeled by primitive objects such as lines, planes, and boxes. Figure 20 (top) shows an example of

using Metaballs to model organic forms like the human body and animal shapes. Figure 20 (bottom) also shows the use of the basic iso-surface rendering technique used to display algebraic surfaces. This was implemented by us as an applet to view the algebraic polynomial equation with form *like f(x,y,z)=0* in three-dimensional space.

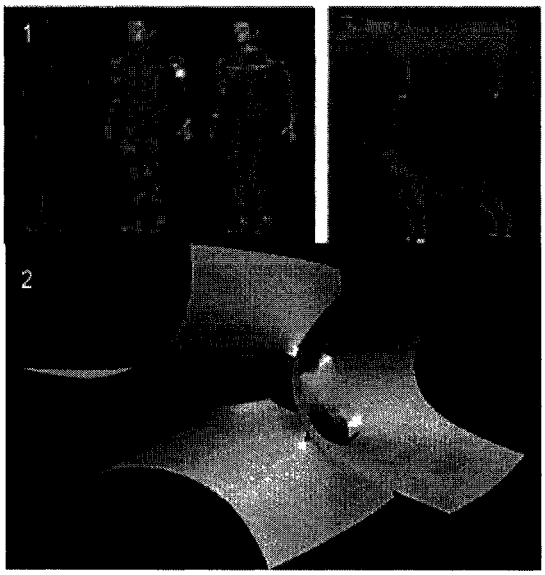


Figure 20. (1) Body modeling (Plankers2001) (2) View the formula for propeller

3.1.3. Metaballs and software visualization

Many applications use Metaballs to represent complex shapes and structural relationships. In (Rilling2002) we apply Metaballs as a metaphor for visualizing software structures. A mapping between software artifacts and Metaballs properties was introduced. The mapping is shown in Table 2. The software classes are directly mapped to Metaballs, and the couplings between classes are mapped to cylindrical links between the Metaballs. Furthermore, shading, color and size of Metaballs can be used to represent different properties of software.

Table 2. Mapping table

artifacts of software	characters of Metaballs
Class	Metaballs
Size of class (lines of code)	Diameter of Metaballs
Package	Color of Metaballs
Coupling between classes	Cylinder between the Metaballs
Strength of coupling	Diameter of cylinder
Hierarchical structure of classes	Cone tree like Metaballs clusters

3.2 Motivation for using Metaballs in software visualization

As discussed in section 2.4.4 Metaphor, investigating new metaphors is an important task in the ongoing research of 3D software visualization techniques. The success of using Metaballs in other areas stimulated us to apply Metaballs as a 3D software visualization technique. We developed our Metaviz tool as a testing platform to evaluate and explore the application of Metaballs in software visualization.

The Metaballs metaphor is a 3D object modeling and rendering technique that blends and transforms an assembly of particles with associated shapes into a more complex 3D shape whose use is highly suitable for animal and other organic forms.

Compared to the traditional node and arc representations, the Metaballs visualization technique has a surface representation that can be described by a mathematical formula. The surface corresponds to a collection of points where their function is equal to a threshold value, also known as the iso-surface.

Metaballs provide a three dimensional picture with smooth connection between Metaballs and shading, which could ease the difficulties of building mental model. Figure

21 illustrates the advantage of the Meatballs approach over a 3D sphere-line graph. Both visuals display the same information and use the same layout. One of problems of the sphere-line graph is that it cannot convey some structural information in the same way as, for example, the Metaballs. The fusion (the thickness of the connection) among two or several Metaballs can be used to show clearly structural dependencies. The fusion can also be used to indicate the relationship among different software artifacts. Shading and blending are other options that can be applied to convey additional information not available in most traditional software visualization techniques.

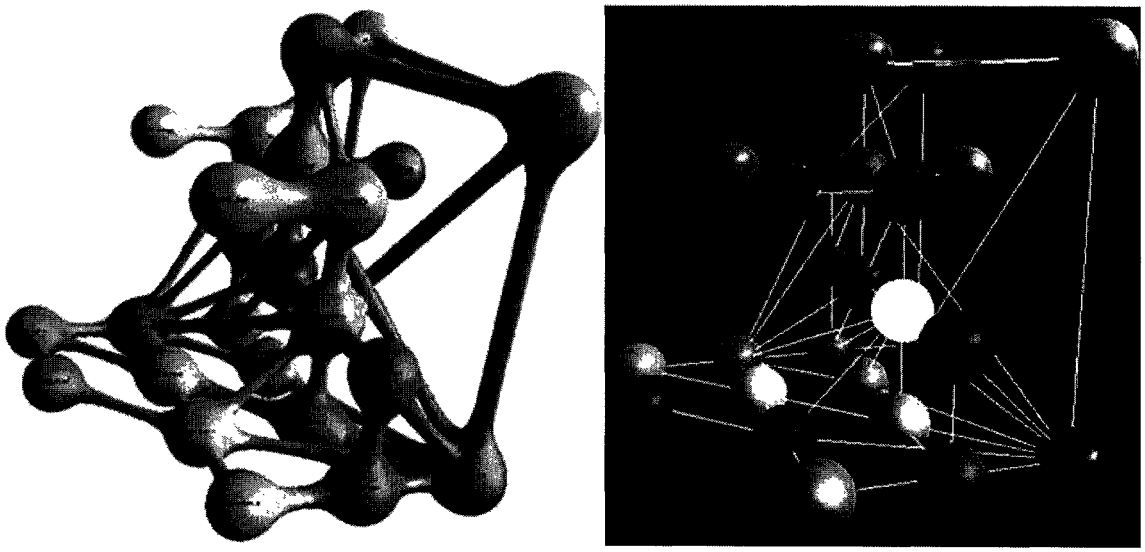


Figure 21. Metaballs vs. sphere-line graph

However, it should be mentioned that the Metaballs has two major disadvantages. (1) The algorithm for rendering Metaballs is slow; (2) triangulated representation of the isosuface makes texture mapping difficult. Section 3.1.2 Performance problem of the Marching Cubes and section 3.4 Text mapping for Metaballs, discuss how we address the rendering and texture problem within our Metaviz project.

3.3 Marching Cubes Algorithm

3.1.1 Introduction to Marching Cubes algorithm

The Marching Cubes algorithm, presented by Lorensen and Cline (1987), uses a divide-and-conquer approach to generate a triangle mesh that approximates the original surface (Lorensen1987). In Marching Cubes, the model is contained in a cube, which is divided into small cubes. Some of the small cubes would intersect with the surface. The algorithm reconstructs the surface from those small cubes. The size of small cube determines how close the reconstructed surface to the original one is. This process can be illustrated in Figure 22, taken from (Bourke1997).

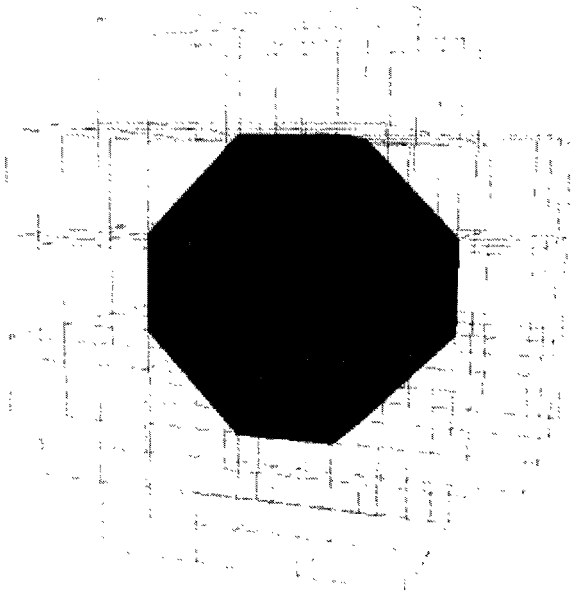


Figure 22. Cube division in Marching Cubes (Bourke1997)

The next step of Marching Cubes algorithm determines if each corner of voxel is inside the surface or outside. This can be done by taking the coordinators of the corner as

input of the function represented the surface, then comparing the output with threshold. Eight corners of voxel can form 256 combinations. Considering complementary and rotational symmetry, Lorensen gives 15 combinations, shown in Figure 23, in his original paper (Bourke1997).

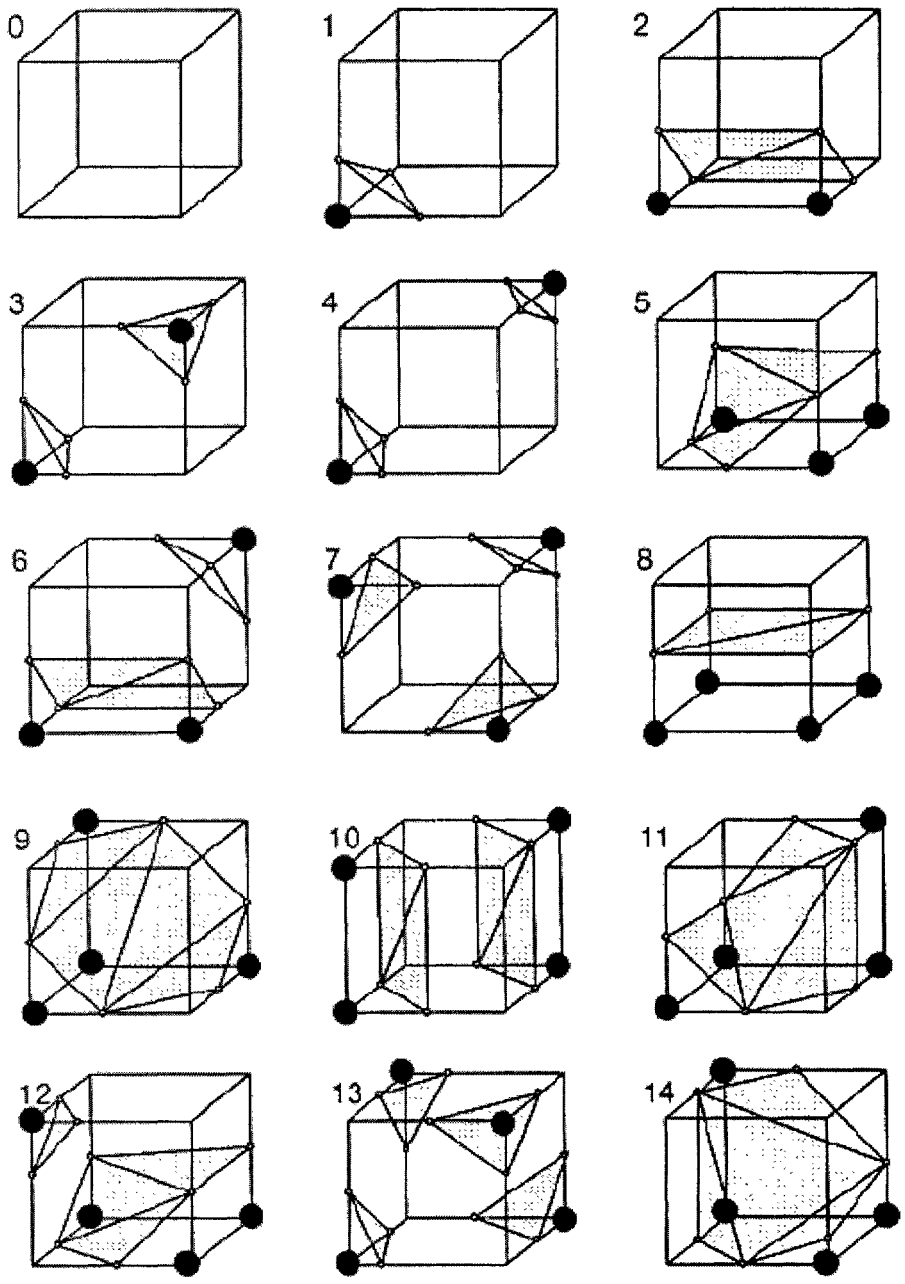


Figure 23. Triangle Cubes (Lorensen1987)

Shoeb finds ambiguity among the 15 combinations, and contributes eight more combinations to solve this problem, shown in Figure 24 (Shoeb1998).

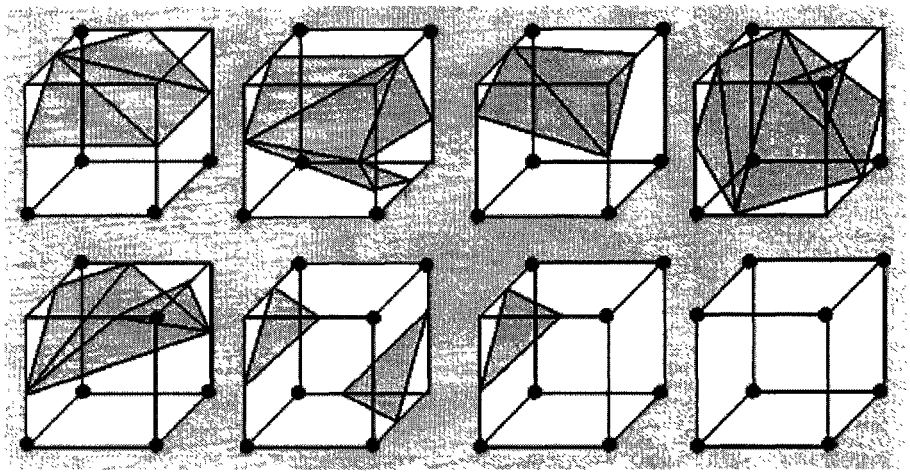


Figure 24. Extra Cubes Combination (Shoeb1998)

Cory Gene (Bourke1997) develops a set of tables, to access the triangles according to a coding bit pattern of eight corners. The bit pattern represents the corners inside or outside of the surface. Considering symmetry, these tables already contains same combinations given by Lorensen in 1987 and Shoeb in1998. The Metaballs package we have implemented is based on Gene's tables.

3.1.2 Performance problem of the Marching Cubes

The speed of rendering a graph is one of the main factors that affect usability of a software visualization technique and the tools implementing these techniques. Nobody wants to use a slow visualization tool no matter how good the algorithm is. Although the Marching Cubes algorithm is claimed as a fast approach for rendering implicit surfaces, it is still slower than parametric rendering, and therefore is less suitable for use in visualization tools. The performance problem in Metaballs rendering is directly related to

the computations carried out by the Marching Cube algorithm that is used to compute the iso-surface of the Metaballs. In the Marching Cubes algorithm, we have to compute the required information for every corner of the small cubes. Such a computation usually is very time consuming, which may includes computations of cube root or even more complex algebra functions.

3.1.3 Reducing unnecessary computation in the Marching Cubes Algorithm

We use a 2D grid graph shown in Figure 25 to simplify the illustration of the unnecessary computations in Marching Cubes algorithm. If we divide the cube recursively, after three iterations, we get 64 cells, as shown in Figure 25. Only 16 of the cells colored in grey are intersected with the surface coloring (shown by the red wiggly line). In order to render a surface of reasonable quality, one usually divides a cube about seven times, and we get $8^7 = 2,097,152$ smaller cubes. This leads to a very small proportion of the cells intersecting with the surface. Only those cells that contribute to parts of the surface need to be fully rendered; other cells that are located either totally inside the surface or outside the surface do not contribute to the surface construction.

