The Java Pyramids Explorer

A Portable, Graphical Hierarchy Browser

Michael Welz

The Java Pyramids Explorer

A Portable, Graphical Hierarchy Browser

Master's Thesis

at

Graz University of Technology

submitted by

Michael Welz

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

20th September 1999

Advisor: Univ.Ass. Dr. Keith Andrews

Der Java Pyramids Explorer

Ein portabler, grafischer Hierarchie Browser

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Michael Welz

Institut für Informationsverarbeitung und Computergestützte neue Medien (IICM),
Technische Universität Graz
A-8010 Graz

20. September 1999

Diese Arbeit ist in englischer Sprache verfaßt.

Begutachter: Univ.Ass. Dr. Keith Andrews

Abstract

The abundance of hierarchically structured information has lead to the development of numerous techniques to visualise information hierarchies. However, many of these techniques, while visually appealing, do not scale well to very large hierarchies.

This thesis presents the Java Pyramids Explorer, which combines a compact three-dimensional information pyramids visualisation with a conventional tree browser, allowing users to rapidly explore even large heirarchies. Written in pure Java, the Java Pyramids Explorer is portable across multiple platforms.

Kurzfassung

Hierarchien werden sehr oft dazu verwendet Informationen zu strukturieren und zu gliedern. Aus diesem Grund wurden zahlreiche Technologien entwickelt um diese hierarchisch strukturierten Informationen darzustellen. Obwohl viele dieser Technologien ansprechende Visualisierungen ermöglichen sind sie für sehr große Hierarchien schlecht geeignet.

In dieser Diplomarbeit wird der Java Pyramids Explorer, ein neuer Ansatz Hierarchien zu visualisieren, beschrieben. Der Java Pyramids Explorer verknüpft die kompakte dreidimensionale Visualisierung von Informations Pyramiden mit einer herkömmlichen Baumstruktur und erlaubt es selbst sehr große Hierarchien zu erkunden. Die Verwendung von reinem Java macht den Java Pyramids Explorer portabel für viele Platformen.

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 REM, the best independent group in the world. Without their famous records this master's thesis had never become reality.

The second chapter of this master's thesis has been finished on the 13th of June and is dedicated to a very important person in my life.

I also want to thank my advisor, Keith Andrews, for his support, suggestions, and his humour during the almost never ending story in correcting draft versions of this thesis.

Last but not least, I especially wish to thank two very special friends of mine, Paula and Ferdi. I hope they will never forget Hong Kong.

Michael Welz
Graz, Austria, September 1999

Credits

- Figure 2.4 was taken from the web site `http://www.cs.umd.edu/hcil/treemaps`, University of Maryland.
- Figure 2.5 was taken from [LRP95].
- Figure 2.12 and Figure 2.13 were taken from `http://www.sgi.com/fun/freeware/3d_navigator.html`.

Contents

List of Figures

Chapter 1

Introduction

This thesis is going to describe the Java Pyramids Explorer which is an application that can be used to visualise and explore hierarchies. The Java Pyramids Explorer uses a technique called Information Pyramids to visualise the hierarchically structured information in 3D. This technique is patent pending by Hyperwave Inc. [Hyp] and was introduced first at IEEE Visualisation 97 [AWP97].

The next chapter, Chapter 2, introduces techniques which have been developed to visualise hierarchies. Most of them were designed to visualise very large hierarchies compactly. Some of these techniques use 2D graphics to display the hierarchy. Tree views, Tree-Maps and the hyperbolic browser are such 2D techniques. Several 3D visualisation techniques, like Cone Trees, are introduced and explained in more detail. The File System Navigator which uses a landscape metaphor and a 2.5-dimensional technique called MAPA are also described.

Chapter 3 describes the programming language and the libraries that have been used to develop the Java Pyramids Explorer application. The first part of this chapter gives a brief introduction to the Java programming language and its strength when developing graphical user interfaces. The next sections describe the class libraries Abstract Window Toolkit and Swing. These libraries were used to develop the graphical user interface. The Java 2D API is used to draw the graphics of the Information Pyramid and it is described in this chapter, too. The chapter closes with a discussion of Java 3D and Java OpenGL.

The Information Pyramids technique is explained in Chapter 4. The purpose and the advantages of this technique are discussed and the developed visualisation method is described. It is introduced how the hierarchy together with additional attributes of the displayed information can be visualised. The possible methods to explore the displayed information and to navigate through the visualised landscape are explained. Two applications which implement the Information Pyramids technique, the 3D Explorer and the 3D Explorer for VRML, are introduced in the last part of the chapter.

Chapter 5 introduces the new concepts behind the Java Pyramids Explorer. Firstly, the disadvantages of the previous implementations of the Information Pyramids technique are listed and the improvements of the new approaches are explained. The main improvements of the Java Pyramids Explorer are the combination of different visualisation techniques, the improved display of the textual descriptions of the nodes in the hierarchy, the usage of a platform-independent graphics library, and the introduction of exchangeable layout managers. Secondly, the choice of views to visualise the hierarchically structured information are introduced. The Java Pyramids Explorer displays the hierarchy in a tree view, a children list, and a pyramid view. Each of these different views are discussed in detail. Finally, the navigational possibilities and the provided navigational aids are explained. It is described how the user can navigate through the hierarchy, how different parts can be expanded or collapsed, and how the user can focus on particular subtrees of the visualisation.

Selected details of the implementation of the Java Pyramids Explorer can be found in Chapter 6. The application was developed using object-oriented techniques and the class hierarchy and particularly those classes that are important to understand the concepts behind the implementation are described in detail. Furthermore the visualisation methods are introduced and the enhancements to the Information Pyramids visualisation are explained. As the Java Pyramids Explorer displays the pyramid in simulated 3D the methods that are used to draw the pyramid view are also introduced.

Finally, Chapter 7 discusses current trends, describes work in progress, and outlines some ideas for future extensions and research.

Appendix A is a user guide to the Java Pyramids Explorer application. It describes the hardware and software prerequisites and the steps needed to install and run the program. The different parts of the application window are described as well as the menu items in the menu bar and the tool bar. The appendix also explains the navigational possibilities and aids provided by the application.

Chapter 2

Techniques for Visualising Large Hierarchies

2.1 Motivation for Visualising Hierarchies

In the early days of software development much research was done to improve the performance and reliability of software and the software development process. Software developers had to solve many problems and open questions and the usability of their software was not so important. This led to products with inconsistent and unintuitive user interfaces and users having to learn how to work with each new product.

As more and more users began to work with computers, hardware and software developers began to think about how users work with their products. A new field, called Human Computer Interaction (HCI), emerged. HCI addresses how user interfaces should look so that they are simple and intuitive to use. During the last few years there have been many publications in this field and hardware and software developers have begun to realise the importance of HCI.

Today much research is being done concerning the design of graphical user interfaces and the visualisation of information [Shn97] [DFAB98] [CMS99]. The rapidly growing amount of information that is being processed by computers requires new techniques to efficiently visualise this information. As an example, the World Wide Web (WWW) grows every day by thousands of new web pages. Considering these web pages to be pieces of information, visualisation techniques, like a simple list of all of these pages, would fail to be understandable and easily comprehensible. In this context one can speak of information overload.

An often used technique to structure and organise information is a hierarchy. Examples of hierarchies would be the hierarchical organisation of employees in a company or the files and directories stored on a hard disk in a computer. Due to information overload, hierarchies are becoming larger and larger. Even the hard disk of an ordinary home PC contains thousands of files. Therefore it is an important question how hierarchies should be visualised. Traditional visualisation techniques, like outlines or tree diagrams, are not ideal for visualising large hierarchies. Many old and new methods are perfect for visualising small, compact hierarchies. Small hierarchies which can be drawn on one sheet of paper or one page on a computer screen, can be visualised properly. If the hierarchy is too large to fit on one page and the user has to scroll to see the whole hierarchy, maybe even over more than one page, the visualisation method fails to be simple and understandable.

