

# **The Hierarchical Visualization System**

A General Framework for Visualizing Information  
Hierarchies Using the Example of Information Pyramids

Werner Putz



# **The Hierarchical Visualization System**

A General Framework for Visualizing Information Hierarchies Using the Example of  
Information Pyramids

Master's Thesis

at

Graz University of Technology

submitted by

**Werner Putz**

Institute for Information Systems and Computer Media (IICM),  
Graz University of Technology  
A-8010 Graz, Austria

11th March 2005

© Copyright 2005 by Werner Putz

Advisor: Ao.Univ.-Prof. Dr. Keith Andrews



# **Das Hierarchische Visualisierungs System**

Ein allgemeines Framework zur Visualisierung hierarchischer Informationen am Beispiel  
der Informationspyramiden

Diplomarbeit  
an der  
Technischen Universität Graz

vorgelegt von

**Werner Putz**

Institut für Informationssysteme und Computer Medien (IICM),  
Technische Universität Graz  
A-8010 Graz

11. März 2005

© Copyright 2005, Werner Putz

Diese Arbeit ist in englischer Sprache verfasst.

Betreuer: Ao.Univ.-Prof. Dr. Keith Andrews



## **Abstract**

In recent years numerous techniques have been developed for visualizing hierarchies. This thesis describes a framework for the visualization of hierarchically structured information called the Hierarchical Visualization System (HVS). This general framework provides a synchronized, multiple view environment for visualizing hierarchies.

A traditional tree view and an implementation of Information Pyramids (the PyramidsBrowser) are provided as example reference browsers in the HVS framework. Other two-dimensional and three-dimensional visualization techniques such as Treemaps, Hyperbolic Browser, Cone Trees, Magic Eye View, and the Walker tree layout have also been implemented within HVS.

Since pure Java is used for the implementation, HVS is portable to any platform.





## **Kurzfassung**

In den letzten Jahren wurden zahlreiche Techniken zur Darstellung von Hierarchien entwickelt. In dieser Diplomarbeit wird das Hierarchische Visualisierungs System (HVS), ein Framework zur Darstellung hierarchisch strukturierter Informationen, beschrieben. Es ermöglicht die Darstellung von Hierarchien in mehreren synchronisierten Fenstern.

Eine traditionelle Baumansicht und eine Implementierung der Informationspyramiden Technik (der PyramidsBrowser) werden als Anwendungsbeispiele von HVS vorgestellt. Andere zweidimensionale und dreidimensionale Visualisierungstechniken wie Treemaps, Hyperbolic Browser, Cone Trees, Magic Eye View und Walker-Tree-Layout sind ebenfalls in HVS integriert worden.

Die Verwendung von reinem Java macht HVS portabel für viele Plattformen.



*I hereby certify that the work presented in this thesis is my own and that work performed by others is appropriately cited.*

*Ich versichere hiermit, diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfsmittel bedient zu haben.*



## **Acknowledgements**

Special thanks to my wife for her patience and understanding while writing this thesis.

I especially wish to thank my advisor, Keith Andrews, for his ideas, support, and correcting the draft version of this thesis.

Last but not least I wish to thank to the person who motivated me to finish this thesis, my new born daughter.

Werner Putz  
Graz, Austria, March 2005



## Credits

- Figures 2.2, 3.4, 2.11, and 4.1 were taken from HCIL [2004] and are used subject to the Copyright Notice of the University of Maryland<sup>1</sup>.
- Figure 4.2 was taken from Visage [2004]
- Figure 4.3 was taken from ShapeVis [2004]
- Figure 4.4 was taken from Polaris [2004]

The following figures are used subject to the ACM Copyright Notice<sup>2</sup> :

- Figure 2.1 extracted from Proc. of SIGCHI'91.
- Figure 2.7 extracted from Proc. of UIST'94.
- Figure 2.10 extracted from Proc. of CHI'96.
- Figure 3.8 extracted from CHI'95 Electronic proceedings.
- Figure 3.10 extracted from Proc. of NPIV'99.
- Figure 3.11 extracted from Proc. of CHI'91.

---

<sup>1</sup>Copyright Notice of the University of Maryland:

All works herein are Copyright (c) University of Maryland 1984-1994, all rights reserved. We allow fair use of our information provided any and all copyright marks, trade marks, and author attribution are retained.

<sup>2</sup>ACM Copyright Notice

Copyright © by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Information Visualization</b>	<b>3</b>
2.1	Linearly Structured Information . . . . .	3
2.2	Multi-Dimensional Information . . . . .	5
2.3	Network Visualization . . . . .	6
2.4	Content-Based Visualizations . . . . .	9
2.5	Spatial Information . . . . .	11
<b>3</b>	<b>Visualizing Hierarchies</b>	<b>13</b>
3.1	Classic Tree Drawings . . . . .	13
3.2	Tree Browsers . . . . .	14
3.3	Treemaps . . . . .	14
3.4	Information Slices . . . . .	17
3.5	Cheops . . . . .	17
3.6	Hyperbolic Browser . . . . .	18
3.7	Magic Eye View . . . . .	19
3.8	Cone Trees . . . . .	20
3.9	3D Hyperbolic Browser . . . . .	21
3.10	File System Navigator . . . . .	22
<b>4</b>	<b>Visualization Toolkits</b>	<b>25</b>
4.1	Snap Together . . . . .	25
4.2	Visage . . . . .	26
4.3	SinVis . . . . .	28
4.4	Polaris . . . . .	29
4.5	InfoVis Toolkit . . . . .	31
4.6	Spotfire . . . . .	32
4.7	Prefuse . . . . .	34
4.8	Information Visualization CyberInfrastructure . . . . .	36

<b>5</b>	<b>The Hierarchical Visualization System (HVS)</b>	<b>37</b>
5.1	The Architecture of HVS . . . . .	37
5.2	Input Modules . . . . .	39
5.3	Hierarchical Data Model . . . . .	41
5.4	Controller . . . . .	43
5.5	Search Result List . . . . .	44
5.6	Visualization Modules . . . . .	45
5.7	Application . . . . .	48
5.8	Synchronization Modes . . . . .	50
5.9	The User Interface of the Application . . . . .	53
<b>6</b>	<b>Information Pyramids</b>	<b>59</b>
6.1	The Pyramids Technique . . . . .	59
6.2	3D Explorer . . . . .	62
6.3	Java Pyramids Explorer . . . . .	63
6.4	The PyramidsBrowser . . . . .	63
<b>7</b>	<b>Selected Details of the Implementation</b>	<b>73</b>
7.1	MainFrame . . . . .	73
7.2	ColorTable . . . . .	74
7.3	VisualizationController . . . . .	75
7.4	The PyramidsBrowser . . . . .	77
7.5	FSDataSource . . . . .	85
<b>8</b>	<b>Outlook</b>	<b>87</b>
8.1	Work in Progress . . . . .	87
8.2	Navigational History and Bookmarks . . . . .	88
8.3	Flexible Mapping of Attributes . . . . .	88
8.4	Comparative Study . . . . .	89
<b>9</b>	<b>Concluding Remarks</b>	<b>93</b>
<b>A</b>	<b>HVS User Guide</b>	<b>95</b>
A.1	Installation . . . . .	95
A.2	Starting HVS . . . . .	95
A.3	HVS . . . . .	95
A.4	HVS Visualizations . . . . .	104
<b>B</b>	<b>Developer Guide</b>	<b>111</b>
B.1	Development of a Tree View Visualization . . . . .	111
	<b>Bibliography</b>	<b>115</b>

# List of Figures

2.1	The perspective wall visualizing linear data structures. . . . .	4
2.2	The Filmfinder application. . . . .	5
2.3	Table lens visualizing baseball statistics. . . . .	6
2.4	Parallel Coordinates visualizing a six dimensional data set. . . . .	7
2.5	The file attribute explorer visualizing the contents of a directory . . . . .	8
2.6	The Harmony Local Map. . . . .	8
2.7	Galaxy of news. . . . .	9
2.8	VxInsight visualizing a set of bibliographic records of articles from journals. . . . .	10
2.9	Websom displaying over a million postings from over 80 newsgroups. . . . .	11
2.10	TileBars visualizing query results. . . . .	12
2.11	The user interface of the Visible Human Explorer. . . . .	12
3.1	The classic Reingold and Tilford tree layout. . . . .	14
3.2	The Microsoft Windows Explorer. . . . .	15
3.3	An example of the layout algorithm of Treemaps. . . . .	15
3.4	The Treemap 4.0 application visualizing a file hierarchy. . . . .	16
3.5	Information slices visualizing a file hierarchy. . . . .	17
3.6	A fully expanded hierarchy and the corresponding representation in Cheops. . . . .	18
3.7	The Cheops visualization with choice boxes. . . . .	19
3.8	The hyperbolic browser. . . . .	20
3.9	A file system visualized in a hyperbolic browser. . . . .	21
3.10	The magic eye viewer. . . . .	22
3.11	A hierarchy visualized using a ConeTree. . . . .	23
3.12	The H3 3d hyperbolic browser. . . . .	23
3.13	A screen shot of the FSN landscape visualizing a file system. . . . .	24
4.1	A multiple view environment created with Snap. . . . .	27
4.2	Creation of a new view by dragging in visage. . . . .	28
4.3	Example of a ShapeVis visualization. . . . .	30
4.4	An analysis example using Polaris. . . . .	31
4.5	The contol panel for treemaps in the InfoVis Toolkit. . . . .	32
4.6	A treemap visualization from the InfoVis Toolkit visualizing a file hierarchy. . . . .	33
4.7	Spotfire exploring a film database. . . . .	34

4.8	A tree structure visualized with Prefuse using a balloon tree layout. . . . .	36
5.1	The hierarchical visualization system. . . . .	38
5.2	The main components of HVS. . . . .	39
5.3	The HVS strategy for filtering. . . . .	42
5.4	Example of filtering a hierarchy in HVS. . . . .	43
5.5	The synchronization architecture of HVS. . . . .	44
5.6	The properties panel used to display details of selected nodes. . . . .	46
5.7	The document type definition of plug-in configuration files. . . . .	49
5.8	The selection dialog for selecting a new data source. . . . .	49
5.9	The standard color selection dialog. . . . .	51
5.10	The color selection dialog. . . . .	52
5.11	The <i>DetailedView</i> mode. . . . .	53
5.12	The hide documents mode. . . . .	54
5.13	The synchronization architecture to fulfill the different synchronization modes. . . .	55
5.14	The Control Panel. . . . .	56
5.15	The filter panel when no Datasource is set. . . . .	56
5.16	The filter panel for the file system Datasource. . . . .	57
5.17	The search panel for the file system Datasource. . . . .	57
5.18	The search result panel . . . . .	58
5.19	The table view of the search result list. . . . .	58
6.1	The 3D Explorer application. . . . .	60
6.2	The 3D Explorer which implements Information Pyramids. . . . .	62
6.3	The JPE application visualizing a file system. . . . .	64
6.4	The placement of labels of the PyramidsBrowser. . . . .	65
6.5	The top surface of leaf node blocks of the PyramidsBrowser are adorned with thumb- nails of the represented documents. . . . .	66
6.6	The sliders of the PyramidsBrowser are used for rotation. . . . .	66
6.7	Freely zooming in the pyramid landscape.. . . .	68
6.8	Maximizing the view of a selected node. . . . .	69
6.9	The animation settings dialog. . . . .	70
6.10	The angle of rotation determines the drawing order of surfaces of a plateau. . . . .	70
6.11	The drawing order of plateaus on top of a base plateau for correct hidden surface elimination with the Painter's algorithm. . . . .	71
6.12	Graceful degradation and different levels of detail during rendering. . . . .	72
7.1	The main window of HVS. . . . .	74
7.2	The different layout algorithms of the PyramidsBrowser. . . . .	83
7.3	The Document Type Definition of the configuration file of the FSDataSource input module. . . . .	86
8.1	An implementation of TreeMaps in HVS. . . . .	88

8.2	An implementation of a hyperbolic browser in HVS. . . . .	89
8.3	A implementation of the Magic Eye in HVS. . . . .	90
8.4	The implementation of the walker layout algorithm in HVS. . . . .	91
A.1	The menu bar. . . . .	96
A.2	The HVS file menu. . . . .	96
A.3	The dialog box to select a data source. . . . .	96
A.4	The dialog box to select a visualization. . . . .	97
A.5	The open configuration file dialog. . . . .	98
A.6	The save configuration dialog. . . . .	98
A.7	The options menu. . . . .	99
A.8	The help menu. . . . .	99
A.9	The tool bar. . . . .	99
A.10	The font selection dialog. . . . .	100
A.11	The color selection dialog. . . . .	100
A.12	The sort panel. . . . .	100
A.13	The filter panel. . . . .	101
A.14	The search panel. . . . .	102
A.15	The search result panel. . . . .	102
A.16	The table view of the search result list. . . . .	103
A.17	The properties panel used to display details of selected nodes. . . . .	104
A.18	Choosing the attributes to display in the properties panel. . . . .	105
A.19	The context menu which pops up on a right mouse click. . . . .	105
A.20	The Information Pyramids visualization. . . . .	106
A.21	The animation settings dialog. . . . .	107
A.22	The plateau settings dialog. . . . .	108
A.23	The dialog box to rename a node. . . . .	109



# Chapter 1

## Introduction

This thesis describes a framework for visualizing hierarchies called the Hierarchical Visualization System (HVS).

The first chapter gives a short overview of information visualization as a research field. The different categories, in which information can be classified are discussed with examples.

Chapter 3 describes techniques for visualizing hierarchies. Since the Hierarchical Visualization System visualizes hierarchically structured information, techniques for visualizing hierarchies are described in some detail. Techniques using 2D graphics are discussed as well as techniques using 3D graphics.

Chapter 4 gives a brief survey of existing toolkits for information visualizations. The presented toolkits are only a small selection of those available. They are selected concerning their different aspects.

The Hierarchical Visualization System is presented in Chapter 5. Firstly, an overview of the architecture is provided. The main modules and components are then described. The main interfaces are explained and the synchronization of visualizations is discussed in detail. At the end of this chapter, the user interface to HVS is described.

Chapter 6 presents the PyramidsBrowser, a three-dimensional visualization of hierarchies using the Information Pyramids technique. At first the Information Pyramids technique is described. The purpose and advantage of this technique are discussed. Extending the technique to visualize the structure and additional content information is presented. The 3D Explorer and the Java Pyramids Explorer, two implementations of the Information Pyramids technique are described afterwards. The concepts and improvements to the PyramidsBrowser are then explained.

Selected details of the implementation are described in Chapter 7. Firstly, the main purposes of the implementation details of HVS are described. The color management and the synchronization of views are explained in detail. Some of the implementation details of the Information Pyramids are described. How to improve the performance is explained in detail. At the end of this chapter the input module for visualizing the structure of a file system is presented.

Work in Progress and some ideas for improvement of the Hierarchical Visualization System are discussed in Chapter 8.

Appendix A provides a user guide for the Hierarchical Visualization System. It describes hardware and software prerequisites, the installation procedure of the application, and the installation of extensions, as well as how to run the application.

Appendix B provides a developer guide for the Hierarchical Visualization System. Using the example of a Tree View, it describes how to incorporate a new visualization into HVS.





## Chapter 2

# Information Visualization

The human visual perception system is able to rapidly and effortlessly recognize changes in size, color, shape and movement. Detecting patterns and finding similarities in visual forms is one of the strengths of humans. Information visualization systems exploit this to compress large amount of textual information into a manageable visual representation. The task performed by information visualization tools is to help users gain insight into raw information.

A related research field is called “scientific visualization”. Scientific visualization primarily represents real world objects such as parts of the human body, vehicles, buildings, or mountains. Information visualization on the other hand deals with abstract data sets which have no given geometry.

Information visualization tools often follow the mantra formulated by Shneiderman [1996] to capture the essence of visualizing an information space:

“Overview first, zoom and filter, details on demand.”

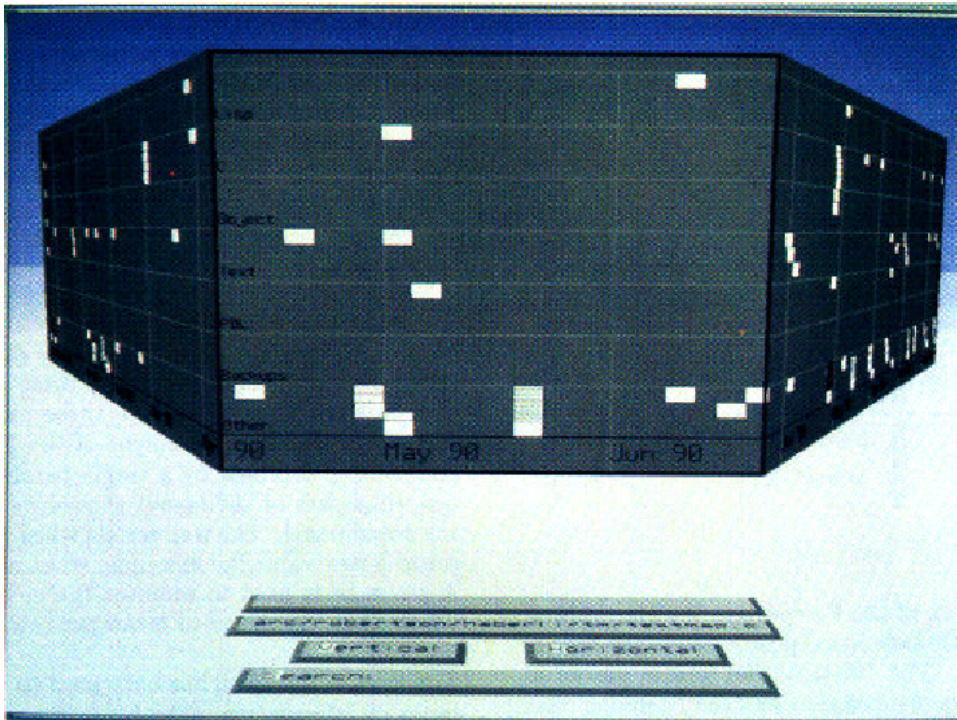
An often cited principle in information visualization is “focus + context”. Focus + context techniques allow the users to view parts of the representation in detail while preserving the remaining context. Fisheye views [Furnas, 1986] use geometric distortion like magnifying glasses to zoom the area of interest. Other techniques use 3d perspective to produce a natural focus on items of interest.

In analogy to Shneiderman [1996] information can be classified into the following categories:

- Linear: textual documents, program source code and tables.
- Hierarchical: Hierarchical structured data has an inherent structure in which items or nodes have a single parent. Hierarchical structured information is discussed in detail in Chapter 3.
- Multi-dimensional: data with more than three attributes.
- Network: Network data are general graph structures consisting of nodes connected by edges.
- Vector space: A vector represents the characteristics of the content of an item.
- Spatial (inherent 2d or 3d data): floor plans, geographic maps newspaper layouts or CAD models.

### 2.1 Linearly Structured Information

The most common form of linearly structured information is a text document. Text documents are normally read from start to finish. Users might take advantage of visualization tools to find lines with

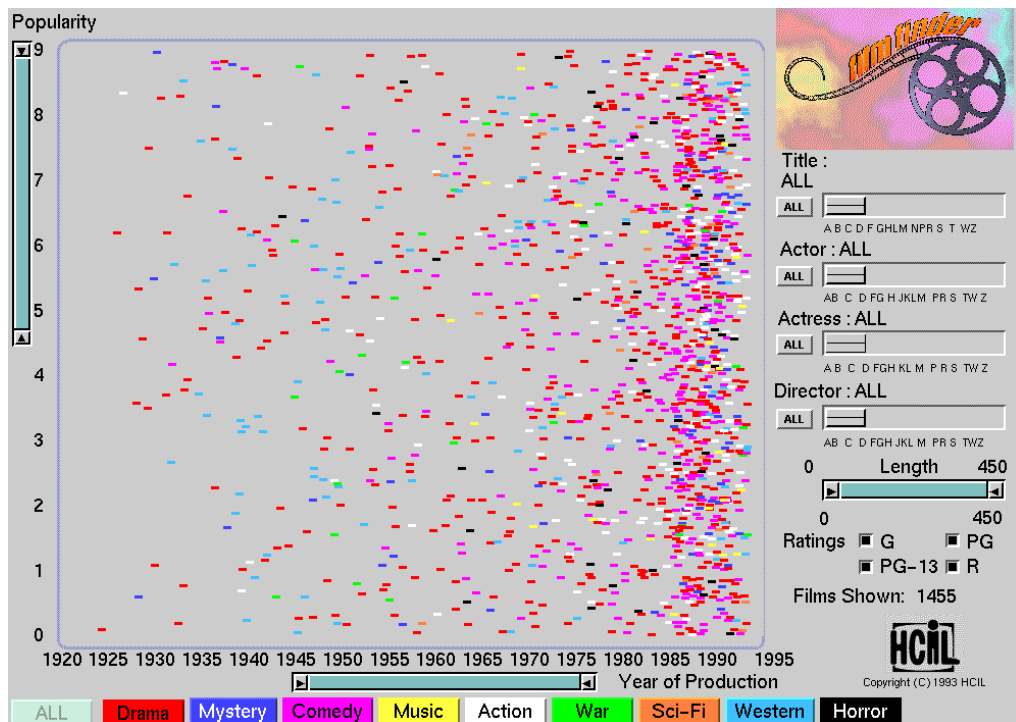


**Figure 2.1:** The perspective wall visualizing linear data structures. [Figure extracted from Proc. of SIGCHI'91. Copyright by the Association of Computing Machinery, Inc.]

specific attributes. For example modern editors used in software development enable users to find changed lines and compare the source code with previous versions.

The Perspective Wall [Mackinlay et al., 1991] (see Figure 2.1 ) approach uses three dimensional perspective to visualize linear information. The linear information is structured across a wall from left to right. Details are displayed in the center of the wall. The two perspective panels on both sides allows users to view the context. The perspective view enlarges the center of the wall and reduces the context with growing distance. The natural focus + context effect is achieved by the 3d perspective.

The Document Lens [Robertson and Mackinlay, 1993] visualizes a multiple page document so that the entire content is visible. This enables users to quickly detect patterns in documents. Zooming into parts of documents allows users to read these parts without losing global context. The Document Lens approach uses a special lens to provide a focus+context display. Using an optical fisheye lens, the distortion would make text hard to read. A simply magnifying lens would obscure the parts of the document near the lens. Instead, Document Lens uses a rectangular magnifying lens with elastic sides. This lens stretches the region outside of the lens to provide a continuous 3d deformation. The resulting truncated pyramid contains the whole document with a magnified focus area. The text in and near the lens is thus readable for users. The movement of the lens is animated so users always understands what is displayed. The lens becomes unreadable when it is moved to closed to the user. This results in a loss of the global context.



**Figure 2.2:** The Filmfinder application uses a starfield display to present films. Sliders are used as dynamic queries to filter the information. [Copyright University of Maryland, all rights reserved.]

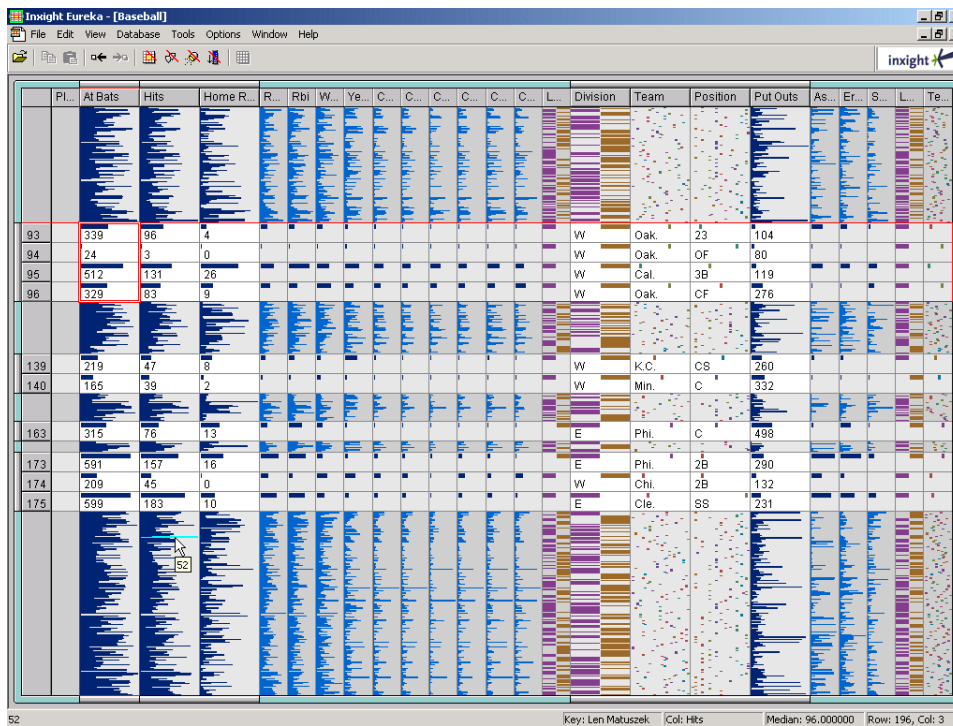
## 2.2 Multi-Dimensional Information

Multi-dimensional information describes items with more than two attributes. For example, a file often has associated metadata attributes such as author, title, modification date and size. These attributes form an  $n$ -dimensional space, within which each item is placed, its position being determined by its attribute values.

Multi-dimensional data is often stored in databases. The FilmFinder [Ahlberg and Shneiderman, 1994] (see Figure 2.2) application is such a database visualization tool. It combines a 2d scatterplot with dynamic queries enabling rapid, incremental and reversible exploration of the features of films. Dynamic queries [Ahlberg et al., 1992] let users move sliders corresponding to attributes in the database, to interactively filter the information. The display reflects the changes in moving sliders simultaneously. Through dynamic queries, users are able to perform task faster than using textual input fields.

Table lens [Rao and Card, 1994] (see Figure 2.3) visualizes the contents of a table using a focus+context technique. A basic feature of a table is that information in rows or columns correlates. The table lens approach distorts cells while preserving rows and columns. This enables users to explore rows and columns as a whole. Multiple focus areas with different focal levels are another feature of table lens. The table lens approach enables users to manage much larger tables than conventional spreadsheet techniques.

Parallel coordinates [Inselberg and Dimsdale, 1990] are another method to visualize multidimensional data. Equidistant vertical lines represent the dimensions of the information space. Each line is



**Figure 2.3:** Table lens visualizing baseball statistics. The focused cells are magnified.

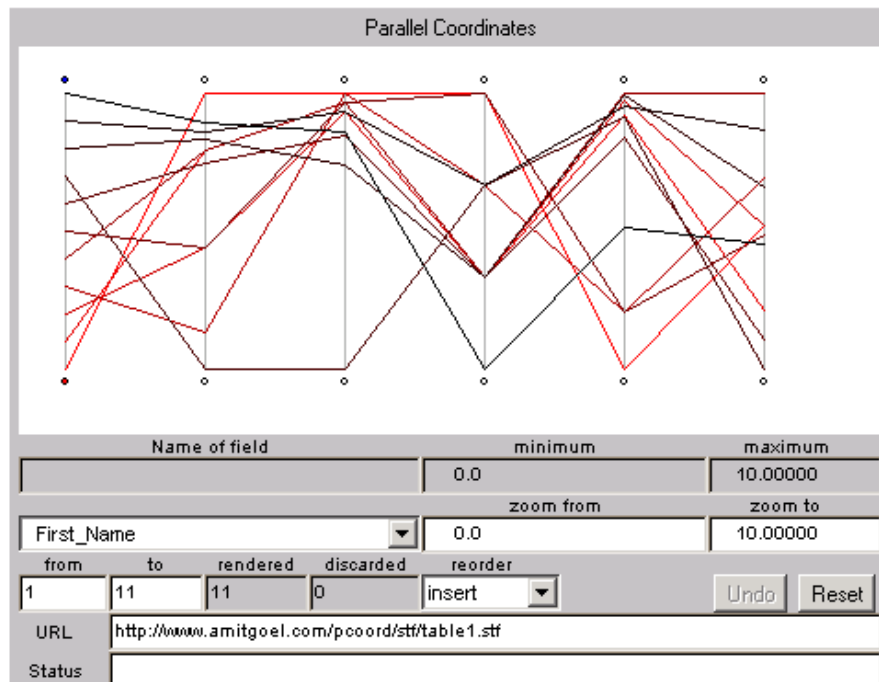
associated with one dimension. Objects are drawn as polylines connecting the positions of an object's attribute value on each dimension. Similar objects will have similar polylines. Figure 2.4 shows an example plot of parallel coordinates. The visualized dataset from Goel [1999] is six dimensional. Each vertical line represents one dimension. The polylines represent the data objects. The points of each line were determined by the object values at each dimension.

Envision [Nowell et al., 1996] displays search results based on their metadata as a matrix of icons. Circled icons represent single documents in a scatter plot like view. Elliptical icons are used to visualize a set of documents located at the same point of the display. Visual attributes such as position, icon size, icon color, and shapes of icons are user configurable. An item summary view is used to visualize details of selected icons.

The FileAttributeExplorer [Weilander, 1999] visualizes the content of a directory tree. Files are arranged in a scatterplot display. The position is determined by the values of two metadata attributes. Group icons are used when too many icons would be placed in the same area. A number in the group icons describes the number of represented files. The full set of metadata attributes is visualized by clicking on the icon. The size and color of icons is used to visualize two further metadata attributes. Zooming into a region of interest is possible for users. To maintain orientation an overview window is used. Optionally, a sortable spreadsheet can be used to display all current files.

## 2.3 Network Visualization

Networks are general graph structures consisting of nodes connected by edges. Examples include web pages and the links between them or a map of the London Underground with stations connected by tube lines. The positioning of nodes is important for visualizing networks. Graphical layout al-



**Figure 2.4:** Parallel Coordinates visualizing a six dimensional data set. Each vertical line represents one dimension. A data object is represented by a polyline which joins the position of the object's attribute values along each dimension.

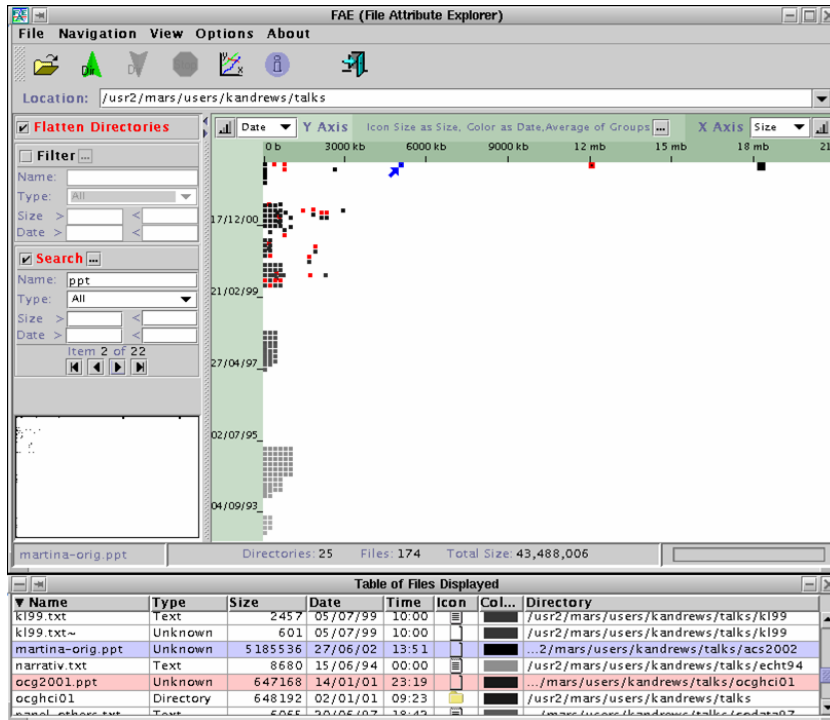
gorithms are used to produce organized layouts representing the associated graph properties. There are several algorithms to remove cycles or reduce crossings of edges. They are used to draw directed graphs. The Harmony Local Map [Schipflinger, 1998] dynamically creates a map visualizing the outgoing and incoming links of a selected document. This approach extends the graph layout algorithm of Eades and Sugiyama [Eades and Sugiyama, 1990] to focus the layout around a specific node. The document in focus is always visualized in the center of the display.

Making use of physical analogies is another method to lay out graphs. One approach is to use a spring model for positioning nodes. Edges are replaced with imaginary springs.

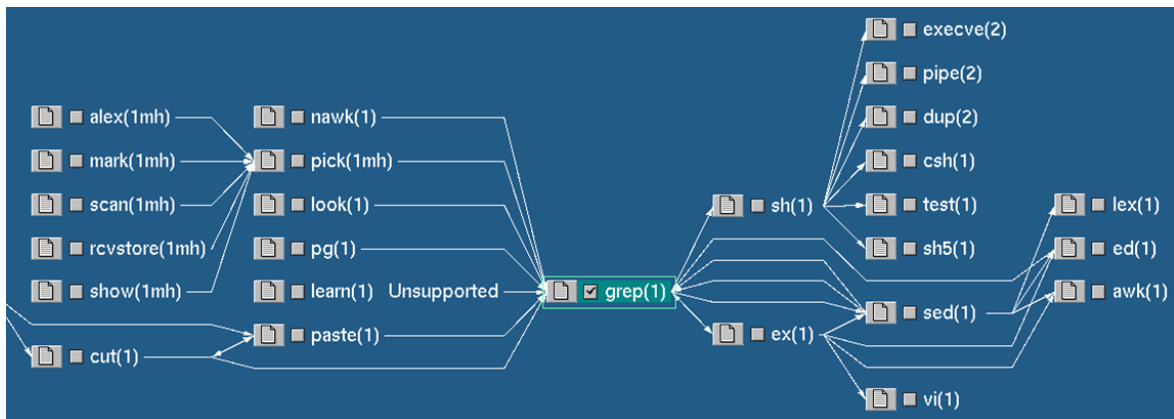
Force-direct placement methods assign each spring with a repulsive force. The nodes are iteratively moved, until the total stress in the system falls below a defined threshold. HyperSpace (formerly Narcissus) [Hendley et al., 1995] is an example of a system that uses a spring model. It visualizes web pages in a self-organizing structure. The attractive force between pages is proportional to the number of links between them. The total number of incoming and outgoing links determines the size of each node.

In energy-based placement methods each spring is assigned an energy. The sum of the potential energy of all springs is defined by an objective function. Numerical methods are used to calculate the minimum of this objective function to determine the position of the nodes.

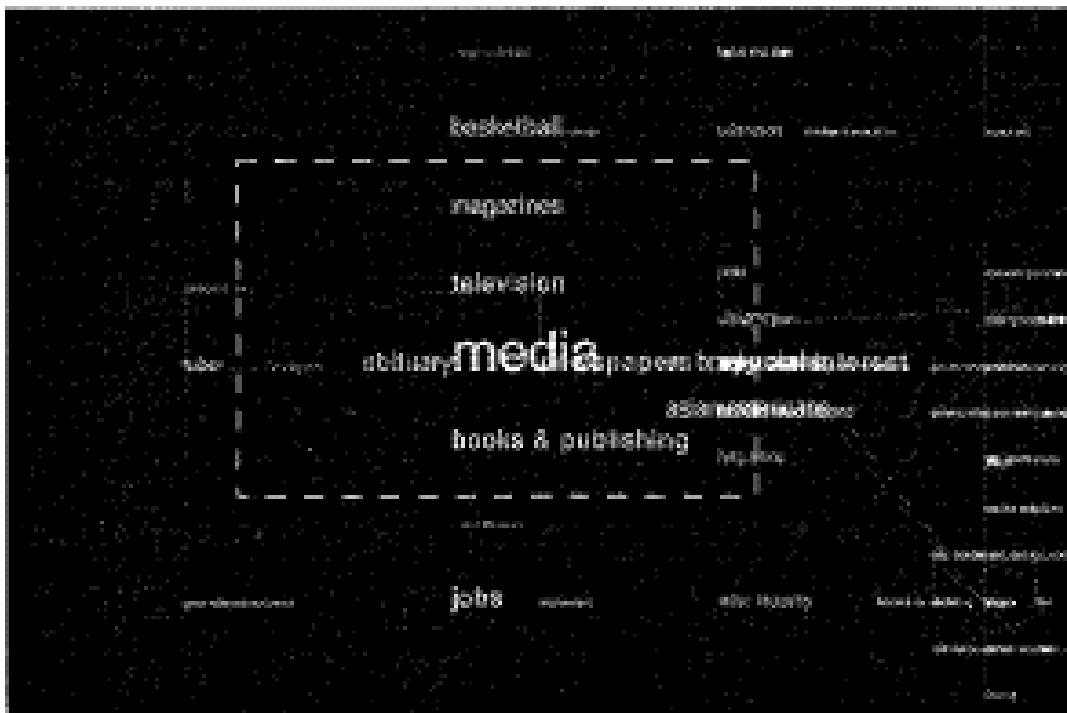
The minimum of this objective function can also be determined with a simulated annealing method. Instead of using a numerical solution, this method uses an approximation algorithm. SemNet [Fairchild et al., 1988], a visualization of knowledge bases, uses such a simulated annealing algorithm to lay out nodes. A three dimensional space is used to visualize knowledge bases as directed graphs.



**Figure 2.5:** The file attribute explorer visualizing the contents of a directory. [Image used with kind permission of Keith Andrews, Graz University of Technology.]



**Figure 2.6:** The Harmony Local Map visualizing a map of the link environment of hypermedia documents. In this case, documents are Unix named pages and all documents upto three links away from the `grep` document are visualized. [Image used with kind permission of Keith Andrews, Graz University of Technology.]



**Figure 2.7:** Galaxy of news. Fluid navigation through related concepts in a network of newswire articles. [Image extracted from Proc. of UIST'94. Copyright by the Association of Computing Machinery, Inc.]

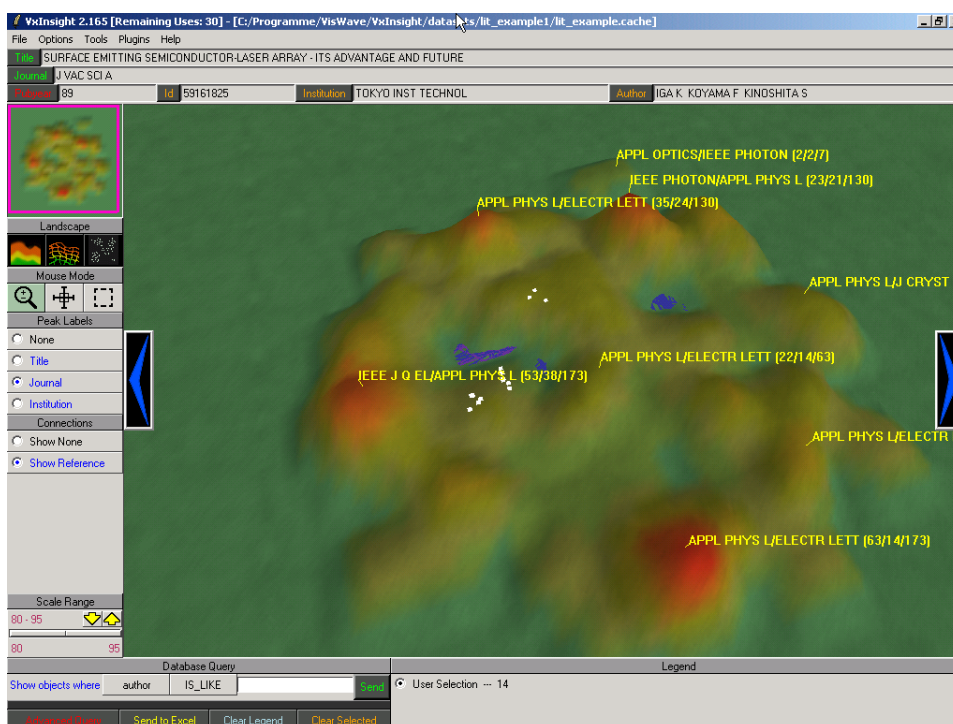
Labeled rectangles represents elements of the knowledge base. These rectangles are connected by lines or arcs. Color coding is used to show the relationships between elements.