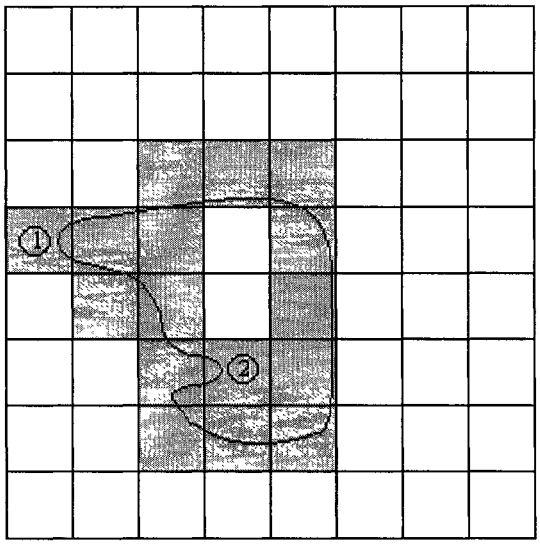


Figure 25. A 2D grid graph assembling Marching Cubes

3.1.4 Optimization

One approach to improve the performance of the algorithm is to optimize the computation complexity, by avoiding any further surface computations in the grid cells that do not contain any parts of the surface. Using recursive division, we check if the cube intersects with the surface. If the cube does not intersect with the surface, we stop any further division and computation for this cube

Figure 26 shows a computation without optimization. Each node in the tree indicates a length computation for a cube, and the complexity of the computation will increase in proportion to size of the tree. After the first division, we get four cubes represented by the second level of the tree. The third level of the tree represents the cubes generated by second division. The leaves of the tree represent 64 smallest cubes.

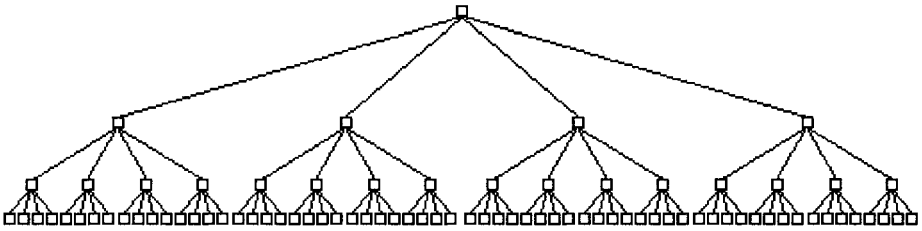


Figure 26. Computing tree before optimization

Figure 27 shows the pruned computing tree by eleminating unnesseceary division and computations. The level of optimization is influenced by several factors: firstly on the iso-surface and the number cells involved in it. Secondly, the number of divisions used to render the surface. .

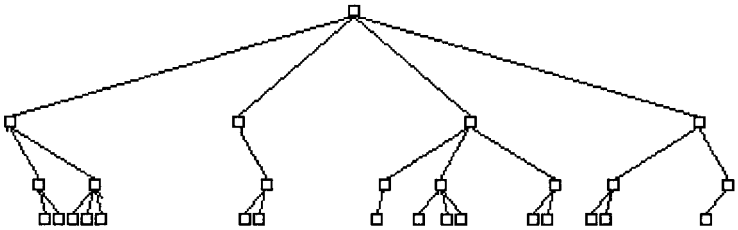


Figure 27. Computing tree after optimization

3.1.5 Cube object intersection

Our optimization of Marching Cubes only works if we are able to judge whether a cube intersects with the surface or not. If the computation of such a judgment costs more time than the computation for each cube, the optimization work fails. A naïve method to check the intersection is based on the eight corners of cube. If all eight corners are inside or outside the surface, it is likely that the cube does not intersect with the surface. However, this judgment does not work on the two circumstances marked ① and ② in Figure 25. All the corners of the grid ① are outside the surface, but the grid

contains part of the surface. By not computing the surface for this grid, will leave a hole in the final rendering surface. Similarly, a hole will occur for the grid ②, with all of its corners being inside the surface, but part of surface is still in the outside of the grid.

In one implementation, our Metaballs class has an abstract method to judge if the cube intersects with the surface. Any subclass with different formulas must implement its own methods that tests if a cube intersects with surface. We have implemented two concrete methods, the cube-sphere and cube-cylinder, which are both used within out Metaviz tool. We have to point out that the optimization does not work when such checking method is more expensive than that of computing corners of cubes. Also the effectiveness of this optimization will decrease when the proportion of cubes intersecting with the surface increases.

3.1.6 Experimental results of our optimization

Table 3 shows our test results for 25 Metaballs that were rendered with and without the presented Marching Cubes optimization. For the experiment, we limited the recursion level to 9. The recursion level directly corresponds to the rendering quality of the Metaballs. If the recursion level is less than 5, the Metaballs surface becomes too coarse (Bourke1997). For a recursion level larger than 9, the rendering time becomes too long to be of any practical use.

Table 3. Test results for the Marching Cubes algorithms optimization. (CPU: P4 2.8GHz, 1GB RAM)

Recursion levels	Non-optimized	Optimized
5	1,093 ms	157 ms
6	7,406 ms	422ms
7	55,469 ms	1,750ms
8	459,609 ms	9,031 ms
9	3,286,978 ms	58,343 ms

3.4 Text mapping for Metaballs

Labeling software entities improves the readability of the visual abstractions considerably and can be achieved by mapping textual labels (entity name) on the surface of the Metaballs. The Marching Cubes algorithm generates a large number of small triangles that are used to render the Metaball. The internal triangle representation makes the texture mapping for displaying text on the Metaballs a major challenge. We address this problem by using billboards and orientation objects to display labels on top of the Metaballs. This technique has another advantage compared to the more traditional texture mapping. When we navigate or reorient ourselves in the 3D space, the Metaballs labels always face toward the camera and therefore improve the readability of the text.

CHAPTER IV. LAYOUT AND CLUSTERING

In the previous chapter we addressed issues with respect to performance optimization of the rendering algorithm used to create the visuals and supporting text mapping for the Marching Cubes algorithm. Another major challenge of current visualization techniques is to make optimum use of the available display space. Layout is always a problem in software visualization, especially for large software. In the following section, we present our new layout algorithm. We also discuss clustering algorithms and how these algorithms can further improve the usability of the layout algorithms

4.1 Review of layout algorithms

Through the use of reverse engineering, one can automatically generate many different types of visual abstractions from the source code. Examples for these include: data flow diagrams, subroutine-call graphs, program-nesting trees, object-oriented class hierarchies, and entity-relationship diagrams (database). For these visuals to be effective and readable, good layout algorithms have to be an essential part of these tools. However, at the current state of the art, most of the current reverse engineering tools still lack support for a good layout algorithms that will improve the readability of the visuals.

The layout problem can, in general, be simply described as: "given a set of nodes (entities) with a set of edges (relations), calculate the position of the nodes and the curve to be drawn for each edge" (Herman2000), satisfying some given criteria.

4.1.1 2D layout algorithms

The few tools that support layout algorithms use rather simple 2D layout algorithms. The current research in 2D layout algorithms focuses on the use of topological graph theory, geometric graph theory, and order theory (Battista 1999). The planarization based on graph theory is applied to a drawing to reduce as much as possible its number of crossings (Jünger 1997). Another approach is to apply physical model to the layout. Sugiyama layout algorithm is such an approach (Sugiyama 1981). Sugiyama combines a force directed algorithm (Eades 1984) with a magnetic field model in his layout algorithm. Figure 28 demonstrates how the magnetic field model can be used to enhance the layout.

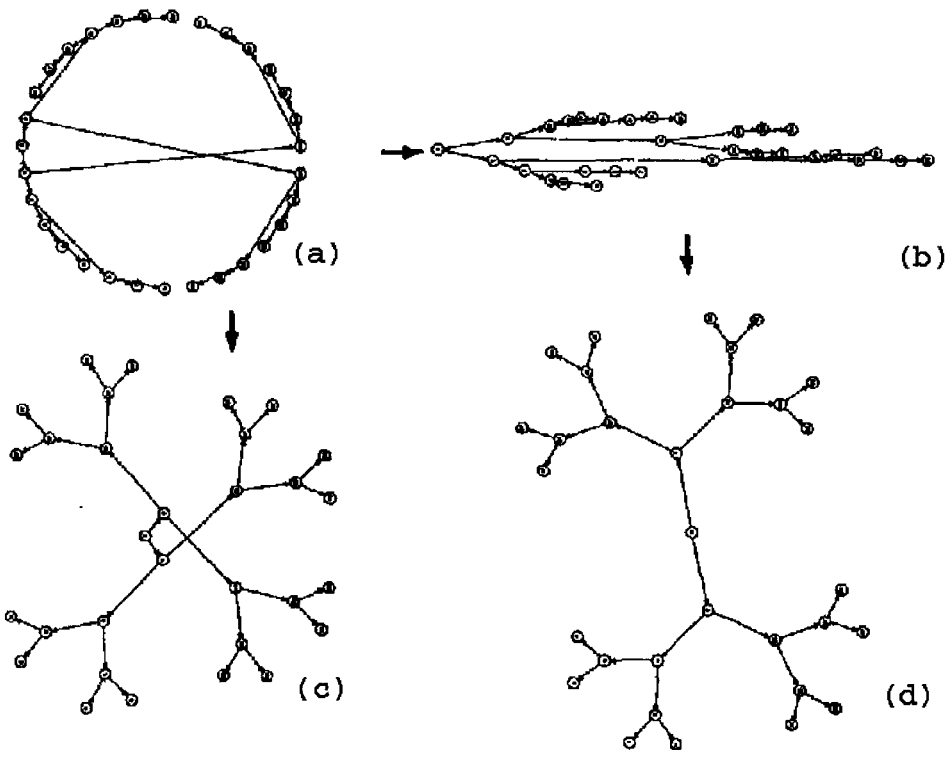


Figure 28. The process of magnetic spring algorithm (a) initial placement (b) layout in no field (c) layout in a strong field (d) layout after two phases

4.1.2 3D layout algorithms

Typically, the viewpoints of 2D drawings are rather limited, because certain rotations and different angles of the drawing might not be meaningful or useful for the user (e.g. viewing a 2D drawing from the bottom). In 3D visualization, however, we walk through the drawing by moving the viewpoint and the view angle. The graph theory driven 2D layout algorithms are no longer suitable for 3D layout algorithms. 3D layout algorithms have to adapt to the requirements of animation, navigation, and interaction. We will discuss two layout algorithms to illustrate the properties of 3D layout algorithms.

The force directed layout algorithm simulates a physical model, which consists of weights and forces, and tries to minimize the total energies in the model. The resulting layout usually exhibits a quite well symmetries and clusters in the network, but it may produce a lot of projection crossing. Therefore, a force-directed layout is suitable for a first-step processing to identify clusters that can help to reduce the number of crossings. Figure 29 is an illustration of force directed layout using VRML for display.

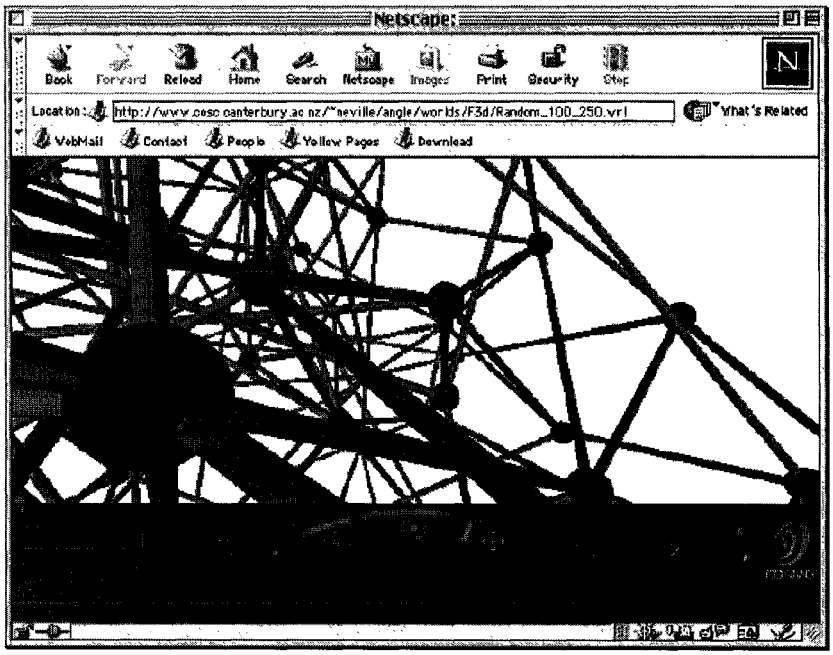


Figure 29. Exploring a virtual world containing a random graph with 100 nodes and 250 edges (Churcher 2001)

3D hyperbolic space is another significant 3D layout research by T. Munzner. By using hyperbolic geometry, one can allocate the same amount of room for each of the nodes in a tree while still avoiding collisions because hyperbolic geometry is one of the non-Euclidean geometries, and there is an exponential amount of room available in hyperbolic space, shown in Figure 30. "The H3 layout algorithm is linear in the number of spanning tree edges (Munzner 2000)". That means the H3 layout is fast enough for animation and interaction.

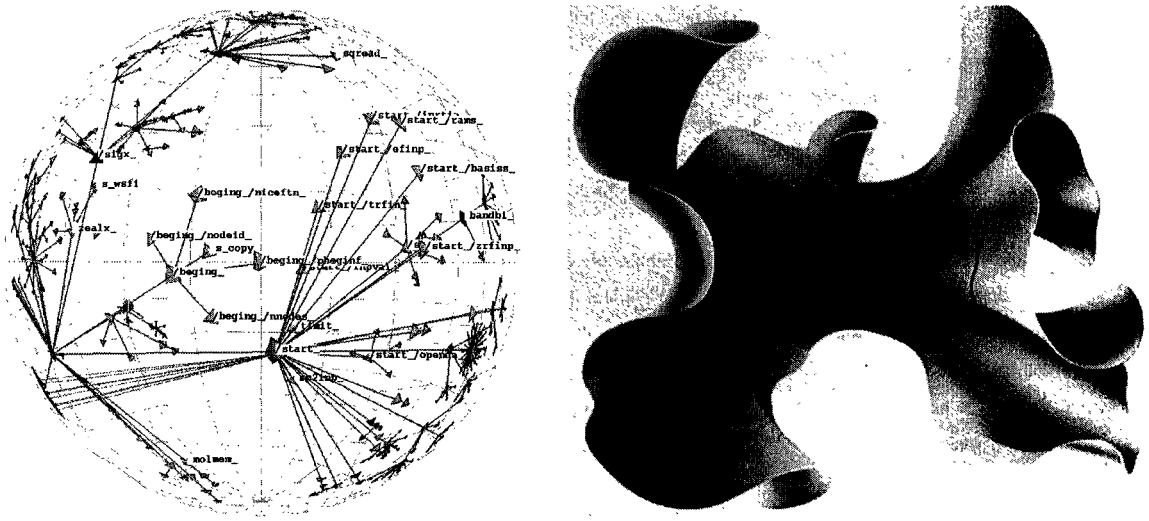


Figure 30. A call graph in 3D hyperbolic space (right); Exponential volume of hyperbolic space (left)

4.1.3 Layout algorithm constrained by program comprehension

Layout can be considered as a multiple-optimization problem. Therefore, any layout only preserves certain drawing properties, and ignores others. Different layouts may be needed to preserve prosperities related to a particular application domain. For

visualizing a large software, we can use zoom in and zoom out, but one problem with this is we lose the context. Focus+Context is another way to avoid such problems by using distortion-based graph drawings.