Methods have been developed to visualise very large hierarchies with even hundreds or thousands of nodes. Some of these methods use two-dimensional visualisations and some of them three-dimensional. As the graphical capabilities of personal computers improve, three-dimensional visuali-

sations are gaining in importance. Users can wear shutter glasses or head mounted displays to explore virtual worlds and interact with objects in these virtual environments. Within these three-dimensional worlds a two-dimensional visualisation of a hierarchy does not seem to be appropriate. In addition to these new visualisations some methods also visualise other attributes of the information together with the hierarchy. For example the Harmony Information Landscape visualises different document types with different colours or shapes.

Attributes which might be visualised include:

- type of information
- quantity of information
- quality of information
- age of information
- owner of information

The remaining part of this chapter gives a short introduction to some of the visualisation techniques, two-dimensional and three-dimensional, which have been developed to visualise hierarchies.

2.2 Tree Views

Tree views use an outline method to visualise hierarchies. They are simple to implement and can also be used in text based systems. Usually, if the user starts to work with a tree, not all subtrees of the hierarchy are expanded. The user can navigate through the tree by expanding and collapsing subtrees. That means the user can decide how deeply or broadly the hierarchy should be displayed.

By opening subtrees the displayed part of the hierarchy grows both in the vertical and the horizontal direction. If there is not enough space to display all information, scrollbars are used to be able to scroll through the displayed parts. However, this is also an important disadvantage of this visualisation technique. The user does not get a good overview of the structure of the entire hierarchy because of the expanding, collapsing, and scrolling.

Nowadays most graphical user interfaces, like the X-Windows based window managers or Microsoft Windows (see Figure 2.1), provide tree views as standard graphical components. Many applications use these standard components and therefore it is easy for users to work with trees. Swing, a new class library for Java[1], provides a tree component that can display all kind of multimedia information like text, graphics, animations, etc (see Figure 2.2).

[1] Java is an object-oriented programming language developed by SUN Microsystems.

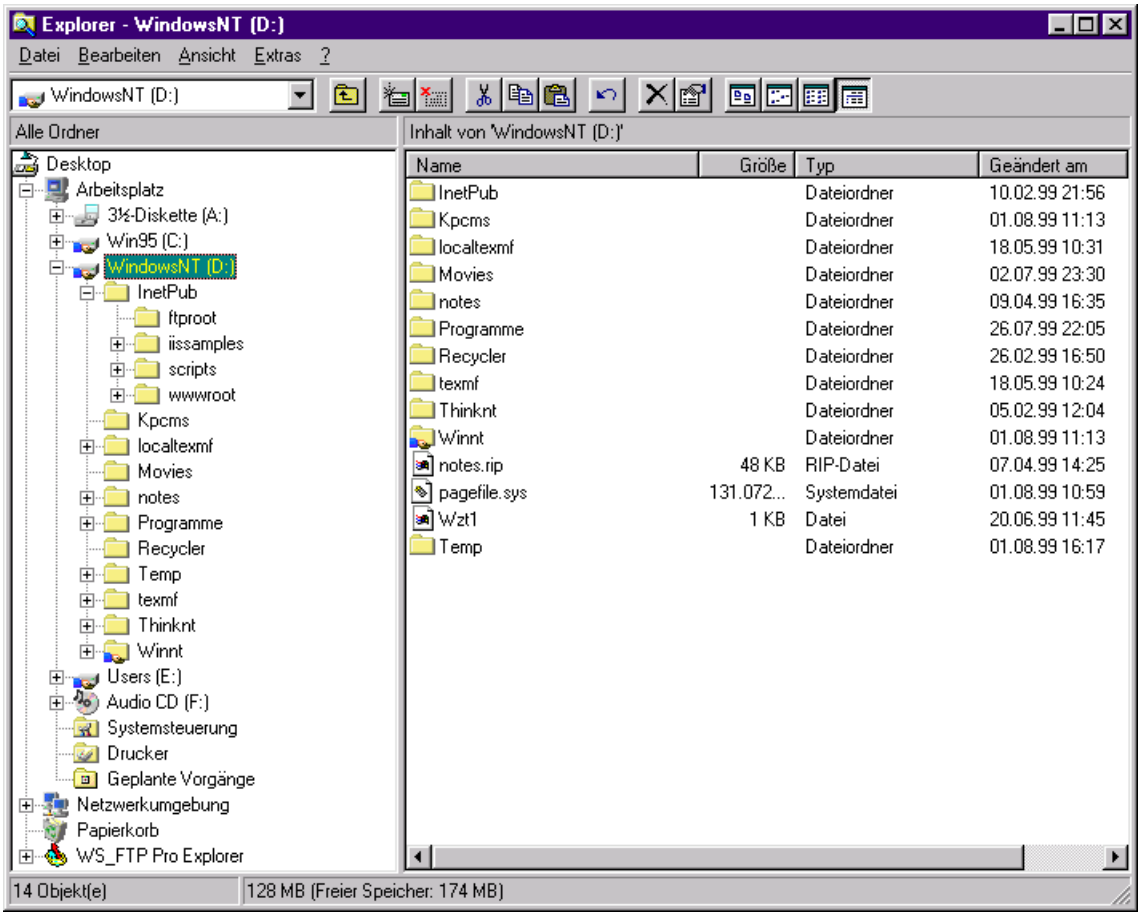


Figure 2.1: The Explorer of Microsoft Windows.

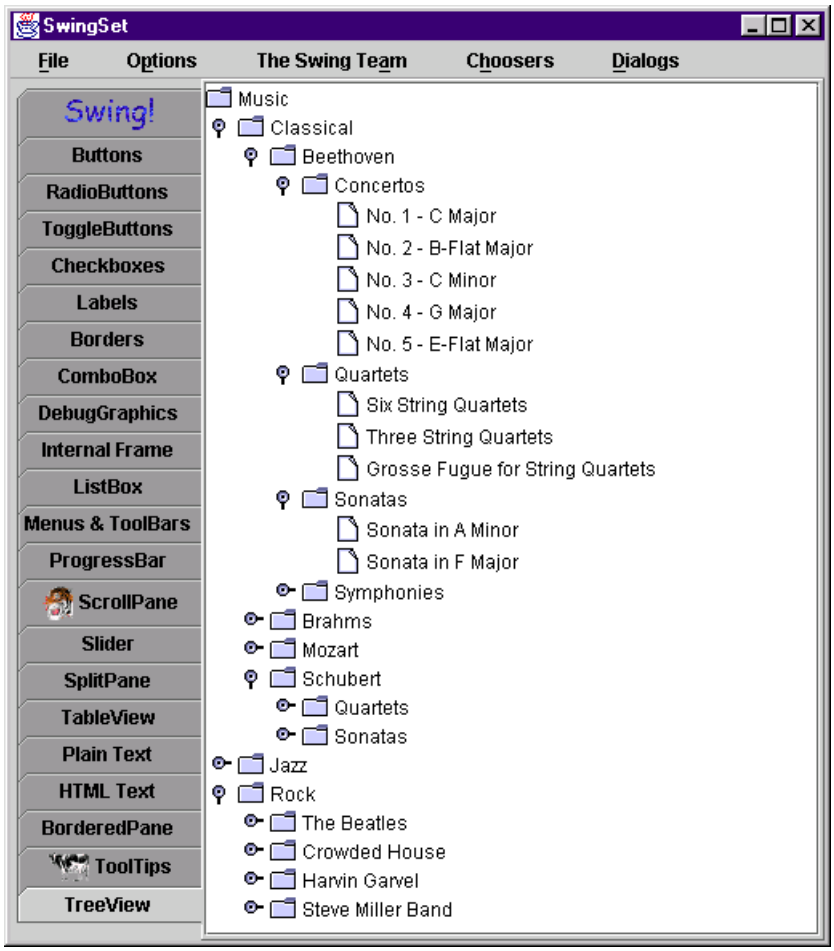


Figure 2.2: An example of the tree component of the Swing package.