Another approach for visualizing a graph is to reduce it to a tree structure and use techniques for visualizing hierarchies. Trees are a special case of graphs. A tree is a directed acyclic graph. In each tree there exists a unique path from the root to each node. The visualization is created using a spanning tree and additional edges are added afterwards.

Semantic networks are networks where nodes represent concepts and links between them represent relations. Semantic zooming techniques are used to manage semantic networks whose visualization does not fit in a single view. These techniques visualize a small part of the network at one time and allow fluid navigation through related objects. The Galaxy of News [Rennison, 1994], for example, creates and visualizes a semantic network of related newswire articles. At first, visualization of root key words is used as an abstract overview of the entire database. Using semantic zooming techniques, users can zoom toward a specific cluster, which leads to article headlines becoming readable (see Figure 2.7). Further zooming eventually makes the full newswire article visible to users.

## 2.4 Content-Based Visualizations

Content-based visualization techniques analyze the content of documents, such as the textual content of a corpus of text documents. One method to characterize documents is the vector space model [Salton et al., 1975]. For text documents a vector of unique words is extracted from each document excluding common terms like “and” and “the”. Each unique word in the document vector is assigned



**Figure 2.8:** VxInsight visualizing a set of bibliographic records of articles from journals. The height and shape of the mountains characterize the number and position of data objects.

a weight. The similarity between documents can be described by vector metrics such as the scalar product.

Methods for mapping the high-dimensional vector space to a lower dimensional display space have been developed. These methods provides a low-dimensional ordination for a set of objects with pairwise distance information preserved as far as possible.

Alternatively, documents can be considered as nodes in a graph. Hence, methods for visualizing graphs (see Section 2.3) can also be used to visualize the corpus of documents.

VxInsight [Davidson et al., 1998] uses a combination of energy-based and force-directed placement. First an ordination is calculated using an energy-based algorithm. This ordination is refined using force-directed placement. Figure 2.8 displays a set of bibliographic records. The height and shape of the mountains characterize the number and position of data objects. The color of a mountain does not contain additional information, but serves to increase the visual perception of height. In Figure 2.8, the labels on the mountain peaks display the name of the journal whose articles are the most common in that area.

WEBSOM [WEBSOM, 2000; Kohonen et al., 2000] uses self-organizing maps to position documents. Self-organizing maps [Kohonen, 2000] use neural networks to produce a graph of input data. Figure 2.9 shows the visualization of over million postings from over 80 newsgroups. The color characterizes the density or the clustering trend of documents. Light (yellow) areas indicate higher density clusters and dark (red) areas empty spaces (ravines). One disadvantage of using self-organizing maps is the often extensive training of neural networks required to produce good results.

TileBars [Hearst, 1995] visualizes query results to provide additional information (see Figure 2.10). Each document is analyzed and represented by a series of rectangles corresponding to topical



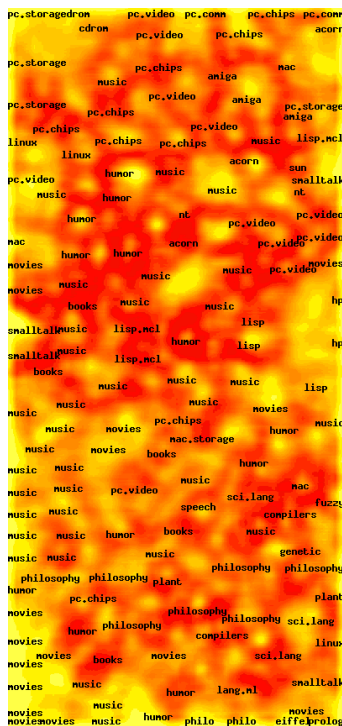


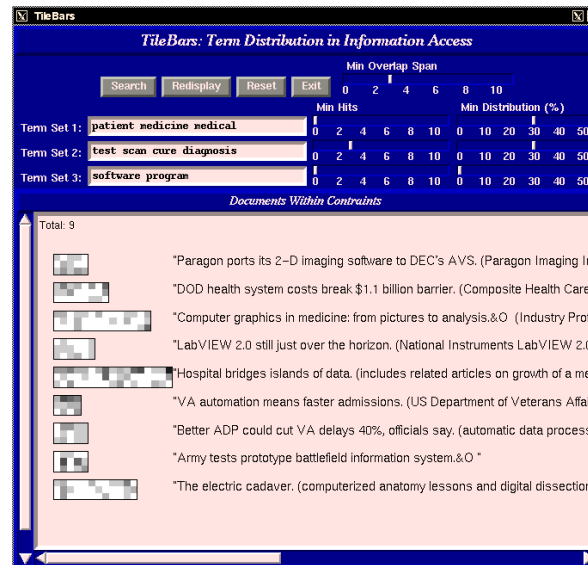
Figure 2.9: Websom displaying over a million postings from over 80 newsgroups.

units in the document. The horizontal length of the rectangle corresponds to the size of the document. Each row in a TileBar visualization represents the level of matching of one term from the query. Shaded squares indicates the occurrences of the query terms. The position corresponds to the part of the document and the darkness indicates the number of hits in that part of the document.

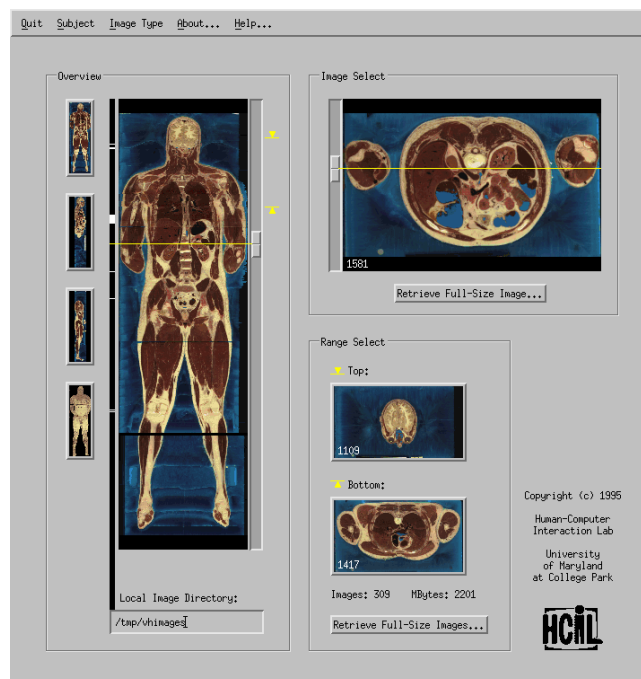
## 2.5 Spatial Information

Spatial information is information which has inherent 2d or 3d geometry. The visualization of spatial information tends to be scientific visualization since it often involves the representation of natural objects.

For example, the Visible Human Project provides an interface to a digital model of a human body. Interactive visualization systems have been built based on this database. The Visible Human Explorer (VHE) [North et al., 1996], for example allows users to remotely explore this library. A miniaturized version of the dataset provides an overview of the library content. The VHE supports downloading full resolution images on demand. In Figure 2.11, the left overview displays a longitudinal front-view cut of the body, to give users a general understanding of the contents of the body. The right viewing window displays an axial section of the position of the horizontal indicator on the overview. Using a vertical slider allows this indicator to be moved to sweep the entire body.



**Figure 2.10:** TileBars visualizing query results. Columns represent topical units of a document and rows correspond to query terms. The darkness reflects the number of matches of a query term in a particular topical unit. [Image extracted from Proc. of CHI'96. Copyright by the Association for Computing Machinery, Inc.]



**Figure 2.11:** The user interface of the Visible Human Explorer. The front-view of the body in the left window acts as overview and the preview image in the right window displays an axial image cut at the current position of the indicator of the overview. Moving the indicator using sliders allows exploration of the entire body. [Copyright University of Maryland, all rights reserved.]

## Chapter 3

# Visualizing Hierarchies

Hierarchical structures are quite common in our society. The file system on a computer or the organization of employees in a company are only two examples of hierarchically structured information. Hierarchical structuring is a powerful method for organizing huge information spaces.

A hierarchy (or tree) has an inherent structure in which items or nodes have a single parent. The root node is the only item not having a parent. Nodes can have child nodes and sibling nodes. Sibling nodes are nodes having the same parent. Each node is reachable along a unique path from the root of the hierarchy.

The amount of hierarchically structured information is rapidly increasing. For example the hard disk of a ordinary home PC contains many thousands of files organized into a hierarchy of folders. So users need powerful tools to understand and manage large hierarchies.

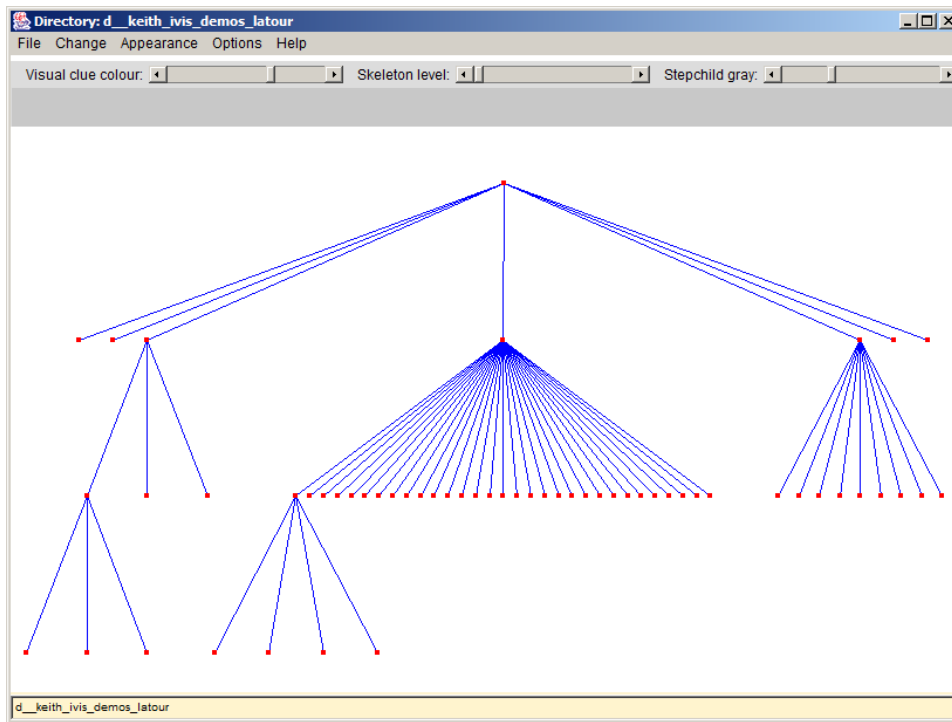
A common way to visualize trees is to use an outline method. This technique is useful for small compact hierarchies which can be drawn in their entirety on a computer screen. For large trees this technique fails due to the amount of scrolling and relatively small visible part of the hierarchy.

Several methods have been developed for visualizing large hierarchies. Some of them use three-dimensional graphics. Using the third dimension, more information can be visualized in the same space. This chapter describes some selected approaches for visualizing hierarchies.

### 3.1 Classic Tree Drawings

The classic tree drawing algorithm for ordered binary trees was formulated by Reingold and Tilford [1981]. The tree is drawn upwards from bottom to top. The y-coordinates of leaf nodes are determined by their level in the tree. The x-coordinate of leaf nodes is initially an arbitrary value. Each subtree is drawn independently. After drawing, the right subtree is shifted so that it is placed as close as possible to the left one. The position of the parent node is determined as the average of the x-coordinates of the child nodes. The y-coordinate is again given by its level in the tree. After placing nodes, the edges are inserted.

To draw trees of unbounded degree this algorithm is adjusted by traversing the children from left to right. The subtrees are placed and shifted according to the order of the children. Since the algorithm tries to place subtrees as close as possible to each other, small subtrees are placed to the left between larger ones. Thus the resulting tree is not symmetrical. Walker II [1980] improves the algorithm of Reingold-Tilford to space out small subtrees in a balanced way. Figure 3.1 shows the classic Reingold and Tilford tree layout.



**Figure 3.1:** The classic Reingold and Tilford tree layout. The tree is drawn from bottom upwards to the top. [Image used with kind permission of Keith Andrews, Graz University of Technology.]

## 3.2 Tree Browsers

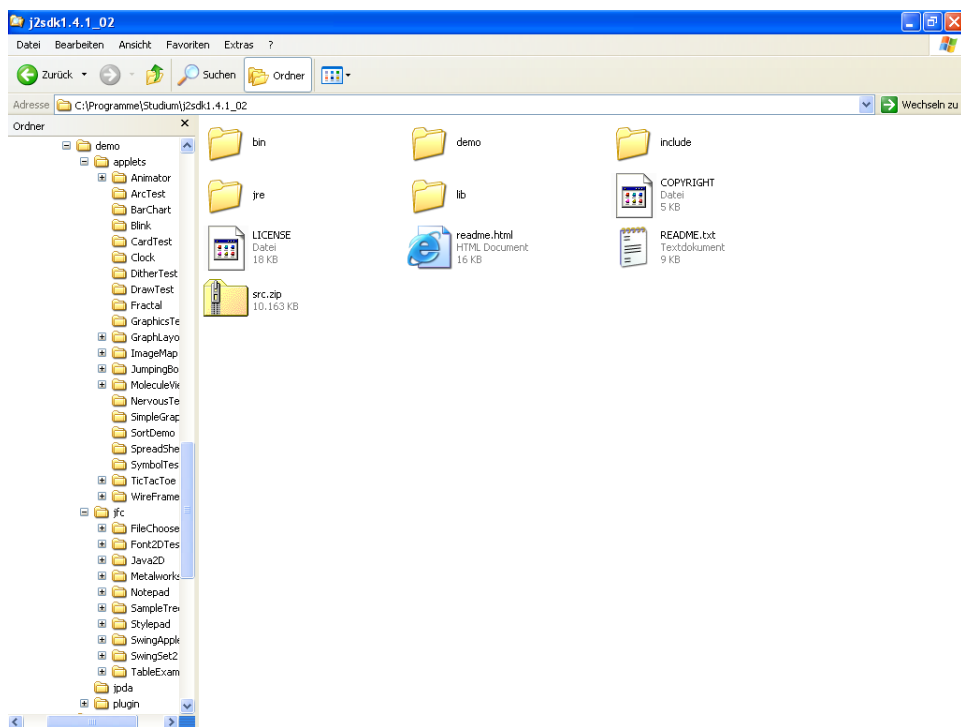
Tree browsers use an outline method to visualize hierarchies. In the initial state typically not all subtrees are expanded. Users navigate through hierarchies by expanding or collapsing subtrees. When a subtree is expanded, the displayed part grows horizontally and vertically.

Scrollbars are used as an additional navigation element, when the displayed part does not fit into the available display area. Exploring large hierarchies in tree browsers leads to a large amount of scrolling. Due to this scrolling and selective expanding, tree views does not provide a good overview of a hierarchy.

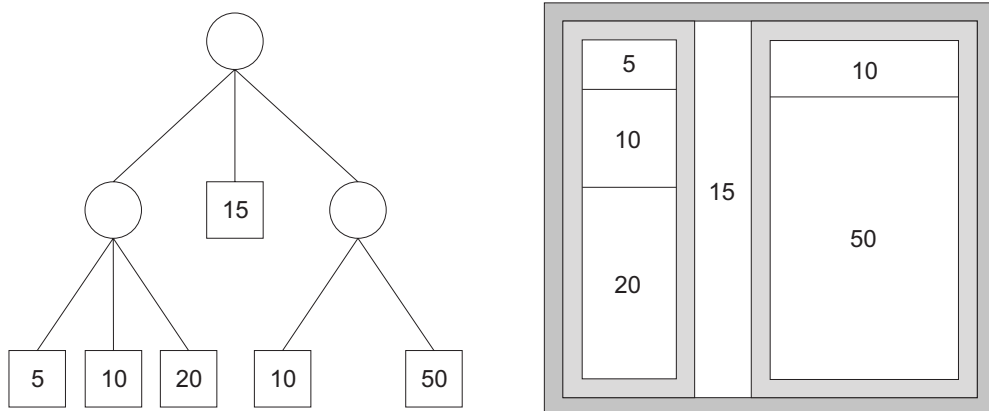
However tree views are easy to implement and are an intuitive representation of hierarchies for users. They are also usable in text-based systems. Most graphical user interfaces, such as Microsoft Windows (see Figure 3.2) and Mac OS, provide tree views as a standard graphical component.

## 3.3 Treemaps

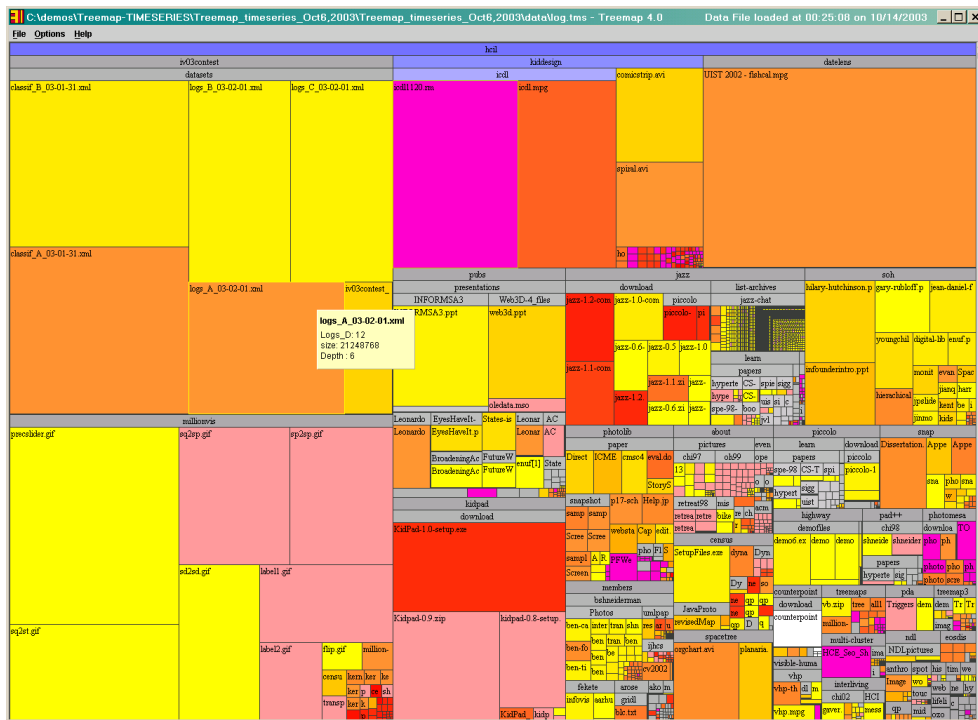
Treemaps use a two-dimensional space filling technique to visualize a hierarchy. This means that the whole display space is used. Each node of the hierarchy is represented by a rectangle. The children of an inner node are drawn inside the rectangle of the parent node. Each node is assigned a weight, often proportional to its size in some sense. The size of the drawn rectangle depends on the weight of the represented node. So the representations of nodes with higher weight are assigned more space on the display area.



**Figure 3.2:** The Microsoft Windows Explorer. A view of directories on the left side and their contents on the right side.



**Figure 3.3:** An example of the layout algorithm of Treemaps. The number in the leaf nodes indicates the weight of the node.

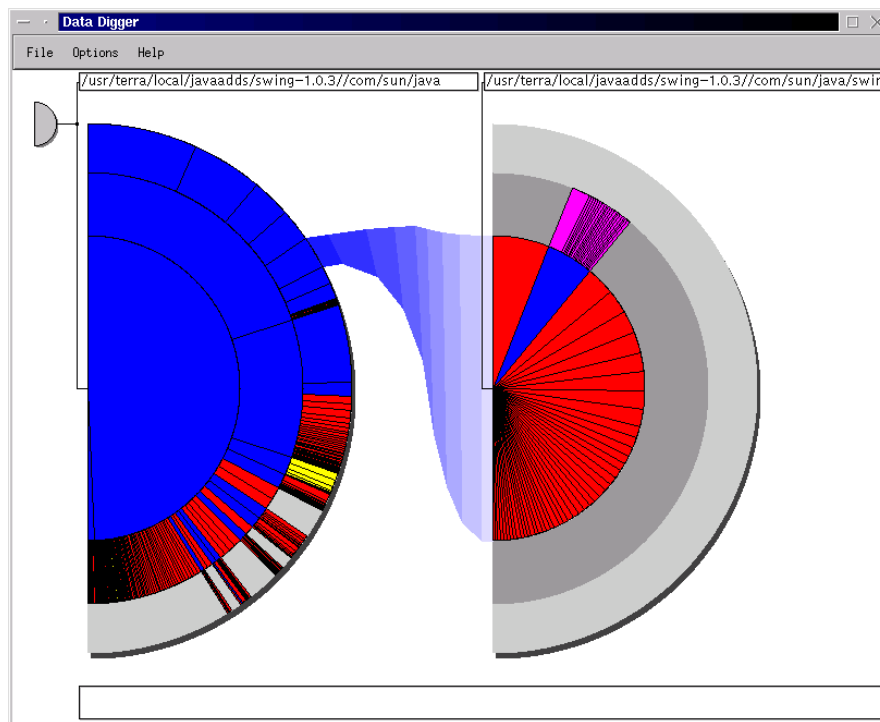


**Figure 3.4:** The Treemap 4.0 application visualizing a file hierarchy. [Copyright University of Maryland, all rights reserved.]

The layout algorithm described in Johnson and Shneiderman [1991] is a “simple slice and dice” algorithm. It starts with the root of the hierarchy. The rectangle for the root is drawn in the size of the available display space. The number of child nodes determines the number of partitions. Each partition is proportional in the size of the corresponding node. The next step is to draw vertical lines to split the rectangle. The position of the line depends on the size of the partition. The algorithm continues recursively, by altering the direction of the partition at each level. So the partitioning direction on even levels is horizontal and on odd levels is vertical. Figure 3.3 illustrates the algorithm with an example.

The simple slice and dice algorithm can often lead to long thin rectangles, hard to see and to compare in size and labels. An approach to avoid this disadvantage are Squarified Treemaps presented in Bruls et al. [2000]. The algorithm is modified to lay out rectangles in horizontal and vertical rows. For each rectangle, a decision is made to either add the rectangle in the same row or add a new row depending on the improvement in the layout. Using squarification the natural order of the elements is lost. Ordered Treemaps, introduced in Shneiderman and Wattenberg [2001] are another approach. The strip Treemap algorithm, one of the ordered Treemap algorithms, is a modification of the Squarified Treemap layout to keep the natural order of the elements.

In general, Treemaps provides a good overview of the hierarchy. They allow users to rapidly recognize large elements in the hierarchy.



**Figure 3.5:** Information slices visualizing a file hierarchy. One directory is fanned out into the right disc. [Image used with kind permission of Keith Andrews, Graz University of Technology.]

### 3.4 Information Slices

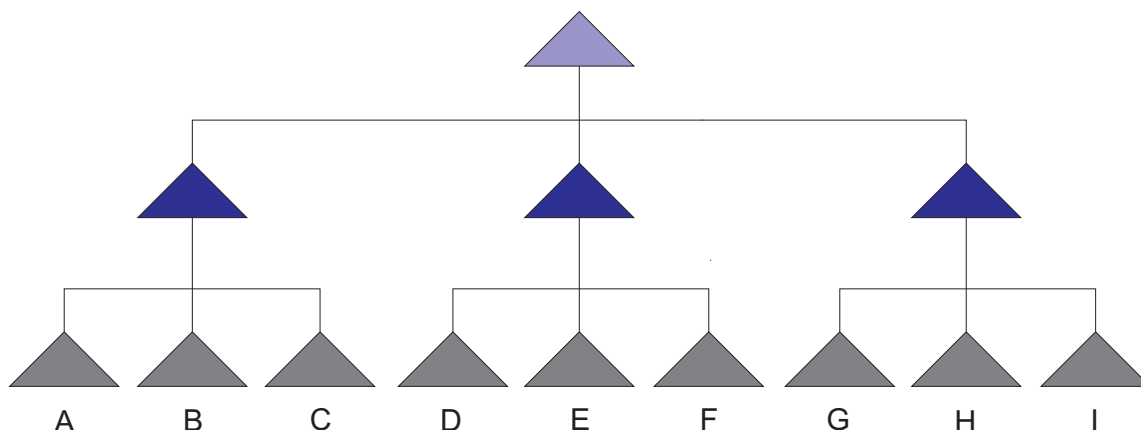
Information slices described in Andrews and Heidegger [1998] are another two dimensional technique for visualizing hierarchies. One or more semi-circular disc are used to lay out trees. Each disc shows several levels of the hierarchy. The exact number of levels is configurable by users. At each level the children are fanned out in the available space. The space used to represent a node is proportional to the size of the node. Cascading of discs allows the visualization of deep hierarchies. Figure 3.5 shows part of a file system visualized with information slices.

The user can navigate through the discs by clicking on inner nodes. When the user expands an inner node on the left disc, its contents are fanned out in the right disc. Expanding an inner node in the right disc iconifies the left disc and moves the right disc to the left. A fresh disc is opened at the right.

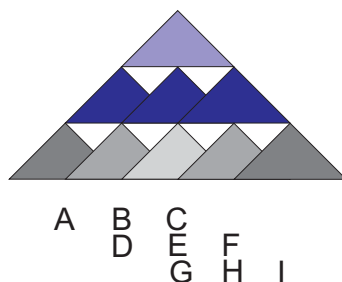
Information Slices allows users to rapidly explore deep hierarchies. Broad hierarchies may lead to extremely thin slices, which can be hard to differentiate. However, users can fan out slices of interest.

### 3.5 Cheops

Cheops introduced in Beaudoin et al. [1996] is a technique for visualizing large hierarchies. It compresses a simple tree drawing to optimize the used display space. In the Walker layout, the space necessary for drawing a tree depends on the number of nodes in the final level. This leads to unused space in higher levels and large distances between branches. Cheops uses triangles to visualize



(a) A fully expanded 3x3 hierarchy.



(b) A 3x3 hierarchy in Cheops representation. Notice the triangle in the middle of the third level represents three nodes simultaneously.

**Figure 3.6:** A fully expanded hierarchy and the corresponding representation in Cheops.

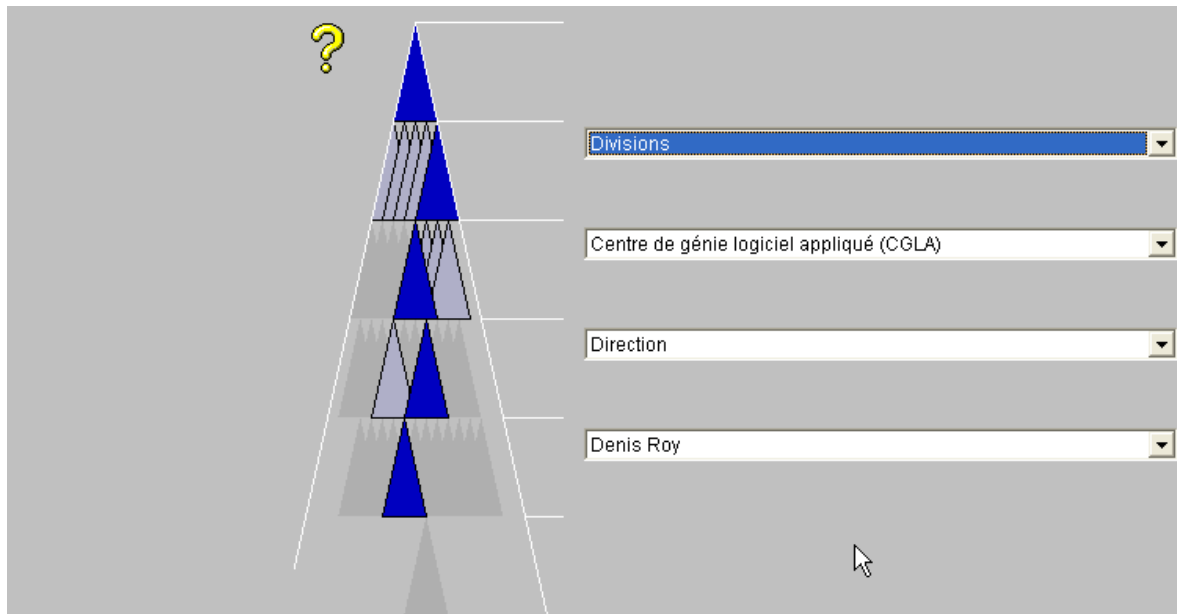
nodes. At each level overloading and multiple use of triangles is used to compress the hierarchy. Figure 3.6(a) shows an expanded hierarchy and Figure 3.6(b) shows the corresponding compressed Cheops representation. As can be seen, Cheops reduces the number of nodes in the third level from nine to five by overloading. The ambiguity of overloaded triangles is resolved when the user selects a triangle. When a node is selected, all of its parents are also selected.

Cheops uses different colors for visualizing uncle nodes, child nodes, singular nodes, overloaded nodes, and unused nodes. The border of nodes is used to indicate the focus point. Hovering over a node highlights its branches. By selecting a node, all triangles not within the selected hierarchy are set to the unused state.

### 3.6 Hyperbolic Browser

The hyperbolic browser uses a focus + context technique to visualize an entire hierarchy. The advantage of using this technique is that users can see the whole hierarchy (the context) when focusing on a particular part.





**Figure 3.7:** The Cheops visualization with choice boxes. The selected values in the choice boxes resolves the ambiguity of overloaded triangles.

A hyperbolic plane is used to lay out the hierarchy. This plane is then mapped to a circular display area. Since a hierarchy grows exponentially with increasing depth, and the circumference of circles in the hyperbolic plane also do, the available space for all nodes is approximately the same. So laying out in the hyperbolic plane is quite simple. The recursive algorithm lays out each node based on local information. Each node is assigned a wedge in the hyperbolic plane to lay out its descendants. The children are placed along an arc in equal distance in this wedge. The distance is calculated to ensure that child nodes have a minimum distance to each other.

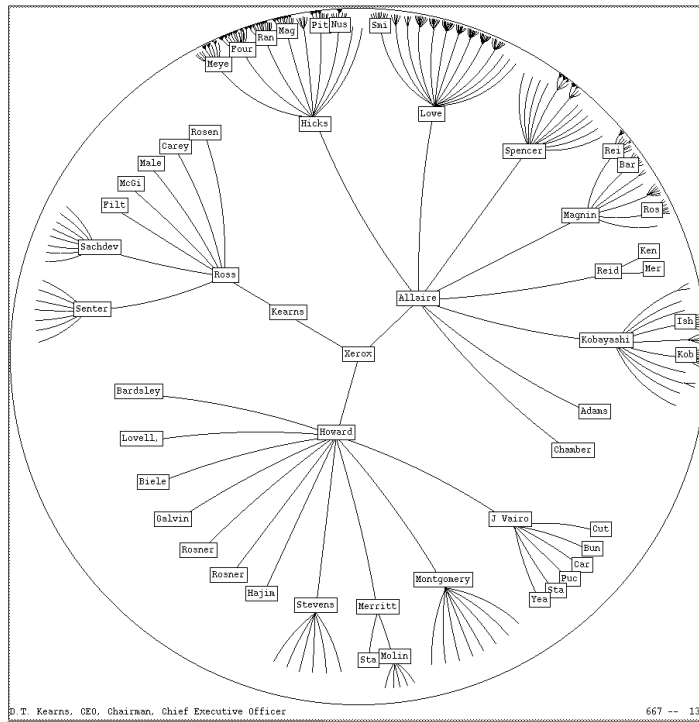
After laying out the tree in the hyperbolic plane, the structure is mapped to the euclidean plane for display. The mapping used is called the Poincaré model. This mapping preserve angles and maps lines in hyperbolic geometry into arcs in euclidean geometry.

The node having focus is displayed in the center of the display area. A change of focus adjusts the focus of the mapping to the display. So the layout in the hyperbolic plane itself does not change. To change the focus a user can click on a point or drag any visible point to another position.

### 3.7 Magic Eye View

The Magic Eye View described in Burger [1999] is another approach which uses the focus + context technique. The hierarchy is first laid out using a 2d algorithm which is similar to the algorithm described in Reingold and Tilford [1981]. The structure is then mapped onto the surface of a hemisphere. Each point on the sphere can be described by two unique angles. Thus the position of each node of the tree in the two dimensional cartesian coordinate system can be mapped directly to spherical coordinates.

The space available on the surface of a hemisphere grows toward the equator. So, for hierarchies placed with their root on the pole of the hemisphere, the space available for child nodes grows with their depth. This effect is similar to hyperbolic geometry (see 3.6).



**Figure 3.8:** The hyperbolic browser. [Image extracted from CHI'95 Electronic proceedings. Copyright by the Association for Computing Machinery, Inc.]

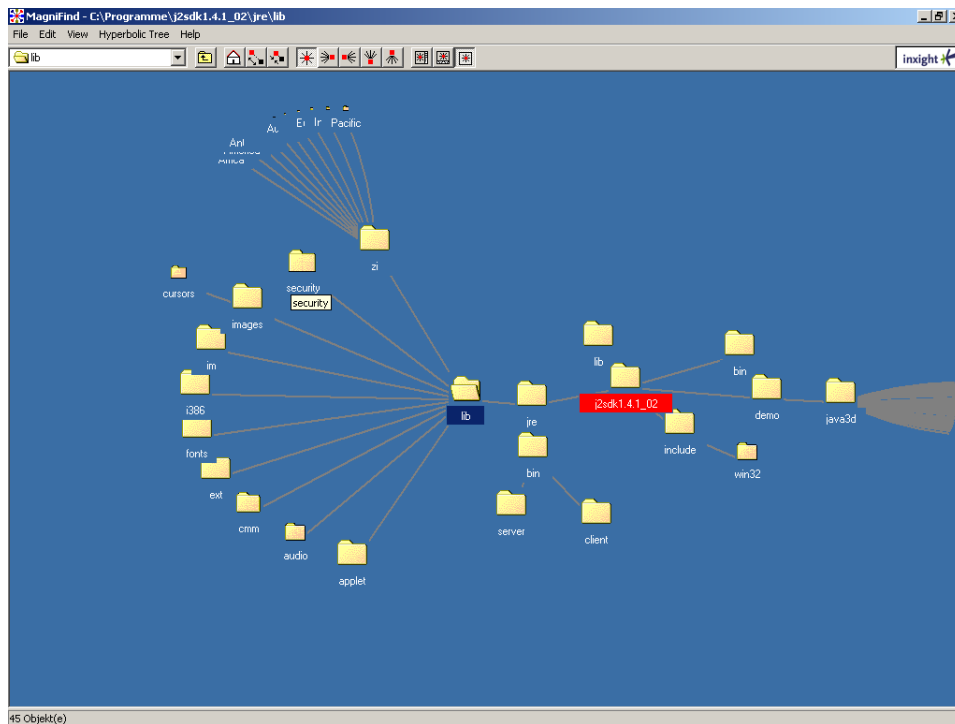
A projection is used to change the focus. For each node of the laid out tree, a ray from the projection center is computed. The intersection points of the rays with the sphere are the visual points of the graph. The projection center is initially located in the point  $(0,0,0)$ . When the projection center is moved, the rays are recalculated and the new positions of the graph are retained as the new intersection points with the hemisphere. Depending on the position of the projection center, the distance between nodes increases or decreases.

Users can change the focus by moving the center along the  $x$ -,  $y$ - and  $z$ -axes. Additionally, the hemisphere can be rotated, translated or zoomed. Figure 3.10 shows the Magic Eye visualizing a hierarchy.

### 3.8 Cone Trees

Cone trees use three dimensional space to visualize hierarchies [Robertson et al., 1991a]. Each node is represented as a card. The root of the hierarchy is drawn at the top of the display area. It forms the apex of a cone. The child nodes are drawn on the base of the cone. Each child may form the apex of a cone at the next level. All cones are of the same height, calculated as the display height divided by the tree depth. The diameters of the cones are reduced at each level, to ensure that the whole hierarchy fits into the available width. Transparency shading is used for the body of the cones, so cones in the back are visible to the user.

To improve the realism of the three dimensional space, perspective projection and lightning techniques are used. With increasing distance to the user, the size of nodes decreases and the color is darker. Shadows of cones are cast on the floor. This conveys additional information about the struc-



**Figure 3.9:** A file system visualized in a hyperbolic browser. This screen shot is produced with Magnifind, a free demo of a hyperbolic browser from Inxight [www.inxight.com](http://www.inxight.com).

ture to the user. Darker shadow implies that there are more nodes in that part of the hierarchy.

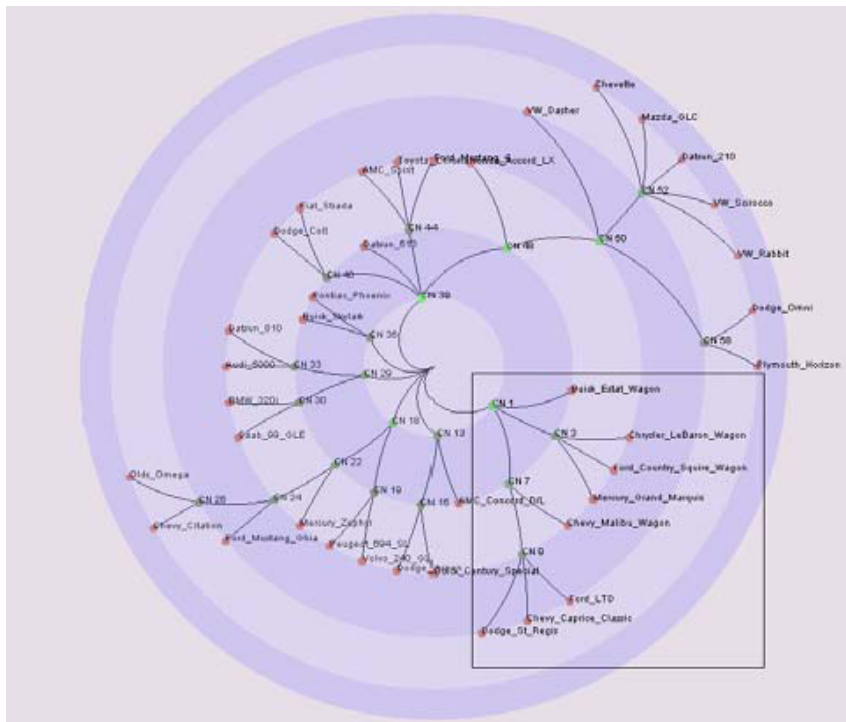
When the user selects a node, the cones are rotated to bring the selected node and all its children to the front. The rotation is animated, so the human visual perception system can track the movement. Additionally, the selected path is highlighted. The rotation of all cones in parallel is also available, to give users an overview of the hierarchy.

In the vertical cone tree layout, labels can be hard to read, and thus only selected nodes are labeled. Cam trees are an approach to solve this problem. They are laid out horizontally from the left to the right rather than from top to bottom.

### 3.9 3D Hyperbolic Browser

The 3D hyperbolic browser (H3) [Munzner, 1997] is another hierarchical visualization technique using three dimensional space. H3 first creates a spanning tree by using domain-specific knowledge for the input graph. So H3 supports visualizing general graphs. The basic layout scheme for the tree nodes is then similar to the recursive drawing algorithm of cone trees. Child nodes are mapped onto the surface of a sphere.

H3 takes advantage of the hyperbolic space in which the surface of the hemisphere grows exponentially with increasing radius. So there is always enough space for the layout. To map the hyperbolic space to the Euclidean space the Klein model is used for projection. It preserves straight lines but distorts angles. Objects near the center are in full size. Approaching the edge of the sphere the projected size shrinks, providing a way to focus a point of interest while preserving a large amount of context.