The SHRiMP (Storey 1997) graph viewer is based on the multiscale Pad++ system. The SHriMP fisheye view algorithm preserves the user's mental map through different strategies to adjust graph layouts.

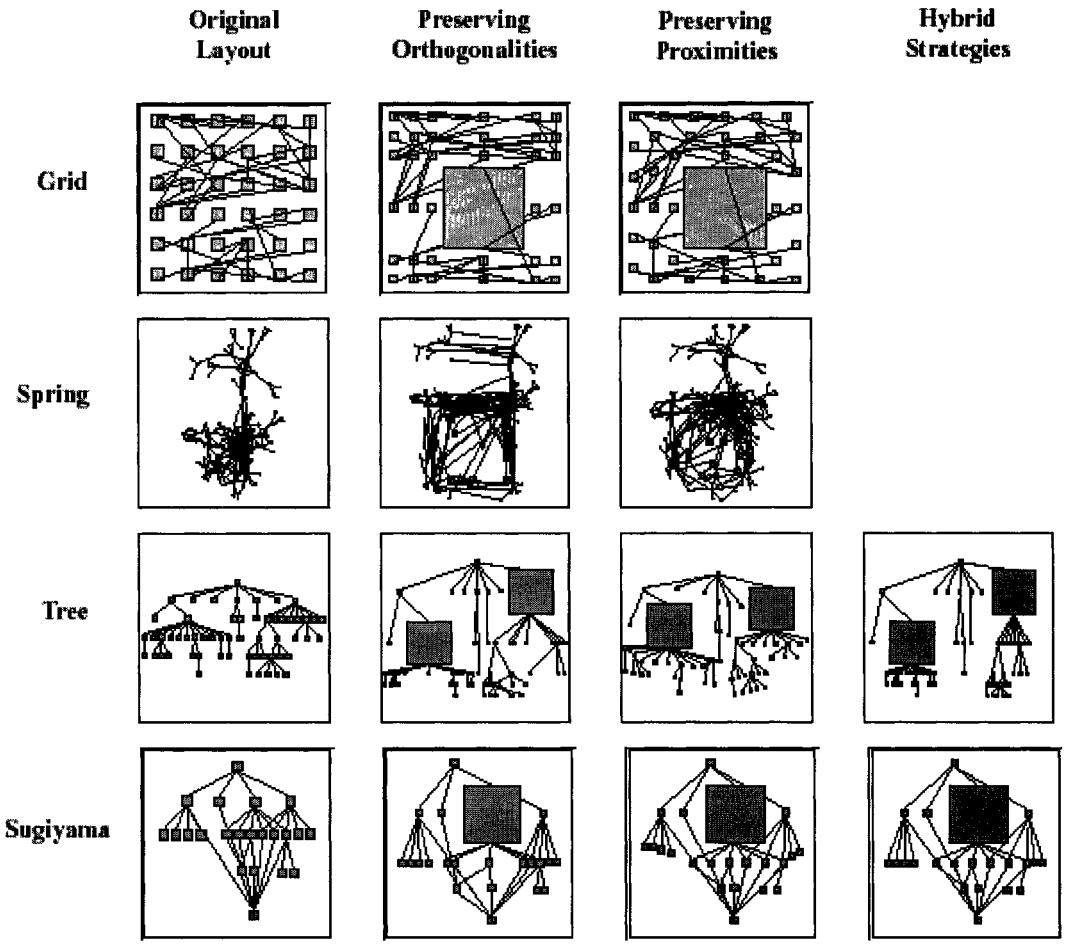


Figure 31. the SHriMP fisheye view algorithm has different strategies to adjust graph layouts while preserving the user's mental map.

4.2 The layout problem - motivation and objective

The graph layout problem is an ongoing research area within the information visualization domain for the last decades. One of the reasons is that many layout algorithms are not scalable. "Few systems can claim to deal effectively with thousands of nodes, although graphs with this order of magnitude appear in a wide variety of applications (Herman2000)." A layout algorithm working well for a small system could become completely useless when the size of the system increases.

Other research topics related to layout algorithms are their time complexity. Table 4 lists the complexity of some 3D layout algorithms. Many layout algorithms may take from minutes to hours for computing an optimized layout. The force-direct layout algorithm and the 3D grid layout, introduced in this section, have a similar magnitude of time complexity. However, for an interactive software visualization tool to be applicable, it needs to have fast response time.

Table 4. The complexity of some 3D layout algorithms

Layout algorithm	H3 layout	Sugiyama layout	Force directed layout
Complexity	$O(n)$	$O(n^3)$	$O(n^3)$

4.3 3D grid layout algorithms

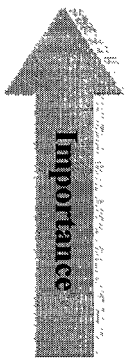
For the Metaviz tool, we use a grid layout approach, where the position of each node corresponds to some integer coordinates. This enables us to reduce the number of possible configurations to a finite number. It also allows for a space efficient

visualization of Metaballs. For example, 1000 entities can be placed within a 10*10*10 grid using the Metaballs visualization. On the other hand, displaying the same 1000 entities using a cone tree, the size of each entity becomes too small at the lower levels of the tree, to be readable or useful to the user.

The layout algorithm is reusable. It can be applied for other visualization approaches that face similar problems with respect to layout and readability. We have therefore developed the layout programs as a separate Java package plug-in that can be reused by other visualization approaches within the CONCEPT project (e.g. UML diagrams, 3D worlds, etc.).

4.3.1 Readability criteria

Since one of the major goals of software visualization is to guide people during the comprehension of software systems, we use readability as the major criterion to evaluate the quality of our layout algorithm. Similar readability criterion has already applied in information visualization can be applied for software visualization. Figure 32 shows the importance of the different criteria for the readability of a visualization technique in general. In our implementation, we take into account many of these criteria, but not all. The objective function is a weighted sum of these numbers with line object crossing having larger weight, projection line crossings less, and length of arcs being least.



Minimize the number of edge crossings

Minimize the "projection crossings"

Minimize the length of edges.

Optimize density distribution

Optimize drawing space (area) Achieved symmetry.

Figure 32. Visualization criteria

Drawing space and density distribution are optimized by choosing the grid size to be the minimum that can accommodate the given number of Metaballs.

4.3.2 3D grid layout as state searching problem

A node at position (x, y, z) with side s occupies the cubical region $grid(v)$ defined by the two grid points (x, y, z) and $(x + s, y + s, z + s)$. An edge $e = (v, w)$ which is represented by a cylinder $cylinder(e, d)$ defined by the point $p1$ at the beginning if $p_v \in grid(v)$, and point p_w at the end if $p_w \in rect(w)$, where p_v and p_w are the positions of node v and w, respectively.

The 3D grid layout problem can be described as follows: How can we place m entities into n^3 positions to satisfy certain readability constrains, where $m <= n^3$.

The computationally intensive nature of 3D layouts has already been shown in some of the existing layout algorithms in other application domains (Munzner 1997; Churcher 2002). For example, the layout of integrated circuits could take several hours on high performance computers. Placing m entities into n^3 positions involves $n^3!/(n^3 - m)!$ cases. In order to use space efficiently, m should be very close to n^3; therefore, the

complexity is almost $O(n^3!)$. To find the best possible layout based on some given constraints one would have to evaluate each possible combination within the given search space. Table 5 illustrates the complexity of the 3D layout problem, depending on n and m. Rather then evaluating all possible $n^3!/(n^3 - m)!$ combinations, one will have to limit the search space that will provide an acceptable layout within given time and space constraints.

Table 5. Complexity of the estimation of 3D layout

The number of entities: m	The number of cells in grid: n^3	The number of cases to evaluate
4	8	1680
8	8	40320
13	27	$2.273*10^{20}$
27	27	$1.089*10^{28}$
32	64	$4.822*10^{53}$

4.3.3 Building up a searching tree

One way to solve the problem is to use a heuristic search for the 3D layout. Building up a search tree is the first task in solving a searching problem.

The search tree starts from a root state and moves on to its children. In the case of a root state, the algorithm randomly places m entities within the given n^3 space. The algorithm swaps some of the entity positions and creates new states for the children. The number of children corresponds to the "branch factor", which directly influences the

complexity of searching algorithm. We calculate the branch factor for certain operators. The result is shown in Table 6.

Table 6. The branch factors of search trees for placing 27 entities into 27 positions

Operator	n entities into n positions	Branch factor
Switching two entities	(n-1)*n/2	351
Switching three entities	(n-1)*...*(n-4)	15,600
Switching four entities	(n-1)*...*(n-5)	$3.6*10^5$
Switching five entities	(n-1)*...*(n-6)	$7.9*10^6$

4.3.4 Using greedy search for grid layout

We used a greedy search as the first searching method for the grid layout, and it failed practically for any application with more than 27 nodes, because the recursive function calls used in the greedy search exhaust the memory of any current computer. This observation provides a good indication about the computation complexity involved in the computation of grid layout.

4.3.5 Using hill climbing for grid layout

The hill-climbing algorithm is a state searching algorithm, with the goal to minimize the memory requirements required to perform the search. A detailed analysis of existing searching algorithms that are studied extensively in the field of Artificial Intelligence can be found in (Russell 1995). The limitation of the hill climbing algorithm is that it can only find a local peak. In our research, we try to overcome this limitation to

a reasonable extent by modifying the hill-climbing algorithm. The modified algorithm will not always be able to find the best state (peak), but often a better state than compared to the traditional hill-climbing algorithm. Additionally, the modified hill-climbing algorithm finds a predefined search goal in less time.

In strategy 1, we compare all children with each other, to identify the child with the best estimation value.

```
CurrentSate = startState;
while(not exceed maximum time){
  BestChild = CurrentState.getBestChild();
  if( BestChild == null)
    Return CurrentState;
  else
    CurrentState = bestChild; }
return CurrentState;
```

Strategy 2 is based on strategy 1, except in strategy 2 we only compare the children with each other until we identify a child that meets the expected estimation value.

```
CurrentSate = startState;
While(not exceed maximum time){
  BetterChild = CurrentState.getBetterChild();
```

```
If (BetterChild == null)
  Return CurrentState;
Else
  CurrentState = BetterChild;}
return CurrentState;
```

4.3.6 Competition hill climbing

In both strategies, once we reach a local peak, the search is complete. In our extended version called *the competition hill-climber*, we modify the hill-climbing algorithm by using a more expensive comparison to break away from the local peak and jump to another hill which has a higher peak than the current hill. For this, we apply a random positioning of the entities into the grid and use these as random starting states. These random starting states will then lead to different peaks. In our implementation, we use ten threads to evaluate ten different starting states using one of the above strategies. Finally, we compare the ten computed peaks and choose peak with the best result. In our example (see Table 7), thread number 7 has the best estimation value.

Table 7. Test result of competition hill-climber (Test condition: P4 2.8GHz, 1 GB RAM, strategy 2 used)

Thread	Projection crossings	Line object cross	Lines length	Objective Function value	Compute time(sec)
0	22	0	125	279	1284
1	10	1	130	221	1134
2	24	1	149	338	953

3	33	0	126	357	1161
4	20	0	126	266	1230
5	17	1	123	263	1353
6	18	0	124	250	1348
7	12	0	114	198	1325
8	16	0	129	241	968
9	11	1	115	213	1329

4.4 Clustering algorithms

We would like to state clearly that layout algorithms on their own often do not provide enough insights and details to be useful for comprehension tasks. The layout algorithms are constrained by the amount of information to be displayed and the limited screen space. Even, if one manages to create a layout, the resulting visual might have far too much information, causing an information overload. Furthermore, layout algorithms are in general only concerned with minimizing the crossings among entities. However, it has to be noted that minimizing the crossings, might not necessarily correspond to the logical view, a programmer or a designer of the system had during its development. Therefore, limiting the number of entities to be displayed and their logical organization to match the user's mental model of the system is one of the key challenges in software visualization. For the visualization of large software systems, it is essential to provide some type of grouping to create a decomposition of the system. The grouping can improve readability (Mancorids1999), by supporting a representation that is closely related to the mental model a programmer forms of a system (Dwyer 2001). The

grouping should map closely to the mental model users form when performing a certain task.

Grouping can be applied to generate suitable abstraction levels and therefore allow for a reduction of the amount of information to be displayed on the screen. Ideally, grouping will also map closely to the mental model a programmer forms during a maintenance task. For example, in the case of the cone-tree and information-cube, entities are grouped into nested semi-transparent containers, while soft-city encapsulates detailed information into buildings. In this thesis, we present two methods for grouping software entities: metric-based grouping and feature-based grouping.

4.4.1. Metric-based grouping.

For the metric-based grouping, we create an internal relation table that is used to access the coupling among different classes. The number of function calls defines the weight of relationships among the different entities (coupling). Our intention is to create a nested graph based on grouping. As the first step, we use a small threshold of coupling to group the top level of graph. In Figure 33, three groups are created in the first step. For a large number of entities, we can further group each sub-node in the next level. A proper threshold should be chosen, so that on each level of the nested graph, the maximum number of entities displayed on the screen will not cause information overload.

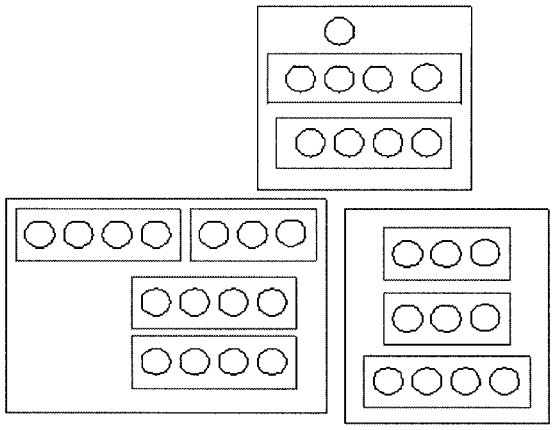


Figure 33. Metric-based grouping

4.4.2 Feature-based grouping

There exist several techniques for identifying features in software systems, e.g. (program slicing (Rilling 2003b), concept analysis (Koschke1999), etc.). For some applications, such as testing and debugging, programmers might be interested in focusing on these features instead of the whole software. Hence, grouping software entities based on their features can help programmers concentrate on the related parts of software and reduce unnecessary work.