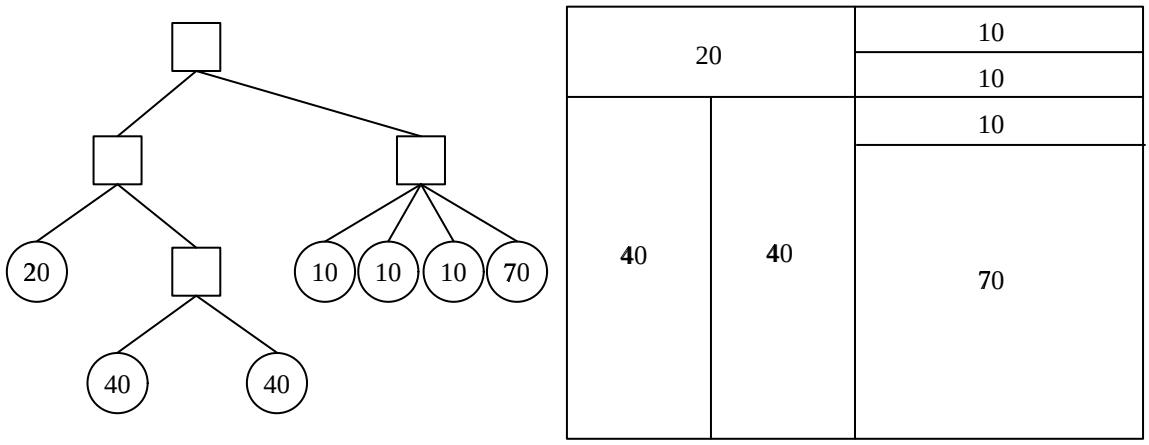


Figure 2.3: An example of a hierarchy and the corresponding Tree-Map. The numbers indicate the sizes of the nodes and the size of an inner node is the sum of its children.

2.3 Tree-Maps

Tree-Maps, as described in [JS91], are an interactive visualisation method which map hierarchical information to a rectangular two-dimensional display. This is done in a space-filling manner, meaning that 100% of the display space is used. Hierarchical structures contain structural information which is associated with the hierarchy and content information which is associated with each tree node. Tree-Maps can visualise both kinds of information.

In order to visualise the structure of a hierarchy, the Tree-Maps method assigns more important information more display space. Each node in the hierarchy is assigned a weight. The user can select between different types of weight, depending on the type of information being displayed (e.g. disk usage, if a directory tree is displayed), as well as other visualisation properties (e.g. colours of different types of nodes). To be meaningful to the user, the size of a node is proportional to its weight.

The display space is partitioned into rectangular bounding boxes to represent the tree structure. Each node in the tree is represented by a rectangle. The children of a node are drawn within the rectangle of their parent. To use 100% of the display space, there is no border around a node. The size of a child node within the rectangular boundary of its parent is proportional to its weight (see Figure 2.3).

Tree-Maps use a recursive drawing algorithm and the drawing is done back to front. A parent node is drawn before its children nodes. After each level of the hierarchy, the partitioning direction changes from vertical to horizontal or vice versa. The time to calculate and draw a Tree-Map is O(n).

Tree-Maps provide a good overview of a hierarchy. However, hierarchies with many leaf nodes lead to visualisations with very small rectangles. Therefore the user has only a rough but no detailed overview. The possibility to zoom into regions with small nodes can help to solve this problem (see Figure 2.4).

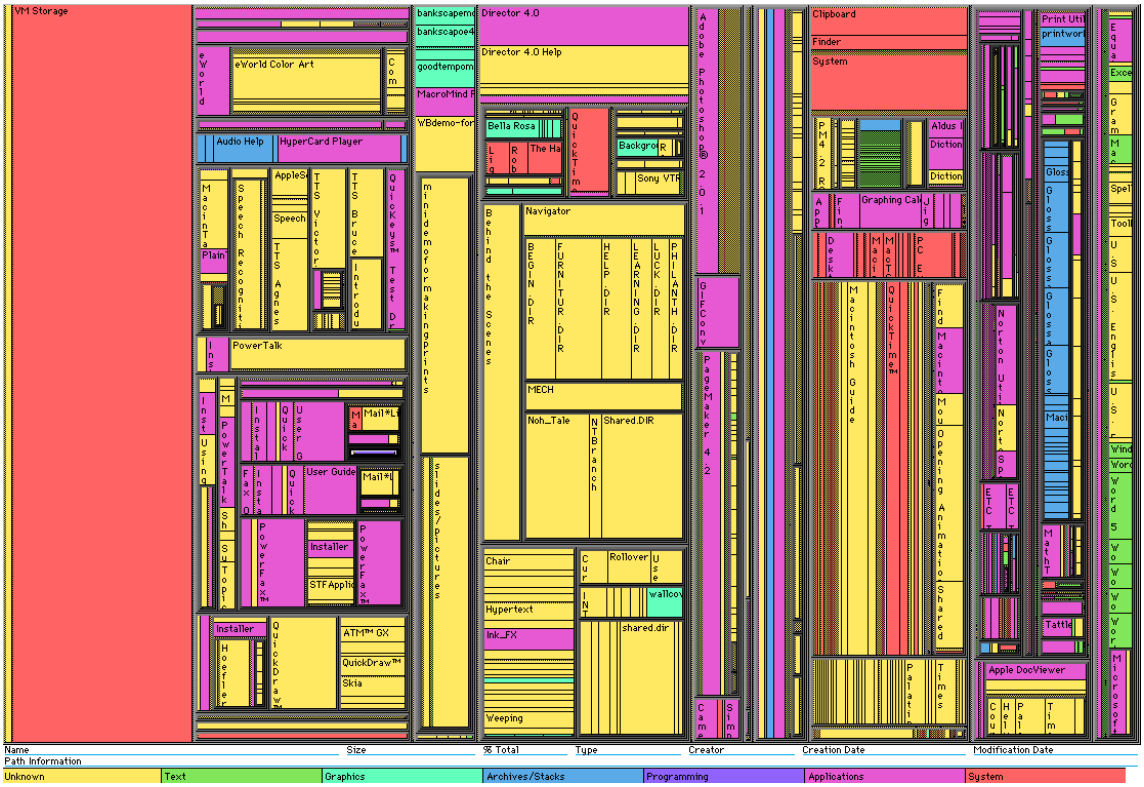


Figure 2.4: The TreeViz application displaying a file system.

2.4 Hyperbolic Browser

The hyperbolic browser [LRP95] uses a focus+context technique for visualising and manipulating large hierarchies. Focus+context means that the entire hierarchy can be seen as context while focusing on particular parts of the hierarchy. The relationship between the focused part and the hierarchy is visible. The hyperbolic browser technique is patented by Xerox Corporation [LR97].