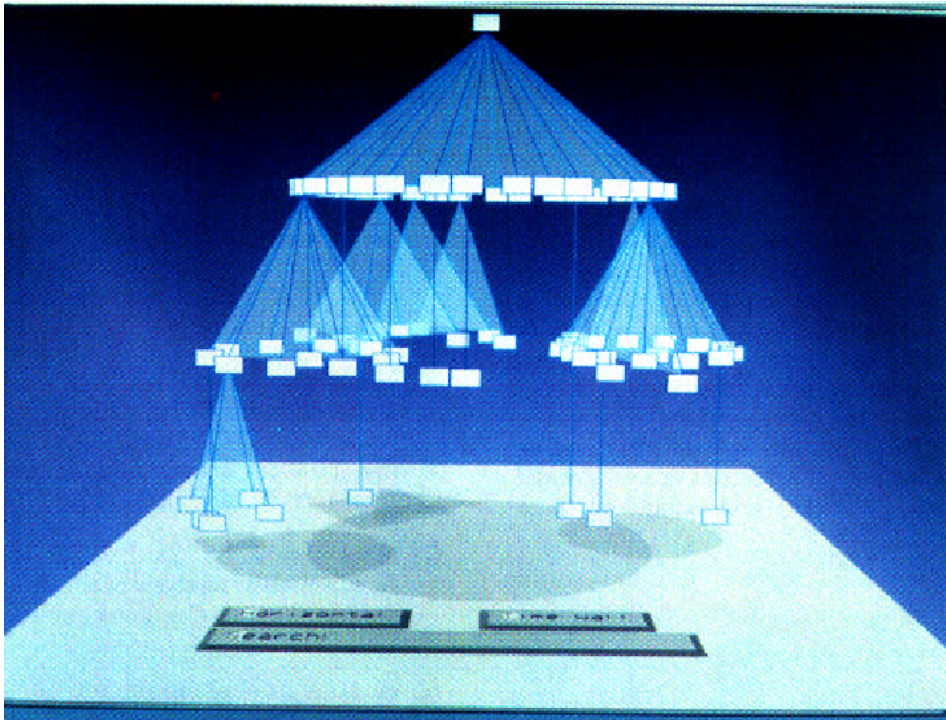
**Figure 3.10:** The magic eye viewer. A geometric mapping is used to project the 2d tree layout onto the surface of a hemisphere. [Image extracted from Proc. of NPIV'99. Copyright by the Association of Computing Machinery, Inc.]

Clicking on a node in H3 highlights it and moves it to the center of the hemisphere. The transition is animated, so users can follow what is going on. Dragging a node from a visible point to another is also possible. To improve usability, color codings and changes of line width is available. Text labels are drawn for nodes where screen areas are greater than a user-defined minimum size.

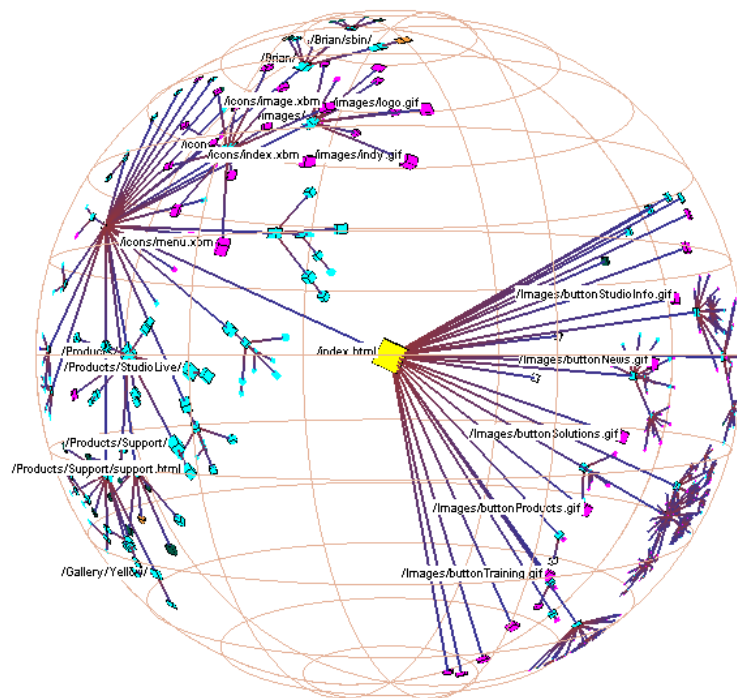
### 3.10 File System Navigator

The File System Navigator (FSN) [Tesler and Strasnick, 1992] uses a three dimensional landscape technique to visualize a file system. Pedestals are used to visualize directories. The height of each pedestal corresponds to the number of files in the represented directory. Files are visualized as boxes on top of the pedestals. Subdirectories recede into the background. The height of the boxes denotes the size of the represented file. Color coding of boxes is used to distinguish the age of files.

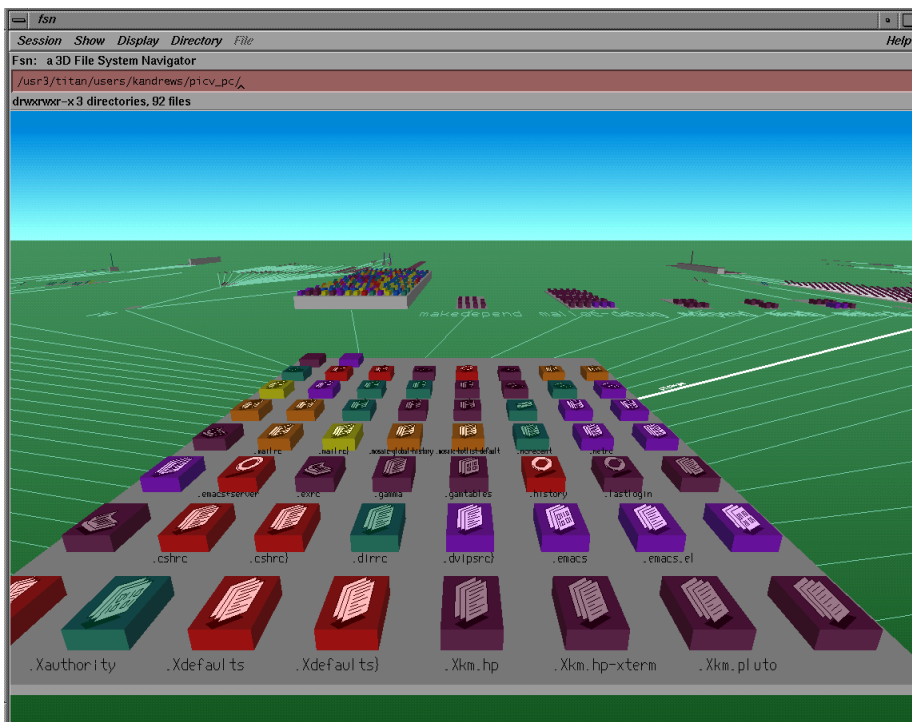
Users can freely navigate through the displayed landscape. An overview map of the landscape is also available. By clicking on the boxes users gain access to the files. It is possible to search for files and highlight them with a spotlight. Zooming to selected elements is another feature of the FSN. Figure 3.13 shows a screen shot of the File System Navigator.



**Figure 3.11:** A hierarchy visualized using a ConeTree. [Image extracted from Proc. of CHI'91. Copyright by the Association for Computing Machinery, Inc.]



**Figure 3.12:** The H3 3d hyperbolic browser.



**Figure 3.13:** A screen shot of the FSN landscape visualizing a file system. The pedestals represents directories, files are on top of them. Subdirectories are arranged in the background. [Image used with kind permission of Keith Andrews, Graz University of Technology.]

## Chapter 4

# Visualization Toolkits

In recent years numerous techniques have been successfully developed to visualize many kinds of data. To perform powerful exploration of information spaces, users need adaptable applications. Multiple view environments allows users to visualize complementary aspects of data. Such a multiple view environment only makes sense if the views are coordinated.

Another aspect of using toolkits is to reduce the implementation costs of visualization applications. By providing standard components in toolkits, such as range sliders, filter methods and dynamic queries, the work involved in implementing a visualization application is reduced.

### 4.1 Snap Together

Snap Together Visualization [North and Shneiderman, 2000] is a user interface for creating and coordinating multiple view environments. Multiple views are often required in exploring data. Microsoft's Windows Explorer for example uses three visualizations to browse the file system: an outline view to visualize folders, a tabular view for files and a small detail view for details of a selected file. The combination of these coordinated views are not changeable. Snap visualization enables end users to combine and coordinate different views to create a new coordinated visualization which adapts their needs.

The conceptual model of Snap is based on the relational database model. The relationship between Snap's concept and a relational database is as follows:

Relational Concept		Snap Concept
Relation	=	Visualization
Tuple	=	Item in a Visualization
Primary key	=	Item ID
Join	=	Coordination

Users typically select a tuple or navigate to a tuple in a visualization. A tuple can be identified by a primary key. So in Snap's terminology these actions are called primary key actions. When a user initiates an action via input, the view responds with a visual feedback. For example, when a user selects a dot in a scatter plot, the selected dot is highlighted.

Foreign key actions, such as load on demand, are often available in visualizations. Snap is able to manage these actions as well. When users coordinate views with Snap, they first load relations into visualizations. After loading, users choose actions in each view when coordinating a pair of views.

The actions of visualization are coupled by their related tuples by the join of relations. There are four possible combinations of join relationships:

- **One-to-One:** This relationship describes a primary key to primary key action. When the user invokes a primary key action on a tuple in one view, the system automatically invokes the latter action on the corresponding tuple in the latter view.
- **One-to-many:** A primary key action in one view can be coordinated with a foreign key action in the other view.
- **Many-to-many:** This relationship is composed by two one-to-many relationships. In Snap users select two one-to-many relations in the desired directions.
- **No relationship:** When there is no relationship between relations, there is no coordination between views.

With the approach of a relational database, coordination of visualizations is reduced to a simple data-relationship problem. Common coordination actions available in Snap are:

- **Brushing and linking:** A selected item in one view highlights the corresponding item in another view. This coordination describes a one-to-one join relationship. Typical usage includes identifying an item when displaying a set of items in different views with different contexts.
- **Overview and detail view:** This one-to-one relationship invokes the detailed view to scroll to the selected item of the overview. Typically items in an overview are visualized smaller and with less detail. This coordination allows direct access to details while providing the context in the overview.
- **Drill-down:** Selecting an item in one view loads the related data in another view.
- **Synchronized scrolling:** Scrolling is coordinated between two views. Scrolling to an item in one view scrolls to the corresponding item in the other view.
- **Details on demand:** When the user selects an item in one view, the corresponding item is loaded and displayed in the other view. The relationship is therefore one-to-one.

Snap's open API is designed to integrate independent visualizations of developers and researchers. Only a few hooks are defined in the API:

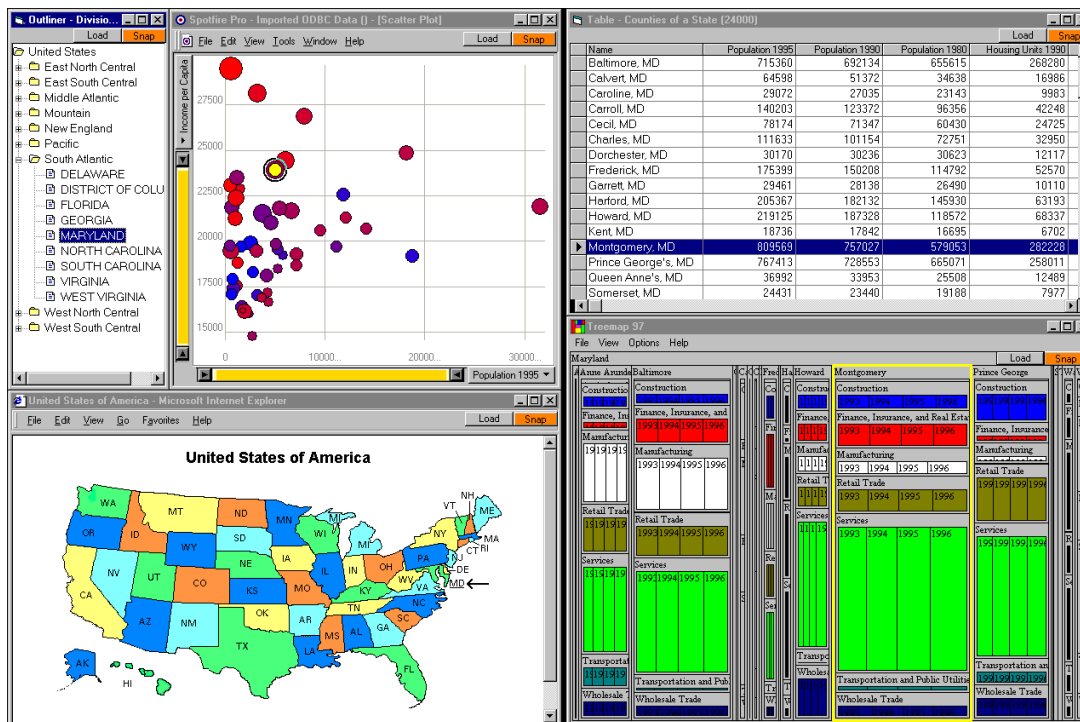
- **Load method:** Snap must be able to send the data to visualizations, when the user loads relation into it.
- **Methods for each primary key action:** When the user invokes an action in the view, this action must send an event to Snap. For Snap it must also be possible to invoke an action in the view.

Figure 4.1 shows a coordinated visualization environment created with Snap.

## 4.2 Visage

Visage [Roth et al., 1996] is a user interface environment for exploring and analyzing information. It is based on an information-centric approach to present information to the user. The information-centric architecture may be thought of as the next step along the path from an application-centric architecture to a document-centric approach in information visualization.





**Figure 4.1:** A multiple view environment created with Snap. In the left the states of the USA are visualized using an outliner, a map and a scatter plot. In the right the table and the Treemap displays detailed information about the selected state. [Copyright University of Maryland, all rights reserved.]

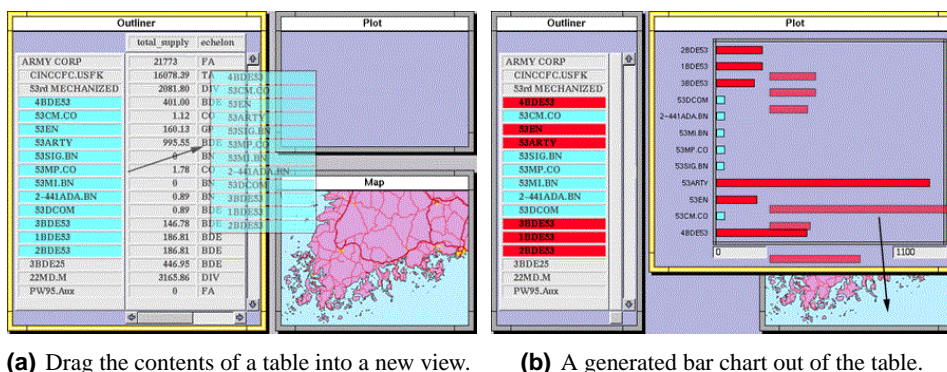
Application-centric architectures use the file as the basic currency. Basic currency means the central focus of user interaction with the system. In fact information resides inside files; files themselves are little of use to the user. To gain information users rely on the use of applications. Each application defines what kind of files can be manipulated by its interface. The contents of files are still out of reach of the user.

In document-centric architectures, the basic currency are documents and no longer files. Documents have some potential meaning in the user’s world. Users are able to manipulate documents directly, so the role of applications is subordinated. Documents may act as containers for other documents, so grouping of meaningful units is possible.

The information-centric approach used in visage is the next step in this trend. Visage uses data elements as the basic currency. So direct manipulation of data at any level is permitted. Visage allows users to drag and drop objects, representing information, between visualizations and applications throughout the visage environment. For example, the user can select parts of a table and drag them into a bar chart display, where the selected data is then visualized (see Figure 4.2). Integrated in Visage are tools for manipulating data. These include tools for filtering and interactive partitioning. Selectively combining subsets of data is also possible. The level of detail is controllable using drill-down and roll-up techniques.

The Visage environment consists of three basic objects:

- **Elements:** In the terminology of Visage, elements are atomic, manipulable graphical objects. Elements are, for example, a point in a scatter plot or a numeric value in a spreadsheet cell.



**Figure 4.2:** Creation of a new view by dragging in visage. [Figures taken from Visage [2004] and used with kind permission of Peter Lucas.]

Each visual element represents exactly one item in the underlying database. However the same object in the database can be represented by multiple visual elements, each in a different view. This one-to-many relationship works as brushing-and-linking coordination between different views.

- **Frames:** Frames act as a pasteboard for elements in Visage. They are lightweight components and provide a grouping function for related elements. Frames are themselves elements and are directly manipulable like atomic elements. They also serve as anchor points for scripts.
- **Scripts:** Scripts are frequently used in Visage. Basic user events are processed by user accessible scripts. Each element in Visage can have an attached script. Merging and filtering the objects in the underlying database is also managed by scripts.

Users can dynamically create visualizations in Visage. Therefore Visage is combined with SAGE [Roth et al., 1994]. SAGE is a knowledge-based automatic graphic design tool.

### 4.3 SinVis

The SinVis Framework [Kreuseler et al., 2000; Kreuzeler and Schumann, 2002] is a scalable framework for visualization of complex data sets. Large unstructured data sets require preprocessing to reduce their size to manageable levels. Filtering out uninteresting items and grouping of similar objects are used. The SinVis Framework provides interactive filtering, clustering, dynamic hierarchy computation and neural networks for preprocessing the information space.

The preprocessing pipeline in SinVis offers interactive user-driven approaches and algorithmic computational procedures to determine the relevant information. User-driven preprocessing procedures are useful because they allow direct consideration of the user's knowledge. The following interactive user-driven methods are integrated into SinVis:

- **Reduction of the number of dimensions:** Users select the most relevant dimensions in visual previews in a tool called the DataTableView. This view is very similar to the Table Lens [Rao and Card, 1994]. It extends the Table Lens by the possibility of grouping entries according to their similarity.

- Filtering out data ranges: Using sliders, users can set data range filters to reduce the information space to relevant items.
- Interactive hierarchy specification: Users can impose arbitrary hierarchical organizations on a given information set. This feature is only available when the information set is not a natural hierarchy. Thus user specific knowledge is used to obtain structures and patterns in data.

Exploiting the similarities of objects in high-dimensional information space is the task of algorithmic methods. SinVis offers:

- Self-organizing maps: extract groups of similar objects. They can be described as a nonlinear projection from an n-dimensional space onto a two-dimensional space. Self-organizing maps provide a useful topological arrangement of information objects in order to display clusters of similar objects. Self-organizing maps are suitable as an overview of an entire collection.
- Dynamic hierarchy computation: a method to achieve a structuring of unstructured information. Using agglomerative clustering algorithms [Kaufman and Rousseeuw, 1990] objects are merged into hierarchical groups according to the similarity between objects and the dissimilarity between groups of objects.

To support different exploration tasks, SinVis offers several visualization techniques. Parallel Coordinates [Inselberg and Dimsdale, 1990], KOAN [Panyr et al., 1996], the Magic Eye View (see 3.7) and ShapeVis are visualization tools of this framework.

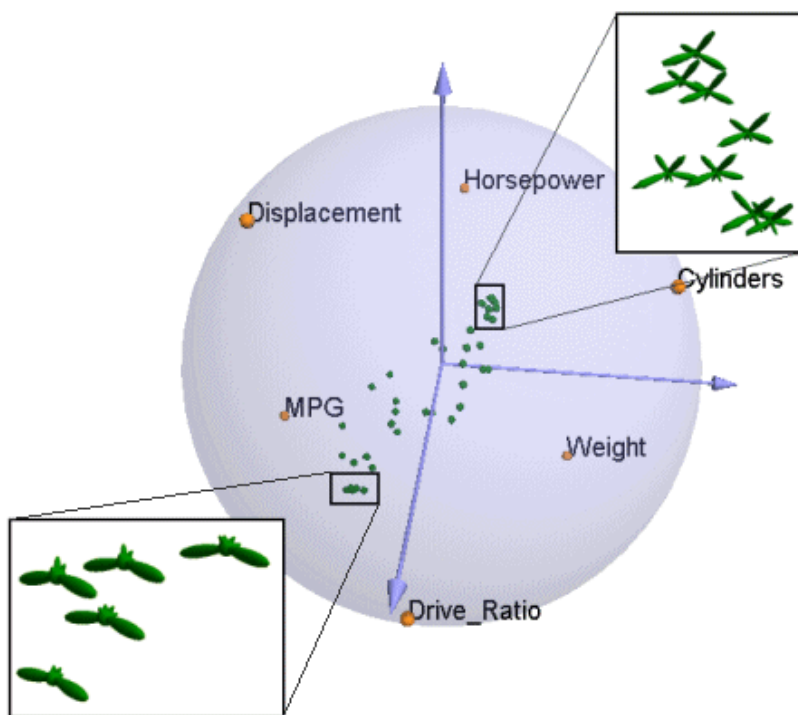
ShapeVis is a visualization tool for exploration of multi-dimensional information. It is based on a spring model. Every dimension of the information space is related to a point  $d_i$  in the visualization space. An information object is mapped to a point  $p$  in a visualization space using springs. Each point  $d_i$  is connected with a spring to the point  $p$ . The stiffness of every spring represents the size of the attribute. The location of the point  $p$  is determined where the spring model is in balance. Objects with different properties may be allocated the same point in the visualization space. In order to solve this problem, ShapeVis (see Figure: 4.3) represents an information object with a small shape rather than a single point. So two objects with different attributes are visualized with different shapes.

## 4.4 Polaris

Polaris [Stolte et al., 2002] is a visualization environment for exploring and analyzing large multi-dimensional databases. Polaris reduces the characterization of fields in tables to two types: quantitative and ordinal fields. Fields in the relational table are partitioned into dimensions and measures. Independent variables in traditional analysis are similar to dimensions and dependent variables to measures. Polaris treats all ordinal fields as dimensions and all quantitative and interval fields as measures.

To support the analysis process Polaris offers:

- Data-dense displays: Visualizations that simultaneously display many dimensions of large subsets.
- Multiple display types: Displays for discovering correlations between variables, finding patterns and uncovering structure.
- Exploratory interface: Rapid change of the viewing data.



**Figure 4.3:** Example of a ShapeVis visualization. The visualized objects are cars with different metadata attributes such as horsepower and cylinders. Objects are placed within a sphere according to their affinity with the six endpoints marked in yellow. The extent of each shape in each dimension indicates the relative affinity of the object to each dimension. [Figure taken from ShapeVis [2004], University of Rostock.]

Polaris provides the following interaction techniques:

- **Deriving Additional Fields:** Simple aggregation such as summation, average, minimum, and maximum can be applied to single fields. Another aggregation function is the counting of ordinal dimensions. Discrete partitioning is used to discretize a continuous domain. Ad hoc grouping allows the user to group different ordinal values for display and data transformations. Threshold aggregation lets the user specify threshold values below which data is uninteresting. This uninteresting data is aggregated into a new category.
- **Sorting and Filtering:** Filtering allows the user to reduce the amount of items displayed and devote more display space to areas of interest. For ordinal fields users can check or uncheck values to display. Dynamic query sliders are used for quantitative fields.
- **Brushing and Tooltips:** Brushing allows users to mark a set of interesting items by surrounding them with a rubber band. Related marks and tuples are identified by a single selected field. All corresponding items are then highlighted in all other panes. Tooltips provide details on demand to the user by hovering over a data point.
- **Undo and Redo:** Unlimited support for undo and redo are provided in Polaris. This allows users rapid, incremental, and reversible exploration of the information space.

Polaris is built within the Rivet [Bosch et al., 2000] visualization environment. This is an environment for rapidly creating visualizations from components.

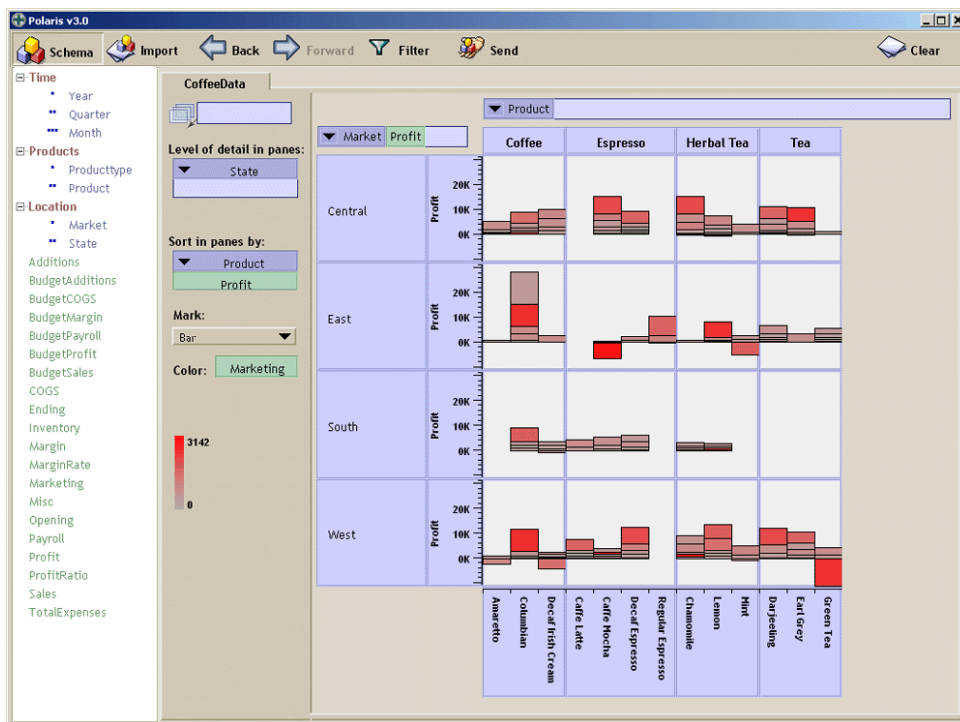


Figure 4.4: An analysis example using Polaris. [Figure taken from Polaris [2004] and used with kind permission of Pat Hanrahan.]

## 4.5 InfoVis Toolkit

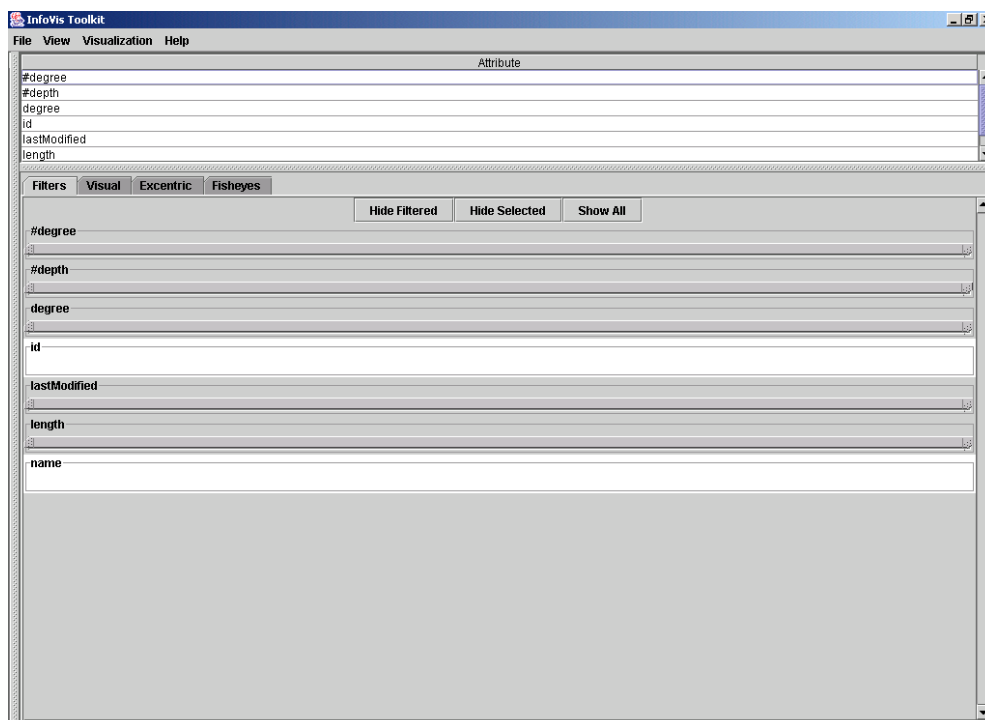
The InvoVis Toolkit [Fekete, 2003] toolkit is a Java-based software architecture for creating information visualization applications and components. Data structures supported by this toolkit are tables, trees, and graphs. Currently supported visualizations are: Scatter Plots and Time Series Parallel Coordinates for tables; Node-Link diagrams, Icicle trees, and Treemaps for trees; Adjacency Matrices and Node-Link diagrams for graphs. The main users of the InfoVis Toolkit are programmers.

The InfoVis Toolkit is organized in four main parts: tables, columns, visualizations, and input/output. The underlying data structure is based on tables. The advantage of using tables is that any data structure can be implemented on top of it. Tables consists of named columns plus metadata and user data. Each row in the table represents a record and each column an attribute.

Trees and graphs are implemented using internal columns to store topological information. The topology for a tree is represented by adding “parent”, “first child” and “next sibling” columns. Selection and filtering is also managed by internal columns. A row is selected when its value in the “selected” column is true.

The attributes stored in table columns are transformed into a visual representation by visualizations. The mapping of attributes into colors is managed by a color visualization component. This defined interface returns a color for a specific row.

To improve the performance of visualizations, redisplay is split into layout and rendering. A recalculation of the layout is only necessary when the structure of the visualized items changes. Dynamic labeling or selection, for example, do not change the layout and are redisplayed without relayout. Filtered items are grayed out for graph and tree drawings and are only removed when the



**Figure 4.5:** The control panel for treemaps in the InfoVis Toolkit.

user selects this option, since the removal operation requires a recomputation of the layout.

During rendering, visualizations maintain a column of shapes and repaint them in the computed color when required. The color of shapes is determined by the color visualization component. Coloring the border of shapes marks selected items. By default selected items have a red colored border and non-selected items have a black border. Dynamic labeling using tooltips is an optional feature of visualizations.

The InfoVis Toolkit provides several components to support interactive manipulation of visualized data. Each visualization comes by default with a control panel. It is organized in tab groups to configure and manipulate the visualization. Factories are used to couple the panel with the visualization. Thus developers can substitute the panel with their own implementation. Figure 4.5 shows the default control panel for Treemap visualizations. The InfoVis Toolkit is designed for developers to rapidly create visualizations.

## 4.6 Spotfire

Spotfire [Spotfire, 2000; Ahlberg, 1996] is a commercial environment for information visualization. One of its design goals was for users to quickly get started with their exploration and visualization tasks. Therefore it creates visualization and manipulation objects automatically. The manipulation objects are dynamic query elements. The data is stored internally in a database. An open Database Connectivity module is used to import relations.

Spotfire provides a rich collection of visualizations to users. The architecture is designed to let users attach graphical objects or glyphs to each object in the database. By default a dot object is

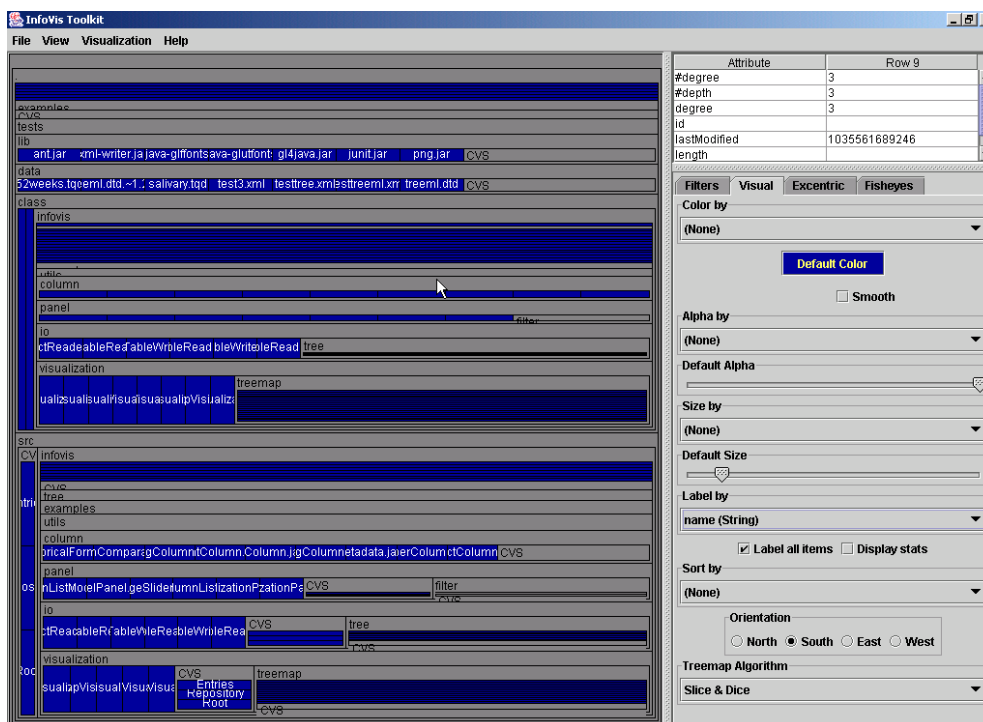


Figure 4.6: A treemap visualization from the InfoVis Toolkit visualizing a file hierarchy.

assigned to database objects.

The basic visualization in Spotfire is the Starfield, an interactive scatter plot with additional features such as zooming, filtering, panning, and details-on-demand. This visualization provides a good starting point for search and query results. Exploration of trends and anomalies is very efficient with Starfields.

User have the possibility to encode properties of database objects in different ways. Visual elements can be colored with different hues or different brightness.

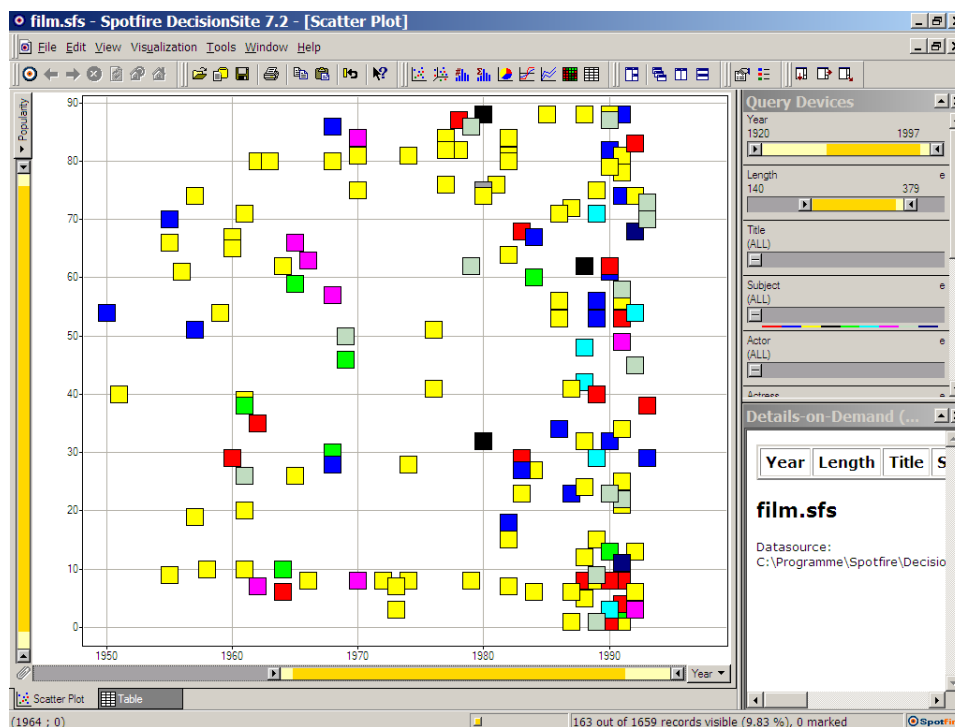
The capability to attach graphical objects to each database object allows users to extend the Starfield visualization to create user-defined views. For example, a geographical visualization can be created by providing background objects defining an appropriate background. Extending this scheme, domain specific and complex visualizations can be defined by users.

Simultaneously creating multiple visualizations and updating all of them interactively gives users a powerful exploration tool. All multiple views are coordinated and synchronized.

Based on user-specific relations Spotfire selects query devices for each attribute. The decision which query device to select is made by simple rules. The following query devices are available in Spotfire:

- Range sliders: for selecting range criteria for attributes with an order relation.
- Alpha sliders: for selecting items from a long list.
- Toggles: for selecting an alternative for an attribute, if only a few alternatives exist.

The resulting query is composed as the conjunction of all query elements. The query elements in their initial state let through all database objects. Users can display details of items by clicking on them.



**Figure 4.7:** Spotfire exploring a film database. Moving the sliders allows users to filter the information in the starfield view.

A predefined approach is used to present data in the database. Exporting this presentation as HTML documents allows users to create a rich presentation of selected data. Figure 4.7 shows a screenshot of exploring a film database using Spotfire.

## 4.7 Prefuse

Prefuse [Heer et al., 2004; Prefuse, 2004] (see Figure 4.8) is another toolkit addressed to developers for information visualization. Similar to the InfoVis Toolkit, it is written in Java and uses the Java2D API. This toolkit supports node-link diagrams, containment diagrams, scatter plots, and time lines. The primary goal of this toolkit is reducing implementation costs by combining existing techniques. This facilitates the development of visualizations. The main features of Prefuse include:

- Input modules for unstructured, graph, and tree structured information.
- Multiple visualizations of one data source and multiple views of the same visualization.
- Graphical transformations such as zooming and panning.
- Support for animations and time based processing.
- A large set of distortion and layout techniques.

To abstract data to visualize, Prefuse provides extensible interfaces for input and output. The entity is the basic data element in Prefuse. It supports any number of named attributes. These are



simply mappings from a name to a value. Node, TreeNode, and Edge are subclasses of Entity and represent structural types.

Filtering in Prefuse terminology is the mapping of abstract data to a suitable visual representation. Firstly, the abstract data is reduced to the visualizable content. VisualItems are created for this content. Each item records properties such size, location and color according to the attributes of the represented item. Three types of VisualItems are provided by Prefuse:

- NodeItems to visualize individual elements
- EdgeItems to visualize a relationship between elements
- AggregateItems to visualize a group of elements

A graph structure is used to maintain the data topology separate from the source data. All VisualItems are created and stored in an ItemRegistry. A caching approach for items and a tracking of their usage are used to support scalability. Managing focused items is also the responsibility of the ItemRegistry.

Actions are the basic component in application development. Actions are composable processing modules which update the VisualItems. The most common types of Actions are:

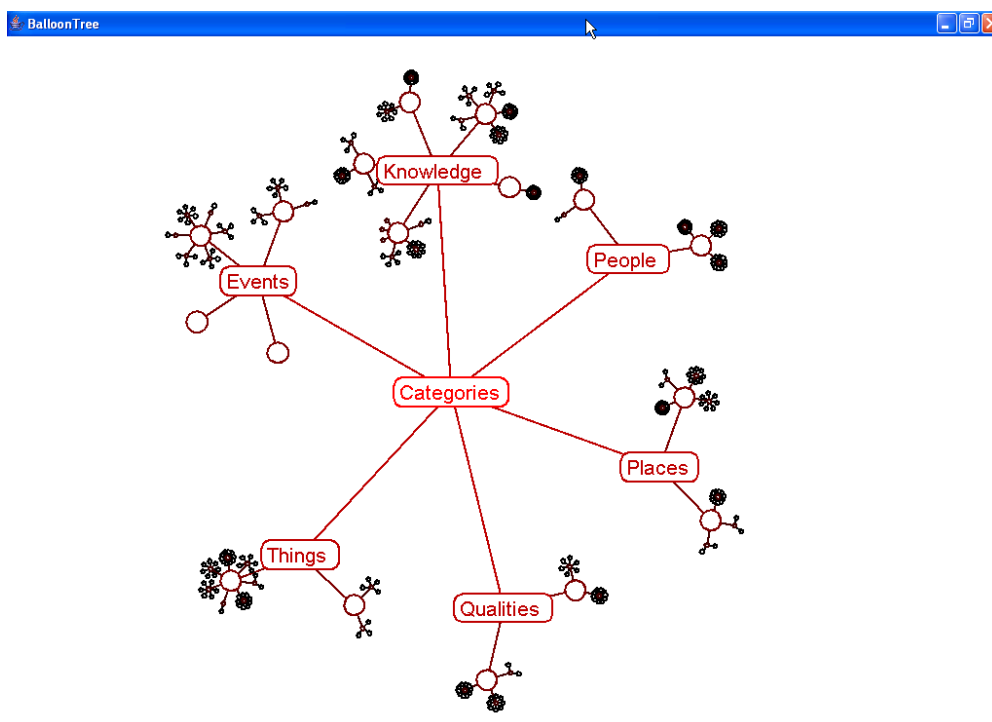
- Filter Actions which control which items are visualized.
- Assignment Actions which manipulate visual attributes, such as location, color and size.
- Animator Actions to achieve animations, such as moving an item.