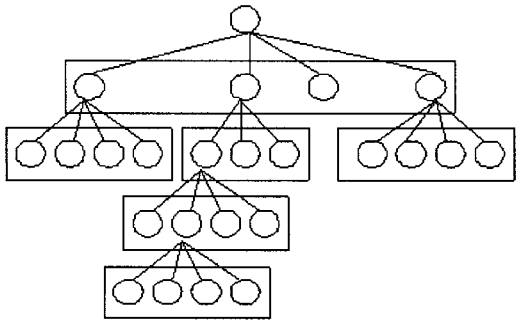


Figure 34. Feature-based grouping

Figure 34 is an example of such a feature-based grouping. A program might be represented as a hierarchical structure, with a feature consisting of several other features.

CHAPTER V. DESIGN AND IMPLEMENTATION OF METAVIZ

5.1 Motivation for using Java 3D platform

Java 3D is a Java API package developed by Sun (Sun 2002). Java 3D is a powerful and high-level package, which reduces development time by providing high level functions that allow programmers to focus on software visualization issues, rather than 3D display implementation issues. In this section, we discuss the advantages of Java 3D by introducing its architecture, and compare it with other graphic APIs.

5.1.1 Introduction to Java 3D

Java 3D is built on top of OpenGL or Direct X graphical engines, and provides a platform independent environment. The architecture of Java 3D is similar to the one used in the Virtual Reality Model Language (VRML). All components for rendering form a tree, called the Virtual Universe. The properties of each node define the appearance of scene graph and its behaviors.

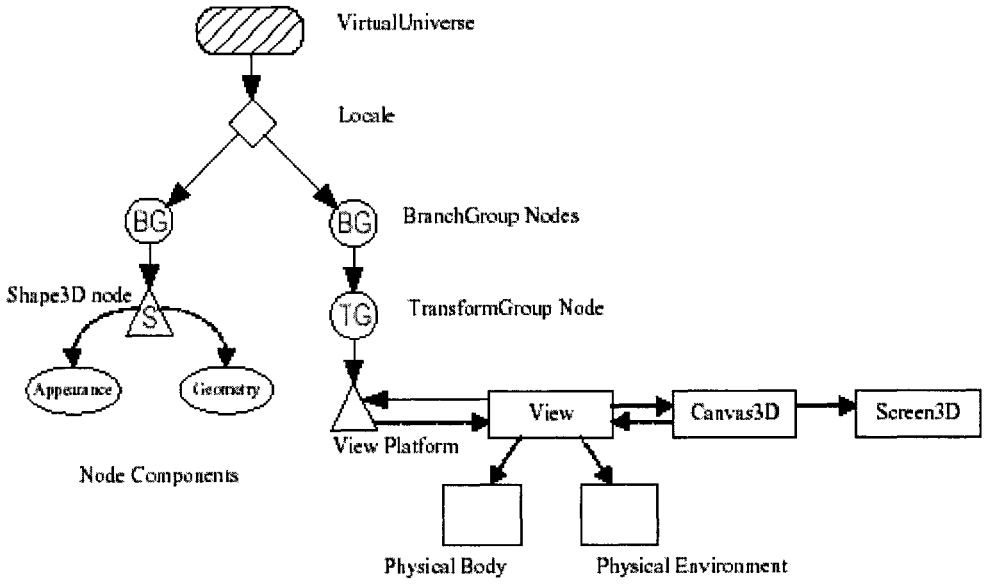


Figure 35. Java 3D architecture (Sun2002)

5.1.2 Java 3D vs. other 3D graphic API

Both Java 3D and VRML (Carey 1997) are easy to learn and to program with; however, Java 3D is more programming oriented, compared to VRML which is a scripting language and based on data structures. Java 3D allows to write dynamic and interactive programs; VRML is more suited to support the display of static models with limited interactions.

OpenGL (Segal 2001) is an industrial standard for computer graphics. It is written in C, and relies on call back functions and global variables for communication among functions. OpenGL is not Object Oriented. These features make OpenGL difficult to be adapted to MFC or OWL. On the other hand, Java 3D encapsulates all global variables in its Virtual Universe, and supports OO programming. Programmers construct all components they need, and insert them into the tree with the root of Virtual Universe.

The Java Virtual Machine will optimize the tree for rendering. Thus, programmers can enjoy most GUI components that Java provides.

Where there are many advantages of Java 3D, we have to mentions its shortcomings as well. Firstly, the Java 3D API is not a standard Java package; computers without Java 3D installation cannot run Java 3D programs. This limits Java 3D usage on the Internet. Secondly, Sun does not take responsibility of using Java 3D compiler. This feature limits the use of Java 3D in critical environments, such as airports or nuclear power stations. The last drawback of Java 3D is incompatibility with Swing.

In term of its high level functionalities, it is one of the best graphic APIs for research in software visualization.

5.2 Architecture of Metaviz

The Metaviz tool consists of three major parts: a clustering and grouping algorithm, the grid-layout, and the Metaballs rendering engine (see Figure 36).

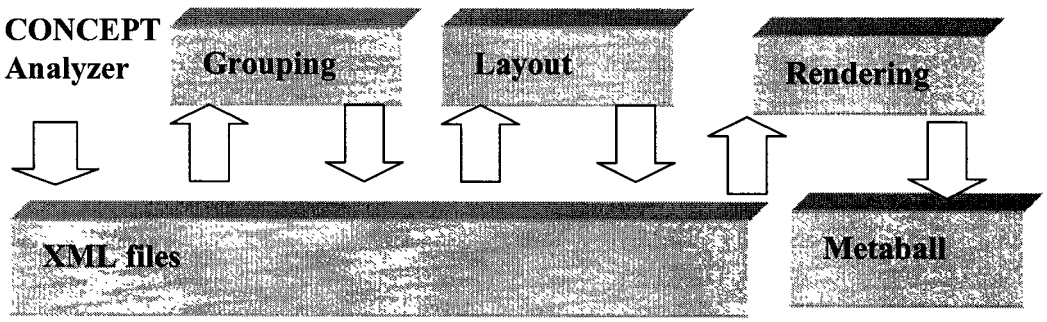


Figure 36. Metaviz pipeline

The grouping part preprocesses the data that CONCEPT Analyzer provides, and makes Metaviz scalable. If the data volume is too large to be visualized on one screen, we have to apply clustering techniques to split the data into groups with reasonable size that

will neither exceed the limit of a computer screen, nor cause information overloading. The clustering support provided by the CONCEPT framework will enhance the scalability of Metaviz, and allows Metaviz to visualize large software systems. The candidates for clustering algorithms have been discussed in section 4.4 Clustering algorithms. The clustering algorithm to be applied depends on the application domain. We have not implemented any clustering algorithm in this research. Instead, we provide this framework that can import and plug-in a third part clustering algorithm, or any clustering algorithm provided by the CONCEPT project, to allow for a testing in the context of large systems.

The grid-layout part uses an XML file as an input, which describes the software artifacts and their internal relationships. Within the Metaviz tool, users can select a layout algorithm from two available strategies. These different algorithms will be discussed more in detail in section 4.1 Review of layout algorithms. The Metaviz tool provides users with continuous feedback about the progress made in the layout optimization by displaying snapshots of the current layout. Once a certain optimum is reached, the layout is complete. The resulting layout will be saved in an XML file to allow further processing.

The rendering engine of the Metaviz tool reads the XML file as an input to generate and render the Metaballs in the 3D space. The rendering engines maps the properties of the software entities to the Metaballs properties. After the completion of the rendering process, the users can navigate through the visuals and apply "an overview, select and zoom" approach to refine the current view (Rilling 2001).

5.3 Integration within the CONCEPT project

For software visualization, Metaviz input the data from CONCEPT Analyzer. We define an XML file as an interface to the CONCEPT project. The XML file contains both information about the software entity properties and internal relationships. Since the XML file is self-explanatory and easy to expand, this interface provides a layer that separates the software analysis concerns from the software visualization concerns.

Besides its software visualization tool interface, Metaviz also provides a programming interface at the component level. All the three components, grouping, layout, and rendering, use XML files as input, and output the results again in XML files (except the output of the rendering engine). For that reason, these components can be reused in other programs. For example, Xiao Hua (Xiaohua 2003) in her thesis project, is applying the grid layout in her visualization tool.

5.4 Experimental results

As part of this research we performed several experiments to explore the capabilities and limitations of Metaviz and its components. In what follows we discuss some of the experimental results as well as the different application domains for the Metaviz tool.

5.4.1 Application example for Metaviz

In this section, we demonstrate how Metaviz can be applied for program comprehension. For illustration purposes, we use the Metaviz program itself as the software to be comprehended. Metaviz consists of 64 classes with a total of approximate

10,000 LOC. The following illustrations will demonstrate how Metaballs in combination with different source code analysis techniques can be used to guide programmers during program comprehension. The examples include: a hierarchical representation, a feature grouping based on program slicing, the visualization of coupling among different software entities and the animation of the layout algorithms.

5.4.2 Hierarchical structure of software

Typically, any larger software system is organized in a hierarchical structure and many software visualization tools are developed for visualizing these hierarchies (e.g., tree structures such as cone-tree, cam-tree, and information-cube). Within the Metaviz tool, we can also visualize such hierarchical structures. Figure 37 shows the structure of our Metaviz tool in a textual form as presented currently in most IDE (left) and the same information represented using our Metaballs approach (right).

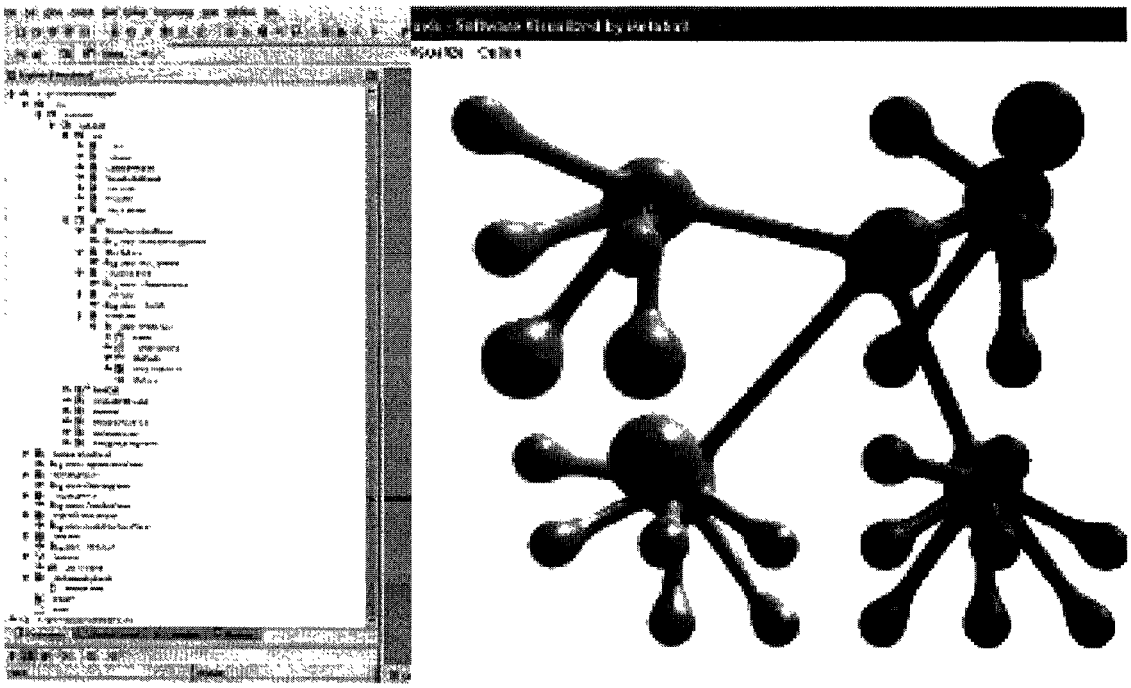


Figure 37. Hierarchical structure in SunONE (left); same structure using Metaviz's inheritance network (right)

5.4.3 Applying the Metaballs metaphor to visualize coupling measurements

Visualizing the internal relationships of software is an essential part of many software visualization tools. For this example, we use the coupling between object classes (CBO) as the weight of coupling among the software artifacts. Figure 38 shows the class coupling within our Metaviz tool. The diameter of the cylinders connecting two Metaballs maps directly to the coupling among these two entities.

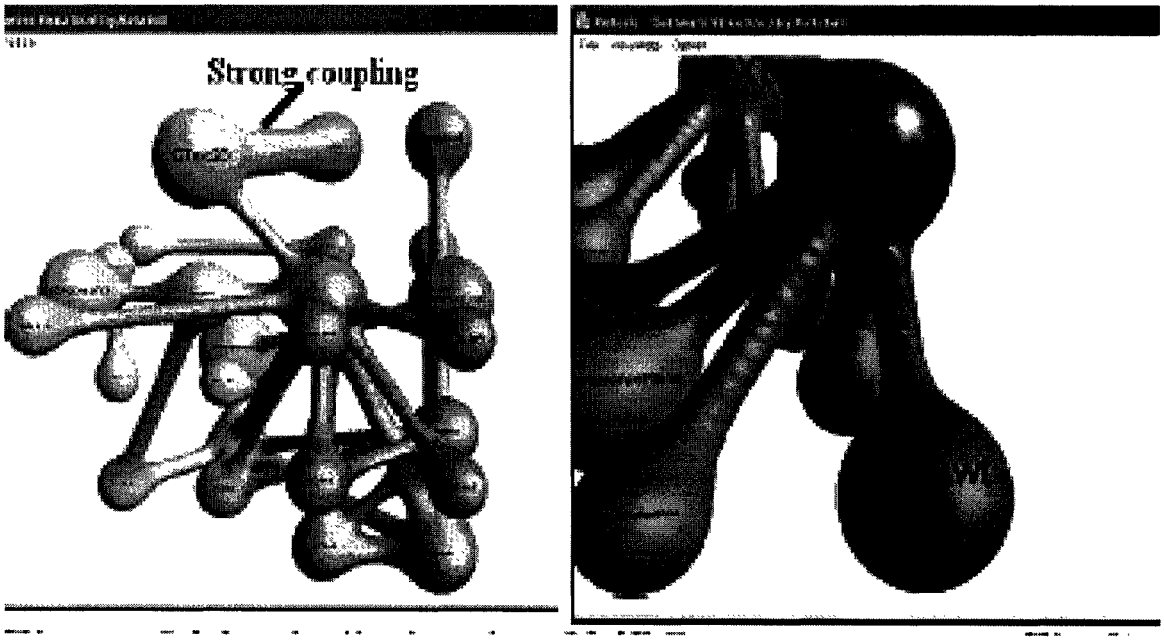


Figure 38. Metaballs visuals of MPC measurements

5.4.4 The process of layout management

Within the Metaviz tool we provide an optimized hill-climbing layout algorithm, the option to visualize the process of layout computation and optimization. The following five snapshots (Figure 39 to Figure 43) are from our layout computation process. Such a process can provide users instant visual feedback of the current layout optimization progress and provides them with the ability to terminate the layout optimization once it meets the users' expectations.