The hierarchy is laid out on a hyperbolic plane (see Figure 2.5) and this plane is mapped onto a circular display space. The hyperbolic plane has the advantage that it is a non-Euclidean geometry in which the circumference of a circle grows exponentially with its radius. The connection to hierarchies is that hierarchies also tend to grow exponentially with depth. Therefore if you want to visualise a hierarchy in hyperbolic space, you have approximately the same space available for parents, children and siblings everywhere in the hierarchy. To layout a node, a wedge of the hyperbolic plane is assigned to it. The wedge angles out from the node. The layout is done recursively and every node places its descendants in the assigned wedge.

For display, the hyperbolic plane has to be mapped to a two-dimensional plane. A conformal mapping, called the Poincaré model, is used. A conformal mapping preserves angles but lines in the hyperbolic plane are arcs in the unit disc.

Initially the root of the hierarchy is displayed at the centre of the display space. The focus can be changed either by clicking on any visible point, bringing the point to the centre, or by dragging any point of the visualisation to any other position. Animated transitions are used to help the user to understand the changes across different views. The layout of the hierarchy is only performed once even if the focus changes. This is possible because a rigid transformation is used to perform change of focus.

2.5 Cheops

Cheops [BPV96] is an approach to visualise even huge hierarchies with millions of nodes. The idea behind the Cheops method is to compress the simple tree visualisation as shown in Figure 2.8. A node in the hierarchy is visualised as a triangle.

If a hierarchy is visualised using a simple tree view visualisation, much space is wasted. If the hierarchy is very deep, the tree view becomes very wide and the available display space is not well used. Cheops takes care of this problem by compressing the tree view. This is done using two techniques; the triangles are drawn overlapping and this reduces the space needed between siblings, and one triangle is reused to display the information of more than one node within a level in the hierarchy. These “overloaded” triangles become unambiguous when their parent is selected.

Figure 2.6 shows the simple tree view visualisation of a fully expanded 3x3 deep hierarchy. To show how Cheops compresses this view, Figure 2.7 shows the Cheops visualisation of the same hierarchy. If, in the Cheops pyramid, the leftmost triangle on the middle level is selected, the three leftmost triangles on the last level represent the nodes A,B and C. Thus by selecting the parent of a node, the overloaded triangles get unambiguous. If a node is selected the path from the selected node to the root node of the hierarchy is highlighted.

A triangle can have different colours and borders. The colour of a triangle visualises its relationship to the selected node. The following relationships are possible:

- Selected node
- Uncle node

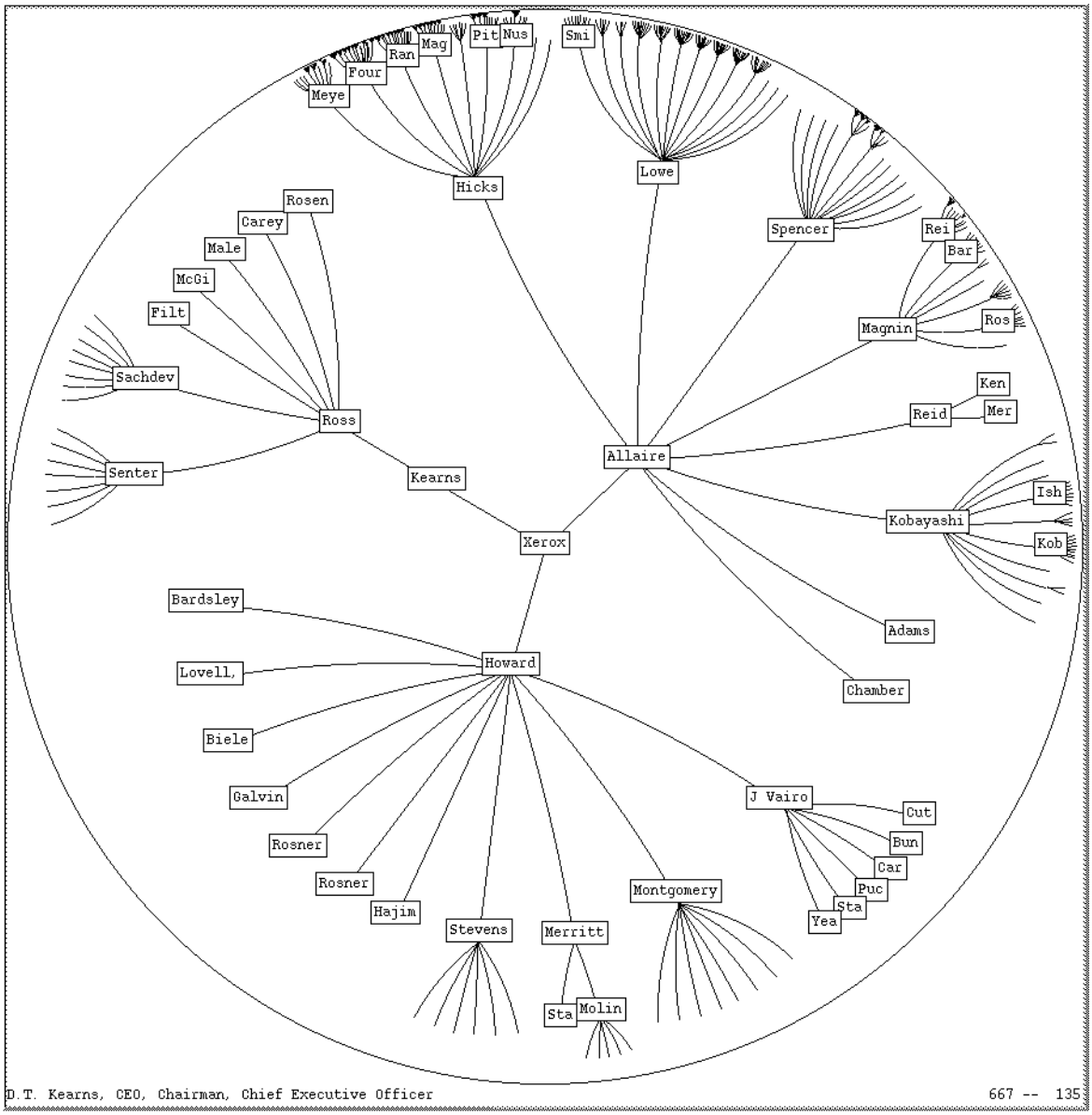


Figure 2.5: The Hyperbolic Browser.

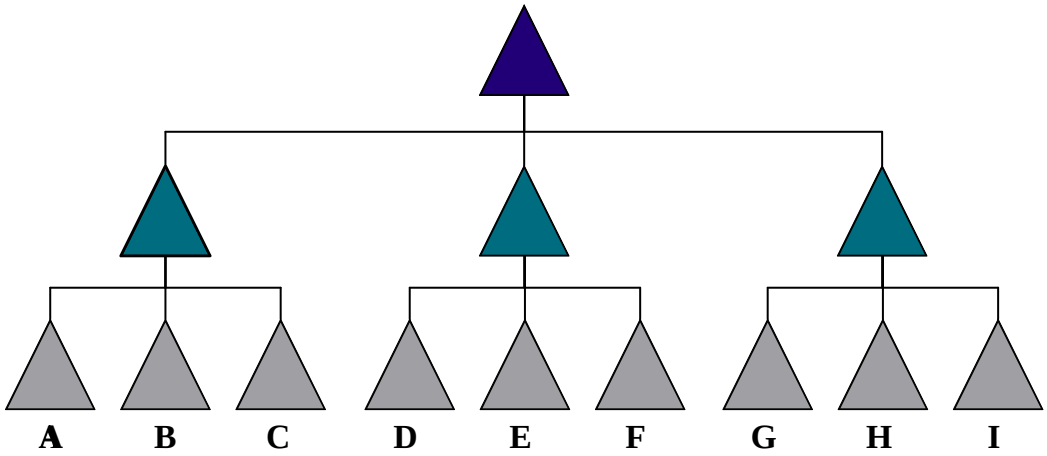


Figure 2.6: A simple tree view of a fully expanded 3x3 deep hierarchy.