Thus Actions are mechanism a for selecting visual items, color assignment, filtering, and layout. ActionLists are created by composing actions. Since ActionLists are themselves Actions they can be used as sub-routines in other lists. Each ActionList executes its Actions sequentially. It can be configured whether an ActionList is executed once or periodically. A scheduler manages the execution of ActionLists.

Renderer components are used to draw visual items. Each Renderer provides functions for drawing an item, computing the bounding box of a drawn item, and to test that a given point lies inside of an item. Default Renderer components for drawing primitives such as shapes, straight and curved edges, and text are included in Prefuse. Which Renderer to use to draw an item is managed by a RendererFactory. Thereby visual appearances can be changed by using different Renderers. A separate display component is used to present the visualized data. It acts as a camera onto the elements in the ItemRegistry. This component takes an ordered list of items and draws them by applying clipping and view transformations. With the help of the Java2D library, zooming and panning mechanism are provided.

The Prefuse library contains components providing advanced functions, which are frequently used in visualizations. These components include:

- Components for layout and distortion. Random, force-directed, top-down, radial and outline layouts are available. All layouts are parameterized and reusable.
- Force Simulation is supported by an extensible configurable library. This library provides functions for simulating n-body forces, spring forces, and drag forces.
- Interactive Controls are provided for common user interaction. These include drag controls for repositioning items, focus controls for updating focus and highlighting settings.



**Figure 4.8:** A tree structure visualized with Prefuse using a balloon tree layout.

- Color Maps are used to assign colors to data elements. These maps are configurable using provided color schemes or automatically generated by analyzing attribute values.
- Keyword search of large data sets is integrated within Prefuse. An approach that enables searching in time proportional to the size of the query string is implemented. The results are available for visualizations by managing a focused set in the ItemRegistry.

## 4.8 Information Visualization CyberInfrastructure

The Information Visualization CyberInfrastructure (IVC) [Baumgartner et al., 2003; IVC, 2004] is a general software repository supporting education and research. It is a collection of individual information visualization algorithms rather than a full toolkit. It provides an architecture in which diverse data analysis, modeling and visualizations can be run.

Different visualizations access the same data set through standard model interfaces. Supported are the standard Java data structures interfaces `TreeModel`, `TableModel`, and `ListModel`. Additionally, a `MatrixModel` and a `NetworkModel` interface are provided. A general interchangeable persistence layer is used for providing these various data models. An XML-based implementation of this layer is used to unify data input and output formats. Software packages implement and use a defined XML schema. Since this schema is in the persistence layer, these software packages can be interchanged and combined through the generated models. To integrate new code, it has to support building of one of the supported models. Thus developers have to build their code with at least one of these supported model types.

## Chapter 5

# The Hierarchical Visualization System (HVS)

As described in Chapter 4 there exist several toolkits for information visualization. Some are addressed to developers and some to end users. Most of the toolkits are designed for visualizing multidimensional data. The Hierarchical Visualization System (HVS) (see Figure 5.1) is a toolkit for visualizing hierarchies exclusively. It addresses the needs of both developers and end users.

HVS is designed to visualize all kinds of hierarchies. Furthermore users should be able to manipulate the visualized hierarchies. Whether these modifications are only simulated in the abstraction or should change the “real world” hierarchy is decidable by users.

Visualizing hierarchies in multiple views is one of the major benefits of HVS. Of course, all views have to be synchronized for exploration. HVS provides an open architecture for developing visualizations. Existing visualization tools and components can be integrated into this framework with only a minimum of modification. For example the standard tree component of the Java *Swing* package must be only extended by the functionality necessary for synchronization.

All visualizations should be created on user request. Thus, the user decides when running the application which visualizations are used to display the hierarchy. Like visualizations the visualized hierarchy should be also exchangeable while the application is running.

This chapter describes the architecture and some of the main features of the Hierarchical Visualization System.

### 5.1 The Architecture of HVS

HVS is designed to visualize all kinds of hierarchically structured information. A multiple view environment helps users in fast exploration of information spaces. Multiple views visualize the same information in more than one window. If the views are not coordinated it is more difficult for users to assimilate the visualized information. Thus a coordination mechanism is absolutely necessary. Since more than one view visualizes the same information and user interactions have to be coordinated, the model-view-controller design pattern (MVC) [Gamma et al., 1995] seems a good solution to accomplish this. The main advantage of MVC is that more than one view visualizes the same model. The MVC divides each component into three parts:

- Model  
The model that holds the data that is displayed.

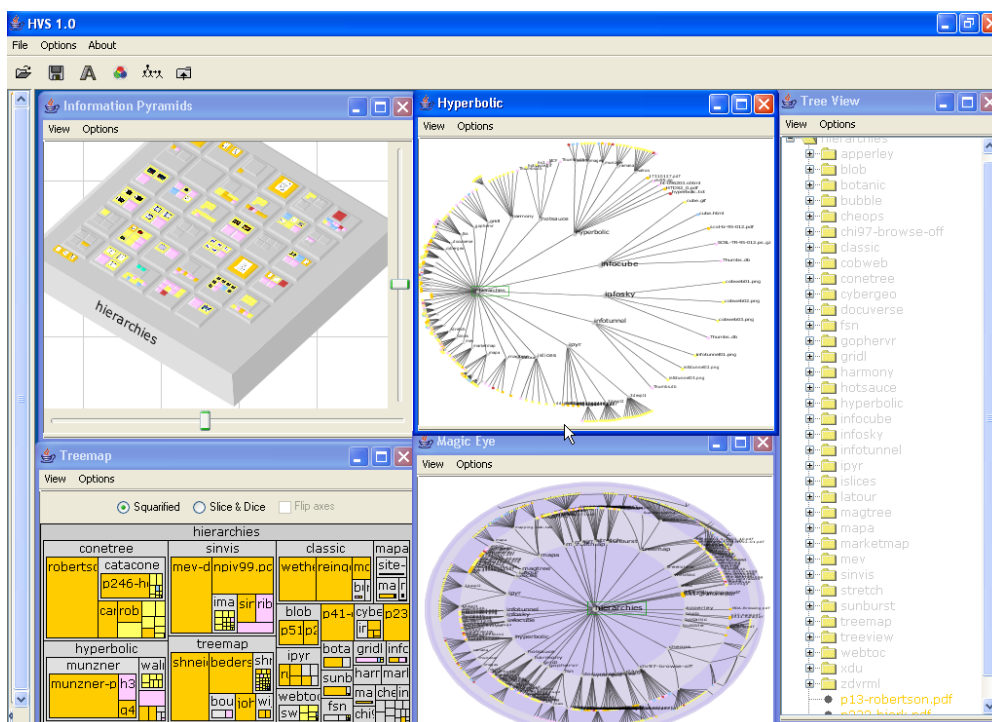


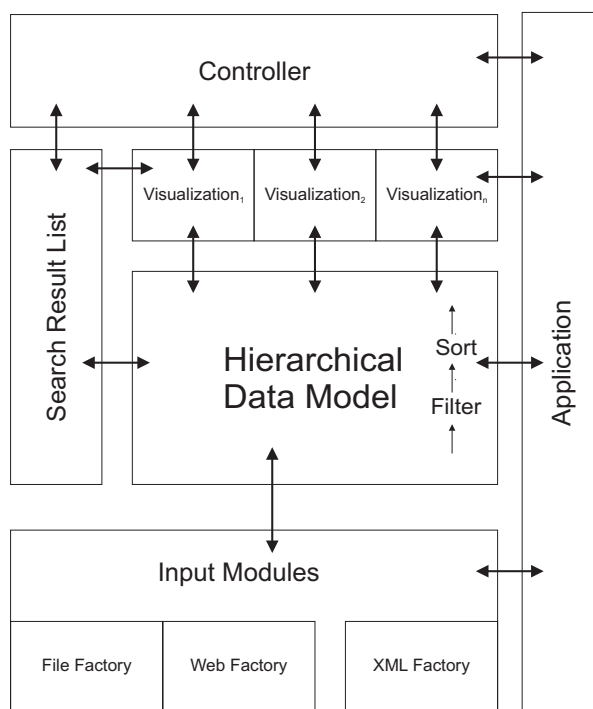
Figure 5.1: The hierarchical visualization system.

- **View**  
The view visualizes the data items of the corresponding model.
- **Controller**  
The controller reacts to user interactions. It updates the view to respond visually to the desired interaction.

Based on this MVC design pattern and the features which HVS should provide, the main components of HVS (see Figure 5.2) are:

- Input Modules
- Hierarchical Data Model
- Visualization Modules
- Controller
- Search Result List
- Application

The controller corresponds to the controller in the MVC pattern. Visualization modules correspond to views. Furthermore they act as user interfaces, since they react to user interactions. The hierarchical data model represents the model part of MVC. The next sections of this chapter describe the main modules in detail.



**Figure 5.2:** The main components of HVS.

## 5.2 Input Modules

Visualizing different types of hierarchies was one main design goal of HVS. Input modules create an abstract model of the “real world” hierarchy. Each hierarchical information model consists of two types of information:

- Content information
- Structural information

The content information in HVS is assigned directly to the nodes. HVS distinguish between inner nodes and leaf nodes. Inner nodes are represented by objects of type *Collection*. *Document* objects represent leaf nodes. Both classes are abstract. Thus the abstract methods have to be implemented when they are subclassed. Since HVS uses only those methods defined by these abstract classes, this allows easily exchanging concrete implementations. *Document* object are assigned a type. This abstraction allows a simple mapping to color attributes in visualizations. This type information can be used to group similar objects. The structural information is not assigned directly to nodes. A separate *InputDataModel* object is used for the structural information. This separation allows manipulation of the structural information without changing node objects.

The content information is not limited to standard attributes like name or size. Any kind of information can be assigned to a node. To get access to this information keys are used. HVS defines three major types of information:

**Numerical information** is information represented by numbers. For example, the number of pages in a document.

**Chronological information** is described by a date. The date when a document was modified is an example of chronological information.

**Textual information** is information represented by a text string. For example, the author of a document represents this kind of information.

Depending on these types of information, input fields for filtering and searching are created. For textual information a single input field is created. Two input fields are generated for the other two types. Thus users can set filters by defining a range. The mapping between keys and type of information is managed by a *DataSource* object. This object also defines all available keys.

Manipulating hierarchies was another design goal of HVS. Thus nodes can be inserted, removed or renamed in HVS. Whether these changes should also manipulate the “real world” hierarchy is decidable by the user. Inserting nodes requires the creation of objects of the desired runtime type. The factory design pattern [Gamma et al., 1995] is a good solution to accomplish this.

Thus each Input Module has to implement the following interfaces and abstract classes defined in the *iicm.hvs.inputfactory* package:

- *DataSource* This abstract class defines all available metadata information. Furthermore it maintains the communication between application and the hierarchy. Furthermore it declares a method to retrieve the factory object for creating the hierarchy.
- *Node* The interface *Node* defines the requirements for objects used as elements in the hierarchical data model. The declared methods to access the content information are:

*getName()* returns the textual name of this node.

*getSize()* returns a numerical value corresponding to the size of this node.

*getLogSize()* The method *getLogSize()* returns the calculated logarithmic size. The logarithmic size of an inner node is determined by the sum of the logarithmic size of the children.

*getMetaData()* This method returns the metadata associated with this node corresponding to a given key. The valid keys and the runtime returned type are managed by the *DataSource* object.

- *Collection* The abstract class *Collection* represents an inner node of the hierarchy. It implements the node interface.
- *Document* Objects of type *Document* represents leaf nodes in the hierarchy. Each leaf node is assigned with a numerical type. This type is mapped in visualizations to a color attribute.
- *InputDataModel* Hierarchically structured information contains structural and content information. The access to structural information is defined by the *InputDataModel* interface. The methods to retrieve structural information are :

*getRoot()* This function returns an object of type *Collection*.

*getSubCollections()* This function returns a *NodeList* object. The elements contained in the returned object are of type *Collection*.

*getDocuments()* This function returns a *NodeList* object. The elements contained in the returned object are of type *Document*.

*getParents()* This function returns a *NodeList* object. The elements contained in the returned object are of type *Collection*.

The *MutableDataModel* interface extends the *InputDataModel* interface for providing manipulation functionality of input modules. It defines methods for inserting, removing and changing nodes and attributes. Of course manipulating hierarchies requires notification of the visual representation. Therefore objects implementing the *InputDataModelListener* interface can be attached to the *InputDataModel* to receive notifications.

The *NodeList* class is a subclass of the *java.util.Vector* class. It provides the possibility of sorting the contained nodes. Since this object already knows its sorting order, a resorting is only done when either the sorting order is changed or an element is inserted. The sorting algorithm used in the *NodeList* is QuickSort.

- *InputFactory* As described inserting nodes requires creation of objects of the correct runtime type. Thus each input component has to provide such a factory object. The abstract class *InputFactory* declares the following methods to fulfill these needs:
  - *createCollection()* Creates a new *Collection* object representing an inner node.
  - *createDocument()* Creates a new *Document* object representing a leaf node.
  - *createInputDataModel()* This method creates an object of type *MutableInputDataModel*. This object provides the structural information of the hierarchy. It is also used for further manipulation of the hierarchy.

Further this object manages the document type information. Thus it declares methods to get all defined document types. The mapping between the abstract document type and the represented textual description used in user interfaces is also managed by this class. Opening documents with external applications is also managed by this object.

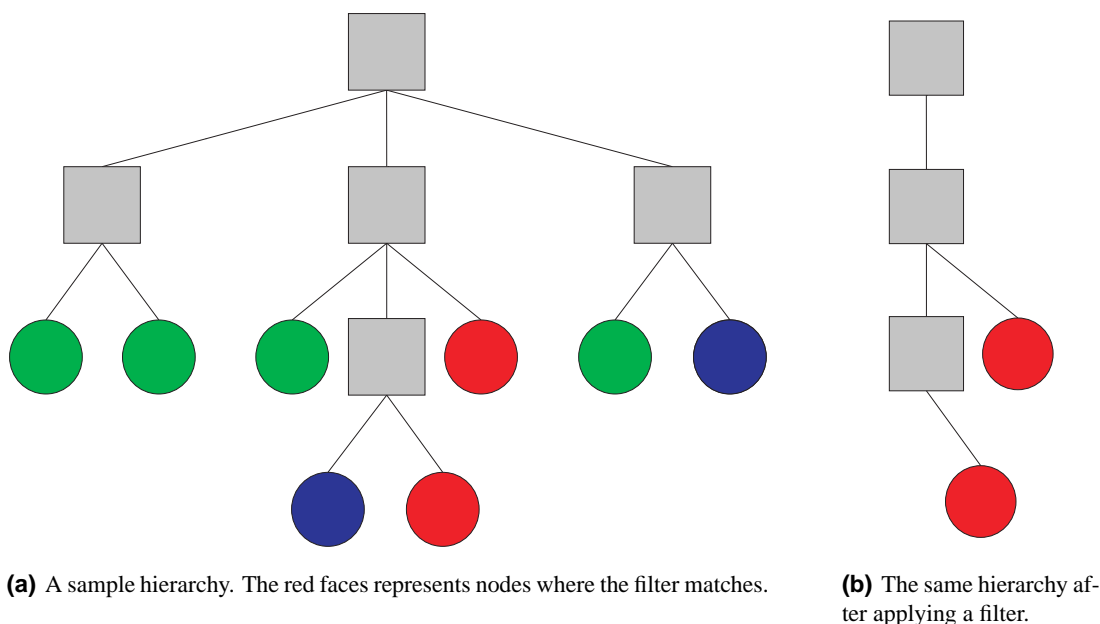
### 5.3 Hierarchical Data Model

The hierarchical data model represents the structural information to be visualized. It acts as a wrapper for the hierarchy created from the various input modules. Thus it reads from the underlying data model. It provides the functionality of sorting and filtering of nodes.

To reduce the amount of data to the parts of interest HVS supports filtering the hierarchy. Filtering means, that a new data model is created containing only elements matching a desired filter. To improve the visibility of interesting parts of the hierarchy empty paths are removed from the filtered data model. Figure 5.3 illustrates the filter strategy in HVS.

This filtering strategy changes the visualized hierarchy dramatically. To ensure that users can track these changes, the filtering is animated. The data model visualized at the beginning of filtering contains only the root node. The hierarchy is traversed in using a breath first search algorithm. When a node matches the applied filter, it and all of its parent nodes are inserted. Of course only those nodes are added to the “filtered” data model, which are not already contained. All views are notified of these changes to update their visual content. Since this procedure requires a certain time, filtering runs in a background thread.

Filtering only changes the structural information of inner nodes. Leaf nodes and the reference to parent node objects are unchanged. So the resulting filtered data model needs only to maintain the new references to child objects. Information about parent objects can be retrieved out of the underlying origin data model. Since the data model is designed as an indirect data model and the node objects do not contain structural information, no objects have to be created during filtering. Since both the structural information of the filtered and that of the origin data model co-exists at the same time, it is easy to switch between them.



**Figure 5.3:** The HVS strategy for filtering.

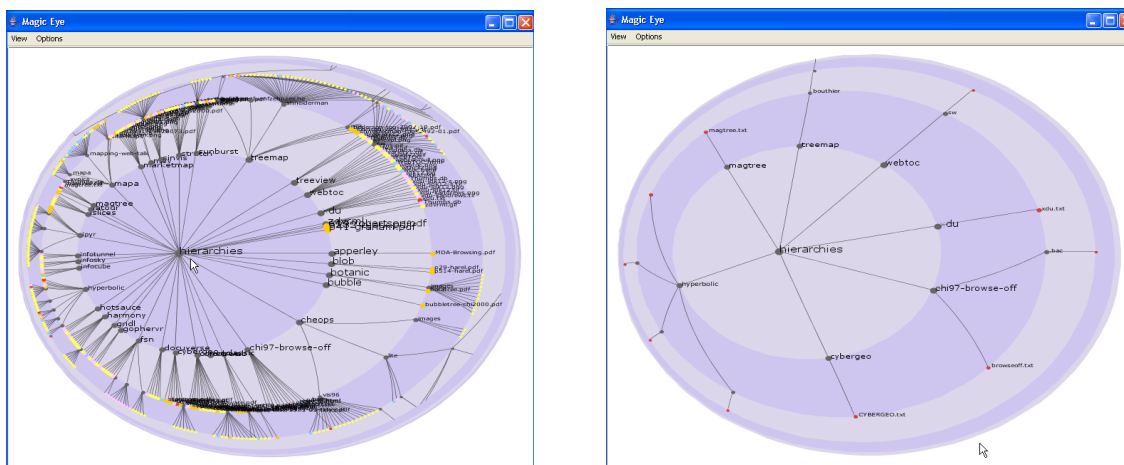
The hierarchical data model also provides the functions for manipulating hierarchies, such as inserting and removing items. These interactions are simply forwarded to the underlying data model. Thus the selected input module has to handle these actions. Therefore it is possible, that these changes also manipulate the “real world”. Of course, the data model has to listen to any changes. Therefore it is registered as a listener to the input module.

The *DataModel* interface declares the methods to access the structural information of the hierarchy. This is the main interface which is accessed by visualizations. The declared methods contains:

- `getRoot()`  
The root *Collection* object is returned by this function.
- `getSubCollections()`  
The returned vector of this function contains the child nodes, which are inner nodes.
- `getDocuments()`  
This method returns the child nodes, which are leaf nodes.
- `getChildren()`  
This method returns the child nodes of a given node.
- `getTreePaths()`  
This method is used to compute all paths from the root of the hierarchy to the given node.

The *MutableDataModel* extends the *DataModel* interface by adding manipulation functionality. Further it is a subclass of the *InputDataModelListener*. Thus it can be attached as listener to the input module. Thus it receives notification of changes in the hierarchy. Furthermore it declares methods to apply filters and change the sorting order of nodes. Methods for attaching listeners are declared as well. These listeners have to implement the *DataModelListener* interface.





(a) A file system hierarchy visualized in the Magic Eye Viewer.

(b) The same hierarchy after a filter has been applied.

**Figure 5.4:** Example of filtering a hierarchy in HVS.

## 5.4 Controller

All views in HVS are synchronized. The synchronization is managed by a controller. Since each visualization has its own user interface, it has to notify the controller. Therefore each visualization sends events to the controller. Of course, visualizations have to provide functions to receive events from the controller. The controller synchronizes the following user interactions:

**Selection** Selected nodes are typically highlighted in some way in visualizations. All selected nodes in one view are also highlighted as selected nodes in other views. Therefore the controller sends an event to all views, indicating that the selection has changed.

**Expansion** Expanding a node in one view causes the corresponding node in other views to be expanded.

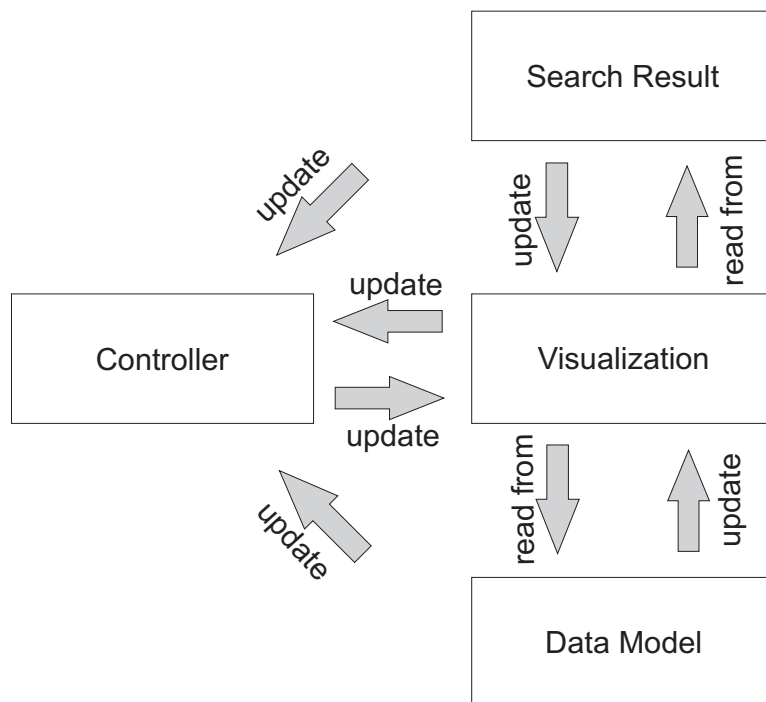
**Navigation** Navigation in a view causes a change of the currently focused node. For example scrolling to a node in an outline view causes other views to change their viewing location to make this node visible.

These actions are not independent of one another. For example, the selection of a node causes the controller to send possibly three events to each view. First, if the parent of the selected node is not expanded, an event is sent to set the parent node expanded. After this a selection event is sent. To be sure that this item is visible, the controller sends a focus event to cause the views to navigate to that item.

Expanding an inner node causes the controller to send an expansion event to set this object in all views into the expanded state. To ensure visibility, an additional focus event is sent.

As described, navigational actions can cause a change of the focused node. HVS differentiates between the following types of change:

**FOCUS** The node identified by this change should be in the visible area of the desired view. This is the default behavior to synchronize navigational actions.



**Figure 5.5:** The synchronization architecture of HVS.

**MAXIMIZE** The identified node should be maximized in the display area. Visualizations with zooming functionality zoom to that item. For tree-based browsers, such as the hyperbolic browser, the node should be centered to magnify it.

**SCROLLING** Indicates that the user scrolls to the focused node.

**SELECT** The focus is changed by a selection.

**EXPAND** The focus is changed by expanding a node. By expanding a node users are typically interested in the child nodes of the expanded one. Therefore visualization should move that node to a position where child nodes are given more display area.

**COLLAPSE** By collapsing a node, users are typically interested in the siblings of the collapsed node. Thus visualizations should move the node to a position where its siblings become more visible. For example, visualizations with zooming functionality would zoom out.

## 5.5 Search Result List

The search result list provides the search functionality of HVS. It receives a filter to apply to each node. A breath first search algorithm is used to traverse the hierarchy. If a node matches the applied filter it is added to the result list.

Visualizations highlight nodes contained in the result list in a suitable way. Thus the search result list have to notify visualizations when the result list has changed. One node of the result list is visualized in different manner. This special node is called the “selected search result”. Using different

highlighting allows users to step through the result list by changing the node representing the selected search result. The result list is sortable by any attribute of the contained nodes.

The interface *SearchResult* declares two methods for determining elements of the search result list. It resides in the package *iicm.hvs.search*. The method *isSearchResult()* returns true if the result list contains the given node. The selected search result is determined using the method *isSelectedSearchResult()*. It returns true if the given node is the selected node in the result list.

## 5.6 Visualization Modules

Visualizations transform the hierarchical information into visual representations. They also perform navigation, zooming, and selection. Several visual attributes such as color, size, labels, thumbnails, and sort order provide meaningful information for users.

The size and labels of visual objects are in general determined by the layout algorithm of a concrete visualization implementation. The sorting order is given by the data model. The mapping of nodes to color and thumbnails is done using a *VisualizationProperties* object. It maps the type of leaf nodes to a color attribute. This mapping is configurable by the user. Associated thumbnails can be achieved using the thumbnail index attribute. The *VisualizationProperties* object maps this index to an image file in the temporary directory of HVS. Since loading thumbnails require certain time, which would negatively influence the performance, thumbnails are loaded using a background thread. Of course, loaded thumbnail images are cached to improve performance.

Selecting items typically causes items to be drawn with a wire box around them. Since HVS supports searching for items by several attributes, visualizations have to provide a way to visualize the search result list. Drawing a wire frame box around the items is a good way to accomplish this. Of course a different color must be used to distinguish between selected items and items of the search result list. For the current selected item in the search result list, another different color is used to differentiate this item.

Dynamic labeling by the use of tool tips is another approach to improve usability. The information provided by tool tips is not limited to the standard attributes, such as name and size, of a node. All available metadata information assigned to a node is also displayed. Since hovering over the display area shows detailed information of items to the user, this technique improves the exploration of the information space.

### 5.6.1 Properties Panel

Each visualization frame provides a properties panel (see Figure 5.6) for displaying several attributes of selected nodes. Since multiple selection is allowed this panel is organized as a table view. The user can decide, whether this panel should be displayed or not. Which attributes to display is also chosen by the user. The displayed nodes are sortable by any attribute. By default, the nodes are sorted by name in ascending order.

Synchronization between the properties panel and the visualization is done automatically by the synchronization mechanism of HVS. Since each visualization sends an event to a controller when the selection has changed, the panel has only to be attached as a listener to that controller. When the user selects a row in the properties panel, a maximize focus event is send to all visualizations. Thereby the selected node in the properties panel is maximized in the visualizations.

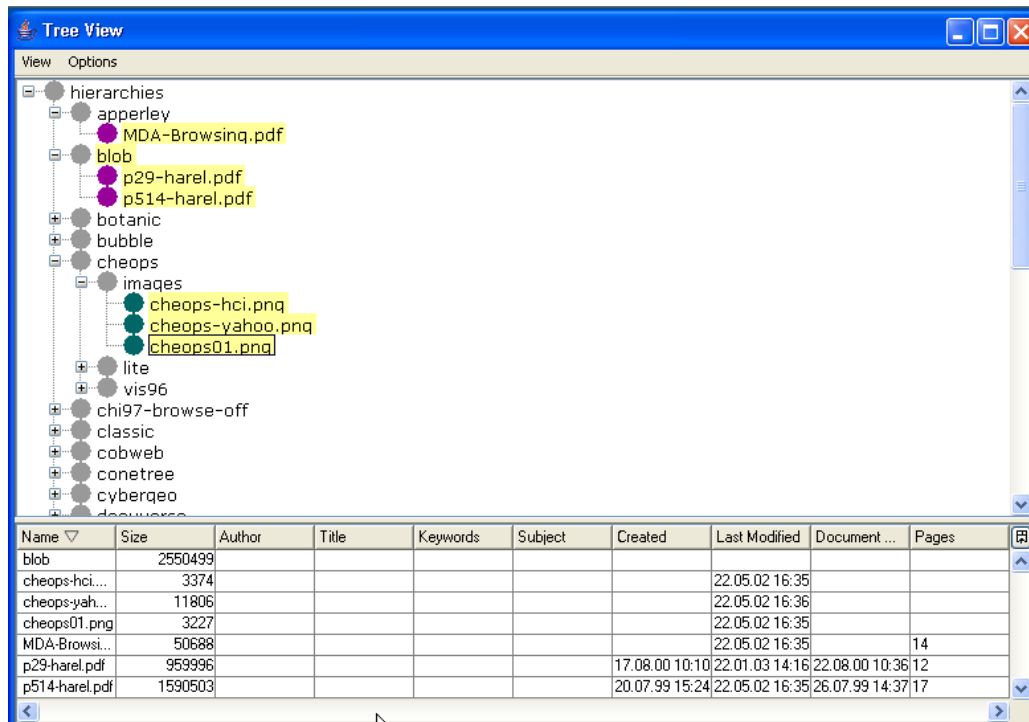


Figure 5.6: The properties panel used to display details of selected nodes.

## 5.6.2 The Main Interfaces for Visualization Development

### Visualization

The abstract class *Visualization* is the base class for all visualizations. The *javax.swing.JPanel* class is the superclass, so it can be placed on every *Swing* component. In order to react to changes, implementations of this class have to implement the following listener interfaces:

- ***ControllerListener***

Reacts to events sent from the *Controller* in order to synchronize this visualization with all other. The declared methods are:

- *focusChanged()* This method is invoked when the focused node in another view has changed.
- *selectionChanged()* The user changing the selection in another view leads to this method being called.

- ***SearchResultListener***

These listeners are notified after the result list has changed. The method *searchResultChanged()* is called after a new list has been set. When the user steps through the result list, this method is invoked as well. Each *Visualization* object is thus registered as a listener of the *SearchResultList* object. To avoid confusion, all listeners are always notified independently of their synchronization state.

- ***DataModelListener***

In order to react to changes in the data model, visualizations have to implement the methods declared in the *DataModelListener* interface.

- ***ExpansionListener***

If the visualization allows navigation by selectively expanding and collapsing nodes, it has to implement the *ExpansionListener* interface too. The method *collectionExpanded()* is called after a node is expanded by the user. When a node is collapsed by the user the method *collectionCollapsed()* is invoked.

The status information provided by the abstract class *Visualization* are:

- *datamodel\_*

The reference of the data model to visualize. This object maintains the structural information of the hierarchy.

- *searchresult\_*

The object to determine that a *Node* object is contained in the search result list.

- *controller\_*

The synchronization object to send notification on user events.

- *popupmenuhandler\_*

This object of type *PopupMenuHandler*. This class manages the standard options of popup menus in visualizations.

The following protected methods are used to synchronize other views:

- *fireFocusChanged()*:

Sends an event to the controller when the focus has changed. Typical user actions initiating such an event are scrolling, panning, and zooming.

- *fireSelectionChanged()*

Sends the current selection to the controller. The controller calculates the focus node using the following strategy: If the selection state in the controller contains all nodes of the current selection, the focus is set to the missing one. If the controller does not contain all current selected nodes, the focus is set to the last element in the current selection.

- *fireCollectionExpanded()*

Sets a collection into expanded state. If the path was previously expanded, all views are updated with the cached paths.

- *fireCollectionCollapsed()*

Sets a collection into the collapsed state. If subcollections are currently expanded they are cached, to expand the complete path when this collection is expanded again.

- *createMultiLineToolTipText()*

This method is used to create a *String* object, used for dynamic labeling.

## **VisualizationProperties**

The purpose of the class *VisualizationProperties* is to map attributes of visualized items to colors. All methods of this class are declared static, so they are accessible for every visualization.

- *getColorForDocument()*  
Determines a color for the desired *Document* object. The color is mapped to the document type.
- *getImageForDocument()*  
If a thumbnail is assigned to document, a *BufferedImage* object is returned. Since thumbnails in four different sizes are defined, the parameter *preferredWidth* is used to determine the thumbnail in the desired size.
- *getSelectionColor()*  
This method returns the color for the visual representation of selections.
- *getSearchResultColor()*  
This method is used to return the color for marking objects as result.
- *getSearchResultSelectionColor()*  
The return value of this function is used to represent the current selected element in the search result.

## 5.7 Application

The application part of HVS manages the basic event handling. It handles the selection and loading of input modules, loading of visualizations, and user interactions for global settings.

### 5.7.1 Plug-in Management

Since HVS is an extensible framework, an easy and simple mechanism has to be found to adapt extensions. A plug-in architecture seems the best solution to accomplish this. Thus the installation procedure is reduced to copying the extension into the *plugin* directory. Another big advantage of using this architecture is that each plug-in can use any external library. Since the *PluginManager* manages the class path settings, users do not have to take care about the Java environment configuration. This task has to be done by developers.

Each plug-in is described by a configuration file. The configuration file is of type XML and defined by a DTD (see Figure 5.7). The defined main elements and attributes are:

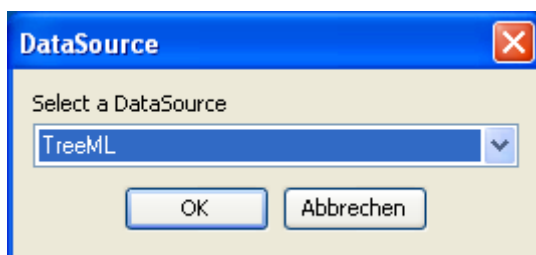
- *extension*:  
The extension point of the plug-in. Since *HVS* is designed to extend visualizations and input modules, the predefined values are:
  - *iicm.hvs.visualization.Visualization* for visualizations
  - *iicm.hvs.inputfactory.DataSource* for input modules.
- *name*:  
The name that appears in the selection box (see Figure 5.8). For visualizations this name is also displayed in the title bar of the window.
- *class*:  
The main class of the plug-in. The fully qualified name is required. This includes the package name.

```

<!ELEMENT plugin (runtime)>
<!ELEMENT runtime ((library | path)*, extension*)>
<!ELEMENT library EMPTY>
<!ELEMENT path EMPTY>
<!ELEMENT extension EMPTY>
<!ATTLIST path name CDATA #REQUIRED>
<!ATTLIST library name CDATA #REQUIRED>
<!ATTLIST extension point CDATA #REQUIRED
                    name CDATA #REQUIRED
                    class CDATA #REQUIRED>

```

**Figure 5.7:** The document type definition of plug-in configuration files.



**Figure 5.8:** The selection dialog for selecting a new data source. Note that the name used in the selection dialog is defined by the plug-in configuration file.

- *path*:  
The path where to find the main class to load. If the plug-in is packed to an archive, the *library* element should be used instead of this.
- *library*:  
This tag is used twice. If this plug-in is organized as a jar file, the library tag is used instead of the path tag. Secondly, all libraries needed for this plug-in are defined using this tag. All these libraries are added to the class path of the desired class loader.

A *Pluginmanager* scans the subdirectories of the *plugin* directory for configuration files. If a plug-in has no configuration file, it will not be selectable. Of course only those plug-ins with correct configuration file and loadable classes will be shown. Thus the *PluginManager* tries to load the main class of each plug-in. If it fails, the failure is logged.

### 5.7.2 Session Management

In HVS the hierarchy and all visualizations are loaded at runtime. Nevertheless, this advantage in flexibility has the disadvantage that that on every restart the user has to configure HVS again. Therefore users would have to spend time to create their working environment on every run of the application. To minimize this time-consuming work and to preserve flexibility, user configurations can be saved.

Thus users create the working environment that fulfills their needs and save it. On a restart of the application they simply have to load a previously saved session. Of course this configuration can be

changed at any time and the modified configuration can be saved. The session management manages the following information:

- The hierarchy which is visualized.
- All visualizations started by the user.
- The synchronization mode of each visualization.
- The size and location of each visualization.
- The size of the main window.

### 5.7.3 Color Management

HVS allows users to define all colors used in visualizations at runtime. The standard color options include:

- Background. The color of the background image of visualizations.
- Collection. The color of objects representing inner nodes.
- Selection. Selected items are highlighted using this color.
- Search Result. Nodes contained in the search result list are marked using this color.
- Selected Search Result. The especially marked node the result list is highlighted in this color.
- Unknown. Leaf node objects which are of unknown type are drawn in this color.

Depending on the selected input module, these standard options are extended by the defined document types. Since each document is assigned a numeric value identifying its type, the resulting mapping is easy.

To improve usability, two configurations are predefined for users:

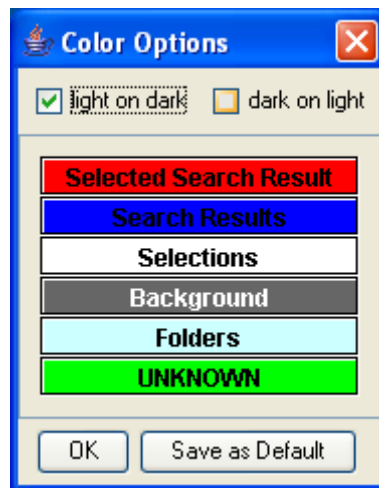
- *light on dark*  
All visualizations have a dark background. Items are drawn with a light color on this dark background.
- *dark on light*  
The background color of visualizations is a light color. The items are drawn in a darker color than the background color.

## 5.8 Synchronization Modes

For effective exploration of hierarchies, HVS offers three different modes of synchronizing views. The synchronization mechanism is designed such that developers of visualizations do not have to take care about this. Users can interactively change the mode of synchronization at any time. The three main modes are:

- Independent versus Synchronized





**Figure 5.9:** The standard color selection dialog.

- Overview versus DetailedView
- Show Documents versus Hide Documents

Each possible combination of this three main modes can be achieved by users.

### Independent versus Synchronized

To allow users to explore a hierarchy without changing other views, visualizations can be set into an independent mode. In this mode, navigational actions, such as expand or select are not passed on to other visualizations. By changing back into synchronized mode, the view is resynchronized with all others.

The observer design pattern [Gamma et al., 1995] is a good solution to provide this functionality. Since only registered listeners are notified from the controller, setting independent mode can be realized by detaching the desired listener. Changing to synchronized mode is then attaching to the controller. Synchronized views must have exactly the same items visible after setting back into synchronized mode. Thus changes are internally stored in the controller. The controller sends events to set the correct internal state of newly resynchronized views.

To ensure continuity, listeners are not only detached from the controller rather than attached to a clone of the controller. So listeners can be resynchronized independently of their state. This simplifies the resynchronization of views, when the data model changes by applying a filter or changing the sorting order.

### DetailedView versus Overview

In *DetailedView* mode a visualization shows only those parts of the hierarchy where items are selected. Creating a detailed view in hierarchies is quite simple. Setting a node of the hierarchy as new root fulfills this enhancement. Deciding which node to take as root is important for the strength of this feature. If a single inner node is selected by the user, this node is the new root of the hierarchy in a detailed view. By selecting a leaf node, the parent node is the new root of the detailed view.