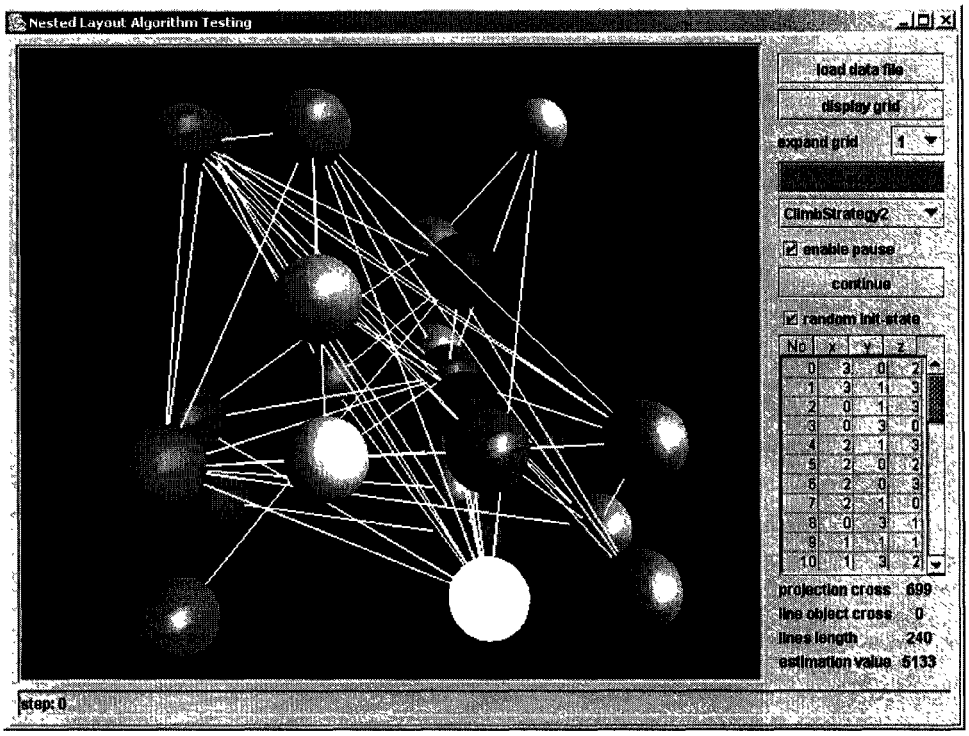


Figure 39. 1st iteration in layout process

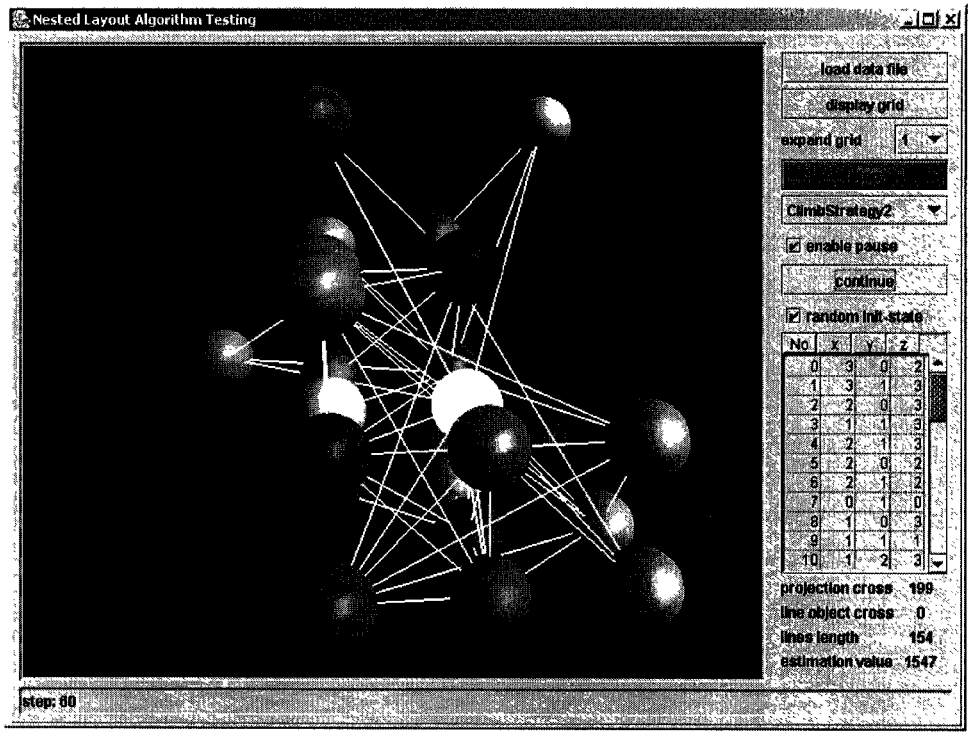


Figure 40. 60th iteration in layout process

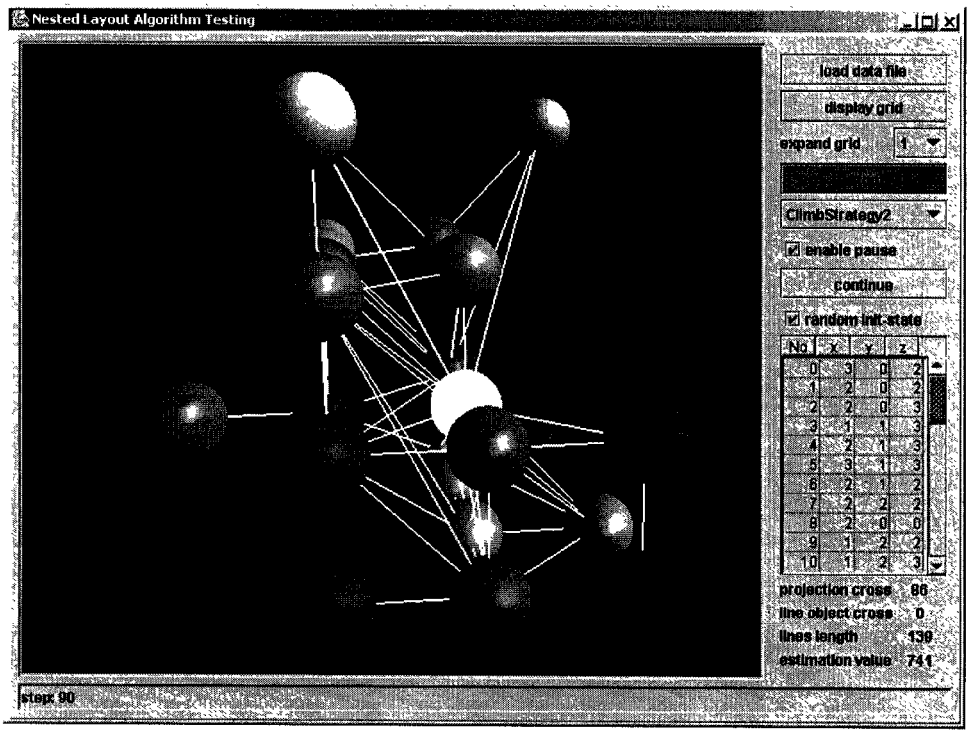


Figure 41. 90th iteration in layout process

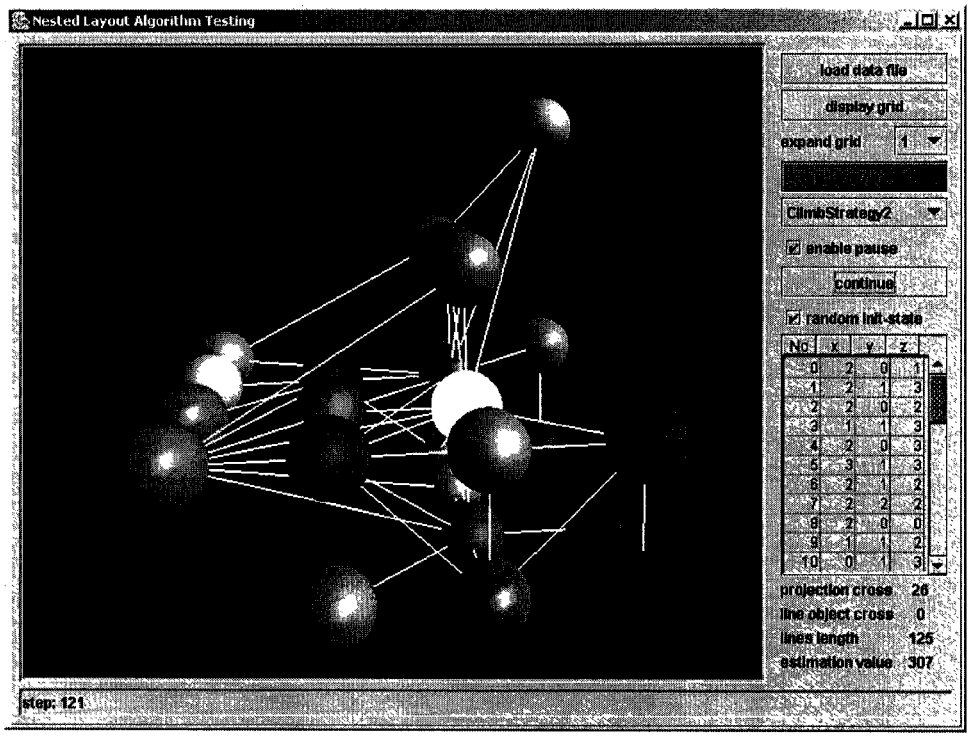


Figure 42. 121st iteration in layout process

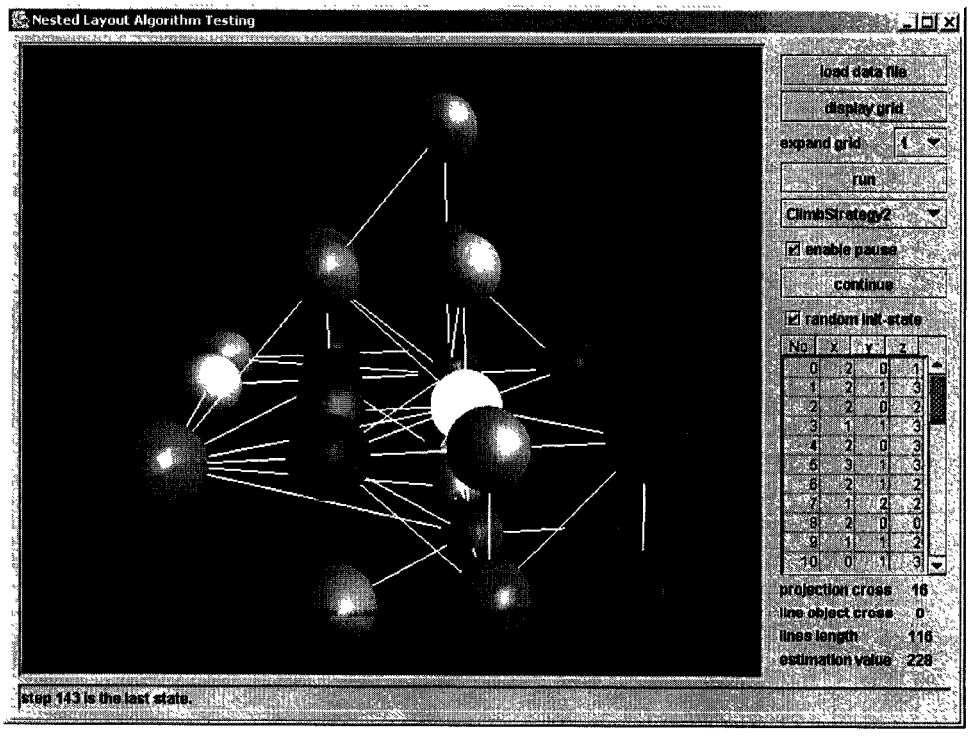


Figure 43. 143rd iteration in layout process

5.4.5 Dynamic software visualization

In the CONCEPT project, an execution trace of a Java program can be recorded. Metaviz extracts information from the execution trace and provides the ability to animate the program executions using the Metaballs metaphor. In Figure 44 to Figure 49, each ball represents a class. The white ball is current execution point. Function-calls from one class to another class are represented by a cylinder with an arrow. The arrow indicates calling direction, while the diameter of the cylinder represents the number of calls.