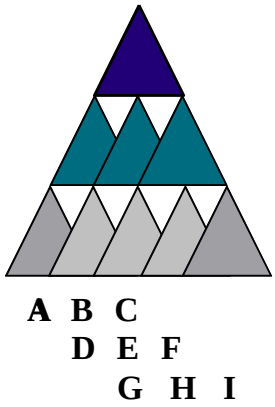


Figure 2.7: The Cheops compressed view of the 3x3 deep hierarchy.

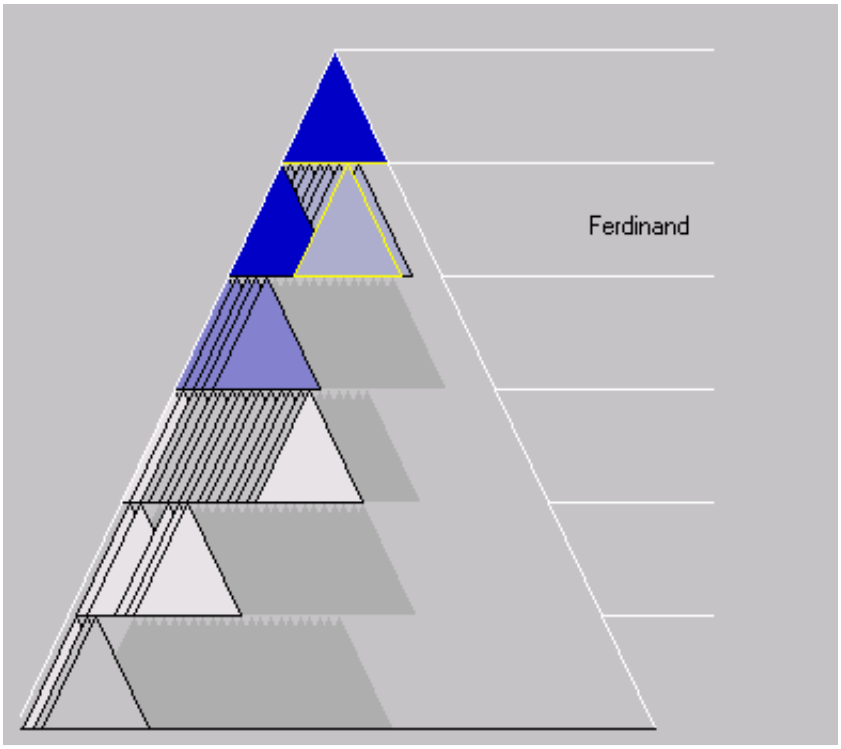


Figure 2.8: The Cheops visualisation of a hierarchy using colour coding and different borders.

- Child node
- Singular node
- Overloaded node
- Unused node

A singular node indicates, that the triangle is not overloaded. The colour of the border of a triangle indicates the current focus point.

If, via a mouse click, a node is selected, all triangles which are not affiliated with the selected node are given the visual representation of an unused node. This has the advantage, that the global context of the hierarchy is not lost. If the user moves the mouse pointer over a triangle without clicking on it, the branch of the underlying node is highlighted. Thereby the user can explore parts of the hierarchy which are not selected without losing the current selection.

2.6 Information Slices

Information slices [AH98] are another two-dimensional technique to compactly visualise large hierarchies. They use one or more semi-circular discs and each disc represents multiple levels of a hierarchy. If more levels have to be shown than one disc can hold, a cascading series of discs are used. At each level in the hierarchy, the space used for a child is proportional to its total size.

Figure 2.9 shows the visualisation of a hierarchy using information slices. The left disc shows some levels of the hierarchy. If more levels of the hierarchy are opened, the right disc is also used

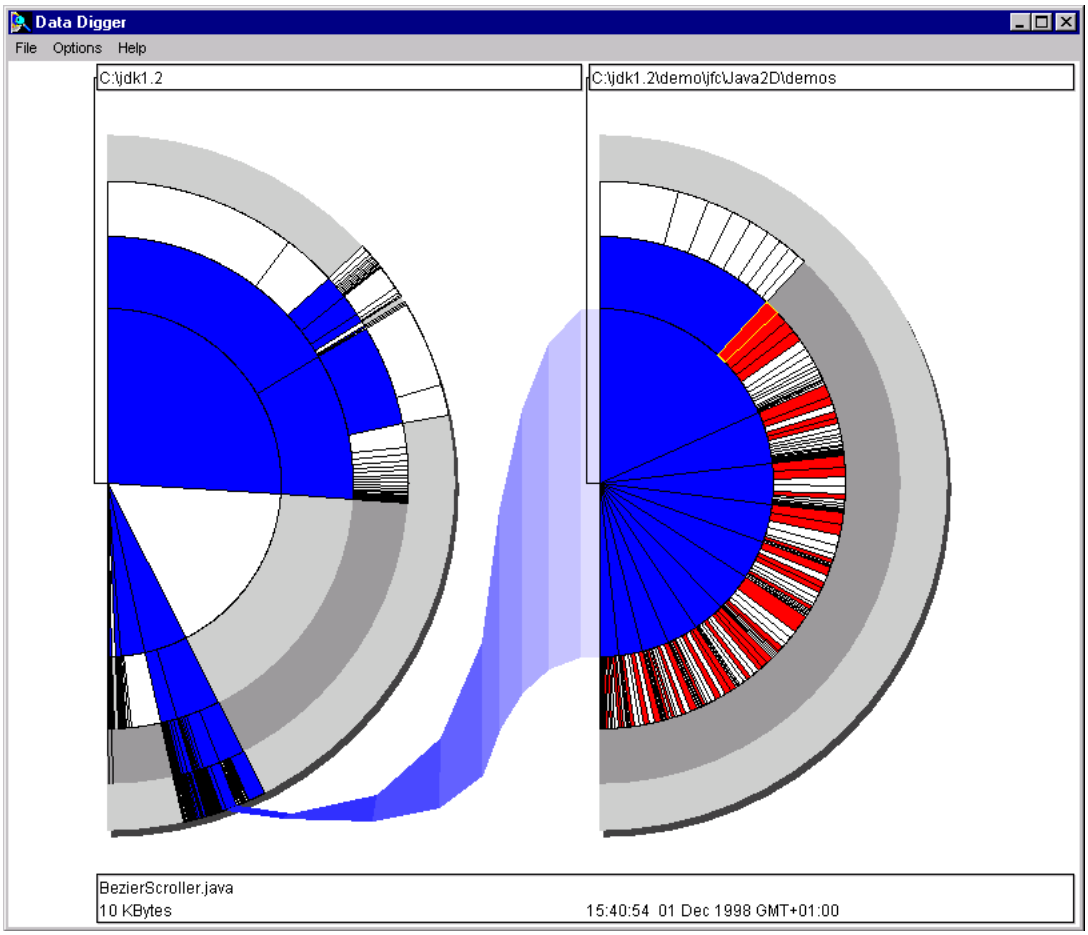


Figure 2.9: The Data Digger application using the Information Slices technology.

to display all opened levels. If still more levels are opened, the leftmost disc is iconified, the levels of the right disc are moved to the left disc and the right disc displays the new levels. The number of levels per disc is user configurable and typically between 5 and 10.