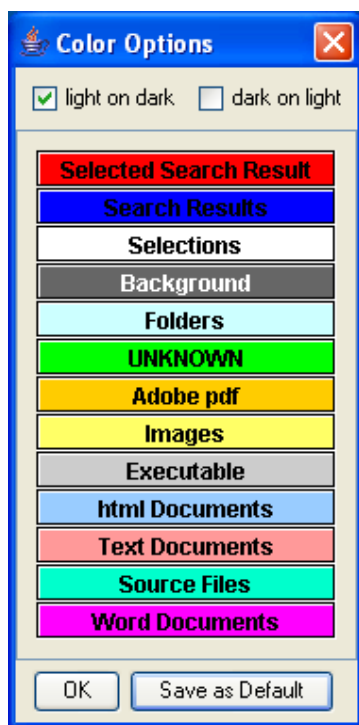


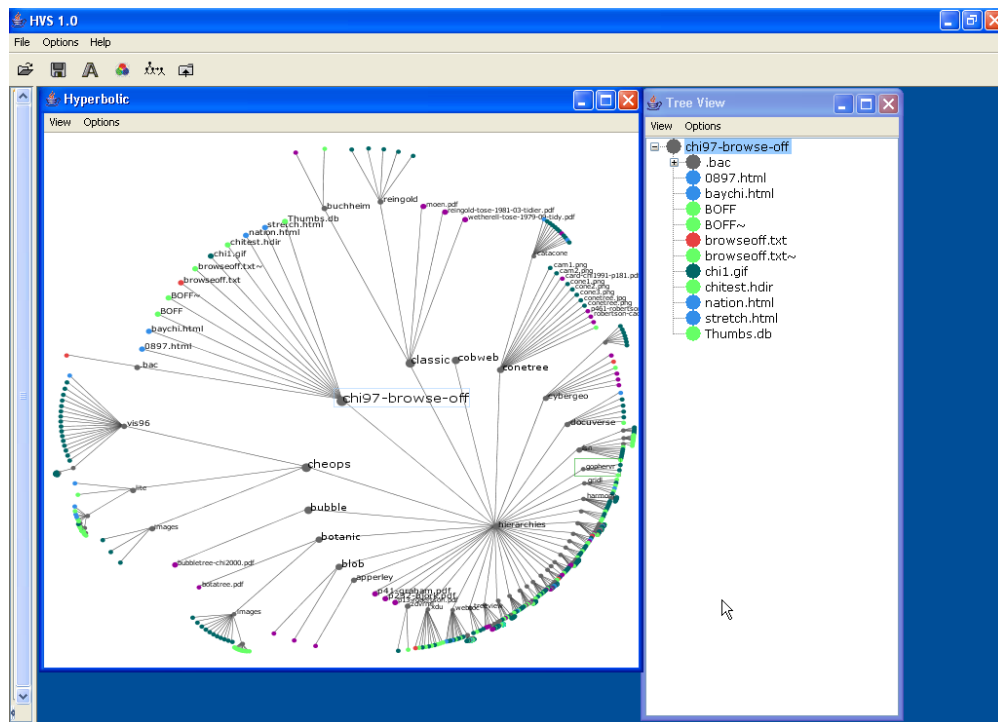
Figure 5.10: The color selection dialog.

When multiple nodes are selected by the user, the new root is calculated as the common parent of all selected nodes. Users do not lose the global context, since other synchronized views not running in *Detailedview* mode continue to visualize the entire hierarchy. Figure 5.11 demonstrates this synchronization mode. The tree view on the right is set into the *Detailedview* mode. The selected inner node in the hyperbolic browser is the new root in the tree view. Hence, the tree view on the right shows the details of any node(s) selected in the hyperbolic browser on the left.

### Show Documents versus Hide Documents

For exploring the structure of the hierarchy HVS offers the option to hide documents. Thus only the structure of the hierarchy is visualized (see Figure 5.12). This enables users to rapidly understand the structure of the hierarchy.

Any combination of these three main modes can be set. To fulfill this enhancement, visualizations are coupled with bridges (see Figure 5.13) to the desired modules (see Figure 5.13). Changing the independent state causes the *ControllerBridge* to detach from one controller and attach to the desired one. To hide documents, the *DataModelBridge* only provides collections to visualizations. Additionally the *SearchResultBridge* is set, to return the parent collection objects as search result. To set the *Detailedview* mode a new root is set to the *DataModelBridge*. This causes visualization of a hierarchy with a different root. As can be seen, changing the mode of synchronization is independent of the implementation of visualizations.



**Figure 5.11:** The *DetailedView* mode. The tree view on the right is in *DetailedView* mode and shows the details of the selected node(s) in the hyperbolic browser on the left.

## 5.9 The User Interface of the Application

HVS offers users visualization of hierarchies with different techniques. All these visualizations are synchronized and can be created at runtime. The number of views and their size should be determinable by users. Thus a multiple window environment is required to allow users to adjust the position and size of each view. Therefore, the panel is assigned a virtual desktop where views reside inside. Furthermore one panel is used to support interactive manipulation of the hierarchy. Additionally, this panel is used to provide feedback on time consuming operations. The elements for providing this visual feedback reside in the bottom area of the panel and are only visible when necessary. The control panel for manipulations resides in the upper area (see Figure 5.14) and is organized into tab groups. The three tabs are Sort, Filter, and Search. The sort tab is visible by default.

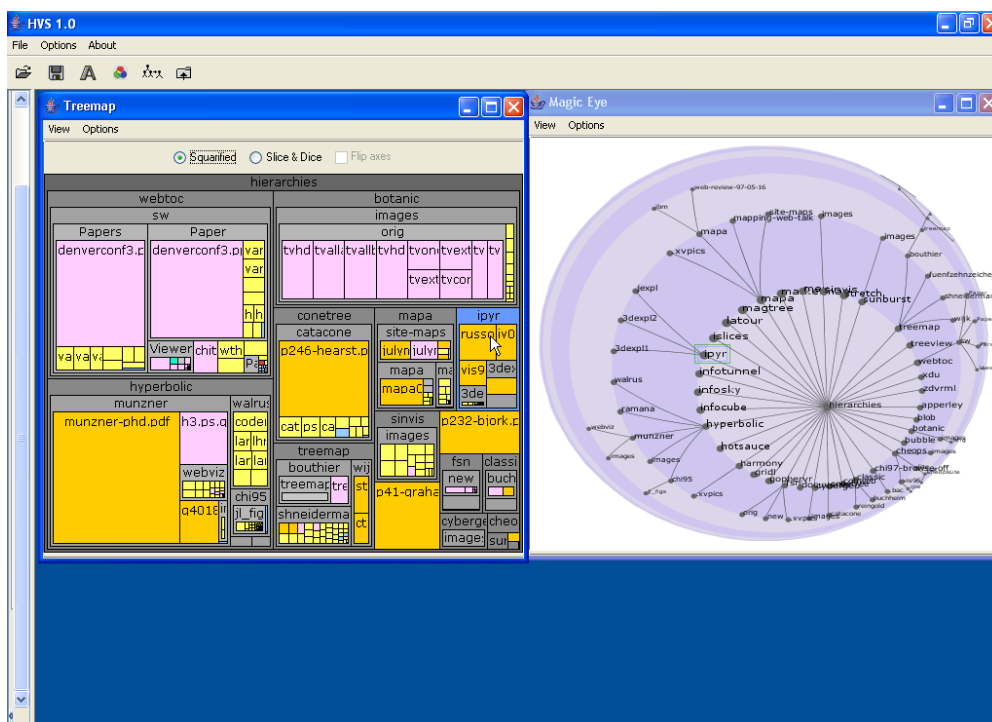
### 5.9.1 Sort

To help users explore a hierarchy, the nodes can be ordered. There are different ways to sort nodes before they are visualized. Of course, sorting the nodes in the data model only changes the arrangement in views, when the layout algorithm does not change the natural order. For example, Squarified Treemaps (see Section 3.3) changes the natural order of elements, therefore sorting nodes does not change the view.

Users can select between the following sort algorithms:

- **Sort by name**

Sorts the nodes by their name. Inner nodes and leaf nodes are sorted in the same manner.



**Figure 5.12:** The hide documents mode. The Treemap visualization on the left and the Magic Eye View on the right are visualizing the same hierarchy. The Magic Eye View is operating in hide documents mode and thereby visualizes only the structure of the tree.

- **Sort by size**

Nodes are sorted by their assigned size.

- **Sort by type**

Leaf nodes are sorted by their assigned type. Since inner nodes are not assigned a type, this order does not change.

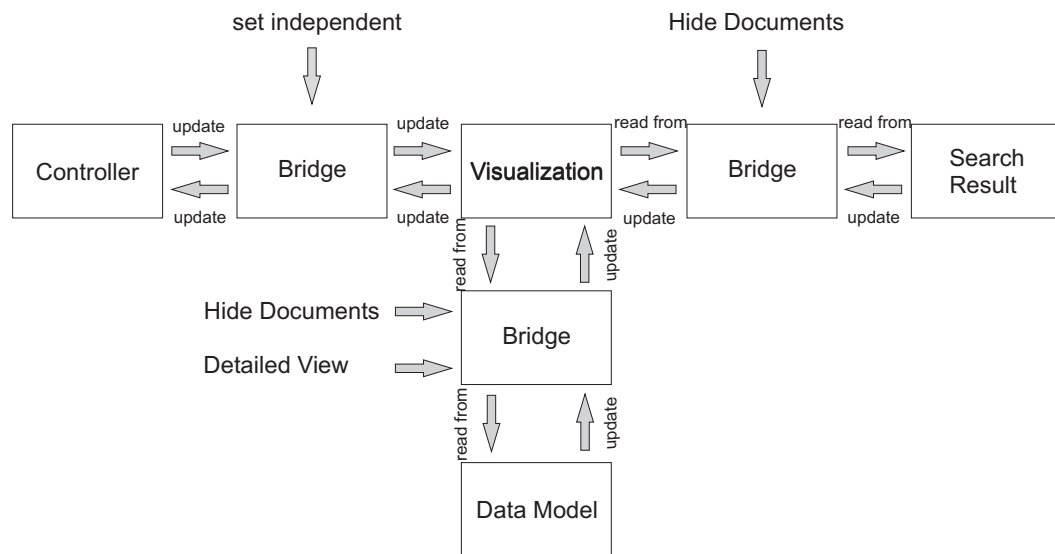
Users can decide whether the nodes should be in ascending or descending order. By default, nodes are sorted by their name and in ascending order.

### 5.9.2 Filter

The filter panel initially displays the name field and the type choice box (see Figure 5.15). The name field can be used to apply a filter on leaf node names. Using the \* wildcard powerful filters can be applied. Possible filters are:

- string matches when the name of a node is exactly “string”
- \*string matches when the name of a node ends with “string”
- string\* matches when the name of a node starts with “string”
- \*string\* matches when the name of a node contains “string”

Since no *Datasource* is set, the type choice box does not contain any values to select. When the user sets a *Datasource* to visualize, additional fields depending on available metadata information are placed onto the filter panel (see Figure 5.16). Depending on the type of metadata information different



**Figure 5.13:** The synchronization architecture to fulfill the different synchronization modes.

input fields are used. For textual fields a single field is used. Two fields are used for numerical and date types. The \* wildcard is not used for these fields; textual fields automatically use the contains strategy.

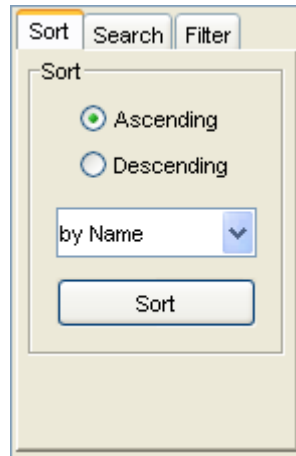
### 5.9.3 Search Facility

An important exploration tool is a search facility. This allows users to navigate to specific items. The search panel initially displays similar to the filter panel only a name field and a type box. Depending on the selected *DataSource*, fields for metadata information are added. Figure 5.17 shows the Search Panel when visualizing a file system. The usage of the provided input fields is like those of the filter panel.

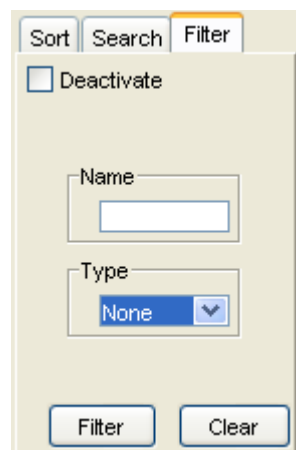
The search result list is visualized by each visualization. Typically a colored wire frame box is drawn around these items. A search result panel (see Figure 5.18) displays the number of elements in the result list. Two buttons allows users to step through the result list. The field displaying the current position in the result list can be used to jump to a particular element.

A table view (see Figure 5.19) displays the result list in more detail. This table visualizes by default all available information of the nodes in the result list. Which attributes of nodes are visualized in the table is selectable by users. In the table view, users can sort the results by any attribute. By default the nodes are sorted by name in ascending order. The position of a node displayed in the search result panel depends on the sorting order of the result list. The table view and the search result panel are synchronized to avoid user confusion.

When the users steps through the result list, HVS sends to each visualization an event to update the visual representation of the particular item. Synchronized views are notified that the focus has changed to the current selected node. Thus synchronized views navigate to that node. For example, an outline view scrolls to this node. Independent views do not change their viewing position since they are not updated.



**Figure 5.14:** The Control Panel.



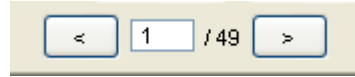
**Figure 5.15:** The filter panel when no Datasource is set.

The screenshot shows a vertical filter panel with a title bar containing 'Sort', 'Search', and 'Filter' tabs. The 'Filter' tab is active. At the top, there is a 'Deactivate' checkbox. Below it are several input fields: 'Name', 'Type' (a dropdown menu currently showing 'None'), 'Author', 'Title', 'Keywords', and 'Subject'. Further down are four date range selectors, each with a '>' button on the left and a '<' button on the right, labeled 'Created', 'Last Modified', 'Document Modified', and 'Pages'. At the bottom of the panel are two buttons: 'Filter' and 'Clear'.

Figure 5.16: The filter panel for the file system Datasource.

The screenshot shows a vertical search panel with a title bar containing 'Sort', 'Search', and 'Filter' tabs. The 'Search' tab is active. At the top, there is a 'Deactivate' checkbox and a 'Search Result List' checkbox. Below these are several input fields: 'Name', 'Type' (a dropdown menu currently showing 'None'), 'Author', 'Title', 'Keywords', and 'Subject'. Further down are four date range selectors, each with a '>' button on the left and a '<' button on the right, labeled 'Created', 'Last Modified', 'Document Modified', and 'Pages'. At the bottom of the panel are two buttons: 'Search' and 'Clear', and below them are three navigation buttons: '<', a text input field, and '>'.

Figure 5.17: The search panel for the file system Datasource.



**Figure 5.18:** The search result panel. Two buttons allows users to step through the result list. The input field can be used to jump to a desired index.

Name	Size	Author	Title	Keywords	Subject	Created	Last Modified	Document Modified	Pages
17310127.pdf	250096	Stefan Sossna (PT...	17310127		TeX output 1999.11....	10.11.99 14:26	22.05.02 16:36	19.11.99 11:53	10
17310392.pdf	147794	Stefan Sossna (PT...	17310392		TeX output 1999.11....	16.11.99 10:06	22.05.02 16:37	19.11.99 11:53	8
2645.pdf	89930					25.05.98 10:32	22.05.02 16:37		10
2657.pdf	114053	Root	91-06.pm			25.05.98 10:33	22.05.02 16:37		17
bederson-to...	3149970					29.09.02 13:45	08.01.04 15:56		22
botatree.pdf	854934		article.dvi			28.06.01 18:07	24.07.02 15:02		8
bubbletree-c...	17473		Microsoft Wo...			10.12.99 17:34	07.11.02 14:21		2
buchheim-g...	249972		walker.dvi			05.07.02 13:56	04.09.02 17:12	05.07.02 13:56	14
cadre_map3...	106668						22.05.02 16:37		1
card-chi199...	1189256	John Nelson	jda2172.tmp			14.07.97 10:44	22.05.02 16:36	14.07.97 10:44	8
cgaa-g4040...	253512					17.06.98 12:41	22.05.02 16:37	17.06.98 13:26	3
chi2000-we...	137794		Untitled Doc...			10.12.99 15:53	16.09.02 13:38		2
ctm.pdf	379407		ctm.dvi			03.09.99 09:38	22.05.02 16:38		6
g4018.pdf	833125					17.06.98 13:12	22.05.02 16:37	17.06.98 13:29	6
HTDS2_0.pdf	189198	Diandra Macias	HTDS for pdf...			10.05.99 09:25	22.05.02 16:36		4
inso2.pdf	26359					05.10.95 16:53	22.05.02 16:37		1
iris-1998-08...	178411					26.06.98 08:12	16.09.02 13:38		8
iui99-starzoo...	433415					18.01.99 21:09	16.09.02 13:38	21.01.99 10:41	1
iv02-165607...	378467	Keith Andrews	Visual Explor...			02.05.02 20:30	22.01.03 14:11	26.07.02 16:02	6
ivis2001-134...	607289	Administrator	Microsoft Wo...			01.09.01 23:10	22.01.03 14:33	18.10.01 10:51	6
johnson-vis...	983765	IEEE	Tree-maps: a...			21.02.98 06:24	08.01.04 16:02	31.10.03 09:21	8
julymp.pdf	593414	dd				12.07.96 18:54	22.05.02 16:37	17.08.96 11:24	1
junemap.pdf	59547	DD				01.07.96 15:11	22.05.02 16:37	09.08.96 18:48	1
magtree-a4...	605983						21.08.02 10:40		33
mapa0v12.pdf	1052934						22.05.02 16:37		14
MDA-Browsi...	50688						22.05.02 16:35		14
mev-da.pdf	5362684		Diplom 71. sdw			05.07.02 12:30	05.07.02 13:31	05.07.02 12:30	72
moen.pdf	839213	DLibSpt3				23.08.01 12:28	04.09.02 17:10	23.08.01 12:28	8
munzner-ph...	8357546					15.06.00 12:47	22.05.02 16:41	15.06.00 12:47	167
npiv99.pdf	3891182		Microsoft Wo...			05.07.02 13:27	05.07.02 14:27	05.07.02 13:27	5
p13-robertso...	39908					17.08.00 10:05	22.05.02 16:35	22.08.00 10:35	1
p232-bjork.pdf	1246529					17.08.00 12:40	22.05.02 16:35	22.08.00 10:36	6
p246-hearsl...	3946289	CDR-13	ihg321.PDF				22.05.02 16:36	13.02.02 10:22	10
p29-harel.pdf	959996					17.08.00 10:10	22.01.03 14:16	22.08.00 10:36	12
p41-graham...	1576504					17.08.00 10:14	22.05.02 16:35	22.08.00 10:36	10
p461-roberts...	203927	John Nelson	jda2172.tmp			14.07.97 11:46	22.05.02 16:36	14.07.97 11:46	2
p514-bjork.pdf	189198					10.05.99 09:25	22.05.02 16:36	10.05.99 09:25	4

**Figure 5.19:** The table view of the search result list. The displayed items are sortable by any attribute.



## Chapter 6

# Information Pyramids

Information Pyramids [Andrews, 2002a; Wolte, 1998; Welz, 1999] are an approach for visualizing and manipulating large hierarchies. They provide a simple and intuitive way to deal with large hierarchies. Hierarchies are visualized in a three-dimensional landscape.

Hierarchically structured information contains two kinds of information. The structural information reflects the relationship between nodes. Content information is associated with each node. The information pyramids technique visualizes both kinds of information. Local focus on a part of a hierarchy while preserving the global context is absolutely necessary for efficient exploration of an information space. The Information Pyramids technique provides mechanism to focus on a particular part of the hierarchy. Since a three-dimensional technique is used, the surrounding elements of the focused one are visualized in the surroundings. This allows users to keep the global context.

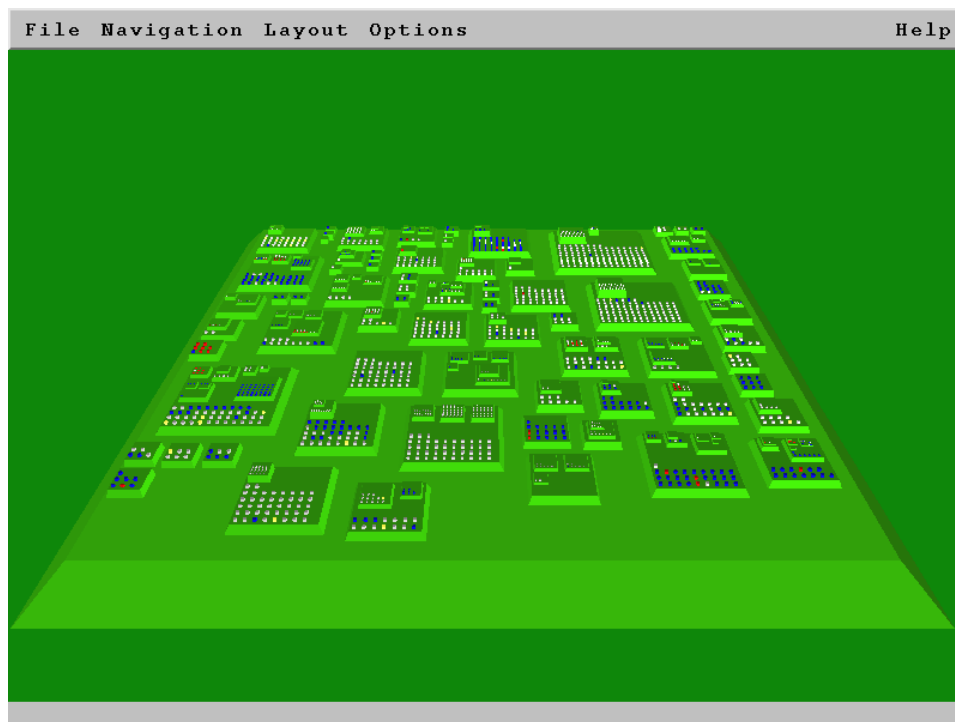
The Information Pyramids technique is similar to the FSN (see Chapter 3.10). Both techniques use a three-dimensional landscape for visualizing a hierarchy. FSN displays only the content information as three-dimensional objects, whereas Information Pyramids uses the third dimension for visualizing structural information. This leads to a better use of the third dimension. Thus a compact representation of the information hierarchy can be achieved. Information Pyramids have also some similarity with Treemaps, since the available display space for child nodes depends on the size of the parent node. Treemaps visualize children within their boundary, the pyramids technique places them on top (see Figure 6.1).

Using three-dimensional space for visualizing a hierarchy has the advantage, that the available display space is used more efficiently. Users can get a good overview, since the whole hierarchy is visible to them. Information Pyramids allow easy visualization of content information. For example, using different sized objects according to an assigned size.

### 6.1 The Pyramids Technique

The Information Pyramids technique visualizes hierarchically structured information as three-dimensional pyramidal objects. Using the third dimension for visualizing the structure of the hierarchy leads to a compact landscape visualization. The name comes from the overall visual impression which looks like a pyramid.

The root of the hierarchy is represented as a square-shaped plateau. All subtrees, represented as smaller plateaus, sit atop of this plateau. Leaf nodes are visualized in different manner to easily distinguish them. Typically leaf nodes are visualized as smaller objects in a different color to inner nodes. The placement of inner nodes starts at the top left corner of the parent's plateau and proceeds



**Figure 6.1:** The 3D Explorer application, the first prototype implementation of the Information Pyramids technique. [Image used with kind permission of Keith Andrews, Graz University of Technology.]

toward the bottom right corner. The placement of inner nodes starts at the bottom left corner. This layout mechanism is applied recursively for all inner nodes of the hierarchy.

Independent of the visualized hierarchy the root plateau is always of the same size. Since child nodes are arranged on top of the root plateau, the pyramid grows only in height. The total height depends only on the number of levels in the hierarchy.

The size of each plateau depends on the number of children and the size of the parent plateau. With growing depth in the hierarchy the size of the plateaus shrinks. Since the plateaus become smaller with their depth in the hierarchy, a zooming mechanism is provided.

### 6.1.1 Leaf Nodes

Each leaf node is visualized as a small object. Its size depends on the size of the plateau this object sits atop. Since the size of plateaus decreases with their depth, the size of objects representing leaf nodes also shrinks. However, the proportion size of leaf node objects to their base plateau is approximately the same.

If all siblings are leaf nodes, the whole space on top of the parents plateau can be used to lay them out. If there exists at least one sibling that is an inner node, only the half space is used for leaf nodes. Leaf nodes can be arranged in different ways. The simplest way is to arrange them in a matrix form. Sorting of nodes by different attribute values may reflect the similarity of nodes. More complex layout algorithm can also be implemented. For example, nodes can be arranged to represent relationships between them.

Leaf node objects can visualize several different attributes. For example, color coding can be used to visualize the type of a file in a file system. The height of the leaf node object can be used to represent an assigned size. Of course, comparing the absolute height of leaf nodes object is meaningful only between siblings.

### 6.1.2 Inner Nodes

The root node and each inner node is represented by a square-shaped plateau. The size of the root plateau is independent of the size of the hierarchy. Thus the root plateau is always the same size.

Inner nodes are placed atop of their parent plateau. The available area must be subdivided in a reasonable way to lay out all child nodes. One approach is to use equal sizes for each plateau. The main disadvantage of this is that plateaus with less children receive the same display size. Thus an inner node with only one child node receives the same display area as an inner node with many child nodes.

Using different sized plateaus provides more information about the structure of the hierarchy for users. Determining the size by the number of containing leaf nodes, for example, allows more efficient use of the available space. Since more space is allocated to plateaus with many child nodes, child plateaus can be drawn larger and more leaf nodes fit into the space. However the determination of size is not limited to the number of leaf nodes. Assigning each node a weight allows its size to vary over different attributes. Determination of large sub-hierarchies can be done by simply comparing the visual size. This allows users to more easily find information. Since the size of each plateau depends on the size of the base plateau, such a comparison is only meaningful for siblings.

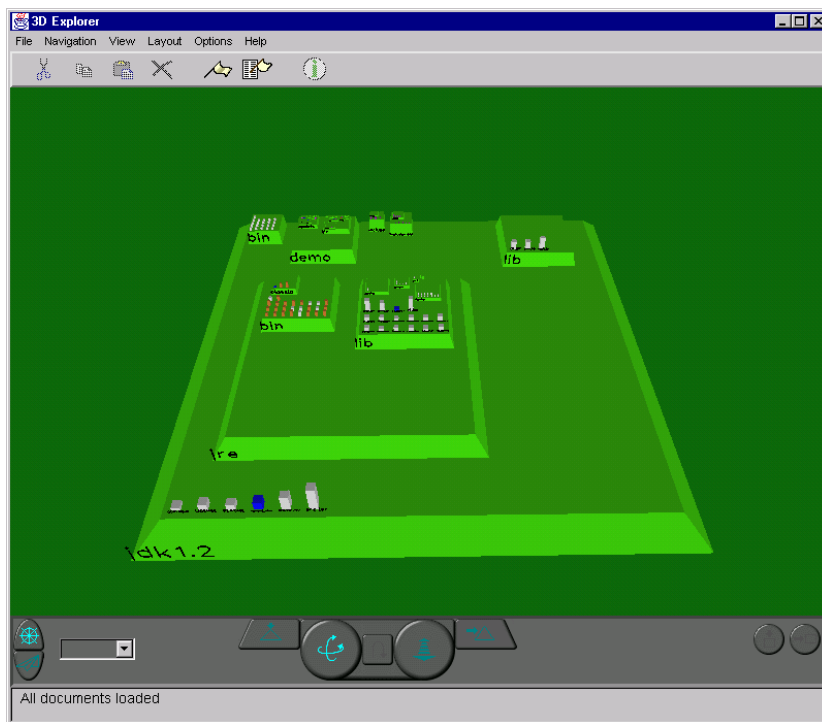
### 6.1.3 Navigation

Suitable navigation facilities are as important as the visual representation for intuitive exploration of a hierarchical information space. Many different navigation methods can be implemented for a compact three-dimensional visualization. In analogy to Wolte [1998], possible navigation aids include:

- Free navigation over the landscape is the best way to gain an overview for users.
- Navigation methods for reaching points of interest. Users should be able to move to quickly to a child or parent node.
- Special views of the hierarchy should be reachable quickly. A top view for example, displays the ground shape of a plateau.
- User-definable views (bookmarks) should be reached without much navigation.
- Providing undo and redo facilities allows users a reversible exploration of the hierarchy.

The Information Pyramids technique provides rich contextual information for users. This information gives users a feeling of the structure of the information space and helps users find specific information.

The movement from one location to another should be smooth animated. Simply switching from one position to another would disorientate users. This could lead to a loss of the global context and the user may have to start exploring the information space again. The animation speed is as important as the animated movement. Too slow animation speed prevents user's continuous work. Too fast animation speed has the same effect as jumping directly from one location to the other. Human-computer interface research suggests that animations should take about one second [Robertson et al., 1991b].



**Figure 6.2:** The 3D Explorer which implements Information Pyramids. Color coding is used to visualize different file types and the height of the leaf nodes corresponds to their size. [Image used with kind permission of Keith Andrews, Graz University of Technology.]

## 6.2 3D Explorer

The 3D Explorer [Wolte, 1998] was the first prototype implementation of the Information Pyramids technique. The programming language used for implementation was Java. Since the platform-dependent library OpenGL was used, the portability was decreased.

The 3D Explorer is a visual exploration tool for file systems. However, the object-oriented design of the application allows to easily exchange the implementation to visualize other hierarchies. The 3D Explorer comes with two different layout algorithms. One uses equally sized plateaus to represent directories. In the other layout algorithm, the size of each plateau corresponds to the size of the represented directory.

Files are represented as small pedestals. Color coding is used to distinguish between different file types. The height of each pedestal is proportional to the size of the represented file. The order of the arranged elements atop plateaus depends on the sorting order. Users can choose from a number of sorting algorithms.

The 3D Explorer provides a rich set of navigational aids for the exploration of the information space. Users can freely navigate around the landscape. When the user approaches the pyramid, more details become visible. Two buttons provide access to front view and top view. Sometimes it is not necessary to visualize all surrounding contextual information of a focused plateau. Therefore “pruning” temporarily designates a particular directory of the pyramid to be the new root. To make the parent visible again, the operation “unprune” is available.

## 6.3 Java Pyramids Explorer

The Java Pyramids Explorer (JPE), introduced in Welz [1999], is a further development of the 3D Explorer. Early user experiments with the 3D Explorer showed that users are not familiar with navigation in three dimensional landscapes. They often lose the global context in spite of the navigational aids available during exploration of the hierarchy.

The JPE application combines the pyramids with a traditional tree and list view (see Figure 6.3). The tree view visualizes only inner nodes of the hierarchy. The list is used to visualize the contents of a selected inner node. In the pyramids view the same tree is visualized as in the tree view. The pyramids view is fully synchronized with the tree view.

In the JPE application the extensive 3d navigational facilities were replaced with three simple sliders. One slider controls the rotation around the x-axis. The second slider is used to rotate the pyramid around the z-axis. For zooming in and out of the pyramid a third slider is provided. Since all views are synchronized, users may also navigate in the tree view. Any navigation can be followed in the pyramids view. This makes sense since users are familiar with using tree views for exploring hierarchies.

To increase portability, the OpenGL 3d graphics were replaced with a custom perspective 3d package implemented with the Java2D graphics library. The Java2D graphics library is part of the Java Development Kit (JDK) and thus users do not have to install an additional library to run the application.

In the JPE application the plateaus are labeled extrinsically. Only the selected node, the parent and a few siblings are labeled. The labels are drawn in the margin of the pyramids view. Labeling more nodes during navigation would facilitate the exploration of the hierarchy. However too many extrinsic labels may confuse users.

## 6.4 The PyramidsBrowser

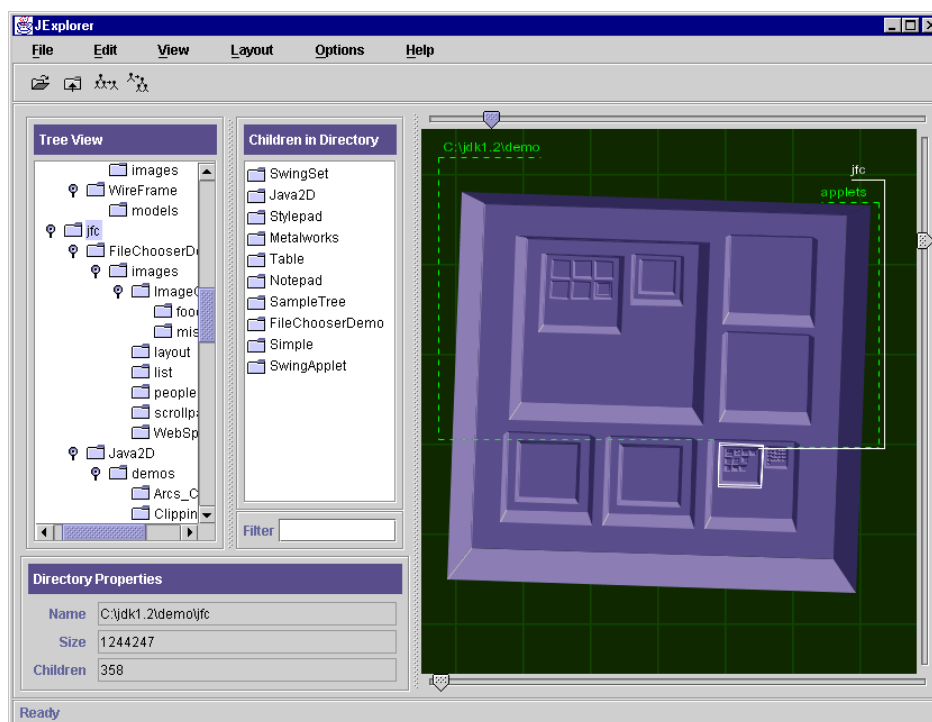
Information Pyramids are a powerful technique for visualizing hierarchies. The compact representation allows visualizing large hierarchies in a manageable way. Both implementations of the Information Pyramids technique scale very well to larger hierarchies.

The PyramidsBrowser is an implementation of the Information Pyramids technique using the HVS environment. Similar to the 3D Explorer application (see Section 6.2) the PyramidsBrowser visualizes both inner nodes and leaf nodes. The placement of the nodes depends on the chosen layout algorithm. The height of blocks representing leaf nodes corresponds to their assigned size.

The PyramidsBrowser produces only a simulated tree-dimensional visualization. Since not all surfaces of the pyramid are visible for the user, navigation possibilities are limited. Nevertheless, this restriction is no disadvantage, because navigational aids can be simplified. As the JPE application, the PyramidsBrowser uses the Java2D API to display the pyramids. Since the Java2D API is already part of the Java Development Kit (JDK) and no additional graphics library is necessary, users do not have to install an extra software package to run the application.

### 6.4.1 Labels and Thumbnails

To find information during exploration, textual description of nodes is absolutely necessary. Therefore, the names of nodes can be displayed on the surface of the corresponding plateau. Of course, not all labels are readable for users. The JPE application uses extrinsic labeling of the most important nodes. Heuristics are used to determine which nodes to label. Labels are drawn for the selected

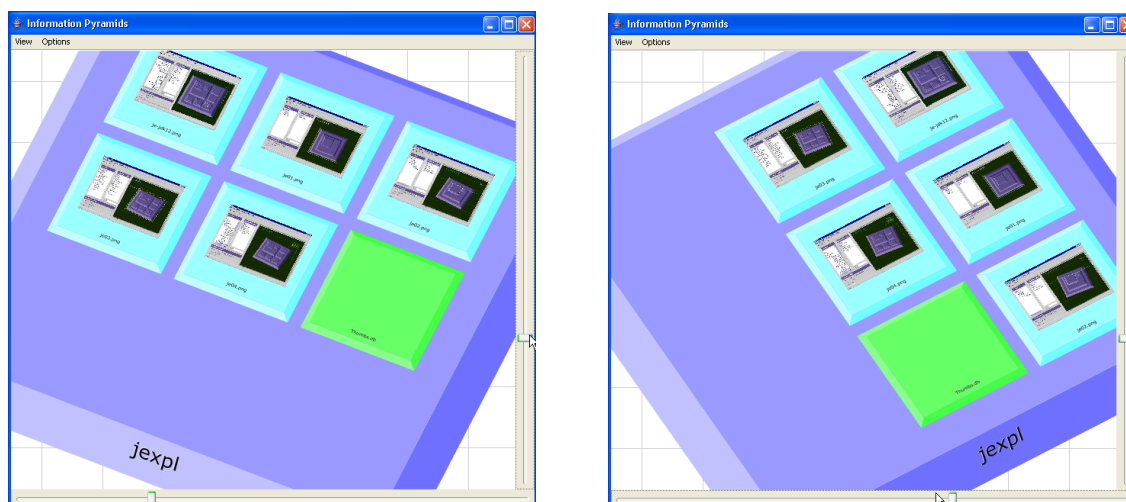


**Figure 6.3:** The JPE application visualizing a file system. The pyramids view in the right visualizes the directories and is synchronized with the tree view in the left. [Image used with kind permission of Keith Andrews, Graz University of Technology.]

node, its parent, and two siblings. Since extrinsic labeling allows only a few of the visible nodes to be labeled, it seems more efficient to place labels on plateaus. Zooming to a particular part of the hierarchy makes these labels readable. This mechanism provides a natural focus + context effect. The focused area is magnified and enlarged while the surrounding context is preserved. This seems a better solution than using heuristics as in the JPE.

The available space to draw labels is limited by the size of a plateau. To produce a uniform appearance, the height of the labels of all plateaus at a certain level should be identical. This height depends on the height of the plateau and the number of the characters of the represented label. The fewer characters the label contains, the more space exists in the width and the font size depends only on the height of the plateau. If the label contains too many characters, the font size must be selected in such a way that all characters can be accommodated in the available width. To produce a uniform appearance, a fixed number of visible characters is used. The font sizes are not precisely identical, because every character needs a different amount of space and labels do not consist of exactly the same characters. However, this difference is small and is not recognizable for users. The number of visible characters is configurable by users. The disadvantage that long names are not completely displayed, can be resolved using tool tips. Tool tips allows users to explore the hierarchy by simply hovering over the landscape.

A shading technique improves the readability of labels. This is achieved by drawing each label twice. First it is drawn in white, then it is moved a little up and to the right and drawn in black. Of course, this technique improves the readability only on light backgrounds. If the color of a plateau is too dark, labels would not be readable. To prevent this effect, black and white are exchanged at a certain minimum brightness of the plateau color. Thereby labels become readable on dark background,



(a) The placement of labels if the rotation angle around the Z-axis is less than 45 degrees.

(b) The placement has changed after rotating the pyramid more than 45 degrees around the Z-axis.

**Figure 6.4:** The placement of labels of the PyramidsBrowser. Notice that inner nodes and leaf nodes are labeled in different manner.

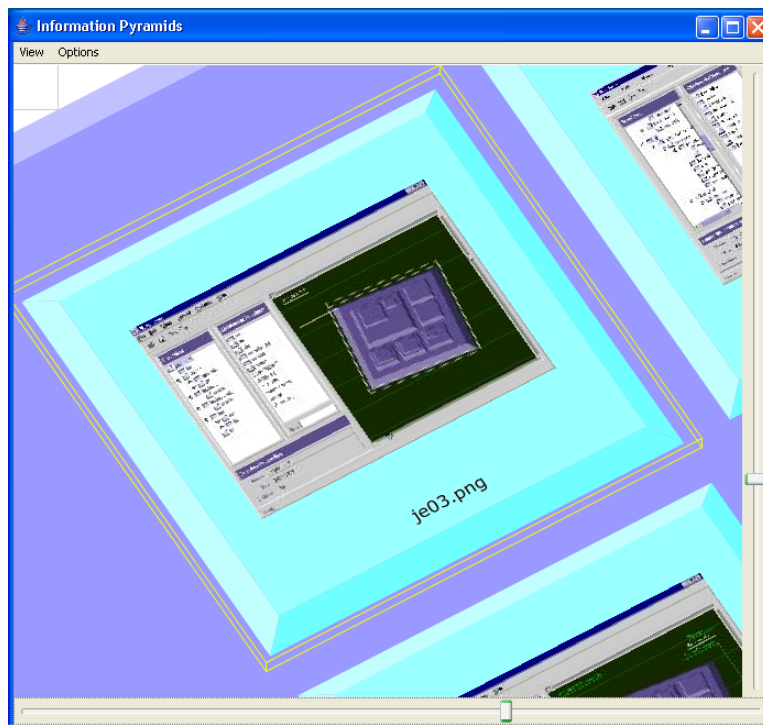
since they appear bright.