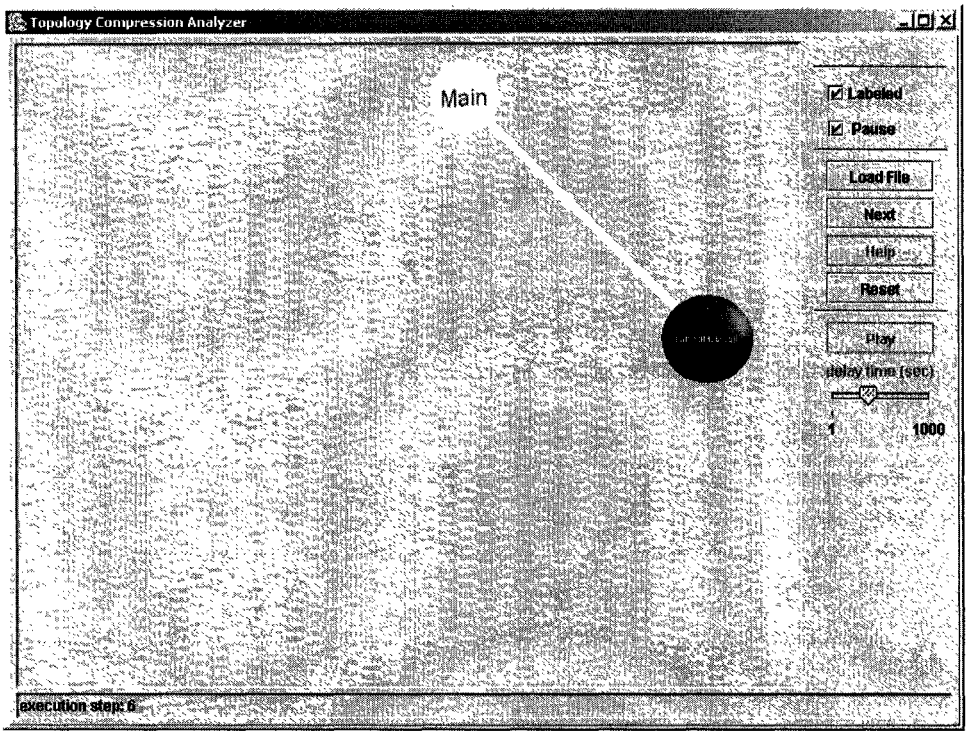


Figure 44. Dynamic visualization in step 6

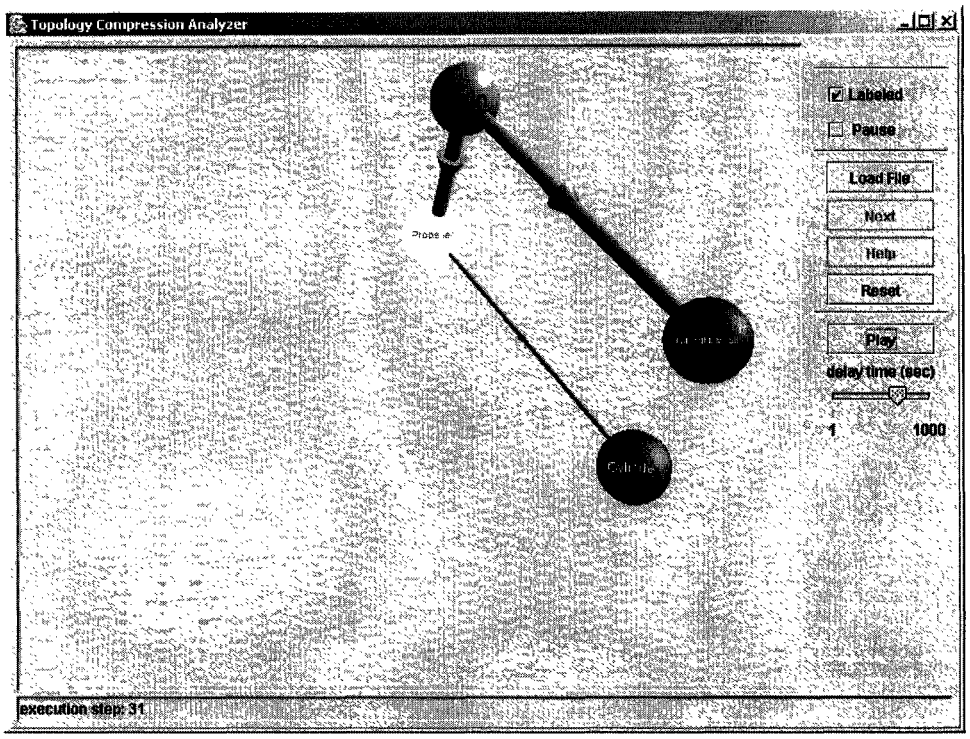


Figure 45. Dynamic visualization in step 31

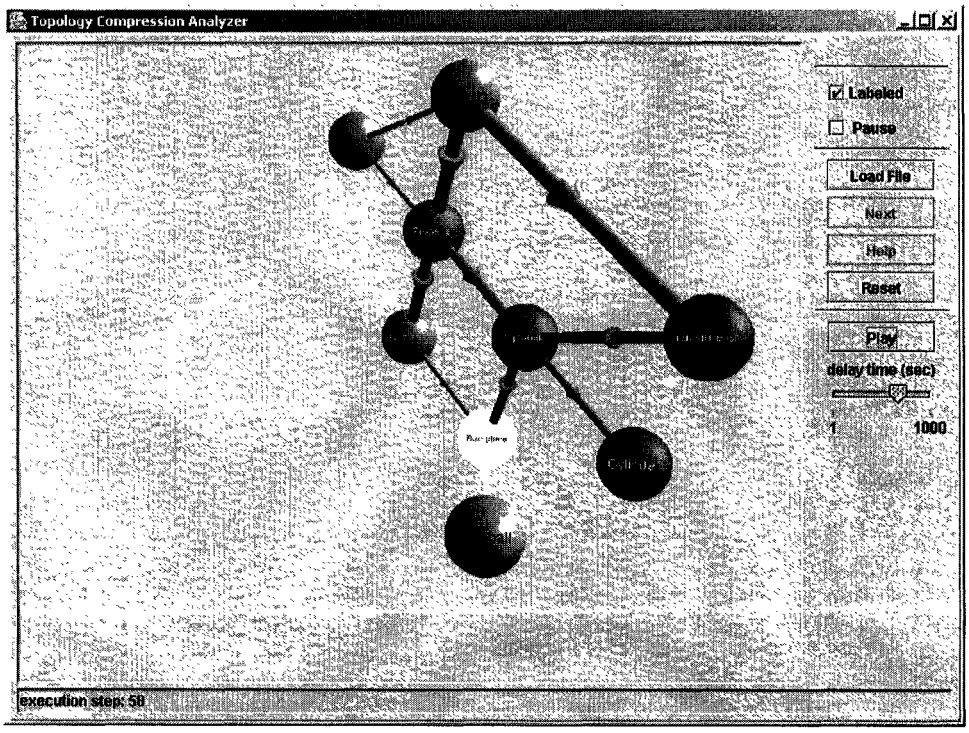


Figure 46. Dynamic visualization in step 50

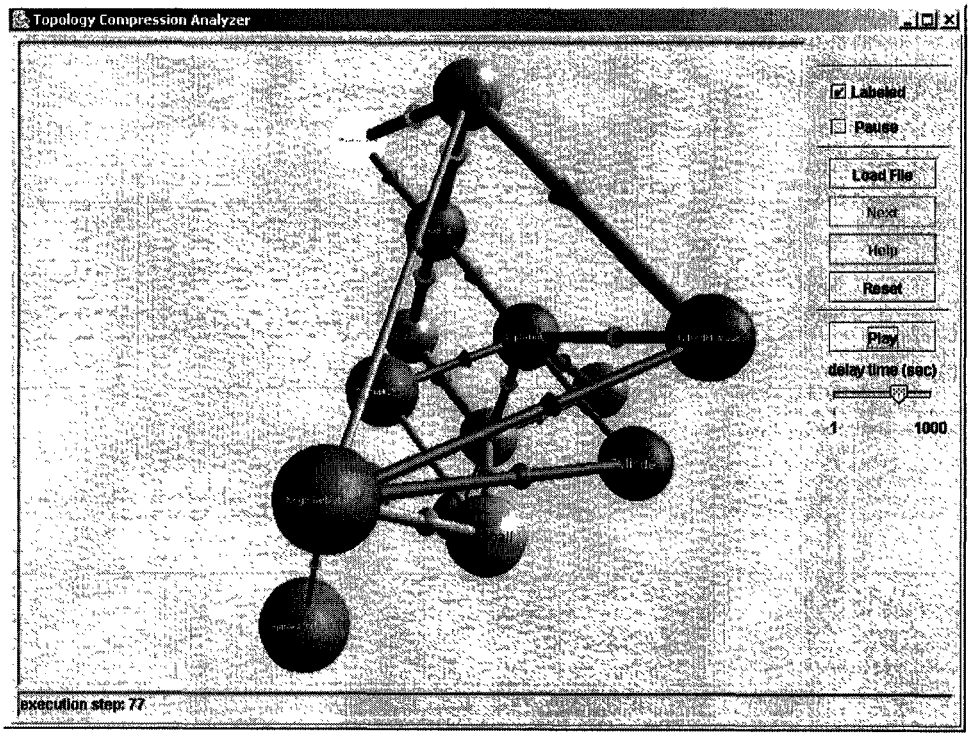


Figure 47. Dynamic visualization in step 77

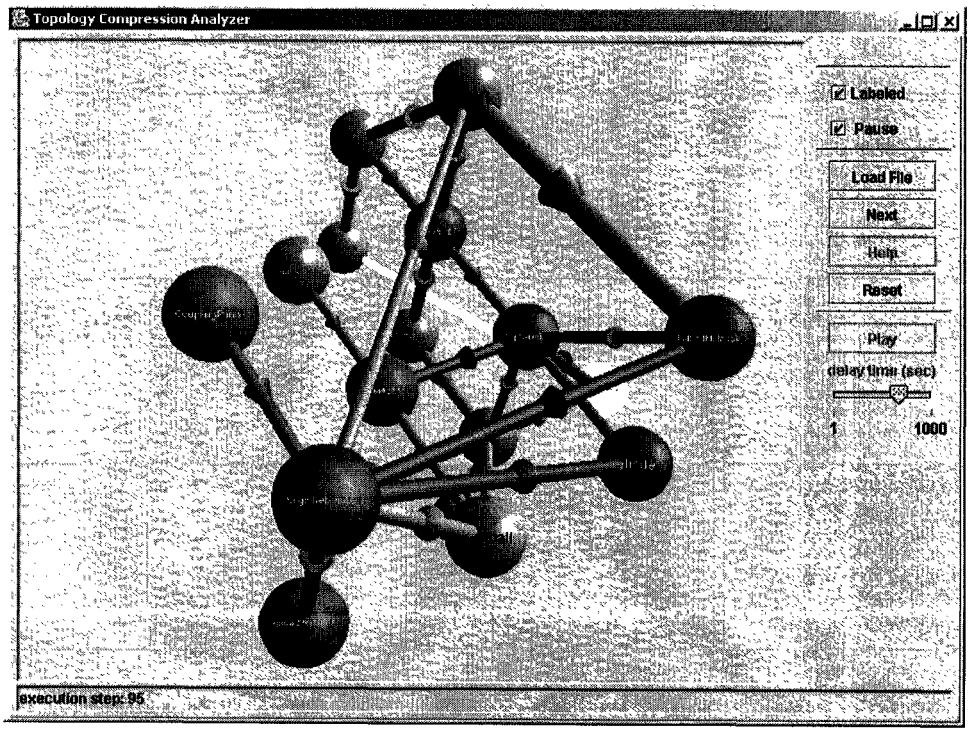


Figure 48. Dynamic visualization in step 95

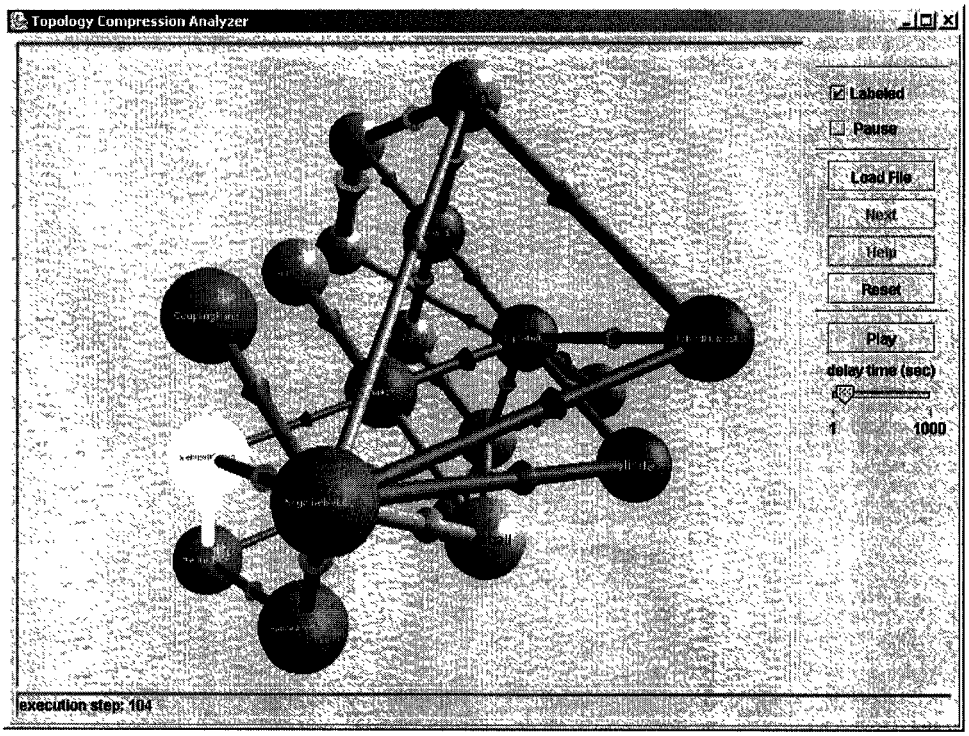


Figure 49. Dynamic visualization in step 104

5.4.6 Applying grid layout to the parser program

The parser program is part of the Concept project. It contains 203 classes and 600 relationships among the classes. Figure 50 is the layout without any adjustment. After ten minutes adjustments, we get a better layout shown in Figure 51. Figure 52 is almost the best result we can get through our grid layout algorithm. Although the last layout is the most readable layout, such large entities and internal relationships still result in information overload.