Early experience showed that information slices are well-suited to navigating very deep hierarchies. They are also useful to get a feeling for the relative sizes of different parts of a hierarchy.

2.7 Cone Trees

Three-dimensional visualisations of hierarchies are used to maximise the usage of available display space and to visualise the whole structure of a hierarchy. Cone Trees (see Figure 2.10) [RMC91] are one method to lay out hierarchies in three dimensions. Every node of the hierarchy is visualised like an index card and is the apex of a cone. A cone in a Cone Tree is the same as a wedge in a tree view.

The root node of the hierarchy is placed near the top of the three-dimensional display space. It is the apex of a cone and the children of the node, leaf nodes as well as non-leaf nodes, are evenly spaced along its base. The next layer of nodes is drawn below the first layer. This is done recursively until the whole hierarchy is drawn. Each cone has the same height, that is the height of the available display space divided by the depth of the tree. The size of each node is calculated in a way that the whole hierarchy fits into the room. The diameter of the cone base is reduced each level. This insures

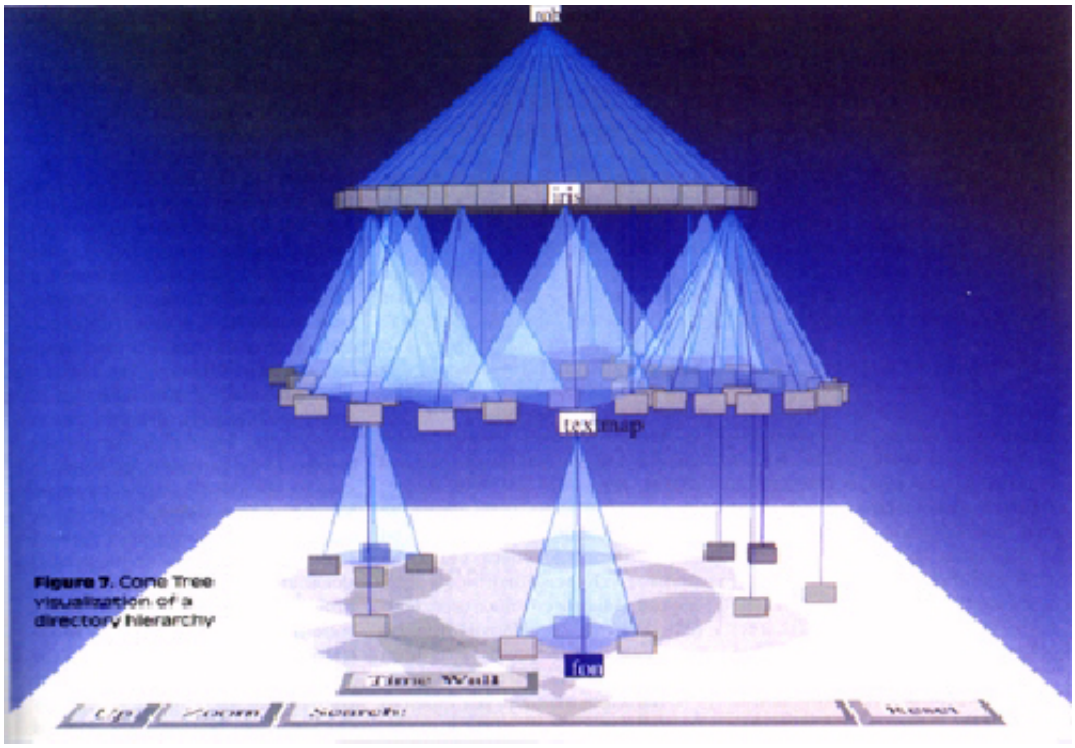


Figure 2.10: The visualisation of a hierarchy using Cone Trees.

that the bottom layer fits in the width of the available display space. The text of a node is only drawn if the node is highlighted. Due to the aspect ratio of the cards the text does not fit very well in the available space. An approach to solve this problem are Cam Trees (see Figure 2.11). Cam Trees are Cone Trees that grow from left to right and not from top to bottom.

When the user selects a node, the Cone Tree rotates and the selected node, as well as each node on the path from the root to the selected node, is brought to front and highlighted. The rotation is animated so that the human perceptual system can track it. The rotation of all necessary cones is done in parallel and is deliberately designed to take about one second. This time seems to be enough for the user to understand what is going on. There is also the possibility that the Cone Tree rotates continuously. This should help to visualise the structure of the hierarchy to the user. A cone is shaded transparently, so that it does not block the view of cones behind it. As some parts of the hierarchy are often not used during navigation and exploration, the user can expand or collapse parts of the hierarchy.

To enhance the user perception of three-dimensional space, some further techniques are used. A three-dimensional perspective transformation results in size changes of nodes depending on their distance to the user. Therefore nodes that are more fare away from the user are drawn smaller. Lighting effects are used to make the whole scene more natural. Thus nodes in the background are drawn darker than nodes in the foreground. Shadows of cones and nodes also provide extra information to the user. The darker the shadow under a particular part of the hierarchy becomes, the more nodes are in this part of the hierarchy.

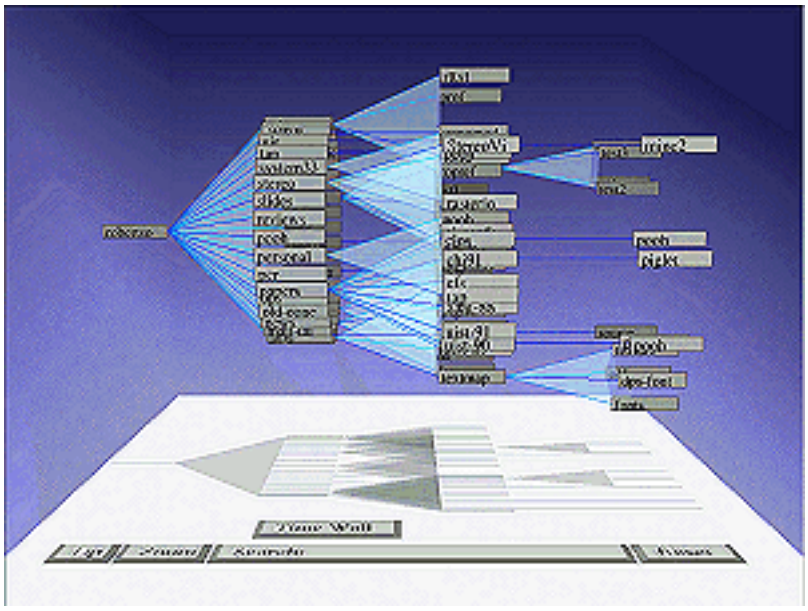


Figure 2.11: The visualisation of a hierarchy using Cam Trees.

2.8 File System Navigator

The File System Navigator (FSN) is a three-dimensional directory browser for the UNIX operating system. The technology behind FSN was developed and patented by SGI [ST96a] [ST96b]. FSN uses a landscape metaphor to lay out a hierarchy.

A hierarchy is laid out on a landscape using a conventional outline method. A directory is visualised as a square pedestal. Lines connecting the pedestals visualise the structure of the hierarchy. The width and the height of a pedestal depends on the number of files in the corresponding directory. The files of a directory are laid out on the top of the pedestal. A file is displayed as a column. The height of a column represents the size of a file. A minimum size is used to prevent too small columns. The colour of a column depends on the age of its file. An icon on the top of a column shows the type of the displayed file. The names of the directories are drawn in front of the pedestals as labels on the landscape (see Figure 2.12).