Leaf nodes and inner nodes are labeled in different manner. Since child nodes are placed on top of plateaus, it make sense to place labels for inner nodes not on the top surface. Thus the front surface seems the best choice to display the textual description of inner nodes. Leaf nodes are labeled at the top surface, since the whole surface can be used. Another reason for this different labeling is helping users distinguishing between leaf and inner nodes. Figure 6.4(a) shows the placement of labels in the PyramidsBrowser. To improve readability, the position of the labels changes from the front surface to the right surface when the pyramid is rotated more than 45 degrees around the z-axis (see Figure 6.4(b)).

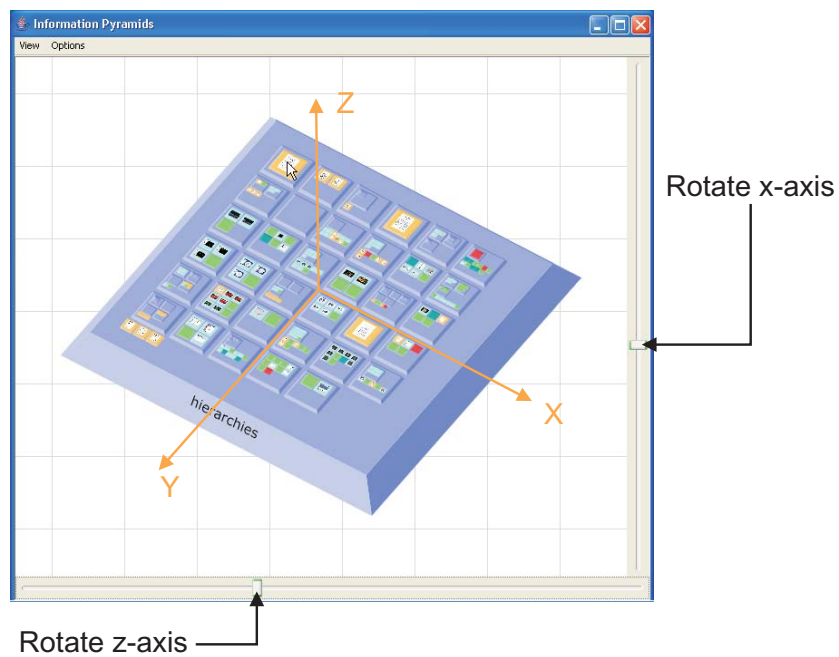
The top surface of a plateau is in general larger than all other surfaces. Thus it is the best place for positioning thumbnails images (see Figure 6.5). It make sense, that the textual description is near the preview image.

## 6.4.2 Navigation

The PyramidsBrowser provides the most important navigation facilities to the user. Since the three-dimensional landscape is simulated, simpler navigational aids can be used for exploring the information space. Users cannot freely move through the three-dimensional landscape. Instead, navigation changes the current view point. Two sliders are used to rotate axes of the pyramid landscape (see Figure 6.6). The slider at the bottom is used to rotate the scenery around its z-axis. Rotating around the x-axis is controlled by the slider on the right. For locally focusing on a particular part of the hierarchy, several navigational aids can be used:



**Figure 6.5:** The top surface of leaf node blocks of the PyramidsBrowser are adorned with thumbnails of the represented documents.



**Figure 6.6:** The sliders of the PyramidsBrowser are used for rotation.



## Zoom

Using the mouse wheel allows users to zoom to a particular part of the hierarchy. The position of the mouse cursor determines the focus of the zoom. This plateau is magnified and kept in the viewing area. Therefore the viewing area has to be adjusted. For smooth zooming the location in three-dimensional space where the mouse cursor points is fixed in the projected two-dimensional image (see Figure 6.7).

## Maximize

Another navigational aid is *maximize*. When the user double clicks on a plateau, this plateau is maximized. Maximizing in this context means that the plateau is zoomed to more or less fill the available screen space. The zoom factor is set to a value that the plateau has the size of the root plateau at zoom factor one. The transition is animated so that users can track the change without losing orientation. This feature allows users to easily navigate to a plateau of interest. Since the plateau is of full size, labels are readable and the thumbnail image (if present) allows a preview of the document.

Maximizing is not limited to one plateau. The user can select a set of nodes to be maximized. Another way to mark plateaus for maximizing is to drag an selection box over the desired nodes. In both cases the bounding box of all selected plateaus determines the zoom level. The center of this bounding box is moved to the center of the display area.

## Pan

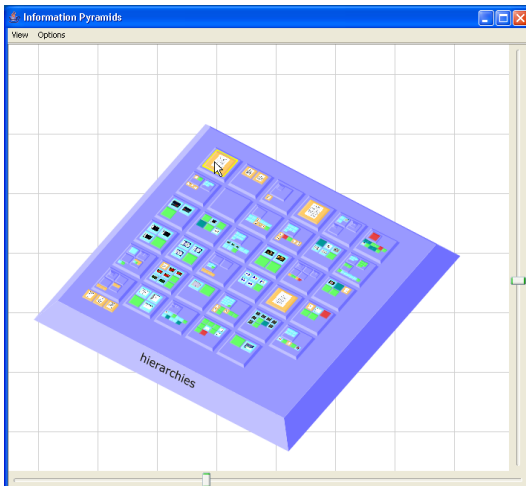
When a particular part of the hierarchy is locally focused, the whole landscape does not fit into the available screen space. The constant sized viewing frame shows only a fraction of the hierarchy to the user at any one time. Panning allows the user to move this viewing frame to change the locally focused part. This enables users to explore the hierarchy at a certain zoom level. Hence, details are visible for users during exploration.

Since visualizations in the HVS environment are all synchronized, the current viewpoint sometimes has to be updated when the user navigates in another visualization. For example, when the user selects a node in another visualization, the plateau representing this node should be visible. *Maximize* this plateau would fulfill this need, but may lead to a loss of global context. Hence the Pyramids-Browser pans the landscape to bring the desired plateau into the visible area. Zooming to a certain level ensures, that the plateau is recognizable for users. How deep to zoom into the scene is configurable by the user. The width of a plateau measured in pixels is a meaningful characteristic for determining the zoom level. Thus users set the desired size of a plateau after zooming and determine thereby the zooming level.

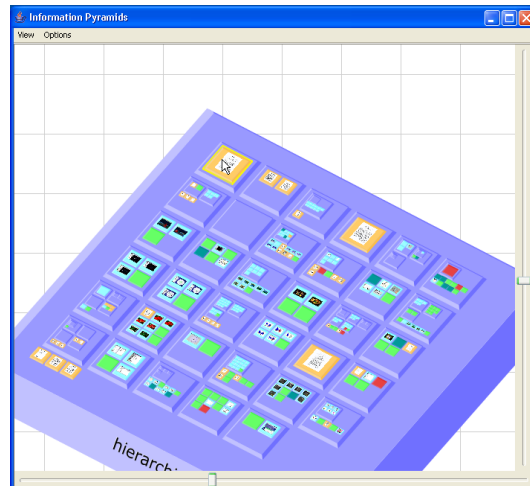
Selected plateaus are drawn with a wire frame box around them. Wire frame boxes are also used to mark plateaus contained in the search result list. The color of these boxes is determined by HVS. If a node is selected and contained in the search result list, a dashed wire frame box in both colors is drawn to especially mark this node.

As described in Section 6.1 the movement from one location to another should be smoothly animated. A conventional pan at first sight would fulfill this requirement. However first zooming out, followed by a short pan, and then zoom to the desired location will help maintain the users global context.

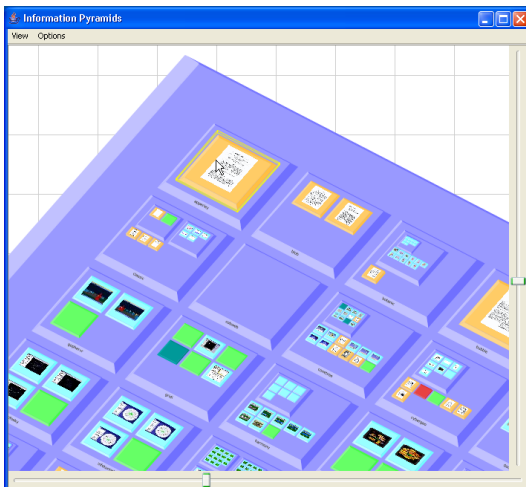
A computational model to calculate an optimal path for zoom and pan is described in van Wijk and Nuij [2003]. To describe the path u,w diagrams are used. Panning denotes u and zooming denotes



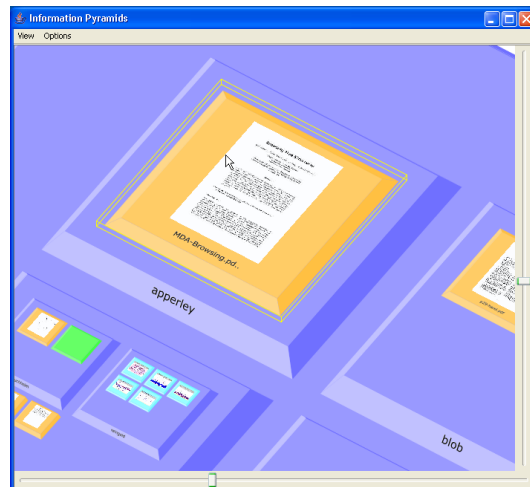
(a) A hierarchy visualized at zoom factor one.



(b) Freely zooming in to the “apperley” directory.

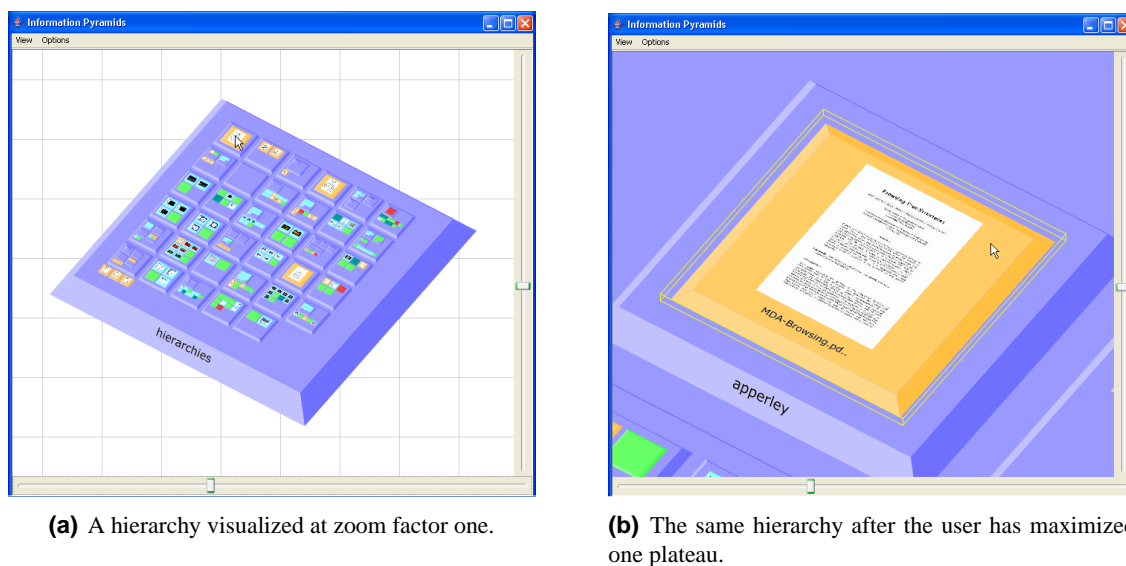


(c) The label becomes visible.



(d) The same hierarchy after the user has continued zooming to the same plateau.

**Figure 6.7:** Freely zooming in the pyramid landscape. Note that the plateau and the position of the plateau relative to the cursor always remain in the same position.



**Figure 6.8:** Maximizing the view of a selected node.

w. Since there is no perceptual difference between horizontal, vertical, and diagonal panning, the optimal path can be considered as a straight line. Analytic solutions for a zoom out - pan - zoom in strategy and for an optimal path are described. Both solutions take the zoom/pan trade-off and the animation speed as parameters.

The PyramidsBrowser uses the optimal path algorithm for animations. Since the variation of the optimal parameters established by user experiments is quite large, the parameters can be adjusted by the user. Figure 6.9 shows the dialog, where users can set these parameters. Users can set these values by adjusting sliders, since absolute values have no meaning for them. A third slider is used to change the zoom-in depth. Thus users can decide how deep to zoom into the scene during animation. This depth is based on the object size measured in pixels.

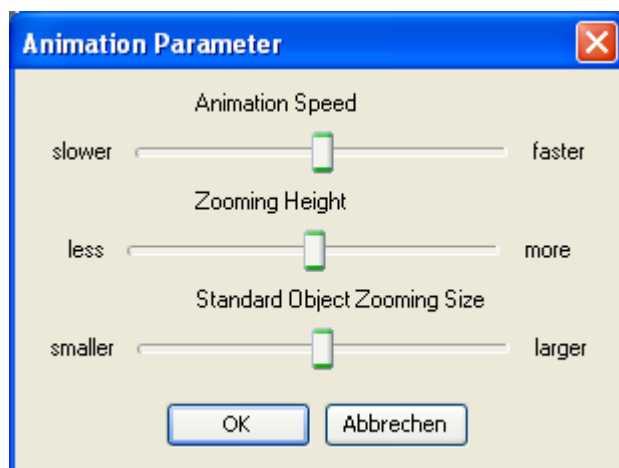
### 6.4.3 Hidden Surfaces and Lightning Model

Painter's algorithm is used to remove hidden surfaces. Objects drawn later cover up those drawn earlier. The name comes from painting. Thus the order of drawing objects determines their visibility.

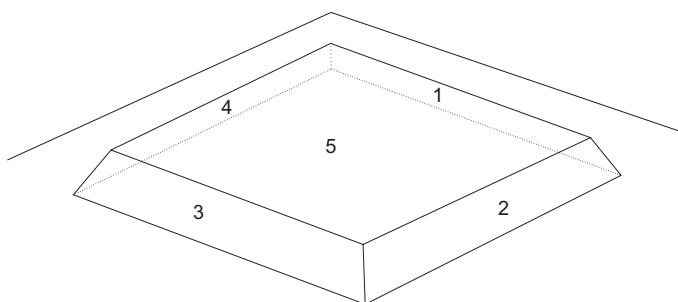
The order of drawing the surfaces of a plateau depends on the rotation around the z-axis. In Figure 6.10, if the angle is less than 45 degrees, surface 1 is first drawn followed by surface 4. If the angle is greater than 45 degrees, surface 4 is drawn before surface 1. The order of the remaining surfaces is independent of the rotation angle. It is surface 2, surface 3 and at last surface 5.

Plateaus are drawn from bottom to top starting with the base plateau. In order to hide non-visible surfaces the drawing order of plateaus on top of a base plateau is important. They are drawn from top left to bottom right (see Figure 6.11). The layout manager must take account of sorting order to hide surfaces which should not be visible.

The most commonly used positioning of a light source is above and to the left of the object. Since the pyramid can be rotated clockwise around its z-axis, both visible lateral surfaces would be dark. Thus the best position for the light source is at the front and to the left of the object. This means that surface 4 and surface 3 are drawn lighter than surface 1 and surface 2. Thereby one lateral surface is



**Figure 6.9:** The animation settings dialog.



**Figure 6.10:** The angle of rotation determines the drawing order of surfaces of a plateau.

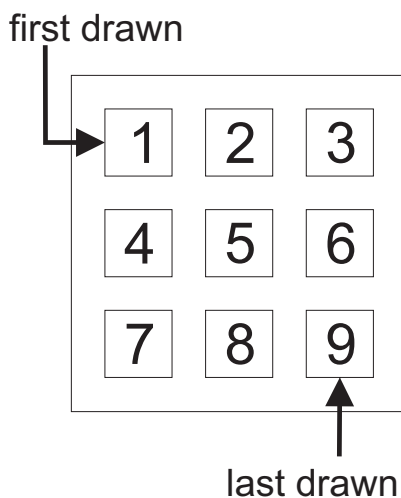
light and one is dark.

Since HVS allows users to change all colors, a method for calculating the colors of the light and the dark lateral surfaces has to be found. The user determines the top color of each plateau. To calculate the colors of the lateral surfaces the color is changed into the HLS color model. The light color is then determined 10 percent lighter than the top color. The dark color is 10 percent darker than the top color. If the top color is too dark or too light, it is capped to produce valid color values.

#### 6.4.4 Rendering Quality and Graceful Degradation

For smooth animations a constant high frame rate is absolutely necessary. The frame rate express as how often the rendered image should be painted in one second. The frame rate should be configurable by users and should be independent of the platform and computer hardware. The number of plateaus to paint influences the paint time as well as the rendering speed of the hardware. During animations it is not necessary to paint all plateaus in every detail. Reducing the level of detail reduces the required time for painting a frame. The following levels of detail are defined:

- FULL: Paint all objects of the pyramid in full detail.



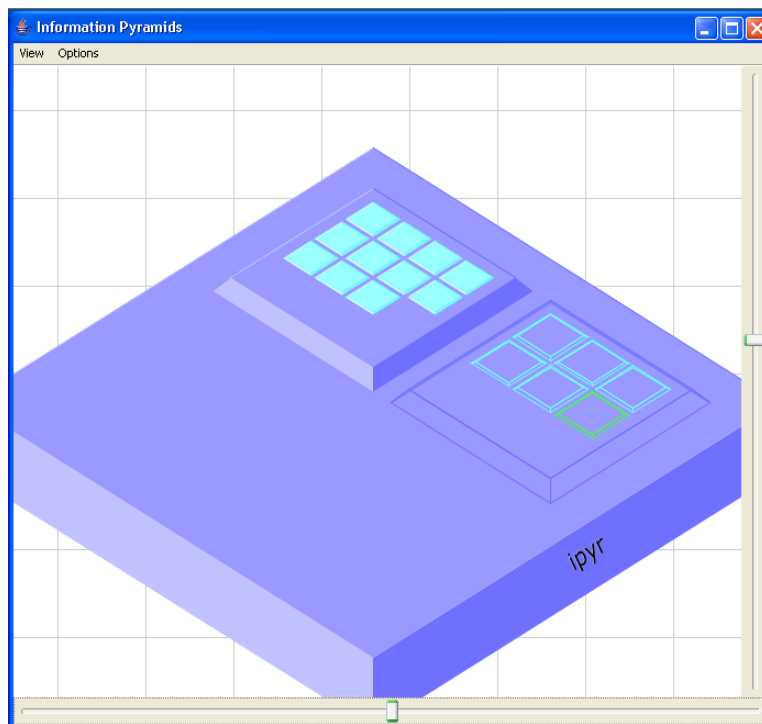
**Figure 6.11:** The drawing order of plateaus on top of a base plateau for correct hidden surface elimination with the Painter's algorithm.

- `ANTIALIASING_OFF`: Sets the Graphics2D rendering hint anti-aliasing to off. This increases the rendering speed approximately by a factor of 10.
- `HIDE_DOCUMENTS_LABELS`: Since rendering labels is a time-consuming operation, not drawing these labels on leaf nodes results in faster rendering.
- `HIDE_LABELS`: Turns off rendering of labels for all nodes.
- `OUTLINE_DOCUMENTS`: Draws leaf node objects only as wire-frames.
- `OUTLINE`: All node objects are drawn as wire-frames.
- `HIDE_DOCUMENTS`: Since the total number of objects to draw influences the rendering time, not drawing leaf nodes reduces the painting time.

To control the frame rate, a mechanism to track the frame rate has to be implemented. Thus the drawing time per frame must be monitored. In order to interactively reduce the details of the painted plateaus, the average time to paint a plateau is tracked. The number of drawn plateaus within a fixed time is determined. Since not all plateaus are of the same size, every plateau is assigned a factor which represents its size proportion. The average time to paint one plateau for each level of detail is kept.

When the plateaus are painted, the number of plateaus to draw is known. The required time to paint is computed by multiplying the average paint time with the number of plateaus. This is done for all levels of detail in ascending order. The level of detail is determined when the computed required time is less than or equal to the available time.

Figure 6.12 illustrates graceful degradation different levels of detail while rendering. The root plateau is painted in full quality. The plateau in the left above the root is painted without labels. The child plateau in the right is drawn as a wire-frame.



**Figure 6.12:** Graceful degradation and different levels of detail during rendering. The root plateau is drawn in full quality, the plateaus in the left subtree are painted without labels, and the subtree at the right is painted as a wire-frame.

## Chapter 7

# Selected Details of the Implementation

This chapter describes some details of the implementation. The programming language used for developing was Java, and in particular the Java Development Kit (JDK) 1.4. This chapter describes a few classes important for a better understanding HVS. HVS currently contains more than 330 classes. About thirty percent of them are inner classes and used for event handling. The next sections describe the purpose of some important classes and their interaction.

The class *Hvs* is the main class of the application. It is used to start the application and to set the look and feel of the *Swing* elements. It opens the main window of the application. The main window is an instance of the *MainFrame* class.

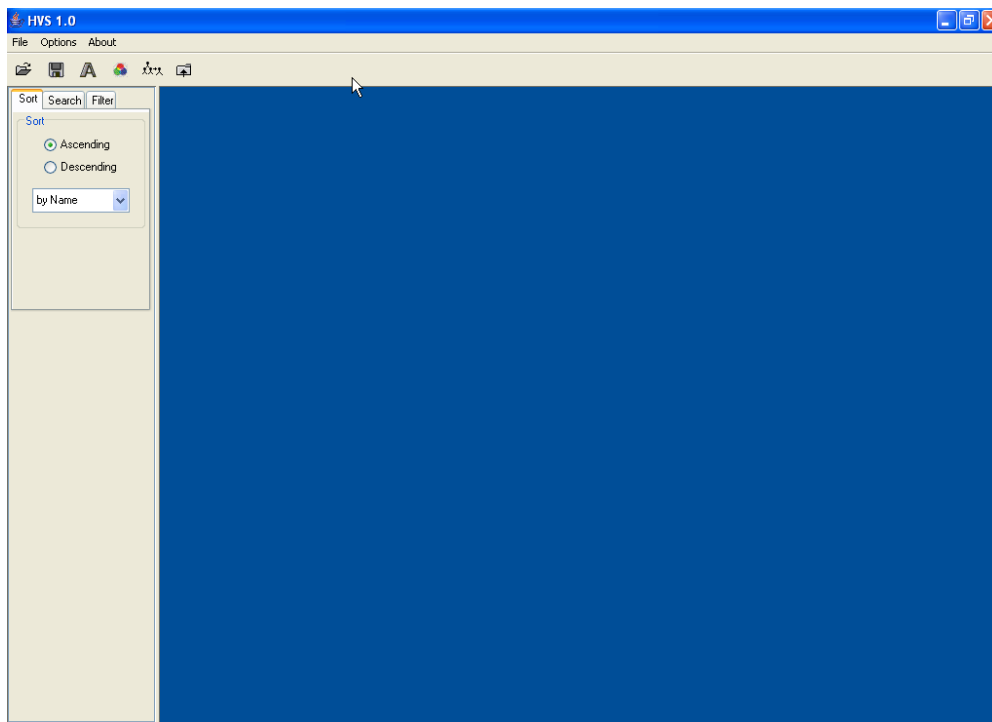
### 7.1 MainFrame

The *MainFrame* class of the package *iicm.hvs.application* has several functions within the application. It is the main window of the multiple window environment for visualizations. The user interface of this class ( Figure 7.1) contains:

- the menu bar
- the tool bar
- the control panel
- the desktop pane

The *MainFrame* class does the whole initialization work. It verifies the HVS directory structure and creates the required directories where missing. After that the default settings are loaded. Default settings are collected in a properties file loaded and managed by the class *HvsDefaults*. Since the properties file only maintains which background color to use, color configuration is loaded afterwards.

Furthermore, the *MainFrame* class manages the event handling of the menu bar and the tool bar. The control panel providing manipulative functionality such as filtering, sorting, and searching handles its own user interaction. For event handling inner classes are used. The advantage of using inner classes is that they have access to the variables of the parent class. The status information managed by the *MainFrame* class contains:



**Figure 7.1:** The main window of HVS.

*visualizationcontroller*\_ This object of type *VisualizationController* manages all visualizations and their synchronization.

*visualizationfactory*\_ A factory class for creating objects of type *Visualization*. This class creates the visualization selected by the user.

*mutabledatamodel*\_ This object of type *MutableDataModel* maintains the hierarchical structure of the information model.

*controllerfactory*\_ The factory object to create a controller managing the synchronization.

*datasource*\_ The *DataSource* object of the desired input module.

## 7.2 ColorTable

The class *ColorTable* manages the color configuration of HVS. It maps integer values to colors. Each integer value represents a type of information. The advantage of this abstraction of mapping is that each attribute can be mapped to a color. Since the type of document objects is defined as an integer greater than zero, they can be simply mapped in this class independent of the input module. Predefined keys are listed all smaller than zero.

Thus the predefined keys are:

```
public static final int TYPE.UNKNOWN = -1;
public static final int TYPE.COLLECTION = -2;
public static final int TYPE.BACKGROUND = -3;
public static final int TYPE.SELECTION = -4;
```



```
public static final int TYPE.SEARCHRESULT = -5;
public static final int TYPE.SELECTEDSEARCHRESULT = -6;
```

The *ColorChooser* class lets the user to change the colors assigned to the keys. For the visual representation of these keys, they have to be mapped to a textual description, since integer values have no meaning for users. This mapping for the predefined keys is done in the *ColorChooser* class itself. For the keys representing defined document types, this mapping has to be made from the input module defining this keys. Since the keys are numbered in ascending order and the desired input module has to return the number of known document types, the dialog shows only defined values. Thus the dialog and the color configuration files can be used for all input modules without adaption.

## 7.3 VisualizationController

The class *VisualizationController* maintains all started visualizations. To manage the synchronization mode of a visualization, an object of type *SynchronizationState* is used. Mapping visualizations to their state objects is one of the main purposes of *VisualizationController*. Thus a change of synchronization mode for a visualization is simply forwarded to the desired state object. Certain changes require synchronization of the visualization. For example, if a visualization is set back into synchronized mode, it has to be resynchronized. This is one of the purposes of this class. The class *SynchronizationState* manages the following status information:

- *independent\_*  
This option is used to switch between independent and synchronized mode.
- *detailedview\_*  
If this option is set, the corresponding visualization is in *DetailedView* mode.
- *hidedocuments\_*  
Setting *hidedocuments\_* true, causes visualizations only to show the structure of the hierarchy.
- *searchresultbridge\_*  
The reference to the *SearchResultBridge* object. This object implements the *SearchResult* interface and manages the behavior of the search result. If the visualization is set to a mode where leaf nodes are hidden, the parent nodes of the search result are highlighted.
- *datamodelbridge\_*  
This object is of type *DataModelBridge* and manages the behavior of the data model when setting the hide documents mode and the *DetailedView* mode.
- *controllerbridge\_*  
This object of type *ControllerBridge* manages the switching of modes from independent mode into synchronized and vice versa.

### 7.3.1 ControllerBridge

The main purpose of the *ControllerBridge* class is to switch between the synchronized controller and the independent controller. It implements the *SynchronizationControllerListener* and the *Controller* interface, so it is notified from the controller. The interface *SynchronizationControllerListener* extends the interface *ControllerListener* by the method *setRoot()*. This supports setting a node as the new root of the hierarchy. This method forwards the request to the desired *DataModelBridge* object. Each visualization is assigned a *ControllerBridge* object and not the real controller. Thus the

*ControllerBridge* object forwards the events sent from visualizations to the desired controller. The events sent from the controller are forwarded to the visualization. This technique supports changing the controller for visualizations at any time. The method used is *setIndependent()*. Hence visualizations do not have to take account of this change. The creation of independent and synchronized controller is managed by a *ControllerFactory* object. This object therefore maintains a reference to that factory object.

## ControllerFactory

The synchronization between different views are managed by a controller. The synchronization mode is changeable at any time by users. When the mode is changed from independent to synchronized, the view has to be resynchronized. However, a resynchronization is also required if the visualized data model changes. For example, after changing the sorting order, the view should visualize previously selected nodes again as selected. The previously focused node should be again the focused node. For synchronized views these states are maintained by the controller. To ensure that resynchronization works as well for independent views, each view is given its own controller set to independent mode.

Since each controller maintains information relating to the currently visualized hierarchy, it has to be updated on changes to the data model. Therefore, it is registered as a listener to the visualized data model. The *ControllerFactory* implements the two following important methods:

***createIndependentController()*** Returns a new instance of a controller, which has the same internal state as the controller for synchronized views. Additionally the newly created object is attached as a listener to the data model.

***getSynchronizedController()*** This method returns the instance of the controller for synchronized views. The independent controller is detached as a listener from the data model.

### 7.3.2 DataModelBridge

The class *DataModelBridge* implements the interface *DataModel* and the *DataModelListener*. It is assigned to the visualizations as the data model to visualize. It is attached as a listener to the class *MutableDataModel*, so it receives events when the hierarchy changes. These events are filtered and forwarded to the appropriate visualizations. This class maintains the reference to the *VisualizationState* object, which maintains the synchronization mode of the corresponding visualization. Furthermore, it maintains the reference to the data model.

When the *DetailedView* mode is set, a new root object is set to this object. Of course, the visualization is notified that the data model has changed. The method *getRoot()* returns the newly set root node. On setting back to the *Overview* mode, this method forwards the request to the underlying data model and the “real” root of the hierarchy is returned. The implementation of this method is:

```
public Collection getRoot()
{
    if (synchronizationstate_.isDetailedView())
        return root_;
    return datamodel_.getRoot();
}
```

Of course, the *DataModelBridge* provides the method *setRoot()* for setting a new root. When the *DetailedView* mode is set and the new root does not equal the current root object, this method notifies the listener that the data model has changed. The return value is in this case true. Thus the

*VisualizationController* object, which calls this method, resynchronizes listeners to show the correct state.

To support visualizing only the structure of the hierarchy, the implementation of the method *getDocuments()* of the *DataModel* interface returns an empty vector. The method *getChildren()* returns therefore a vector containing only *Collection* objects. The source code of these two methods looks like:

```
public Vector getDocuments(Collection parent)
{
    if (synchronizationstate_.isHideDocuments())
        return new Vector();
    return datamodel_.getDocuments(parent);
}
public Vector getChildren(Collection parent)
{
    if (synchronizationstate_.isHideDocuments())
        return datamodel_.getSubCollections(parent);
    return datamodel_.getChildren(parent);
}
```

## 7.4 The PyramidsBrowser

This section describes some details of the *PyramidsBrowser* implementation. The main class of the *PyramidsBrowser* is *PyramidsBrowser*. It subclasses the *Visualization* class of HVS. It manages the communication between HVS and the visualization using the Information Pyramids technique. It handles the general options of this visualization. The panel which visualizes the pyramid is *InformationPyramid2D*.

### 7.4.1 Redisplay

The landscape is not rendered directly onto the screen for performance reasons. Instead, a double buffer strategy is used, whereby the landscape is first rendered into an off-screen image. The advantage of using a cached off-screen image is that the scene has only to be rendered on user interactions. For example, when tool tips are used, hovering over the landscape forces a repaint of the landscape after every mouse movement. Since the rendered image does not change when the tool tip window appears, the off-screen image is drawn on the screen rather than rendering again. Moving the window of the application is the second example where painting the off-screen image can be used.

The rendering of the pyramid landscape is split into three steps:

- Layout of the plateaus.
- Compute graphic primitives.
- Render the graphic primitives into an off-screen buffer.

The layout of plateaus only changes when nodes are inserted into or removed from the data model. Thus a computation of the layout has only to be done in such a case. The abstract class *VRNode* is used to maintain this layout information.

Plateaus consists of five surfaces. Each surface is represented as a graphic primitive, more specially as a *java.awt.Shape* object. The class *VRSurface* is used for this purpose. It implements the *Shape* interface and maintains the graphical primitive as a *GeneralPath* object. Additionally

each *VRSurface* object knows which surface it represents. The shape interface declares the method *intersects()*. This method returns true if the bounding boxes of two shapes intersect. If the bounding boxes of two arbitrary shapes intersect, it is possible that the shapes themselves do not. Therefore *VRSurface* implements a method to test the intersection of two shapes. The algorithm first tests the intersection of the bounding boxes of two shapes. Only if the bounding boxes intersect, the intersection of the shapes is tested. If the intersection area is not empty, the two shapes definitely intersect.

The class *VRPlateau* maintains all graphic primitives required for drawing a plateau. Thus it holds an array of *VRSurface* objects. When a plateau is selected or is a search result, a wire frame box is drawn around it. This wire frame box is not maintained by this class. Nevertheless, the graphic primitives for drawing this box are computed when the box is rendered onto the screen. The *VRPlateau* class has two methods for drawing itself:

***paintWireFrame()*** Draws this plateau as wire-frame.

***paintShapes()*** Fill all shapes of this plateau with the desired color.

When the represented node is selected or is contained in the search result list, both methods compute the wire frame box at the beginning. Therefore the represented *VRNode* must be known. Since selected items are drawn with a surrounding wire frame box, the *VRPlateau* objects themselves do not change on selection. Thus a recomputation of this drawing objects is not necessary. They only have to be drawn again into the off-screen buffer.

The class *VRPlateau* has three further important methods:

***contains()*** The method *contains()* returns true if a given coordinate lies within the area of this plateau. The *java.awt.Shape* interface declares a method *contains()* which satisfy this for the *shape* object. Thus all shapes has to be passed and called the *contains()* method for every *VRSurface* object. If one of the shapes contains the given coordinates, they lie inside of the plateau. A hit detection mechanism is thereby realized. The code fragment for this implementation looks like:

```
public boolean contains(double x, double y)
{
    for (int i = surfaces_.length - 1; i >= 0; i--)
    {
        if (surfaces_[i].contains(x, y))
        {
            return true;
        }
    }
    return false;
}
```

***intersects()*** Tests the intersection of one of the shapes of this plateau with an arbitrary shape. The algorithm to detect the intersection is similar to that for the *contains()* method, except the method *intersect()* is called on every *VRSurface* object.

***paint()*** This method forwards the request for painting to the corresponding *VRNode* object.

## 7.4.2 VRNode

Each *VRNode* object corresponds to a node of the visualized hierarchy and represents a plateau. *VRCollection* objects are used to represent inner nodes. Leaf nodes are represented by objects of type

*VRDocument*. Both classes extends the *VRNode* class. There are several reasons for using different objects for inner nodes and leaf nodes. Firstly, the algorithm for reducing the level of detail during animations (see Section 6.4) has different behavior for leaf and inner nodes. Secondly, the placement of labels is completely different. Thirdly, leaf node objects can visualize thumbnail preview images. The relationship between *VRNode* objects corresponds to the structure of the visualized hierarchy. Each *VRCollection* object maintains a reference to its parent and its child nodes. *VRDocument* objects only maintain a reference to their parent.

The class *VRCollection* and the class *VRDocument* are both subclasses of the abstract class *VRNode*. They implement the two abstract methods:

- *computeVRPlateaus()*
- *paint()*

The method *computeVRPlateaus()* is used to compute the graphic primitives of all plateaus. This method is optimized to compute only the necessary minimum number of plateaus. The creation of the corresponding *VRPlateau* object is forwarded to the *InformationPyramid2D* object, which maintains all objects. The computation starts at the root *VRNode* and goes down the hierarchy. Only when the return value of this function is *PLATEAU\_OK*, does computation continue on child plateaus.

When a plateau is so small that it is not visible, then its child plateaus also cannot be visible. Hence, it makes no sense to compute child plateaus of such plateaus. The return value therefore is *PLATEAU\_TOSMALL*. When the layout algorithm uses equally sized plateaus, then all siblings are also too small to be visible and computation of siblings can be stopped immediately.

Furthermore child plateaus do not have to be computed when the plateau lies outside the visible area to the left, right or top. Since parallel projection is used and pyramids only grow upwards, children of such plateaus can never become visible. *PLATEAU\_OUTSIDE* is the return value in this case.

If a plateau lies outside the visible are to the bottom it is also not visible. However, its children may become visible, since the pyramid grows upwards. Hence, in such a case the function returns the value *PLATEAU\_OK*.

The implementation of the *computeVRPlateaus()* for the *VRCollection* class looks like:

```
public int computeVRPlateaus()
{
    int retValue=pyramid_.createVRPlateau(this);
    if (retValue !=InformationPyramid2D.PLATEAU.OK)
        return retValue;
    pyramid_.createLabelShape(node_);
    if (children_==null)
        layoutChildren();
    for (Iterator it= children_.iterator(); it.hasNext();)
    {
        VRNode vrNode=(VRNode) it.next();
        int ret=vrNode.computeVRPlateaus();
        if (ret==InformationPyramid2D.PLATEAU.TOSMALL)
        {
            if (vrNode.isVRDocument() &&
                pyramid_.getLayoutManager().isDocumentsEqualSize())
                return InformationPyramid2D.PLATEAU.OK;
            if (!vrNode.isVRDocument() &&
                pyramid_.getLayoutManager().isCollectionEqualSize())
                return InformationPyramid2D.PLATEAU.OK;
        }
    }
}
```

```

    }
    return InformationPyramid2D.PLATEAU_OK;
}

```

The *paint()* method draws the corresponding *VRPlateau* object in the given level of detail.

### 7.4.3 InformationPyramid2D

The panel *InformationPyramid2D* displays the three-dimensional visualization of the Information Pyramid. It is a subclass of *javax.swing.JPanel* and holds the status information of the pyramid. It reacts to user interactions and forwards the painting requests to the desired visual representation (VR) objects.

The status information managed by this class contains:

- *root\_*  
This object represents the root node of the hierarchy. The type of this object is *VRCollection*.
- *image\_*  
The rendered image of the pyramid. This image is painted in response to operating system repaint requests.
- *layoutmanager\_*  
The reference of the layout manager used to compute the dimensions and sizes of plateaus. The type of this object is *VRLayout*. An exchange of the layout manager during runtime causes a re computation of the visualization. To avoid confusing users the focused plateau is kept in the display area.
- *renderer\_*  
This object is used to calculate the two-dimensional representation of plateaus. It holds the status information of the currently displayed rendered image.
- *vrplateaufactory\_*  
This factory object of type *VRPlateauFactory* is used to create *VRPlateau* objects. These objects hold the surfaces of plateaus. The *VRPlateauFactory* class tries to reuse *VRPlateau* objects rather than creating new objects. Since object creation and garbage collection requires certain system resources, reusing objects increases the speed of the application.
- *levelcontrol\_*  
To reach a constant high frame rate during animation the quality of rendering is gracefully reduced. *LevelControl* is the object is used to manage the reduction in level of detail of the plateaus.
- *fullqualitypaint\_*  
During animation the landscape is rendered in less detail for a constant high frame rate. Since rendering requires certain time, the landscape is drawn in a background thread when the pyramid becomes stationary again. The *FullQualityPaint* class renders the landscape into an off screen image. If the user interacts with the pyramid, this rendering is interrupted immediately.

*InformationPyramid2D* does not render the plateaus of the pyramid directly. Instead, a double buffer strategy is used. The *paint()* method paints only the rendered off-screen image onto the screen. On repaint requests from the operating system, for example when the window is moved, the pyramid does not have to be computed again. Only when the window is resized, the *paint()* method renders the off screen-image and draws it onto the screen.

#### 7.4.4 VRLayout

Layout manager objects are used to compute the size and position of plateaus in the pyramid. The three-dimensional coordinates of the base plateau are fixed. All other plateaus are laid out on top of the base plateau. Every *InformationPyramid2D* object holds the reference to the used layout manager. All layout manager objects have to implement the abstract method *layoutSubCollections()* defined in the abstract class *VRLayout*.

This method calculates the size and position of all children of a given node. The method returns an object of type *Vector*. The elements of the returned vector are of type *VRNode*. It is necessary that the elements of the vector are sorted by their position in the three-dimensional space. In order to remove hidden surfaces, the plateaus are sorted from back to front and from left to right.