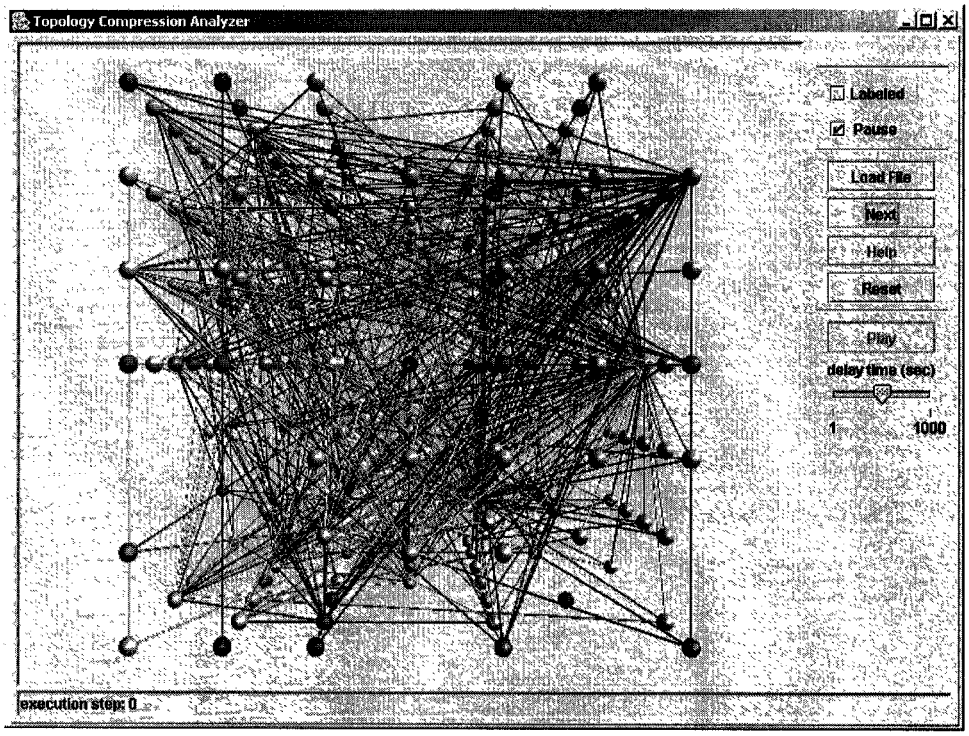


Figure 50. The initial layout

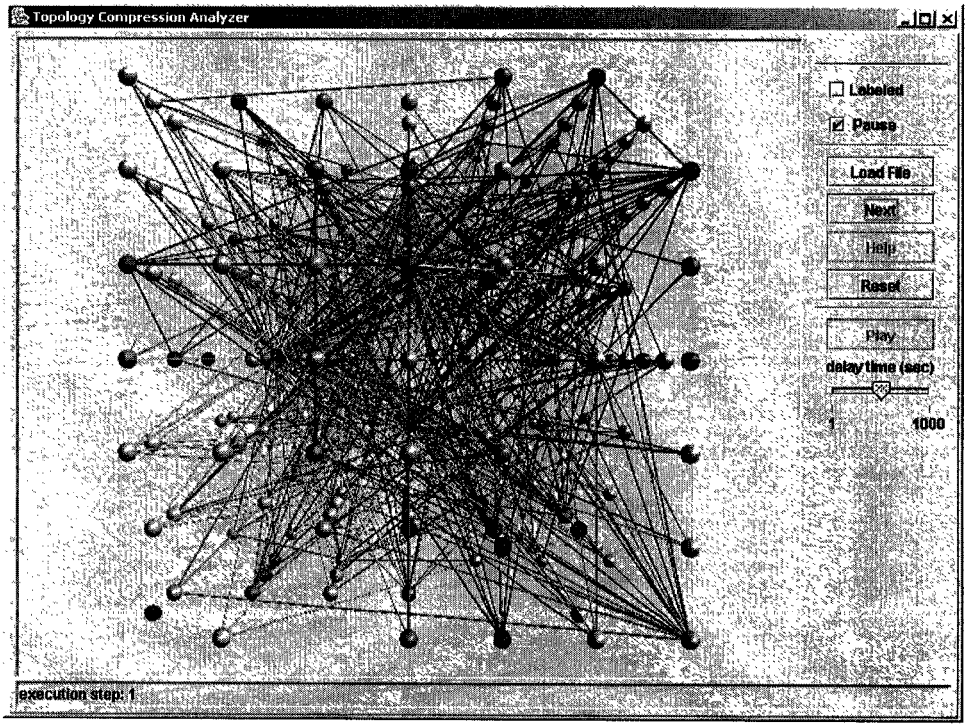


Figure 51. The layout after ten minutes adjustments

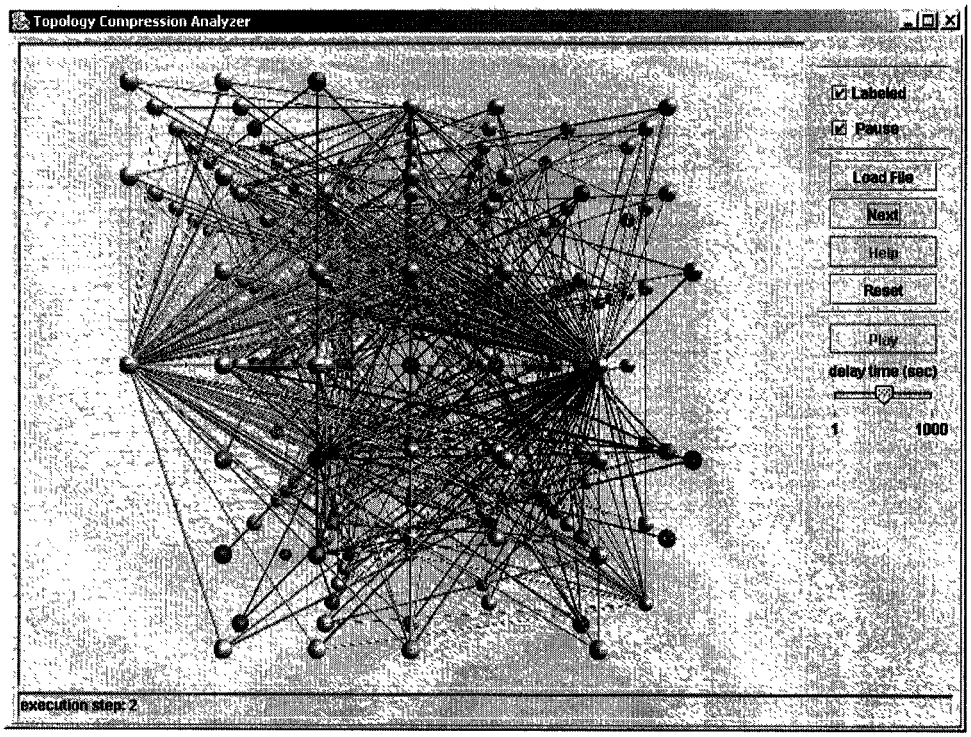


Figure 52. The layout after two hours adjustments

CHAPTER VI. CONCLUSIONS AND FUTURE EXTENSIONS

We have presented a software visualization tool, Metaviz, in which we address three important visualization issues: the Metaballs metaphor as a basis for software visualization, 3D grid based layout, and grouping. We further discussed implementation issues and the use of Java 3d as the choice of our implementation environment. Furthermore, we discuss the integration of Metaviz within the CONCEPT project and illustrate some potential applications for our system. We believe that the Metaballs metaphor can provide programmers with some guidance in building a suitable mental model of a software system. Our grid layout technique improves the readability of 3D visuals on a screen. In particular its ability to dynamically animate the layout optimization process by showing the current optimization progresses helps the user to be an active participant in this task (if required). Grouping based on object coupling and slicing based features are introduced to avoid information overloading. Using the code of the Metaviz program itself, we have demonstrated how these three key features can ease the cognitive burden associated with the visualization of large software systems. We have integrated the Metaviz tool in the CONCEPT program comprehension environment. Extensive experimentation with very large programs is needed before we can conclusively claim high effectiveness of this 3D visualization metaphor. Several issues remain for future work, issues like the scalability of the Metaballs for very large systems, further improvements to the layout and grouping algorithms. In particular, the area

combining domain knowledge representation with the Metaballs visualization might be a challenging avenue for future research.

BIBLIOGRAPHY

(AbsInt 2003) AbsInt © 1998-2003. Last modified on 4 July 2003. URL: http://www.absint.com/aicall

(Battista 1999) G. di Battista, P. Eades, R. Tamassia, and I.G. Tollis, "Graph Drawing: Algorithms for the Visualization of Graphs". Prentice Hall, 1999.

(Bourke1997) Paul Bourke. "*Polygonising a scalar field*". May, 1997. URL: http://astronomy.swin.edu.au/~pbourke/modelling/polygonise/

(Carey 1997) R. Carey and G. Bell. "*The Annotated VRML 2.0 Reference*". URL: http://www.web3d.org/resources

(Churcher 1999) Neville Churcher, Lachlan Keown,Warwick Irwin. "*Virtual Worlds for Software Visualization*". URL: www.cosc.canterbury.ac.nz/research/ RG/svg/lachlan-msc/thesis.pdf

(Churcher 2001) Neville Churcher and Alan Creek, "*Building Virtual Worlds with the Big-Bang Model*". Australian Symposium on Information Visualization, (invis.au 2001)

(Dwyer 2001) Tim Dwyer. "*Three Dimensional UML using Force Directed Layout. Australian Symposium on Information Visualisation*". Australian Symposium on Information Visualisation, (invis.au 2001), Sydney, NSW, Australia, ACS, 2001.

(Eades 1984) P. Eades. "*A heuristic for graph drawing*". Congressus Numerantium". 42:149--160, 1984.

(Eades 1999) P. Eades, "*Drawing Free Trees*", Bulletin of the Institute for Combinatorics and its Applications, pp. 10– 36, 1992. http://www.dia.uniroma3.it/~gdt/, 1999.

(Euler 1736) Euler. "*The Seven Bridges of Königsberg*". URL:http://mathforum.org/isaac/problems/bridges1.html, 1736

(Favre 2001) Favre J.M., "*GSEE: a Generic Software Exploration Environment*", 9th Intenational Workshop on Program Comprehension (IWPC'2001), Toronto, Canada, May 2001, pp. 233-244.

(Feijs1998) L. Feijs, and R. D. Jong. "*3D Visualization of Software Architectures*". Com. of the ACM, 41(12), Dec. 1998.

(Gogolla 1999) Martin Gogolla, Oliver Radfelder, Mark Richters. "*Towards Three-Dimensional Representation and Animation of UML Diagrams*". UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings.

(Haibt 1959) Haibt, L. M. "*A Program to Draw Multi-Level Flow Charts*". In Proceedings of The Western Joint Computer Conference, 15 (pp. 131-137). San Francisco, CA:

(Harel 1992) Harel D, "Biting *the Silver Bullet, Toward a Brighter Future for System Development*", IEEE Computer 25(1), 1992, pp 8-20.

(Hemmje1994) M. Hemmje, C. Kunkel, and A. Willet: "*LyberWorld A Visualization User Interface Supporting Fulltext Retrieval*", Proc. of ACM SIGIR'94, ACM Press, 1994.

(Hendley 1995) B. Hendley and N. Drew. "*Visualization of complex systems*". University of Birmingham(UK). 1995.

(Herman 2000) Ivan Herman, Guy Melancon, and M. Scoot Marshall, "*Graph Visualization and Navigation in Information Visualization: a Survey*", IEEE Transactions on Visualization and Computer Graphics, Vol6 No 3, 2000 Computer Science, 1995. (Ischia, Italy) IEEE Computer Society Press, 1998.

(James 1982) James F. Blinn. "*A generalization of algebraic surface drawing*". ACM Transactions on Graphics, 1(3):235--256, July 1982

(Johnson 1991) Brian Johnson and Ben Shneiderman. "*Tree-maps: A space-filling approach to the visualization of hierarchical information structures*". In G.M. Nielson and L. Rosenblum, editors, proc. Visialization '91, pages 284–291, Los Alamitos, CA, October 1991. IEEE Computer Society Press.

(Jünger 1997) M. Jünger, S. Leiper, and P. Mutzel, "*Pitfalls of using PQ-trees in automatic graph drawing*". In G. Di Battista, editor, Graph Drawing (Proc. GD '97), vol. 1353 of *Lecture Notes in Computer Science*, pp. 193-204. Springer-Verlag, 1997.

(Knight 1998) Knight, C. "*Visualization For Program Comprehension: Information And Issues*". University of Durham, computer Science Technical Report 12/98.

(Knight 1999) C. Knight and M.Munro, "*Comprehension with[in] Virtual Environment Visualisations*", Proceedings of the IEEE 7th International Workshop on ProgramComprehension, pp4-11, May 5-7, 1999.

(Knight 2001) Knight, C. and Munro, M. "*Software Visualization conundrum*", University of Durham, Department of Computer Science Technical Report 05/01, July 2001.

(Knight 2000) Knight, C., and Munro, M. "*The Power of (Software) Visualization*", University of Durham, Department of Computer Science Technical Report 01/00, January 2000.

(Korel 1998) Bogdan Korel, Juergen Rilling. "*Program Slicing in Understanding of Large Programs*". IWPC 1998: 145

(Koschke 1999) Koschke. R., "*An Semi-Automatic Method for Component Recovery*", Proceedings of the Sixth Working Conference on Reverse Engineering, pp.256-267, Atlanta, October 1999.

(Laguna 1997) M.R. Laguna, R. Martí, and V. Vals, "*Arc Crossing minimization in Hierarchical Digraphs with Tabu Search*", Computers and Operations Research, Vol. 24, No. 12, pp. 1165–1186, 1997.

(Lanza 2002) Michele Lanza. "Software Visualization Introduction". University of Berne, Switzerland. URL: iamwww.unibe.ch/~scg/Teaching/OORPT/05Visualizing.pdf

(Lorensen 1987) Lorensen, W.E. and Cline, H.E., "*Marching Cubes: a high resolution 3D surface reconstruction algorithm*", Computer Graphics, Vol. 21, No. 4, pp 163-169 (Proc. of SIGGRAPH), 1987.

(Mancorids 1999) S. Mancorids. B. S. Mitchell, Y. Chen, E. R. Gansner "*Bunch: A clustering tool for the recovery and maintenance of software structures*" In Proc; IEEE Inter. Conference on Software Maintenance, IEEE Computer Society Press, 1999, pp 50-59.

(Mark 2001) Mark A. Foltz Design Rationale Group, "*Graph Exploration for Software Archeology*". MIT Artificial Intelligence Laboratory, September 2001 URL:http://www.ai.mit.edu/projects/drg/

(Mayrhauser 1998) Mayrhauser, A., A. M. Vans, "*Program Understanding Behavior During Adaptation of Large Scale Software*", Proc. of the 6th Intl. Workshop on Program Comprehension, pp. 164-172, Italy, 1998.

(Melançon 1998) G. Melançon and I. Herman, "Circular *Drawings of Rooted Trees*", Reports of the Centre for Mathematics and Computer Sciences. Report number INS–9817, available at: http://www.cwi.nl/InfoVisu/papers/circular.pdf, 1998.

(Munzner 2000) T. Munzner. "*Interactive Visualization of Large Graphs and Networks*". *Ph.D. dissertation*, Stanford University, June 2000. URL:http://graphics.stanford.edu/papers/munzner_thesis/

(Myers 1986) Myers, B. A. "*Visual Programming, Programming by Example, and Program Visualization: A Taxonomy*". In M. Mantei & P. Orbeton (Ed.), *Proceedings of Human Factors in Computing Systems (CHI '86)*, (pp. 59-66). New York: ACM Press.

(Myers 1988) Myers, B. A. "*The State of the Art in Visual Programming and Program Visulization"*. (Technical Report No. CMU-CS-88-114). Computer Science Dept., Carnegie-Mellon University, Pittsburg, PA.

(Myer 1990) B.A. Myers. "*Taxonomies of Visual Programming and Program Visualization*". Journal of Visual languages and Computing, Vol. 1, pp97-123, 1990.

(Panagiotis 1990) Panagiotis K. Linos, "*Layout Methods for Improving the Readability of Graphical Reorientations for Programs*", 1990. URL: http://www.wbmt.tudelft.nl/pto/research/publications/Dissertation_Willemse/Dissertatie.ps.gz

(Plankers 2001) Ralf Plankers and Pascal Fua. "*Articulated Soft Objects for Video-based Body Modeling*". Proc. 8th Int. Conference on Computer Vision, Vancouver, Canada, July 2001

(Price 1992) B. A. Price, R. M. Baecker, and I. S. Small, "*A Principled Taxonomy of Software Visualization*", Journal of Visual Languages and Computing, Vol. 4, No. 3, pp211-266, 1992.

(Rajlich 2001) Václav Rajlich, Norman Wilde, Michelle Buckellew, Henry Page "*Software Cultures and Evolution*". Computer,September 2001 (Vol. 34, No. 9) pp. 24-28

(Rekmoto 1993) Jun Rekimoto, Mark Green, "*The Information Cube: Using Transparency in 3D Information Visualization*". Proc. of the 3rd Workshop on Infor. Technologies & Systems (WITS'93), pp. 125-132

(Rilling 2001) Rilling J., "*Maximizing Functional Cohesion of Comprehension Environments by Integrating User and Task Knowledge*", 8th IEEE Working Conference on Reverse Engineering (WCRE 2001), Stuttgart, Germany, October 2001, pp. 157-165.

(Rilling 2002) Juergen Rilling and S. P. Mudur, "*On the Metaballs to Visually Map Source Code Structures and Analysis Result on 3D Space*", Department of Computer Science, Concordia University. in Proceedings of Ninth Working

Conference on Reverse Engineering, Richmond, VA, October 29-Novemver 1 2002, pp. 11-24.

(Rilling 2003a) Juergen Rilling, Jianqun Wang, S. P. Mudur. "*Metaviz - issues in software visualization beyond 3D*". Department of Computer Science, Concordia University. VISSOFT 2003 -- 2nd Annual "DESIGNFEST" ON VISUALIZING SOFTWARE FOR UNDERSTANDING AND ANALYSIS

(Rilling 2003b) Juergen Rilling, Tuomas Klemola: "*Identifying Comprehension Bottlenecks Using Program Slicing and Cognitive Complexity Metric*". IWPC 2003: 115-124

(Robertson 1993) G. G. Robertson, S. K. Card, and J. D. Mackinlay. "*Information visualizing using 3D interactive animations*". Comm. of the ACM, 36(4), 1993.

(Roman 1993) G.C. Roman and K. C. Cox, "*A Taxonomy of Program Visualization Systems*", IEEE Computer pp11-24, December 1993.

(Russell 1995) S. Russell and P. Norvig, "*Artificial Intelligence A modern approach*", Prentice Hall, 1995.