To navigate through the displayed scene, different methods can be used. The user has the possibility to freely fly around in the scene using the mouse. If the user selects an object, a zoom method can be used to fly to the selection. Interesting places and viewpoints can be marked . The marked places are displayed in a listbox and can be visited again by selecting. A back button makes it possible to undo navigational steps. To make selections visible, a selected object is highlighted with a spotlight. An overview window can be opened to show a two-dimensional outline of the entire hierarchy (see Figure 2.13).

In spite of the fact that FSN uses three dimensions to visualise a hierarchy, this method does not have many advantages over a two-dimensional visualisation. This is because the third dimension is used to display the pedestals and columns of the directories and the files only. The structure of the hierarchy is just a two-dimensional outline drawn on the landscape.

Another visualisation technique that uses similar display methods is the Harmony Information Landscape. It was part of the Harmony project at the IICM. The Harmony Information Landscape [Eyl95] [Wol96] is used to visualise the hierarchy of a Hyperwave server.

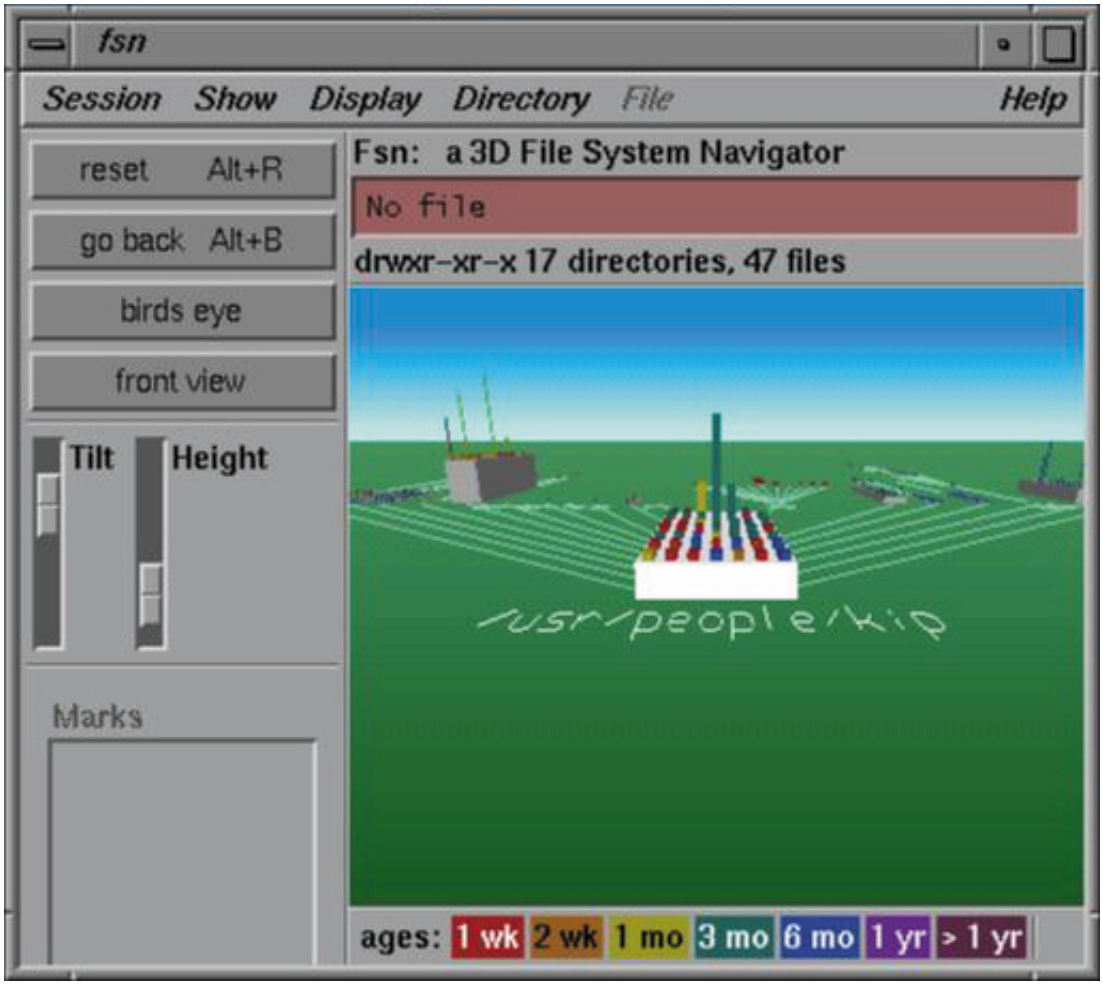


Figure 2.12: The FSN application window.

Figure 2.13: The overview window of the FSN application.

2.9 MAPA

MAPA [DK98] is a web mapping service of Dynamic Diagrams, Inc. It provides enhanced facilities to navigate through large web sites and it can be used to visualise a hierarchical structure of the site.

The MAPA system extracts the hierarchical structure from an arbitrary web site, stores it in a database and provides visualisational and navigational facilities. The design of MAPA makes it most useful for web sites from 500 to 50,000 web pages.

MAPA consists of four main modules:

- **Web Walker**
  The web walker is used to gather link information from a website.
- **Organizer**
  The organizer creates a hierarchical structure of the link information gathered by the web walker and stores the hierarchy and the link and meta-information in a central database.
- **Link List**
  The link list is a CGI-generated web page that shows a summary of all the links relating to a particular document. The link list gets the links from the central database.
- **Visualisation Applet**
  The visualisation applet is a Java applet that visualises the hierarchy, which is stored in the database, as a map.

The map, generated by the visualisation applet, is a local map (see Figure 2.14). It always shows the location of a single focus page within the overall structure of the hierarchy. The visualisation of the map consists of the following parts:

- **The focus page**
  The focus page is in the center of the map and shows the current active document. Positioned in the center of the map, it should help to visualise the current position of the user within the hierarchy.
- **The ancestry of the focus page**
  The orange pages, placed on the lower left of the map, are the parent pages of the focus page. They visualise the path to the root page of the hierarchy.
- **The child pages**
  The child pages of the focus page have a green colour. All child pages are shown. To be able to draw them without an overlay, they are placed behind the focus page along the horizontal diagonal of the isometric projection.
- **The grandchild pages**
  The blue coloured grandchild pages are placed behind the child pages, in parallel rows. They are shown to be able to go directly to a grandchild page of the focus page and to have a visual indication of the extents of the current part of the hierarchy.

Navigation in MAPA can be done by using two different techniques. Either the user selects one of the pages on the map and this page is going to be the new focus page (map navigation), or the user can go to any other web site (site navigation). To see the name of a page the user can move the mouse pointer over it and the name is shown above the page. If the user moves over a child page, the names of all child pages appear.