Three different layout managers are available in the Pyramidsbrowser:

***VRBoxLayout*** The *VRBoxLayout* uses equally sized plateaus to represent inner nodes. Leaf nodes are represented by smaller boxes. The ratio which leaf nodes are visualized smaller than inner nodes is configurable by users. Plateaus are arranged in a matrix. The default height of each plateau representing inner nodes is seven percent of the square size. The height of leaf node objects is logarithmically scaled relative to the available inner node height corresponding to the size of the represented node. If all siblings are leaf nodes, the whole area is used to visualize them. Otherwise leaf nodes are arranged at the front and inner node plateaus at the back.

***VRSizeSortedLayout*** The *VRSizeSortedLayout* class is a layout manager that uses different sized plateaus to represent inner nodes. The dimension of a plateau is proportional to the size of the corresponding inner node. The size of leaf node objects is reduced by a user-configurable factor. This factor is determined as the ratio of the size of the leaf node's plateau to the size of the smallest inner node plateau. This layout manager tries to maximize the size of each visualized plateau to use the available display space efficiently. Thereby it arranges leaf nodes and inner nodes in the same manner. The layout algorithm is the same as Michael Welz described in Welz [1999] for arranging inner nodes:

1. Sort the inner nodes by their size in descending order.
2. Compute the total size of all children which are inner nodes and the percentage of those nodes compared to their total size. The percentage of a node is used as the size of its plateau. If the percentage of a node is smaller than a given minimum size, increase its percentage to the minimum size. For leaf nodes set the minimum size for leaf nodes.
3. Place the biggest plateau in the top left corner of the square shaped display area.
4. Position the second largest plateau to the right of the first one.
5. Compute the minimum bounding rectangle (MBR) of the used display area.
6. As long as the MBR does not change, place the following plateaus below the second plateau. As the nodes are sorted descending order, the sizes of the following plateaus are equal to or smaller than the size of the second plateau. Therefore the MBR can only get higher but not wider.
7. Compute the MBR if the next plateau would be placed to the right of the last MBR and if it would be placed below it. Compare both MBRs with a square and use the position where the MBR is more equal to a square.
8. If the plateau is placed to the right of it step 6 is repeated. If it is placed below the MBR the following plateaus are placed to the right of the plateau until the MBR gets wider.
9. Repeat step 7 and step 8 until all plateaus are arranged.

10. Now the positions of the plateaus are computed but the sizes of them are too large to fit into the available display space. Therefore the plateaus have to be shrunk to fit on the top of the plateau of the parent.

The height of the inner node's plateaus are calculated as the user-configurable height proportion in percent multiplied by the size of the largest plateau. The default value of the height proportion is 7 percent. The height of leaf node plateaus corresponds to their size.

***VRNumberOfChildrenLayout*** This layout algorithm is similar to the *VRSizeSortedLayout* algorithm, except the size of plateaus is determined by the number of children rather than their size.

Figure 7.2 illustrates the effect of these three different layout algorithms. In all three figures, the same hierarchy is visualized only the layout algorithm is changed.

### 7.4.5 Drawing Textual Labels Onto Surfaces

For realistic representation, labels must be deformed and moved to the respective surface. However, characters are only available as a two-dimensional surfaces and plateaus are computed in three-dimensional space. The two-dimensional shapes to render are determined using parallel projection. This computation is not possible for labels since labels are already complex shapes without explicit coordinates.

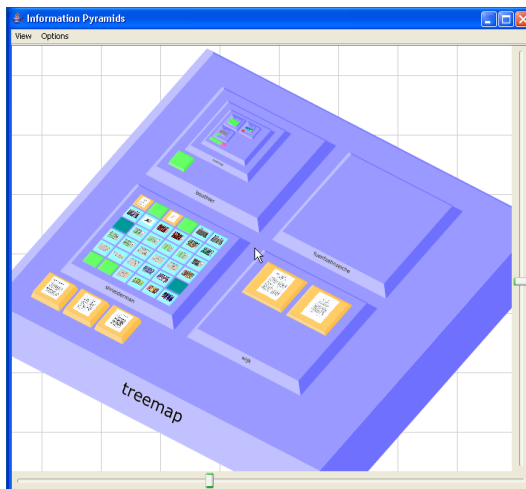
One of the strengths of Java2D is that shapes can be manipulated in many ways. It allows transformation of any graphic object when rendered. Transformations can include translation, rotation, scaling, shearing, or a combination of these. Thus for rendering labels a transformation has to be found to fulfill this requirement.

The *java.awt.geom.AffineTransform* class is used for transformations in the Java2D API. This class builds an affine transformation. Affine transformations keep parallel lines parallel when objects are transformed. To compute an affine transform which manipulates the label in suitable manner, the problem is reduced to computing the transformation for an arbitrary point. If the transformation works fine for one point in the coordinate space it will work for all points as well.

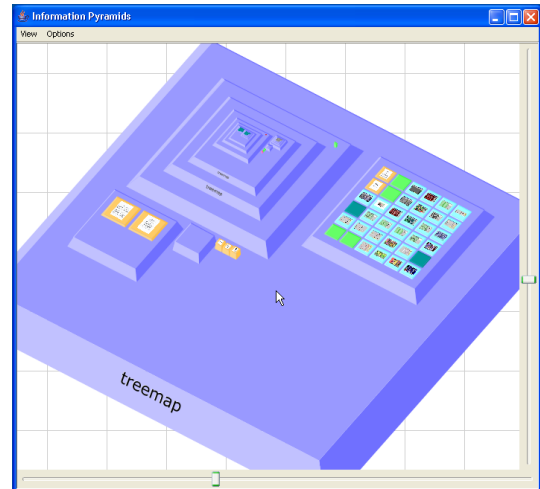
1. Take point at the bottom left of the label. Set its z coordinate to zero.
2. Translate in the x-direction by half the label width. The center of the label is in the center of the coordinate space.
3. Scale the size of the label to fit into the available area.
4. Compute the angle between the front surface and ground surface of the plateau. Rotate around the x-axis by this angle. The label is now parallel to the front surface.
5. Translate to the center of the front surface. The label is now at the front surface if the pyramid is not rotated.
6. Rotate around the x-axis and z-axis by the rotation angle of the pyramid.
7. Compute the two-dimensional coordinates using parallel projection.

The resulting transformation modifies each coordinate of the shape. Transforming the user coordinate space has the same effect as creating a transformed shape. The coordinate space is transformed with the computed affine transform before the label is rendered.

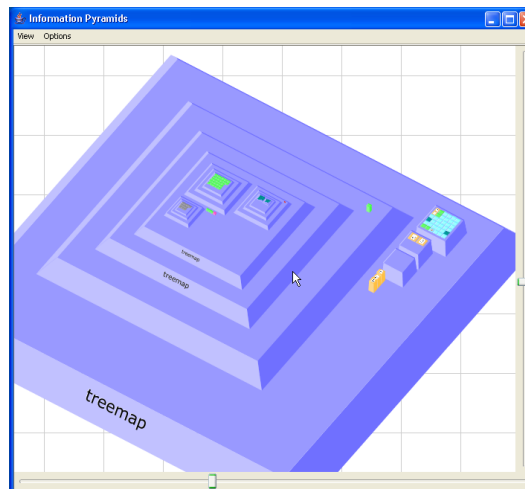




(a) The pyramid using the VRBoxLayout layout manager. Nodes are sorted according to the current sort order.



(b) The pyramid using the VRSizeSortedLayout layout manager. Nodes are sorted in decreasing order of their size.



(c) The pyramid using the VRNumberOfChildrenSortedLayout layout manager. Inner nodes are sorted in decreasing order by their number of children.

**Figure 7.2:** The different layout algorithms of the PyramidsBrowser. All three screenshots illustrate the same hierarchy.

If the transformed label is too small to be visible, it is not drawn. The minimum height of drawn labels was determined through experimentation as four pixels. The code fragment of the method for creating such an affine transform looks like:

```
public AffineTransform getAffineTransform(double[] point3D,
    double alpha, double scale, double tx, double ty, double height)
{
    alpha = -90d + alpha / InformationPyramid2D.TORAD;
    double cosL3 = Math.cos(alpha * InformationPyramid2D.TORAD);
    double sinL3 = Math.sin(alpha * InformationPyramid2D.TORAD);
    if (multiplier_ * zoom_ * scale *
        (sinL1_ * cosL1_ + cosL3 * cosL2_ *
         cosL1_ - sinL3 * sinL2_) * height < 4)
    {
        return null;
    }
    aft_.setTransform(multiplier_ * zoom_ * scale * cosL1_,
        multiplier_ * zoom_ * scale * sinL1_ * cosL2_,
        -multiplier_ * zoom_ * scale * cosL3 * sinL1_,
        multiplier_ * zoom_ * scale * (cosL3 * cosL2_ *
         cosL1_ - sinL3 * sinL2_),
        view_center_x_ + multiplier_ * zoom_ * (
            (scale * tx + point3D[0]) * cosL1_ -
            (scale * ty * cosL3 + point3D[1]) * sinL1_),
        view_center_y_ + multiplier_ * zoom_ * (
            (scale * tx + point3D[0]) * sinL1_ * cosL2_ +
            (scale * ty * cosL3 + point3D[1]) * cosL1_ *
            cosL2_ - (scale * ty * sinL3 + point3D[2]) *
            sinL2_));

    return aft_;
}
```

### 7.4.6 Hit Detection

When the user clicks on a plateau, the system responds by drawing a wire frame box around the clicked plateau. The plateau corresponding to the two-dimensional coordinates of the mouse position have to be found. For this purpose the *Renderer* object maintains all painted *VRPlateau* objects. The algorithm to find the clicked plateau passes the vector from the back to the front. It calls the *contains()* method, which returns true if the given coordinates hit one of the surfaces of the plateau. The first object which returns true is the derived plateau. The source code that looks for the corresponding *VRPlateau* object looks like:

```
public VRPlateau getVRPlateauForLocation(double x, double y)
{
    for (int i = plateaus_.size() - 1; i >= 0; i--)
    {
        if (plateaus_.get(i).contains(x, y))
        {
            return plateaus_.get(i);
        }
    }
    return null;
}
```

### 7.4.7 Panning

Panning is a navigational aid panning resulting in two-dimensional movement of the rendered image. The pyramids landscape represents a three-dimensional building which can be rotated. When the pyramid is rotated, the three-dimensional point in the center of the viewing area should be fixed to avoid confusion to the user. This point must therefore be known when rotating.

A possible solution is to compute this point when the pyramid is rotated. When this computation is done during rotation the animation looks like trembling. Hence, a better solution is to determine this three-dimensional point during panning animations. Since this point corresponds to the center of the visible area it is called *ViewCenter3D*.

The *z* coordinate of the *ViewCenter3d* is determined as the highest visible *z* coordinate of the plateau in the center of the display area. This seems the best choice, since the rotation around the *z*-axis keeps the top of the visualized pyramid in the viewing area. Of course this computation is only possible when the projective ray through the center of the viewing area hits at least one plateau. This limits the movement possibilities of panning. This disadvantage barely arises, since panning is used when the pyramid is magnified. Furthermore, it avoids the possibility of moving the pyramid out of the visible area.

Since parallel projection is used only the three-dimensional rotated *x*- and *y*- coordinate of the pyramid influences the two dimensional coordinates. Thus the third dimension has to be determined using geometrical identities. The normal line of the *x*-*y* plane through the center of the viewing area is used. Depending on the intersected surface of the plateau the *z*-coordinate is determined:

- When the normal line intersects the top surface, the *z* coordinate is determined by an arbitrary point on this surface.
- The computation of the intersection of two straight lines is used to compute the *z* coordinate when the normal line hits a lateral surface. The first line is given by the normal line. The desired surface is reduced to a line. For example, the view of the left surface in the *x*-*z* plane is a straight line. This is the second line used to compute the intersection.

After determination of the *z* coordinate, the *x*- and the *y*-coordinate can be computed from the image by inverting the projection.

## 7.5 FSDataSource

The *FSDataSource* module reads in the hierarchical structure of a file system. The visualization of the directory hierarchy is not limited to the standard information provided by typical file browsers. Metadata attributes are extracted out of known documents. Of course, extraction is only possible when a file contains this information. The extraction procedure takes a long time and requires much computation.nal performance. To improve performance, the attributes are stored into a temporary file. To assign attributes to the object representing the associated file, the absolute path of the file is used as key. To take account of changes of files, attributes are validated before assigning them to the desired object. Since a file system returns when a file was last modified, this attribute is suitable for validation.

All metadata information is stored in one file and all attributes are always available. When only a part of the file system is visualized, keeping all metadata attributes in main memory would cause a high memory footprint. Since the path of the selected root of the hierarchy is known when creating the abstract data model, only those attributes have to kept, whose path is descendant of the path to the root.

```
<!ELEMENT FileHandler (Entry*)>
<!ELEMENT Entry (Type, Description, Application?, Filter+)
<!ELEMENT Type (#PCDTATA)>
<!ELEMENT Description (#PCDTATA)>
<!ELEMENT Application (#PCDTATA)>
<!ELEMENT Filter (#PCDTATA)>
```

**Figure 7.3:** The Document Type Definition of the configuration file of the FSDataSource input module.

The following metadata attributes are generated from files where possible:

- Author
- Title
- Subject
- Number of pages
- Keywords
- Comments
- Creation date
- Document modified
- File modified

Of course, only certain files contain more information than the file last modified attribute. Additionally, thumbnail images in four different sizes are created from images and Adobe PDF documents.

Documents are organized in types. These different types are mapped to color attributes. The type is determined by the extension of the file. Figure 7.5 shows the Document Type Definition of the type configuration file. The declared tags are:

- Type. A numeric value defining the assigned type.
- Description. The description which appears in the user interfaces.
- Application. The application used to open documents of this type. This field does not have to be set on Microsoft Windows operating systems.
- Filter. Defines the extensions of files which should be assigned to that type.

# Chapter 8

## Outlook

### 8.1 Work in Progress

Current work in progress is concentrating on integrating different kinds of hierarchical visualization techniques, including Treemaps, hyperbolic browser, magic eye view, and cone trees.

**Treemaps** A Treemap visualization was developed and integrated in the HVS environment. The user can choose between two different layout algorithms. Implemented are the simple slice and dice algorithm and the squarified treemap algorithm. Figure 8.1 shows a file system visualized as Treemaps.

Further development of the Treemap visualization involves adding zooming functionality. A zooming algorithm, similar to that used in the Information Pyramids technique, where users can simply zoom to a particular part of the hierarchy by scrolling the mouse wheel is the next step in development. The zooming mechanism could be extended with panning. Smooth animations would be necessary since the visualization has to respond to user interactions in other views.

**Hyperbolic Browser** An implementation using hyperbolic geometry is integrated in the HVS environment for demonstration and research purposes. The first impressions of our developed prototype is that the navigation in the hyperbolic space feels natural. Of course, this implementation is fully synchronized with all other views in HVS. The direction of the layout algorithm can be chosen: radial, left, right, top, and bottom. Figure 8.2 shows the prototype hyperbolic browser using the radial layout direction.

**Magic Eye View** The magic eye view (see Figure 8.3) is another visualization technique currently being integrated in HVS. The simple two-dimensional layout is mapped onto the surface of a hemisphere using geometrical projection. Work is ongoing, since the movement of the projection center does not feel as natural as in the hyperbolic browser.

**Walker Layout** For demonstration purposes, the Walker layout algorithm is another implemented visualization (see Figure 8.4). This implementation provides zooming functionality. The tree can be stretched in the x- and y-directions. Scroll bars are used if the tree does not fit into the available screen area.

**Cone Trees** Work is currently going on to develop a second three-dimensional visualization for HVS, the well-known cone trees technique using OpenGL for Java.

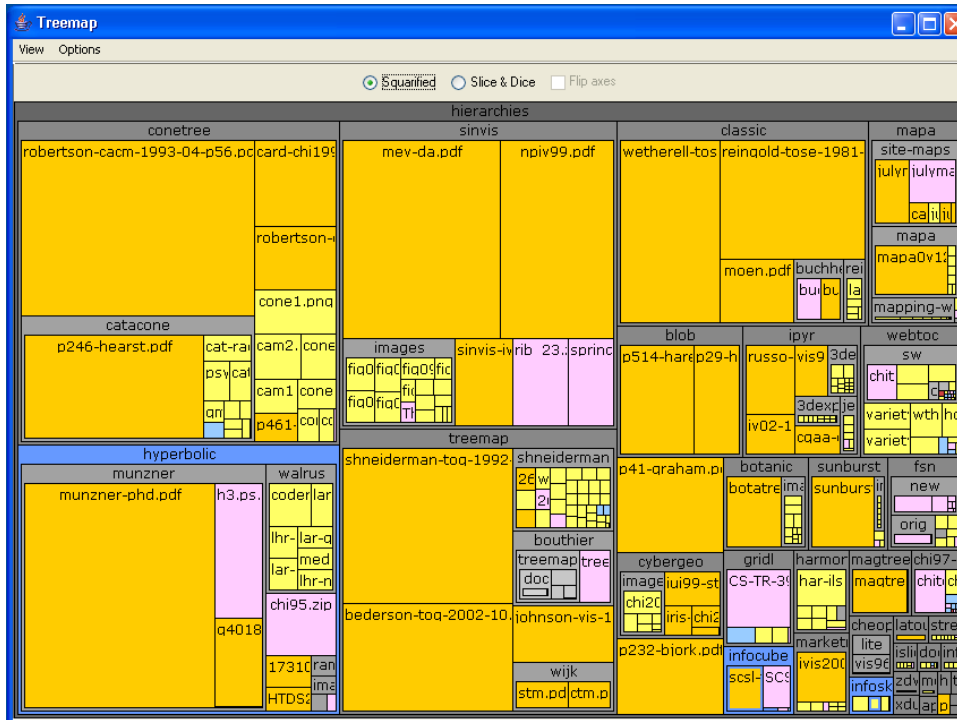


Figure 8.1: An implementation of TreeMaps in HVS.

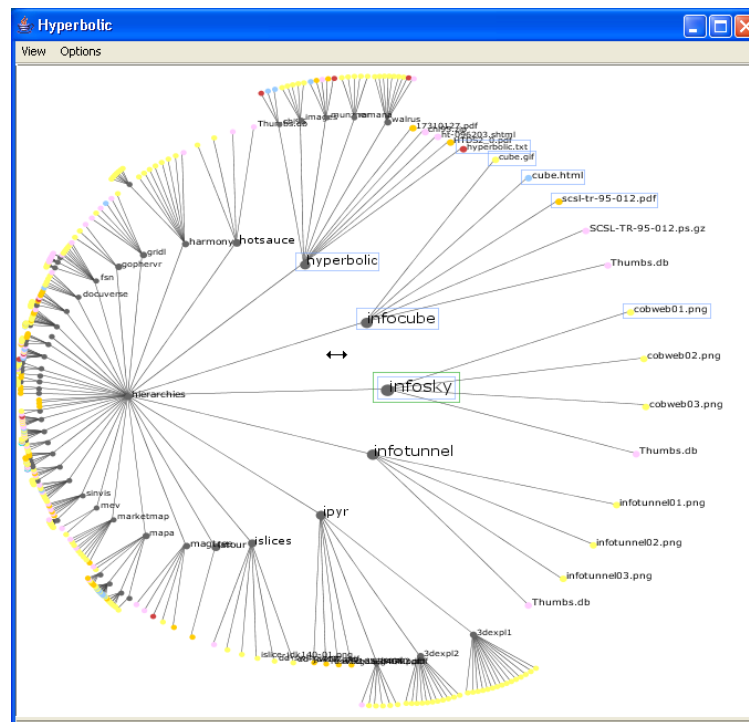
## 8.2 Navigational History and Bookmarks

Navigational history is not currently implemented. It would be useful to have unlimited undo and redo operations in exploring hierarchies. This would allow users rapid interactive and reversible exploration of the hierarchy. For navigational history user interactions have to be logged. Since all user interactions are sent as events to a controller, this logging would be quite simple. A *Forward* and a *Back* button would have to be included into the tool bar to navigate through the history. Additionally, a list view of all logged navigation points should be provided. This would allow the user to directly jump to the desired location.

Another navigational aid that can be implemented are *Bookmarks*. It would be nice to mark items in a hierarchy and recall them by simply selecting in a separate list. This can be done by either marking nodes in the hierarchy or marking positions in the landscape. Preferable is to mark nodes, since positions depend on the used layout algorithm.

## 8.3 Flexible Mapping of Attributes

The current implementation of HVS has static color mapping. Thus the application maps the type of a document to a color in the visualization. The input module for the file system uses the extension of files to identify the document type. It would be nice to map an arbitrary attribute to colors in visualizations. For example, the age of files could be visualized using color attributes. The decision which attribute to map to color attributes should be made at runtime. A possible solution is to provide a menu item for input modules to choose the mapping of an attribute to a document type. Each input module has to implement this mapping and a user interface to configure this mapping.



**Figure 8.2:** An implementation of a hyperbolic browser in HVS.

## 8.4 Comparative Study

The HVS framework provides a powerful tool for comparing different visualization techniques for hierarchies. Thus it seems to be an optimal tool for an comparative study. The aim of the experiment should be to find out difference in user performance between different visualization techniques for different tasks and different hierarchies.

A number of representative participants have to be acquired. Typical tasks are then performed by these participants. For an objective result, all test have to be run on the same platform, since the performance depends on the hardware, operating system, and maybe installed software packages. The facilitator should conduct a training session for HVS and each tool. After training, the participants should have time for free exploration to get a feeling for the tools in HVS. The time of each task should be measured. Typical tasks to perform might include:

- Locate a file determined by a path.
- Locate all files of a given type.
- Determine the most common file type.
- Locate the largest file.
- Locate the largest file of a certain type.

After measuring the time it takes to perform a set of tasks, a post test questionnaire could be administered to assess the subjective impressions of the participant. Subjective feelings of users are

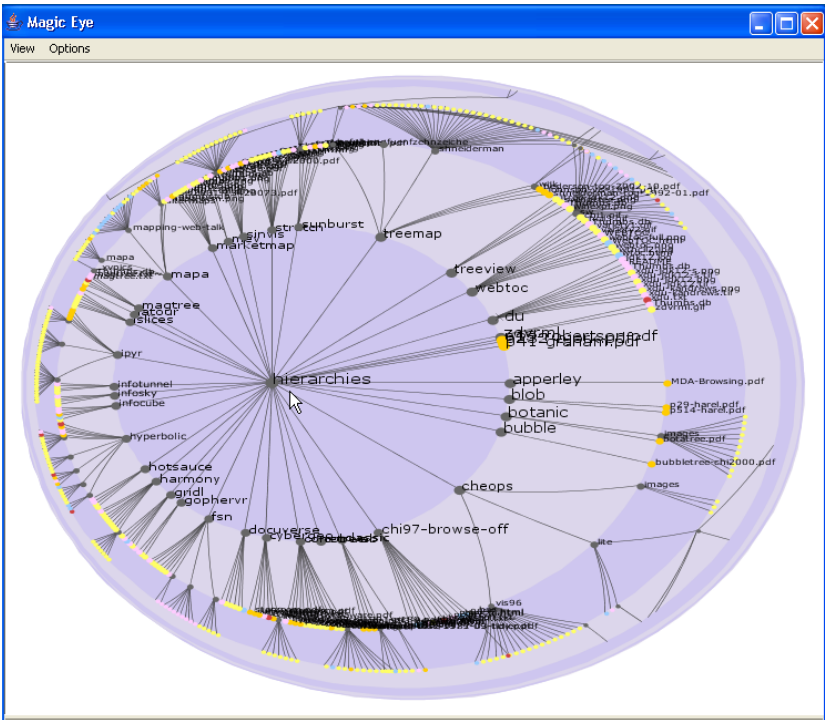
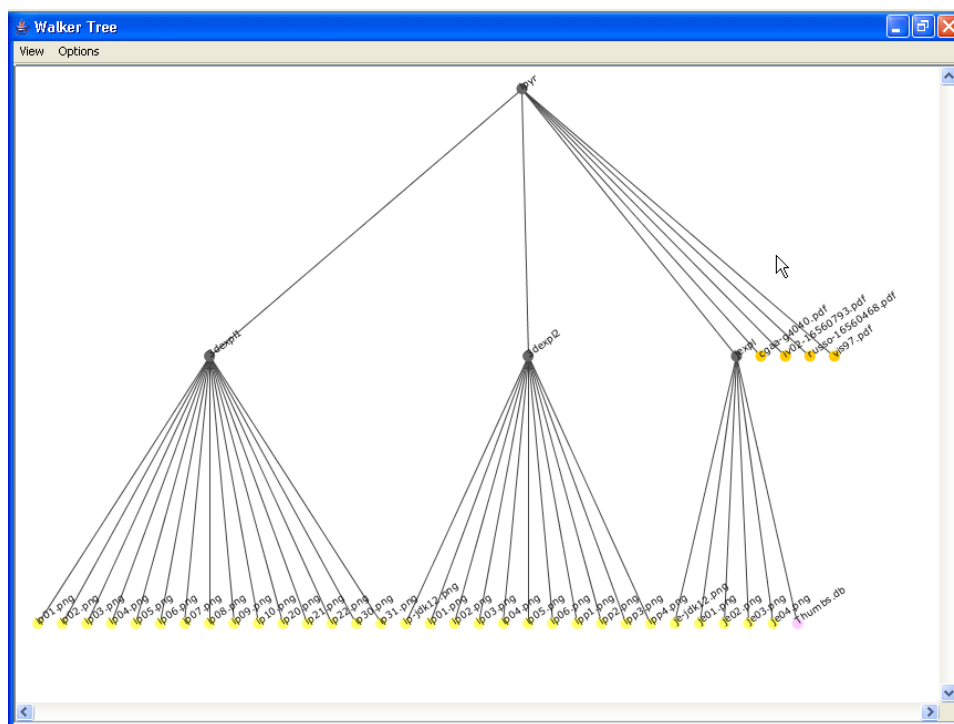


Figure 8.3: A implementation of the Magic Eye in HVS.

considers as well as the objective timing to finishing each task. The comparison of the different techniques would allow to determine the best technique for specific tasks and types of hierarchies.





**Figure 8.4:** The implementation of the walker layout algorithm in HVS.



## Chapter 9

# Concluding Remarks

This thesis presented the Hierarchical Visualization System, a framework for visualizing hierarchically structured information. A traditional tree view browser and the PyramidsBrowser, a visualization tool using the Information Pyramids technique, are provided as a sample of flexibility.

Chapter 2 gave a short overview of the information visualization research field. Tools for the different types of information were presented. Techniques for visualizing hierarchies are discussed in detail in Chapter 3. Some toolkits for building information visualizations were described in Chapter 4.

The Hierarchical Visualization System was discussed in detail in Chapter 5. The architecture and main modules of HVS were presented. As part of this framework, the PyramidsBrowser using the Information Pyramids technique was described in Chapter 6. New navigational aids were developed as well as improved labeling of the plateaus. In Chapter 7 selected details of HVS and the PyramidsBrowser were described. Techniques to improve the performance and the layout of plateaus were discussed in detail.

The thesis concluded with an outline of work currently in progress, and some ideas for future research.



# Appendix A

## HVS User Guide

### A.1 Installation

To install HVS the zip archive has to be uncompressed. When uncompressing the archive, a directory structure is created. The *plugin* directory contains all standard plug-ins of *HVS*. The *userconfig* directory is initially empty. It is used to store the session files. The *lib* directory contains the *log4j* library. The *colorconfig* directory contains two *XML*-files. These two files contain the color definitions of HVS. One file for a dark background color and one file for a light background.

The Java environment has to be set correctly before the application can be started. The application requires JDK 1.4.x or later.

#### A.1.1 Installation of a Plug-in

The installation of a plug-in simply involves copying it into the *plugin* directory. Developers design their plug-in mostly as a zip archive, users only have to uncompress the archive in the *plugin* directory. Each functioning plug-in has a configuration file. The Java class files are either in a subdirectory or in a jar file. The installation process is completed by restarting the application.

### A.2 Starting HVS

HVS is started under Windows using the *hvs.bat* file. The batch script looks like:

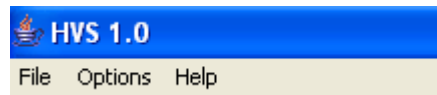
```
java -Xms32M -Xmx256M -jar hvs.jar
```

The Java option *-Xmx256M* allows the Java virtual machine to use 256 Mega bytes of main memory. This is sometimes necessary for HVS to have enough space to create thumbnail images.

### A.3 HVS

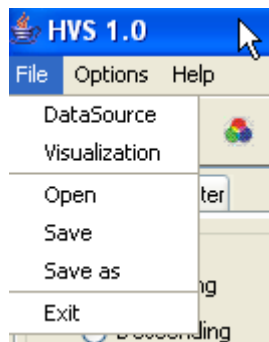
#### A.3.1 The Menu Bar

The menu bar provides easy access to the main functions available in HVS. Three menus are provided: File, Options, and Help.



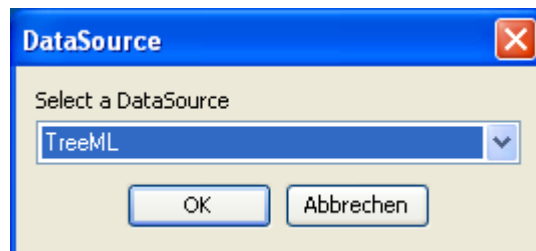
**Figure A.1:** The menu bar.

## File Menu



**Figure A.2:** The HVS file menu.

**DataSource:** Opens a dialog box (see Figure A.3) to select a hierarchy. All opened views are updated to visualize the newly selected hierarchy.



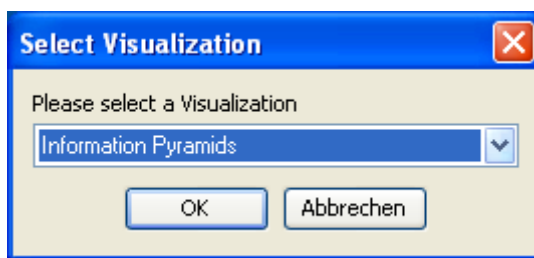
**Figure A.3:** The dialog box to select a data source.

**Visualization:** Opens a dialog box (see Figure A.4) to select a visualization. The selected visualization is opened in a new window and displays the hierarchy.

**Open:** Opens a dialog box (see Figure A.5), where a configuration file can be selected. This configuration is afterward loaded.

**Save:** Saves the session data to the current selected configuration file. If no configuration file was previously selected by loading or saving, a dialog appears to choose one.

**SaveAs:** Opens a dialog box (see Figure A.6) to select a configuration file. The session data is then stored to this file.



**Figure A.4:** The dialog box to select a visualization.

**Exit:** Closes the application.

### Options Menu

**Color:** Opens a dialog box where the user can set different color attributes. Depending on the selected input, the look of this dialog varies. Figure A.11 shows the standard options of this dialog. After modification, the new configuration can be saved as default configuration. Two different configurations can be chosen:

- “light on dark”. The background is drawn in dark color, all items are drawn in a lighter color.
- “dark on light”. The background color is light, all other items are drawn in dark colors.

**Font:** Opens a dialog (see Figure A.10) where the user can set the font used in visualizations. The font family and the size can be configured.

### Help Menu

**About:** Opens a window which displays some information about HVS.

## A.3.2 The Tool Bar

The tool bar contains the following buttons ordered from left to right:

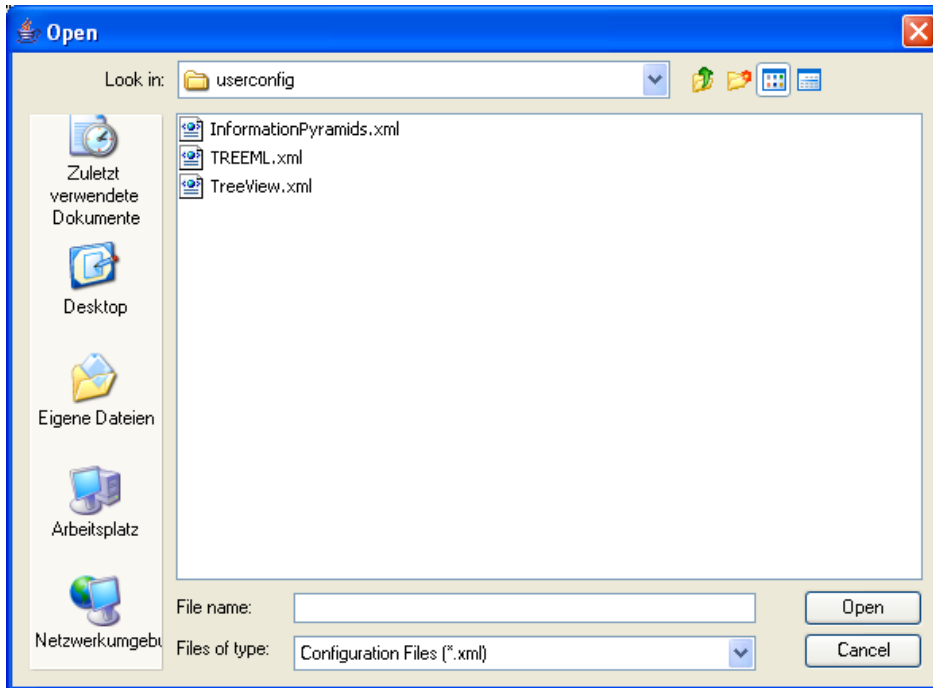
**Open Configuration:** Opens a dialog box where a previously saved configuration file can be selected. The selected configuration file is parsed and creates the visualization environment with the stored hierarchy.

**Save Configuration:** Saves the session data to the currently selected configuration file. If no configuration file was previously selected by loading or saving, a dialog appears to choose one.

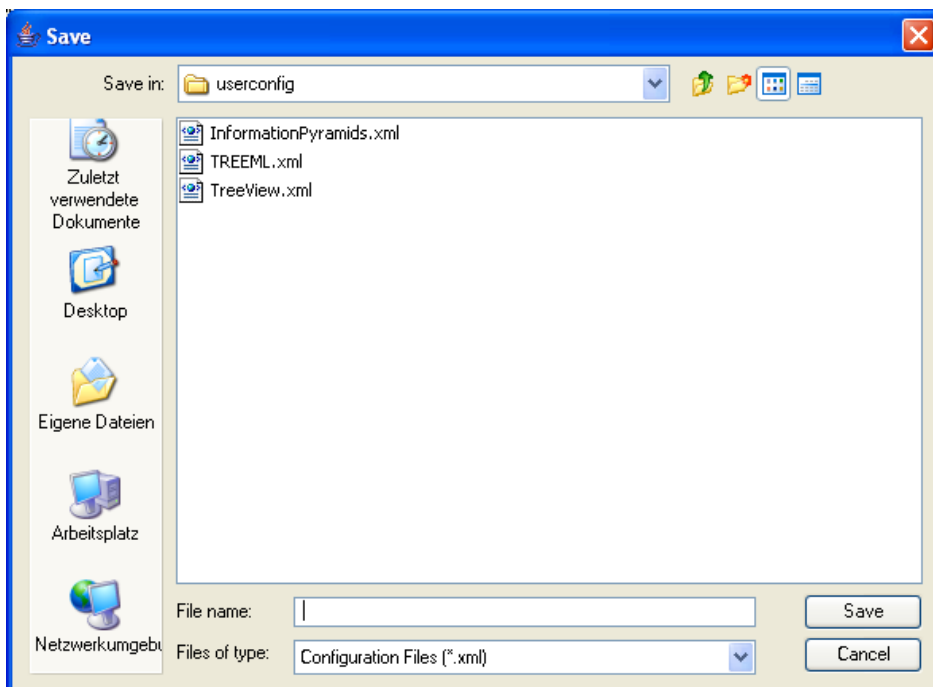
**Font:** Opens a dialog box (see Figure A.10) where the user can set the font options of visualizations.

**Color:** Opens a dialog box (see Figure A.11) where the user can set the color options of visualizations. The contents of the dialog depend on the selected *DataSource*.

**Make Root:** Makes the currently selected inner node the new root of the hierarchy. If multiple inner nodes or leaf nodes are selected, an error dialog appears.

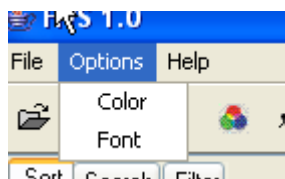


**Figure A.5:** The open configuration file dialog.

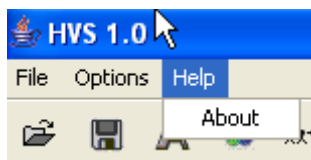


**Figure A.6:** The save configuration dialog.





**Figure A.7:** The options menu.



**Figure A.8:** The help menu.

**Make Parent As Root:** Makes the parent node of the current root the new root of the hierarchy.

### A.3.3 The Control Panel

#### Sort Panel

The sort panel (see Figure A.12) allows sort criteria for nodes to be specified. The available options are:

- by name. Sort the nodes by their name.
- by size. Sort the nodes by their size.
- by type. Sort the leaf nodes by their type.

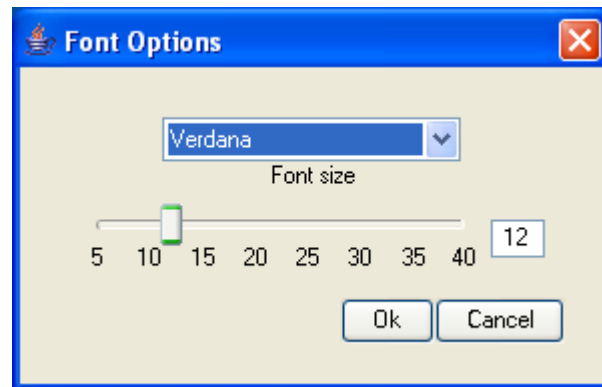
The sorting order can be either ascending or descending.

#### Filter Panel

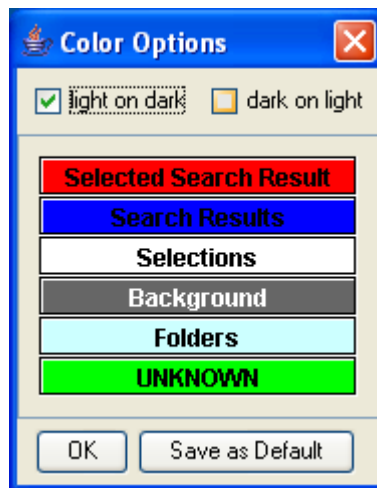
The filter panel (see Figure A.13) allows the application of filters to the hierarchy. The name input field is used to apply a filter to the name field of leaf nodes. With the help of the \* wildcard many useful filters can be specified:



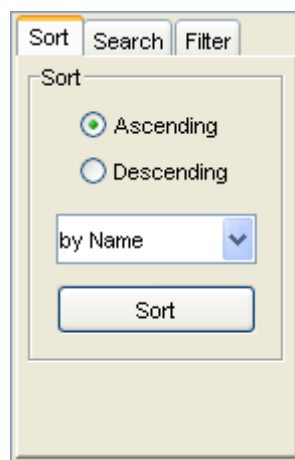
**Figure A.9:** The tool bar.



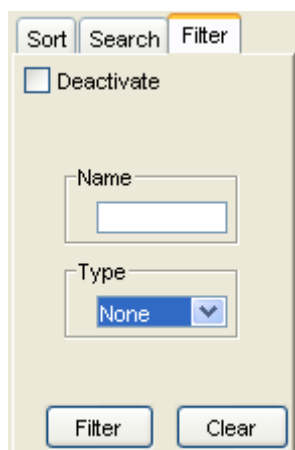
**Figure A.10:** The font selection dialog.



**Figure A.11:** The color selection dialog.



**Figure A.12:** The sort panel.



**Figure A.13:** The filter panel.

- string matches when the name of a node is exactly “string”
- \*string matches when the name of a node ends with “string”
- string\* matches when the name of a node starts with “string”
- \*string\* matches when the name of a node contains “string”

The combo box allows to filter leaf nodes by their type. The available type information depends on the kind of hierarchy being visualized.