(Sanlaville 2001) Sanlaville R., Favre J.M., Y. Ledru, "*Helping Various Stakeholders to Understand a Very Large Software Product*", European Conference on Component-Based Software Engineering, Sept. 2001.

(Sarita 2001) Sarita Bassil, Rudolf K. Keller. "*Software Visualization Tools: Survey and Analysis*". Ninth International Workshop on Program Comprehension (IWPC'01) May 12 - 13, 2001

(Sarkar 1992) M. Sarkar and M.H. Brown, "*Graphical Fish–eye views of graphs*", Human Factors in Computing Systems, CHI '92 Conference Proceedings, ACM Press, pp. 83–91, 1992.

(Sarkar 1994) M. Sarkar and M.H. Brown, "*Graphical Fisheye Views*", Communications of the ACM, Vol. 37 No. 12, pp. 73– 84, 1994.

(Scanlan 1989) Scanlan, D. "*A Structured Flowcharts Outperform Pseudocode: An Experimental Comparison*". IEEE Software, 6(5): 28-36.

(Segal 2001) Mark Segal, Kurt Akeley. "*The OpenGL Graphics System: A Specification*", Version 1.3. SGI, August 14, 2001.

(Shenghua 2003) Shi Shenghua, Master Thesis to be published. Computer Science Department, Concordia University. 2003.

(Shoeb 1998) Shoeb. "*Improved Marching Cubes*", enuxsa.eas.asu.edu/~shoeb/graphics/improved.html

(Storey 1997) M.-A. D. Storey, K. Wong, F. Fracchia, and H. Muller. "*On integrating visualization techniques for effective software exploration*". In Proceedings of the IEEE Symposium on Information Visualization, IEEE Visualization, 1997

(Storey 1999) Storey M.-A., Fracchia F. and Müller H.., "*Cognitive Design Elements to support the Construction of a Mental Model During Software Exploration*", Journal of Software Systems, special issue on Program Comprehension, v 44, pp.171-185, 1999.

(Sun 2002) Sun Microsystems. Inc. (2002, June). "*The Java 3DTM API Specification*" Version 1.3. Palo Alto. CA. USA

(Sugiyama 1981) K. Sugiyama, S. Tagawa, and M. Toda. "*Methods for visual understanding of hierarchical system structures*". IEEE Transactions on Systems, Man, and Cybernetics, 11(2):109--125, February 1981

(Sugiyama 1995) Kozo Sugiyama, Kazuo Misue: "*Graph Drawing by the Magnetic Spring Model*".. Journal of Visual Languages and Computing 6(3): 217-231 (1995)

(Tip 1995) Tip F., "*A survey of program slicing techniques*", Journal of Programming Languages, 3(3), pp. 121-189, September 1995.

(Wesley 1991) Addison-Wesley, 1997. Brian Johnson and Ben Shneiderman. *"Tree-maps: A space-filling approach to the visualization of hierarchical information structures*". In G.M. Nielson and L. Rosenblum, editors, proc. Visialization '91, pages 284–291, Los Alamitos, CA, October 1991. IEEE Computer Society Press.

(Xiaohua 2003) Xiaohua, Department of Computer Science, Concordia University. "*2D & 3D UML-based Software Sisualization for Object Oriented Progrom*". Master thesis in 2003.

(Yonggang 2003) Yonggun Department of Computer Science, Concordia University "*Automatic Design Pattern Recovery*". Master thesis in 2003.

(Young 1997) Peter Young and Malcolm Munro. "*A New View of Call Graphs for Visualizing Code Structures*". http://vrg.dur.ac.uk/misc/PeterYoung D

(Yong1998) Young, P., and Munro, M. "*Visualizing Software in Virtual Reality*". In Proceedings of the IEEE 6th International Workshop on Program Comprehension

(Yong1999) P. Young and M. Munro, "Visualizing *Software in Cyberspace*", Ph.D. Thesis, University of Durham, October 1999.

APPENDIX. CLASS DIAGRMS FOR METAVIZ

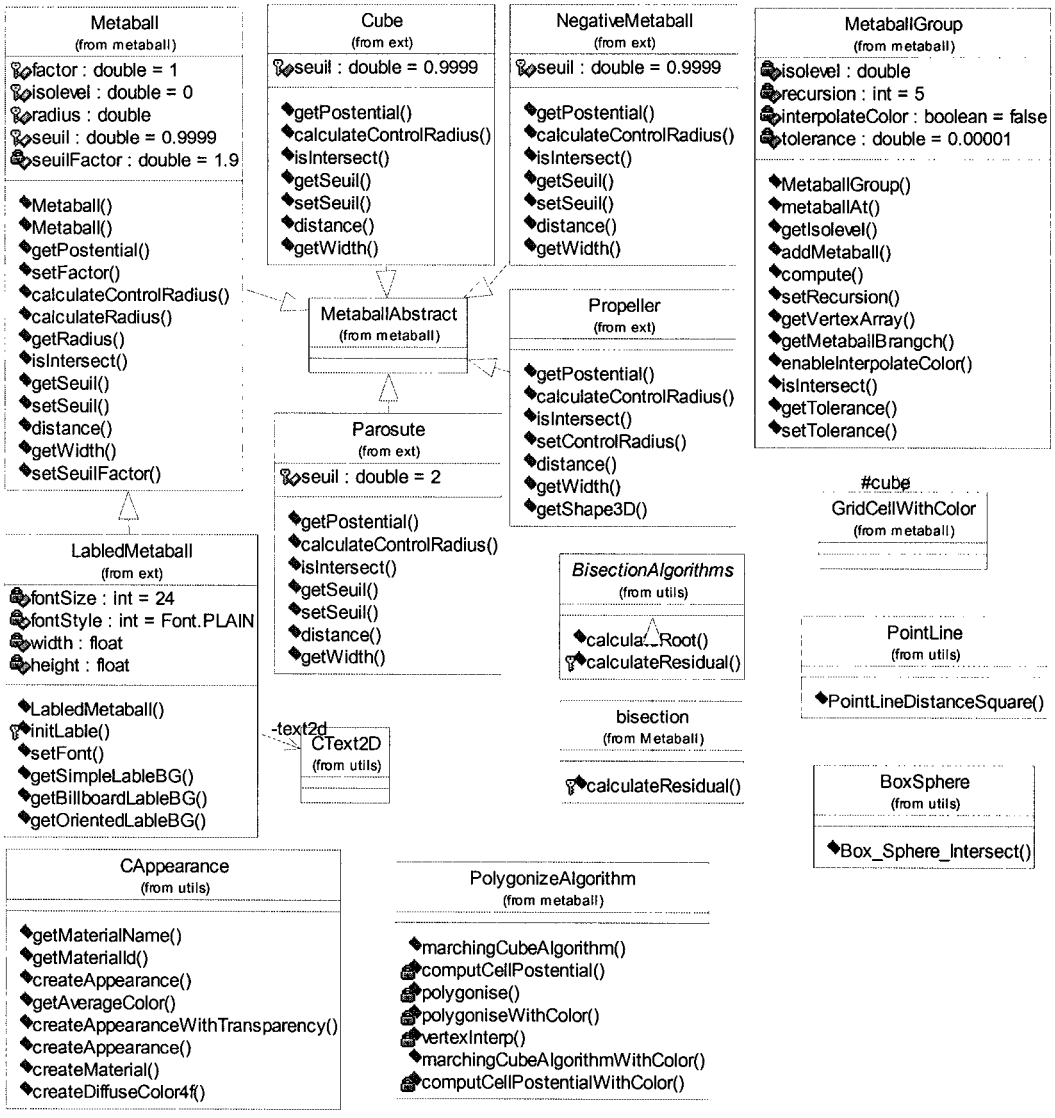


Figure 53. Metaballs class diagram

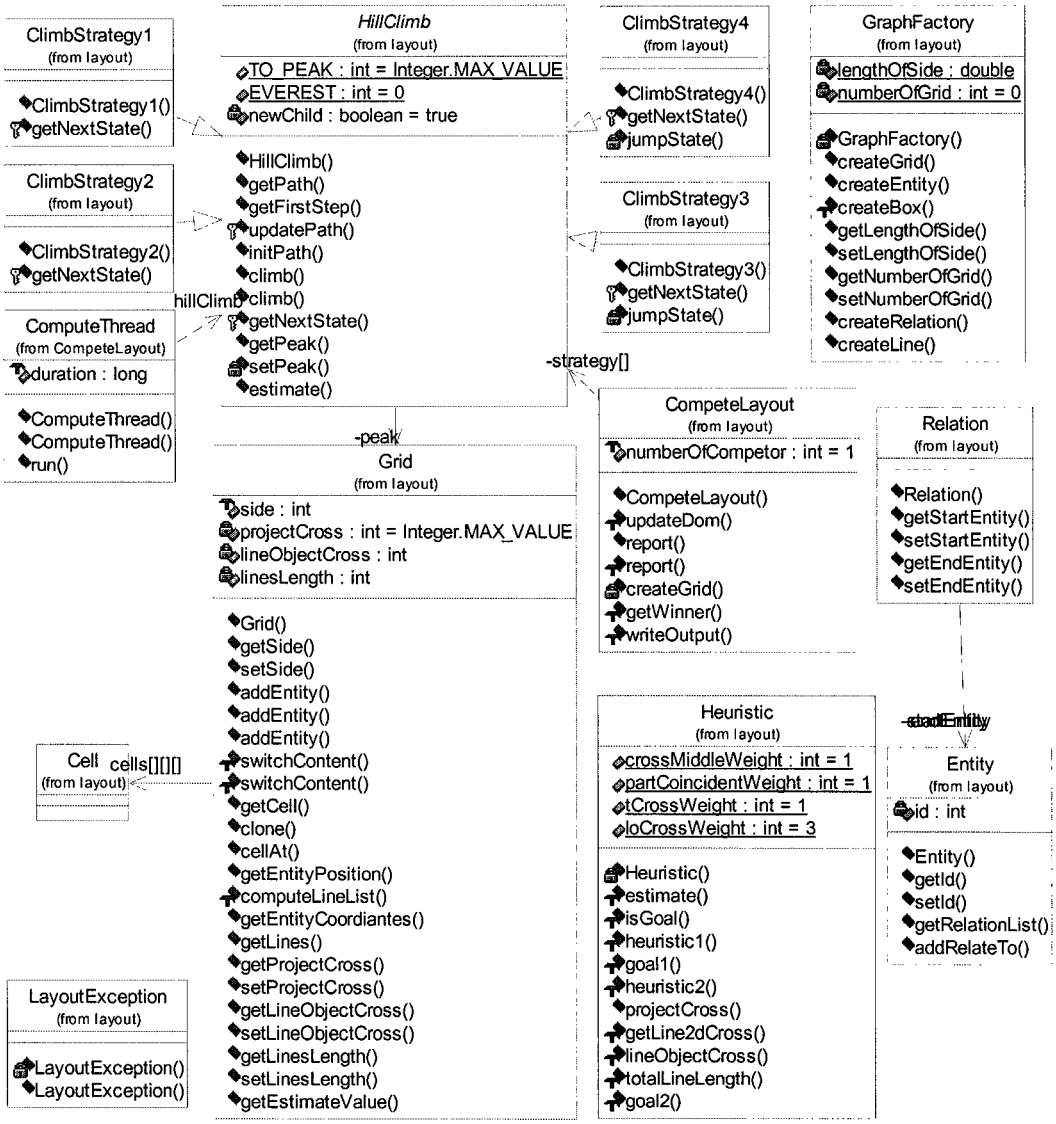


Figure 54. Grid layout class diagram

Figure 55. Uitlity class diagram

Figure 56. Dynamic visualization class diagram

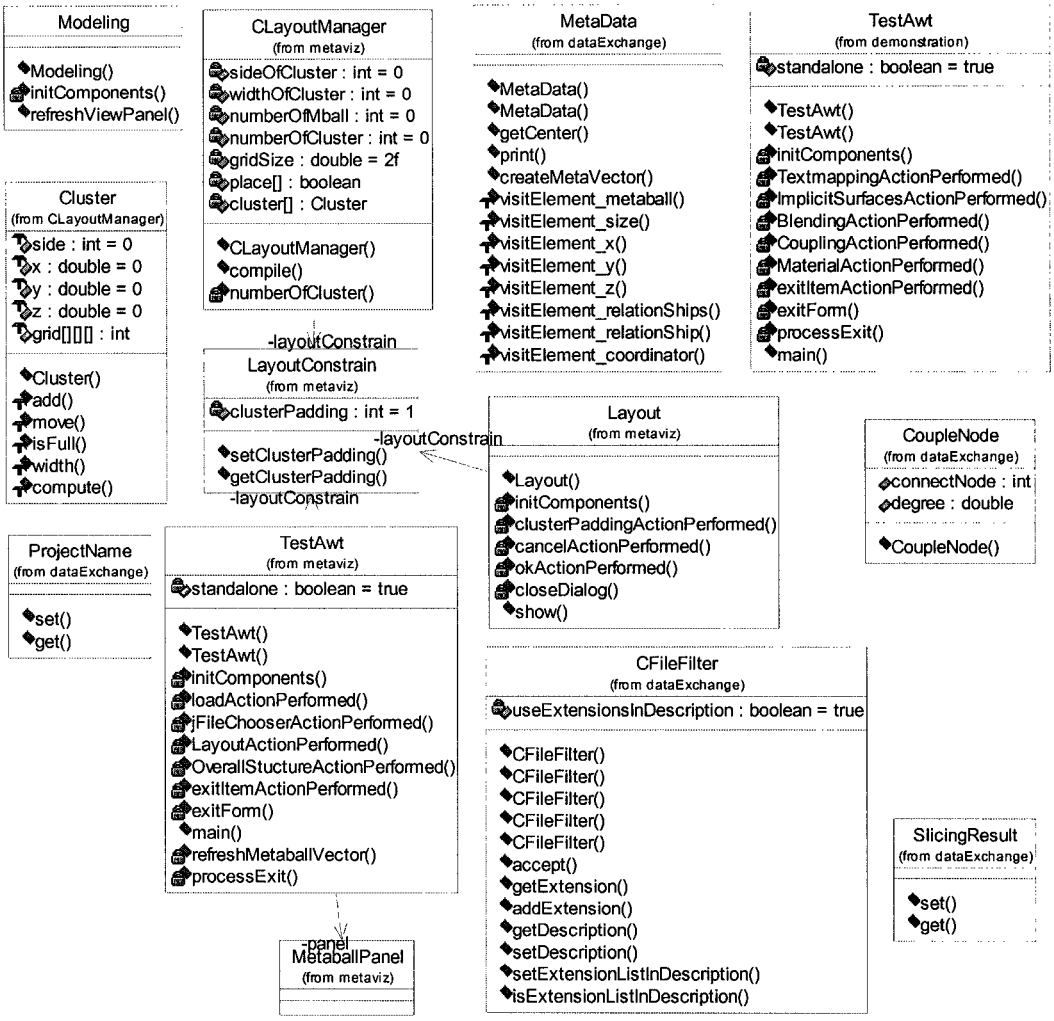


Figure 57. Metaviz class diagram