### Search Panel

The search panel (see Figure A.14) allows searching for specific items in the hierarchy. The name input field is used to apply a string to compare with the name field of nodes. With the help of the \* wildcard useful search string can be generated:

- string matches when the name of a node is exactly “string”
- \*string matches when the name of a node ends with “string”
- string\* matches when the name of a node starts with “string”
- \*string\* matches when the name of a node contains “string”

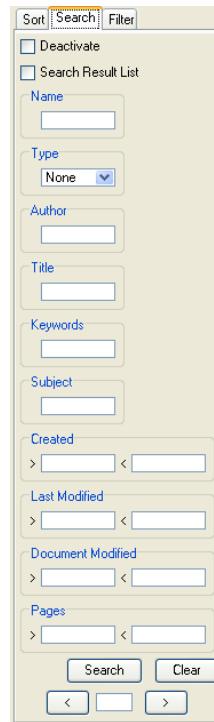
The combo box allows searching for leaf nodes by their type. The available type information depends on the kind of hierarchy being visualized.

### Search Result Panel

The search result panel (see Figure A.15) visualizes the number of elements in the result list. The currently visualized item is also displayed. With the help of the two buttons, stepping through the result list is possible.

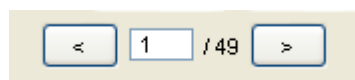
### Search Result Table View

The search result list can additionally be visualized in a table view (see Figure A.16). By default, the table view visualizes all attributes of nodes. The set of attributes visualized is user-configurable.



The search panel is a vertical form with a light beige background. At the top, there are three tabs: 'Sort', 'Search', and 'Filter', with 'Search' being the active tab. Below the tabs are two checkboxes: 'Deactivate' and 'Search Result List'. The form contains several input fields: 'Name', 'Type' (a dropdown menu currently set to 'None'), 'Author', 'Title', 'Keywords', and 'Subject'. Below these are four date range selectors, each with a greater-than sign on the left and a less-than sign on the right, labeled 'Created', 'Last Modified', 'Document Modified', and 'Pages'. At the bottom of the form are two buttons: 'Search' and 'Clear'. Below the buttons are three small navigation buttons: a left arrow, a text input field, and a right arrow.

**Figure A.14:** The search panel.



The search result panel shows a horizontal navigation bar with a light beige background. It contains three buttons: a left arrow, a text input field containing the number '1', and a right arrow. To the right of the text input field is the text '/ 49'.

**Figure A.15:** The search result panel.

Name	Size	Author	Title	Keywords	Subject	Created	Last Modified	Document Modified	Pages
17310127.pdf	250096	Stefan Sossna (PT...	17310127		TeX output 1999.11...	10.11.99 14:26	22.05.02 16:37	19.11.99 11:53	10
17310392.pdf	147794	Stefan Sossna (PT...	17310392		TeX output 1999.11...	16.11.99 10:06	22.05.02 16:37	19.11.99 11:53	8
2645.pdf	89930					25.05.98 10:32	22.05.02 16:37		10
2657.pdf	114053	Root	91-06.pm			25.05.98 10:33	22.05.02 16:37		17
bederson-to...	3149970					29.09.02 13:45	08.01.04 15:56		22
botatree.pdf	854934		article.dvi			28.06.01 18:07	24.07.02 15:02		8
bubbletree-c...	17473		Microsoft W/o...			10.12.99 17:34	07.11.02 14:21		2
buchheim-g...	249972		walker.dvi			05.07.02 13:56	04.09.02 17:12	05.07.02 13:56	14
cadre_map3...	106668						22.05.02 16:37		1
card-chi199...	1189256	John Nelson	jda2172.tmp			14.07.97 10:44	22.05.02 16:36	14.07.97 10:44	8
cgaa-g4040...	253512					17.06.98 12:41	22.05.02 16:37	17.06.98 13:26	3
chi2000-we...	137794		Untitled Doc...			10.12.99 15:53	16.09.02 13:38		2
ctm.pdf	379407		ctm.dvi			03.09.99 09:38	22.05.02 16:38		6
g4018.pdf	833125					17.06.98 13:12	22.05.02 16:37	17.06.98 13:29	6
HTDS2_0.pdf	189198	Diandra Macias	HTDS for pdf...			10.05.99 09:25	22.05.02 16:36		4
inso2.pdf	26359					05.10.95 16:53	22.05.02 16:37		1
iris-1998-08...	178411					26.06.98 08:12	16.09.02 13:38		8
iui99-starzoo...	433415					18.01.99 21:09	16.09.02 13:38	21.01.99 10:41	1
iv02-165607...	378467	Keith Andrews	Visual Explor...			02.05.02 20:30	22.01.03 14:11	26.07.02 16:02	6
ivis2001-134...	607289	Administrator	Microsoft W/o...			01.09.01 23:10	22.01.03 14:33	18.10.01 10:51	6
johnson-vis...	983765	IEEE	Tree-maps: a...			21.02.98 06:24	08.01.04 16:02	31.10.03 09:21	8
julymap.pdf	593414	dd				12.07.96 18:54	22.05.02 16:37	17.08.96 11:24	1
junemap.pdf	59547	DD				01.07.96 15:11	22.05.02 16:37	09.08.96 18:48	1
magtree-a4...	605983						21.08.02 10:40		33
mapa0v12.pdf	1052934						22.05.02 16:37		
MDA-Browsi...	50688						22.05.02 16:35		14
mev-da.pdf	5362684		Diplom 71.sdw			05.07.02 12:30	05.07.02 13:31	05.07.02 12:30	72
moen.pdf	839213	DLibSpt3				23.08.01 12:28	04.09.02 17:10	23.08.01 12:28	8
munzner-ph...	8357546					15.06.00 12:47	22.05.02 16:41	15.06.00 12:47	167
npiv99.pdf	3891182		Microsoft W/o...			05.07.02 13:27	05.07.02 14:27	05.07.02 13:27	5
p13-robertso...	39908					17.08.00 10:05	22.05.02 16:35	22.08.00 10:35	1
p232-bjork.pdf	1246529					17.08.00 12:40	22.05.02 16:35	22.08.00 10:36	6
p246-hearst...	3946289	CDR-13	ihg321.PDF				22.05.02 16:36	13.02.02 10:22	10
p29-harel.pdf	959996					17.08.00 10:10	22.01.03 14:16	22.08.00 10:36	12
p41-graham...	1576504					17.08.00 10:14	22.05.02 16:35	22.08.00 10:36	10
p461-roberts...	203927	John Nelson	jda2172.tmp			14.07.97 11:46	22.05.02 16:36	14.07.97 11:46	2
p514-kool.pdf	1500000					20.07.98 15:24	22.05.02 16:36	26.07.98 14:37	17

Figure A.16: The table view of the search result list.

### A.3.4 The Visualization Frame

The visualization frame displays a visualization. It provides a standard menu bar for all visualizations.

#### The Menu Bar

**View Menu** This menu is standard for all visualizations.

**Independent:** Set the synchronization mode of this visualization into independent or synchronized. If independent is set, navigational actions in other visualizations do not change this one.

**Detailed View:** If this option is selected, this visualization acts as a detailed view. This means that the root of this view is the common parent of all currently selected nodes.

**Hide Documents:** If this option is set, no leaf node objects are visible in the visualizations.

**Show Properties:** If this option is set, the properties panel is shown (see Figure A.17) at the bottom of the window.

**Close:** Closes this view.

**Options Menu** The appearance of this menu is determined by the respective visualization.

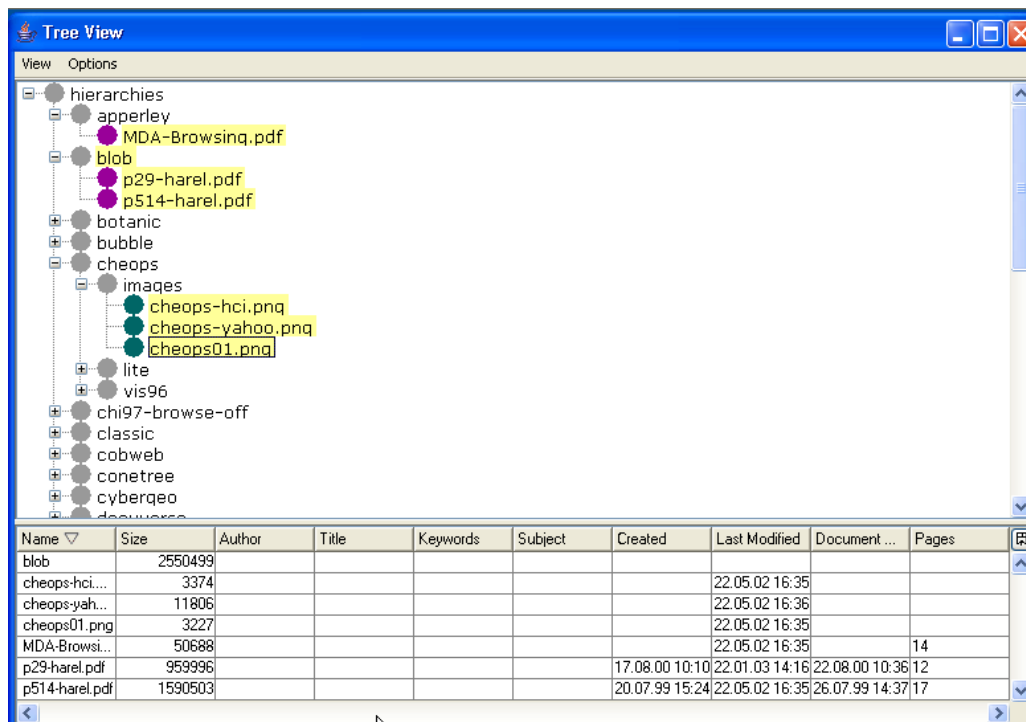


Figure A.17: The properties panel used to display details of selected nodes.

### A.3.5 Properties Panel

The properties panel provides information about all selected nodes. It is structured as a table view. By default, it displays all available metadata information of the selected nodes. Which attributes to show is configurable by clicking on the header field above the vertical scrollbar (see Figure A.18). The table can be sorted on any displayed attribute by clicking on the desired header field. By default, the nodes are sorted by their name in ascending order. Clicking on a header field changes the sorting order.

## A.4 HVS Visualizations

### A.4.1 Tree View Visualization

The Tree View visualizes the hierarchy using an outline method. The Options menu allows the activation of tool tips.

#### Mouse Functions

**Left mouse button - single click:** Selects that object which the mouse cursor points at. A colored wire frame is painted around this object. If the Control key is pressed during the click, the object is added to the current selection, if it was previously selected, it is removed.

**Left mouse button - double click:** Expands or collapses the node, when the mouse points to an inner node representation.

Name ▾	Size	Author	Title	Keywords	Subject	Created	Last Modified	Document ...	Pages	Author
3dexpl1	298124									✓ Title
3dexpl2	263143									✓ Keywords
cgaa-g4040...	253512					17.06.98 12:41	22.05.02 16:37	17.06.98 13:26	3	✓ Subject
cube.gif	10778						22.05.02 16:37			✓ Created
cube.html	1472						22.05.02 16:37			✓ Last Modified
infosky	258154									✓ Document Modified
infotunnel	93921									✓ Pages
ipyr	2389397									
iv02-165607...	378467	Keith Andrews	Visual Explor...			02.05.02 20:30	22.01.03 14:11	26.07.02 16:02	6	
jexpl	198966									
russo-16560...	661394					18.07.02 10:20	22.01.03 14:26	26.07.02 16:01	6	
...	...					...	...	...	...	

**Figure A.18:** Choosing the attributes to display in the properties panel.



**Figure A.19:** The context menu which pops up on a right mouse click.

**Left mouse button - dragging:** Paints a rectangle over the dragged area. If the mouse button is released, all objects under this rectangle are selected.

**Right mouse button - single click:** Opens a context menu (see Figure A.19) for manipulation of the hierarchy. The selectable options are:

**Insert:** Nodes can be inserted into the hierarchy. The available submenu items are :

**Collection:** Opens a dialog box to create and insert an inner node.

**Document:** Opens a dialog box to create and insert a leaf node.

**Remove:** Removes that node which the mouse points to from the hierarchy. If the node is an inner node, all of its children are also removed.

**Rename:** Opens a dialog box (see Figure A.23) to rename the node the mouse point to.

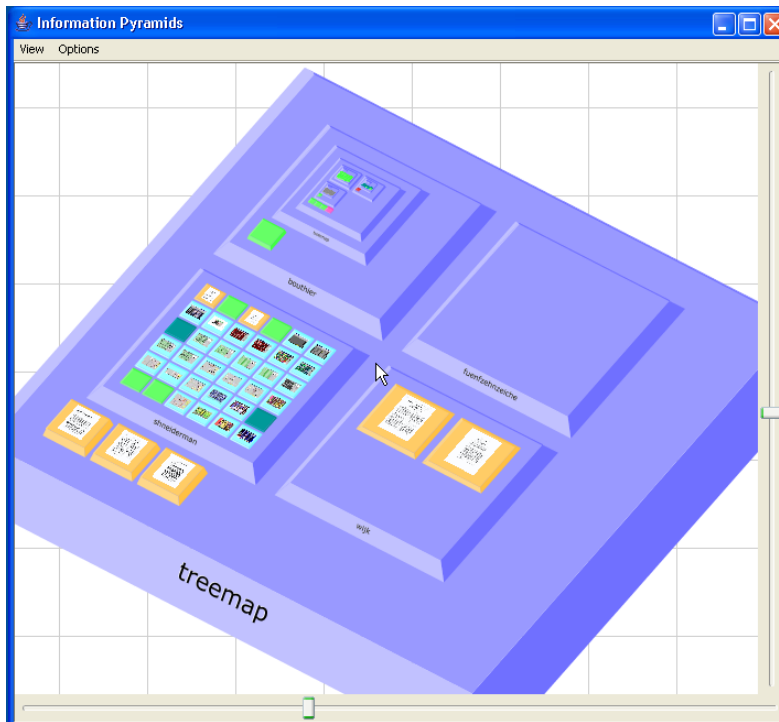
**Open:** If the mouse points to a leaf node object, the application tries to open this document in the corresponding application

**Mouse wheel - single click:** When the mouse cursor is pointing at a leaf node object, HVS tries to open this item with the associated application.

**Mouse wheel - rotate:** Moves the scroll bar up or down depending on the direction of movement.

## A.4.2 Information Pyramids Visualization

The Information pyramids technique visualizes the hierarchy as three-dimensional landscape (see Figure A.20). The pyramid can be rotated around the x-axis using the slider on the right. The slider at the bottom rotates the pyramid around its z-axis.



**Figure A.20:** The Information Pyramids visualization.

### Options Menu

**Show Tooltips:** If this option is set, tool tips are visualized when the user hovers over an item.

**Show Labels:** If this option is set, the names nodes are displayed.

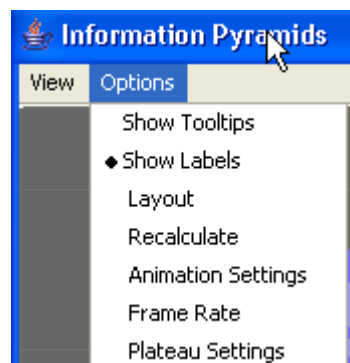
**Layout:** Opens a window where a layout manager can be selected.

**Recalculate:** Lay out all plateaus again by refreshing the size parameters.

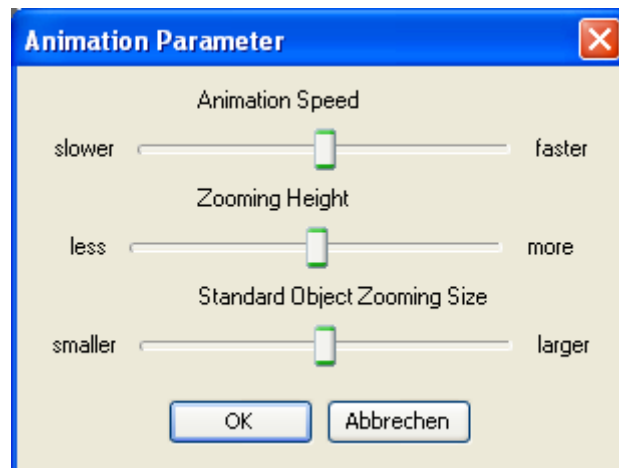
**Animation Settings:** Opens a window where the parameters for animation can be adjusted.

**Frame Rate:** Opens a window where the frame rate can be set.

**Plateau Settings:** Opens a window where the dimensions of plateaus can be modified.







**Figure A.21:** The animation settings dialog.

**Animation Settings Dialog** The animation settings dialog lets users adjust the parameters used in animation sequences. The adjustable parameters are:

**Animation Speed:** The speed of the camera movement.

**Zooming Height:** How high to zoom out during navigation.

**Standard Object Zooming Size:** The width of the focused plateau measured in pixels determining the zoom factor.

**Plateau Settings Dialog** The options of this dialog let users to adjust the visual impression of the pyramids. The adjustable parameters are:

**Plateau Minimum Size:** The minimum width of displayed items. If the minimum size is reduced by moving the slider to the left, more items are displayed.

**Plateau Height Proportion:** The height of plateaus in percentage of its size.

**Plateau Border:** The border of each plateau. Inside this border, the wire frame box for highlighting plateaus is drawn.

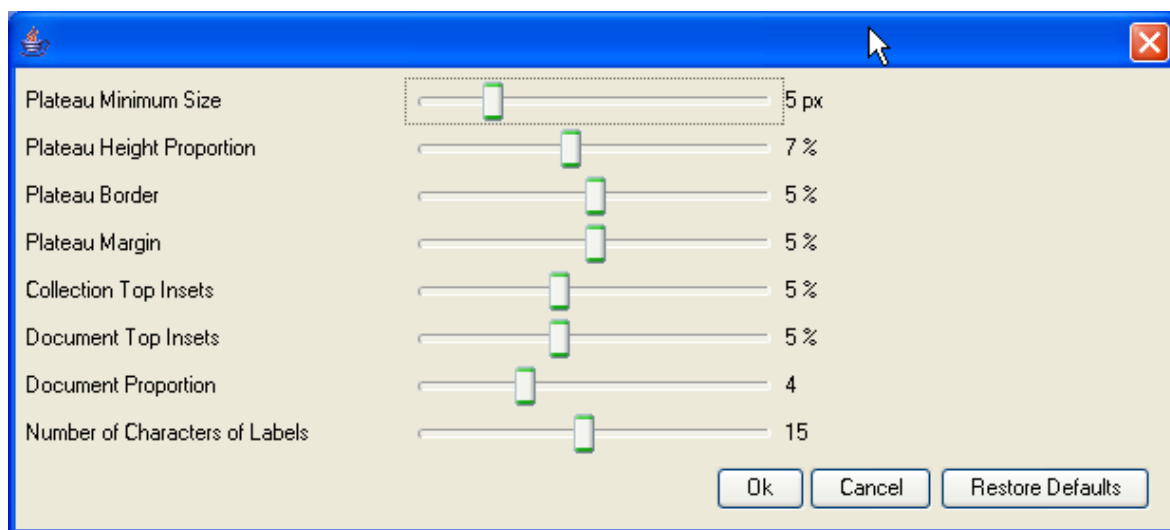
**Plateau Margin:** The margin which is set when placing children.

**Document Top Insets:** The top insets for plateaus representing leaf nodes relative to the size of the plateau.

**Collections Top Insets:** The top insets for plateaus representing inner nodes relative to the size of the plateau.

**Document Proportion:** The ratio between plateaus representing inner nodes and document nodes.

**Number of Characters of Labels:** The number of characters displayed in labels.



**Figure A.22:** The plateau settings dialog.

## Mouse Functions

**Left mouse button - single click:** Selects the object which the mouse cursor points to. A colored wire-frame is painted around this object. If the mouse does not point to any object, the current selection is cleared.

**Left mouse button - double click:** Maximizes the object which the mouse is pointing at.

**Left mouse button - dragging:** Paints a rectangle over the dragged area. If the mouse button is released, all objects under this rectangle are selected.

**Right mouse button - single click:** Opens a menu (see Figure A.19) for manipulation of the hierarchy. The selectable options are:

**Insert:** Nodes can be inserted into the hierarchy. The available submenu items are :

**Collection:** Opens a dialog box to create and insert an inner node.

**Document:** Opens a dialog box to create and insert a leaf node.

**Remove:** Removes the node which the mouse points to from the hierarchy. If the node is an inner node, all of its children are also removed.

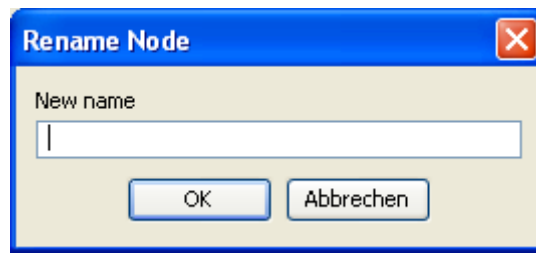
**Rename:** Opens a dialog box (see Figure A.23) to rename the node being pointed to.

**Open:** If the mouse points to a leaf node object, the application attempts to open this document in the corresponding application.

**Right mouse button - dragging:** Changes into panning mode. The landscape moves during dragging in the same manner as the mouse cursor.

**Mouse wheel - single click:** When the mouse cursor is pointing at a leaf node object, HVS tries to open this item with the associated application.

**Mouse wheel - rotate:** Depending on the direction, the user can zoom in or out. The location of the mouse cursor determines the center of the zooming area.



**Figure A.23:** The dialog box to rename a node.

**Mouse wheel - dragging:** Paints a rectangle over the dragged area. If the mouse button is released, all objects under this rectangle are maximized.



## Appendix B

# Developer Guide

The Hierarchical Visualization System is designed for developers to extend. This chapter describes how to develop a new visualization for HVS.

### B.1 Development of a Tree View Visualization

Since HVS is designed to integrate standard visualization tools, the development of a tree view is quite simple. The first step in developing a visualization is creating a new directory in the plugin directory of HVS. In this newly created directory a plug-in configuration file have to be created. For the Tree View this file looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin>
  <runtime>
    <path name="classes"/>
    <extension point="iicm.hvs.visualization.Visualization"
      name="Tree_View"
      class="iicm.hvs.visualization.jtree.JTreeVisualization"
    />
  </runtime>
</plugin>
```

The next step in creation of the Tree View visualization is to create the main class. The *Swing* component *javax.swing.JTree* is used to visualize the tree structure. The *JTree* component takes a data model, which implements the *javax.swing.tree.TreeModel* interface. Thus an adapter class has to be written, to transform the interface of the HVS data model to that used in the tree.

The main class of the *TreeView* has to subclass the class *iicm.hvs.visualization.Visualization*. The class *TreeView* implements the *ExpansionListener* interface, since expanding and collapsing of nodes should be synchronized. The source code fragment thereby looks like:

```
public class TreeView extends Visualization implements
    ExpansionListener
{
  private JTree tree_;
  public TreeView(DataModel dataModel, Controller controller,
    SearchResult searchResult, PopUpHandler popUpHandler,
    JFrame frame)
  {
    super(dataModel, controller, searchResult, popUpHandler, frame);
    ....
  }
}
```

```

        tree_ = new HvsJTree(treeModel)
        add(new JScrollPane(tree_));
        ....
    }
}

```

To display the nodes in the colors defined in the HVS environment, a modified cell renderer has to be implemented. This cell renderer must be a subclass of the *javax.swing.DefaultTreeCellRenderer* class. The only method to overwrite is *getTreeCellRendererComponent()*. This method returns the displayed component. The implementation looks like the following:

```

public Component getTreeCellRendererComponent(JTree tree ,
        Object value , boolean selected , boolean expanded ,
        boolean leaf , int row , boolean hasFocus)
{
    super.getTreeCellRendererComponent(tree , value , selected ,
        expanded , leaf , row , hasFocus);
    setText(((Node) value).getName());
    if (value instanceof Document)
    {
        setForeground(VisualizationProperties.getColorForDocument(
            (Document) value));
    }
    else
        setForeground(VisualizationProperties.getCollectionColor());
    if (searchresult_.isSelectedSearchResult((Node) value))
    {
        setBorder(BorderFactory.createLineBorder(
            VisualizationProperties.getSearchResultSelectionColor(), 2));
    }
    else
    {
        if (searchresult_.isSearchResult((Node) value))
        {
            setBorder(BorderFactory.createLineBorder(
                VisualizationProperties.getSearchResultColor(), 2));
        }
        else
            setBorder(null);
    }
    return this;
}

```

For synchronization with other views, the *TreeView* has to send events on user interactions to HVS. For full synchronization, events have to be sent when:

- The selection in the tree changes.
- A node is expanded or collapsed.
- The user scrolls in the tree.

To send those events the class *visualization* provides the following methods:

```

protected final void fireFocusChanged(FocusEvent event)
protected final void fireSelectionChanged(SelectionEvent event)
protected final void fireCollectionExpanded(ExpansionEvent event)
protected final void fireCollectionCollapsed(ExpansionEvent event)

```

The *JTree* component supports the attachment of listeners to receive notifications on user interactions. All listeners have to implement the interface *javax.swing.event.TreeExpansionListener* to receive a notification when a node is collapsed or expanded. For listening on events which are created when the user selects a node, the interface *javax.swing.event.TreeSelectionListener* has to be implemented. The method *valueChanged()* has to be implemented and looks like:

```
Vector selectednodes = new Vector();
TreePath[] p = tree.getSelectionModel().getSelectionPaths();
if (p != null)
{
    for (int i = 0; i < p.length; i++)
        selectednodes.add(p[i].getLastPathComponent());
}
fireSelectionChanged(new SelectionEvent(JTreeVisualization.this,
                                        selectednodes));
```

The TreeView visualization is compiled using the following command:

```
javac -cp hvs.jar *.java
```

When the compiled class files are not part of the Java *CLASSPATH* environment, they have to be placed into the correct plug-in directory. The application is then started as usual using the command:

```
java -Xms32M -Xmx256M -jar hvs.jar
```





# Bibliography

- Ahlberg, C. (1996). *Spotfire: An Information Exploration Environment*. SIGMOD Rec., 25(4):25–29. 32
- Ahlberg, C. and Shneiderman, B. (1994). *Visual Information Seeking using the FilmFinder*. In Conference Companion on Human Factors in Computing Systems (CHI'94), pages 433–434. ACM Press. 5
- Ahlberg, C., Williamson, C. and Shneiderman, B. (1992). *Dynamic Queries for Information Exploration: an Implementation and Evaluation*. In Proceedings of the Conference on Human Factors in Computing Systems (CHI'92), pages 619–626. ACM Press. 5
- Andrews, K. (2002a). *Visual Exploration of Large Hierarchies with Information Pyramids*. In Proceedings of the Sixth International Conference on Information Visualization (IV'02), pages 793–798. IEEE Computer Society Press. 59
- Andrews, K. (2002b). *Visualising Information Structures: Aspects of Information Visualisation*, Professorial Thesis, Graz University of Technology. <ftp://ftp.iicm.edu/pub/keith/hbil/visinfo.pdf>.
- Andrews, K. and Heidegger, H. (1998). *Information Slices: Visualizing and Exploring Large Hierarchies using Cascading, Semi-Circular Discs*. In Proceedings of the 1998 IEEE Symposium on Information Visualization (InfoVis'98), Late Breaking Hot Topics Paper, pages 9–11. IEEE Computer Society. <ftp://iicm.edu/pub/papers/ivis98.pdf>. 17
- Baumgartner, J., Börner, K., Deckard, N. J. and Sheth, N. (2003). *An XML Toolkit for an Information Visualization Software Repository*. In Proceedings of the 2003 IEEE Symposium on Information Visualization (InfoVis'03), Poster Compendium, pages 72–73. IEEE Computer Society. 36
- Beaudoin, L., Parent, M.-A. and Vroomen, L. C. (1996). *Cheops: A Compact Explorer for Complex Hierarchies*. In Proceedings of the 7th Conference on Visualization (Vis'96), pages 87–92. IEEE Computer Society Press. <http://www.istop.com/~maparent/paper.html>. 17
- Bosch, R., Stolte, C., Tang, D., Gerth, J., Rosenblum, M. and Hanrahan, P. (2000). *Rivet: A Flexible Environment for Computer Systems Visualization*. Computer Graphics, 34(1):68–73. 30
- Bruls, M., Huizing, K. and vanWijk, J. J. (2000). *Squarified Treemaps*. In de Leeu, W. and van Liere, R., editors, Proceedings of the Joint Eurographics and IEEE TVCG Symposium on Visualization (VisSym'00), pages 33–42. IEEE Computer Society. 16
- Burger, T. (1999). *Magic Eye View: Eine neue Fokus + Kontext Technik zur Darstellung von Graphen*. In german, University of Rostock, Institute for Computer Graphics. [http://www.icg.informatik.uni-rostock.de/Diplomarbeiten/1999/Thomas\\_Buerger/](http://www.icg.informatik.uni-rostock.de/Diplomarbeiten/1999/Thomas_Buerger/). 19

- Card, S. K., MacKinlay, J. D. and Shneiderman, B. (1999). *Readings in Information Visualization : Using Vision to Think*. Morgan Kaufmann. ISBN 1558605339.
- Davidson, G. S., Hendrickson, B., Johnson, D. K., Meyers, C. E. and Wylie, B. N. (1998). *Knowledge Mining With VxInsight: Discovery Through Interaction*. Journal of Intelligent Information Systems, 11(3):259–285. 10
- Eades, P. and Sugiyama, K. (1990). *How to Draw a Directed Graph*. Journal of Information Processing, 13(4):424–437. 7
- Fairchild, K. M., Poltrock, S. E. and Furnas, G. W. (1988). *SemNet: ThreeDimensional Representation of Large Knowledge Bases*. In Guidon, R., editor, Cognitive Science and Its Applications for Human-Computer Interaction, pages 201–233. Lawrence Erlbaum Associates. 7
- Fekete, J.-D. (2003). *The InfoVis Toolkit*. Technical report, INRIA, University Paris. <http://www.lri.fr/~fekete/InfoVisToolkit/>. 31
- Foley, J. D., van Dam, A., Feiner, S. K. and Hughes, J. F. (1996). *Computer Graphics — Principles and Practice*. The Systems Programming Series. Addison-Wesley, second edition in c edition. ISBN 0201848406.
- Furnas, G. W. (1986). *Generalized Fisheye Views*. In Proceedings of the Conference on Human Factors in Computing Systems (CHI'86), pages 16–23. ACM Press. 3
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc. ISBN 0-201-63361-2. 37, 40, 51
- Goel, A. (1999). Parallel Coordinates Applet. <http://www.amitgoel.com/pcoord/index.html>. 6
- HCIL (2004). HCIL. <http://www.cs.umd.edu/hcil/pubs/screenshots/>. iii
- Hearst, M. A. (1995). *TileBars: Visualization of Term Distribution Information in Full Text Information Access*. In Proceedings of the Conference on Human Factors in Computing Systems (CHI'95), pages 59–66. ACM Press/Addison-Wesley Publishing Co. 10
- Heer, J., Card, S. K. and Landy, J. A. (2004). *Prefuse: A Toolkit for Interactive Information Visualization*. <http://jheer.org/publications/2005-prefuseCHI.pdf>. 34
- Hendley, R. J., Drew, N. S., Wood, A. M. and Beale, R. (1995). *Case Study: Narcissus: Visualising Information*. In Proceedings of the 1995 IEEE Symposium on Information Visualization (InfoVis'95), page 90. IEEE Computer Society. 7
- Inselberg, A. and Dimsdale, B. (1990). *Parallel coordinates: A Tool for Visualizing Multi-dimensional Geometry*. In Proceedings of the 1st Conference on Visualization (VIS'90), pages 361–378. IEEE Computer Society Press. 5, 29
- IVC (2004). IVC. <http://iv.slis.indiana.edu/>. 36
- Java2D (2004). Java2D. <http://java.sun.com/docs/books/tutorial/2d/>.
- Johnson, B. and Shneiderman, B. (1991). *Tree-Maps: A Space-filling Approach to the Visualization of Hierarchical Information Structures*. In Proceedings of the 2nd conference on Visualization (Vis'91), pages 284–291. IEEE Computer Society Press. <http://www.cs.umd.edu/hcil/treemaps/>. 16

- Kaufman, L. and Rousseeuw, P. J. (1990). *Finding Groups in Data - An Introduction to Cluster Analysis*. Wiley-Interscience. ISBN 0471878766. 29
- Knudsen, J. (1999). *Java 2D Graphics*. O'Reilly & Associates Publishing Co., Inc. ISBN 1-56592-484-3.
- Kohonen, T. (2000). *Self-Organizing Maps*. Springer. ISBN 3540679219. 10
- Kohonen, T., Kaski, S., Lagus, K., Salojärvi, J., Paatero, V. and Saarela, A. (2000). *Organization of a Massive Document Collection*. IEEE Transactions on Neural Networks, Special Issue on Neural Networks for Data Mining and Knowledge Discovery, 11(3):574–585. 10
- Kreuseler, M., Lopez, N. and Schumann, H. (2000). *A Scalable Framework for Information Visualization*. In Proceedings of the 2000 IEEE Symposium on Information Visualization (InfoVis'00), pages 27–36. IEEE Computer Society. <http://www.informatik.uni-rostock.de/~mkreusel/SInVis/infovis.html>. 28
- Kreuseler, M. and Schumann, H. (2002). *A Flexible Approach for Visual Data Mining*. IEEE Transactions on Visualization and Computer Graphics, 8(1):39–51. 28
- Mackinlay, J. D., Robertson, G. G. and Card, S. K. (1991). *The Perspective Wall: Detail and Context Smoothly Integrated*. In Proceedings of the Conference on Human Factors in Computing Systems (CHI'91), pages 173–176. ACM Press. 4
- Munzner, T. (1997). *H3: Laying Out Large Directed Graphs in 3D Hyperbolic Space*. In Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97), page 2. IEEE Computer Society. 21
- North, C. and Shneiderman, B. (2000). *Snap-together Visualization: A User Interface for Coordinating Visualizations via Relational Schemata*. In Proceedings of the Working Conference on Advanced Visual Interfaces (AVI 2000), pages 128–135. ACM Press. 25
- North, C., Shneiderman, B. and Plaisant, C. (1996). *User Controlled Overviews of an Image Library: A Case Study of the Visible Human*. In Proceedings of the first ACM International Conference on Digital Libraries (DL'96), pages 74–82. ACM Press. 11
- Nowell, L. T., France, R. K., Hix, D., Heath, L. S. and Fox, E. A. (1996). *Visualizing Search Results: Some Alternatives to Query-document Similarity*. In Proceedings of the 19th annual international ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'96), pages 67–75. ACM Press. 6
- Panyr, J., Fürer, T. and Preiser, U. (1996). *Kontextuelle Visualisierung von Informationen*. In Proceedings des 19. Oberhofer Kolloquiums über Information und Dokumentation, pages 217–228. In German. 29
- Polaris (2004). Polaris. <http://graphics.stanford.edu/projects/polaris/>. iii, 31
- Prefuse (2004). Prefuse. <http://prefuse.sourceforge.net/>. 34
- Rao, R. and Card, S. K. (1994). *The Table Lens: Merging Graphical and Symbolic Representations in an Interactive Focus + Context Visualization for Tabular Information*. In Proceedings of the Conference on Human Factors in Computing Systems (CHI'94), pages 318–322. ACM Press. 5, 28

- Reingold, E. M. and Tilford, J. S. (1981). *Tidier Drawing of Trees*. IEEE Transactions on Software Engineering, 7(2):223–228. 13, 19
- Rennison, E. (1994). *Galaxy of News: An Approach to Visualizing and Understanding Expansive News Landscapes*. In Proceedings of the 7th annual ACM symposium on User Interface Software and Technology (UIST'94), pages 3–12. ACM Press. 9
- Robertson, G. G. and Mackinlay, J. D. (1993). *The Document Lens*. In Proceedings of the 6th annual ACM symposium on User Interface Software and Technology (UIST'93), pages 101–108. ACM Press. 4
- Robertson, G. G., Mackinlay, J. D. and Card, S. K. (1991a). *Cone Trees: Animated 3D Visualizations of Hierarchical Information*. In Proceedings of the Conference on Human Factors in Computing Systems (CHI'91), pages 189–194. ACM Press. 20
- Robertson, G. G., Mackinlay, J. D. and Card, S. K. (1991b). *Information Visualization using 3D Interactive Animation*. In Proceedings of the Conference on Human Factors in Computing Systems (CHI'91), pages 461–462. ACM Press. 61
- Roth, S. F., Lucas, P., Senn, J. A., Gomberg, C. C., Burks, M. B., Stroffolino, P. J., Kolojechick, A. J. and Dunmire, C. (1996). *Visage: A User Interface Environment for Exploring Information*. In Proceedings of the 1996 IEEE Symposium on Information Visualization (InfoVis'96), pages 3–12. IEEE Computer Society. 26
- Roth, S. F., Kolojechick, J., Mattis, J. and Goldstein, J. (1994). *Interactive Graphic Design using Automatic Presentation Knowledge*. In Proceedings of the Conference on Human Factors in Computing Systems (CHI'94), pages 112–117. ACM Press. 28
- Salton, G., Wong, A. and Yang, C. S. (1975). *A Vector Space Model for Automatic Indexing*. Communications of the ACM, 18(11):613–620. 9
- Schipflinger, J. (1998). *The Design and Implementation of Harmony Session Manager*. Master's thesis, Graz University of Technology, Institute for Processing Information and Computer Supported New Media. <ftp://ftp.iicm.edu/pub/thesis/jschipf.pdf>. 7
- ShapeVis (2004). ShapeVis. [www.informatik.uni-rostock.de/~mkreuse/SInVis/infovis.html](http://www.informatik.uni-rostock.de/~mkreuse/SInVis/infovis.html). iii, 30
- Shneiderman, B. (1996). *The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations*. In Proceedings of the 1996 IEEE Symposium on Visual Languages (VL'96), pages 336–343. IEEE Computer Society. <ftp://ftp.cs.umd.edu/pub/papers/papers/3665/3665.ps.Z>. 3
- Shneiderman, B. and Wattenberg, M. (2001). *Ordered Treemap Layouts*. In Proceedings of the 2001 IEEE Symposium on Information Visualization (InfoVis'01), pages 73–78. IEEE Computer Society. <http://www.computer.org/proceedings/infovis/1342/13420073abs.htm>. 16
- Spence, R. (2001). *Information Visualization*. Addison-Wesley. ISBN 0-201-59626-1.
- Spotfire (2000). Spotfire. <http://www.spotfire.com/>. 32
- Stolte, C., Tang, D. and Hanrahan, P. (2002). *Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases*. IEEE Transactions on Visualization and Computer Graphics, 8(1):52–65. 29

- Swing (2004). Swing. <http://java.sun.com/docs/books/tutorial/uiswing/>.
- Tesler, J. D. and Strasnick, S. L. (1992). *FSN: The 3D File System Navigator*, Silicon Graphics, Inc. <ftp://ftp.sgi.com/sgi/fsn>. 22
- van Wijk, J. J. and Nuij, W. A. A. (2003). *Smooth and Efficient Zooming and Panning*. In Proceedings of the 2003 IEEE Symposium on Information Visualization (InfoVis'03), page 90. IEEE Computer Society. 67
- Visage (2004). Visage. <http://www.maya.com/visage/base/scenario.html>. iii, 28
- Walker II, J. Q. (1980). *Positioning Nodes For General Trees*. *Software—Practice and Experience*, 10(7):553–561. 13
- WEBSOM (2000). WEBSOM. <http://websom.hut.fi/websom>. 10
- Weilander, E. (1999). *Metadata Visualization: Visual Exploration of File Systems and Search Result Sets based on Metadata Attributes*. Master's thesis, Graz University of Technology, Institute for Processing Information and Computer Supported New Media. <ftp://ftp.iicm.edu/pub/thesis/eweit.pdf>. 6
- Welz, M. (1999). *The Java Pyramids Explorer: A Portable, Graphical Hierarchy Browser*. Master's thesis, Graz University of Technology, Institute for Processing Information and Computer Supported New Media. <ftp://ftp.iicm.edu/pub/thesis/mwelz.pdf>. 59, 63, 81
- Wolte, J. (1998). *Information Pyramids: Compactly Visualising Large Hierarchies*. Master's thesis, Graz University of Technology, Institute for Processing Information and Computer Supported New Media. <ftp://ftp.iicm.edu/pub/thesis/jwolte.pdf>. 59, 61, 62