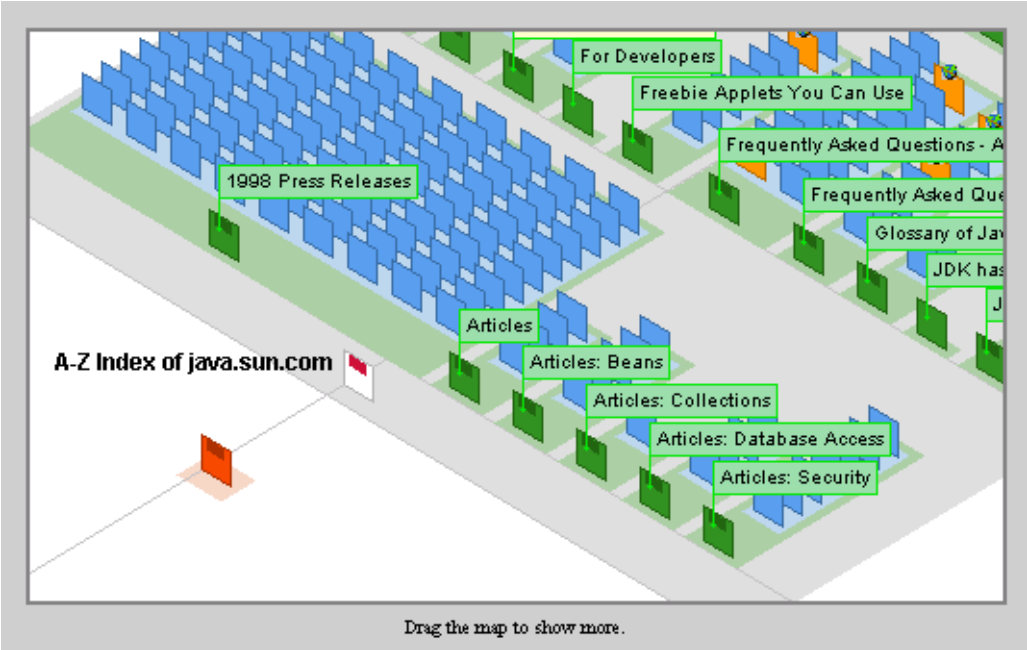


Figure 2.14: The MAPA visualisation of the hierarchy of a webserver.

Chapter 3

Graphical User Interfaces with Java

3.1 Introduction to Java

Java is an object-oriented programming language developed by Sun Microsystems which emerged in the early 1990's [AG96] [Javc]. Java has many similarities with older programming languages like C, C++ and Smalltalk, but its design makes it easy to develop platform-independent, object-oriented software. As Ken Arnold and James Gosling have noted in [AG96]:

> "The Java programming language (hereafter called simply "Java") has been warmly received by the world community of software developers and Internet content providers. Users of the Internet and World Wide Web benefit from access to secure platform-independent applications that can come from anywhere on the Internet. Software developers creating applications in Java benefit by developing code only once, with no need to "port" their applications to every software and hardware platform.
>
> For many, Java is known primarily as a tool to create applets for the World Wide Web. "Applet" is the term Java uses for a mini-application that runs inside a web page. An applet can perform tasks and interact with the user on their browser page without using resources from the Web server after being downloaded. Some applets may, of course, interact with the server for their own purposes, but that's their business.
>
> Java is indeed valuable for distributed network environments like the Web. However, Java goes well beyond this domain to provide a powerful general-purpose programming language suitable for building a variety of applications that either do not depend on network features, or want them for different reasons. Java's ability to execute code on remote hosts in a secure manner is a critical requirement for many organisations.
>
> Other groups use Java as a general-purpose programming language for projects where machine independence is less important. Java's ease of programming and safety features help produce debugged code quickly. Some common programming errors never occur because of features like garbage collection and type-safe references. Modern network-based and graphical user interface-based applications that must attend to multiple tasks simultaneously are catered to by Java's support for multithreading, while the mechanisms of exception handling, ease the task of dealing with error conditions. While its built-in tools are powerful, Java the language is itself a simple language in which programmers can quickly become proficient."

The most significant difference to programming languages like C or C++ is the platform independence of Java. The compiler compiles programs written in Java into a bytecode which is translated

into machine code during runtime. The Java Virtual Machine is the interpreter which performs the translation. Bytecode for the Java Virtual Machine is equivalent to the machine code for a processor. The only part of Java that is not platform-independent is the Java Virtual Machine. Since many different virtual machines for the most important operating systems exist this is no disadvantage. The programmer using Java does not have to think about this at all. Programming languages that use an interpreter (think about the old and famous BASIC) tend to be slow but Java programs are almost as fast as programs written in C or C++. The object oriented design of the written software lets programs often perform even faster.

Maybe platform independence is the most important advantage of Java but a lot of other things make this programming language also very interesting. Using Java, the programmer does not have to take care about the storage management of his software. Java provides automatic storage management, called the garbage collector. Memory is automatically allocated and disallocated for the objects that are used. Simple to use exception handling is also provided as well as support for threads. As concurrent programming gains importance this seems to be very useful. A rich set of class libraries and the integrated API for graphical user interfaces are further advantages of Java. All these things together make the Java programming language very interesting to expert and novice users alike.

With Java a programmer can write every program that could be written with any other language. The capability to run within a virtual machine makes systems possible that go beyond the scope of traditional approaches. Java programs can run as different types of applications. Examples include:

- **Stand-alone Application**
  A stand-alone application is the same as a traditional application. The bytecode is usually loaded from a local disc and executed by a Java Virtual Machine. A stand-alone application can use local and remote resources and has no security restrictions.

- **Applet**
  An applet is a Java program that is executed within the Java Virtual Machine of a webbrowser. The bytecode of the applet is loaded from a remote server. Applets are often used for visualisations, animations or enhanced user input that would not be possible with HTML. Since an applet is executed on the computer of the user special security considerations have to be made. An applet is, by default, not allowed to access the local file system and cannot make network connections to servers other than the one it was loaded from.

- **Servlet**
  Servlets are Java programs that run within the Java Virtual Machine of a server. The kind of server is not further specified, and might be a webserver, a ftp-server, a mail-server or any other type of server. Nowadays servlets are most often used to dynamically generate web content. Similar to CGI-scripts they produce HTML output by a request of a user. By default, a servlet has no security restrictions. They have the same rights as stand-alone applications. Only if the bytecode of a servlet is loaded from a remote server are security restrictions applied.

- **Embedded System**
  Java can be used to write programs for embedded systems. If the embedded system, e.g. a small chip on a credit card, provides a virtual machine, bytecode can be loaded and executed. Who knows, maybe the microwave oven of the future uses Java to warm our meals.

The following sections give an overview of programming graphical user interfaces with Java. Basic knowledge of Java, C, C++ or a similar programming language is important for understanding the given samples.

3.2 The Abstract Window Toolkit

Even the first version of the Java Development Kit (JDK) provided a library to build graphical user interfaces. This library is called the Abstract Window Toolkit (AWT). As Java is a platform-independent programming language, the AWT has also to be platform-independent and the developed graphical user interfaces should look good and consistent on all platforms. These design goals were not achieved in Java 1.0. The produced user interfaces look equally mediocre on all systems and there are many bugs, even in the newest version of Java 1.0. Although the core class libraries of version 1.0 are very well designed, the AWT programming model is not really object-oriented and is also awkward.

With Java 1.1 a new version of the AWT came along. The most important modification in the new version of the AWT was the new event model. It is much clearer and it is object-oriented designed. The design of graphical user interfaces became much easier and most of the bugs of version 1.0 were eliminated. The AWT version, described hereafter, is the one used in Java 1.1. Except of the event handling there are many similarities between the AWT's of Java 1.0 and 1.1.

The user interface of a program not only consists of the graphical objects a user can see on the display. Sounds, feelings and other methods to communicate with the user are also parts of the user interface. Since this thesis is about the visualisation of information, this section describes only the graphical components of the AWT. It should be mentioned, that the AWT provides mechanisms for other types of interaction, e.g. playing sounds, too.

The Basic Components of the AWT

The AWT provides many basic components that are most often used to build graphical user interfaces. AWT components not only include standard components like buttons and labels, but also higher level components like a file dialog window. Each basic component has the look-and-feel of the used window manager. For example, a button does not look the same if a Macintosh system is used instead of a Windows based system. All of the components provide events to react on user input or modifications.

The components provided for user input are:

- Buttons
- Canvas
- Checkboxes
- Choices
- Labels
- Lists
- Menus
- TextFields
- TextAreas
- Scrollbars

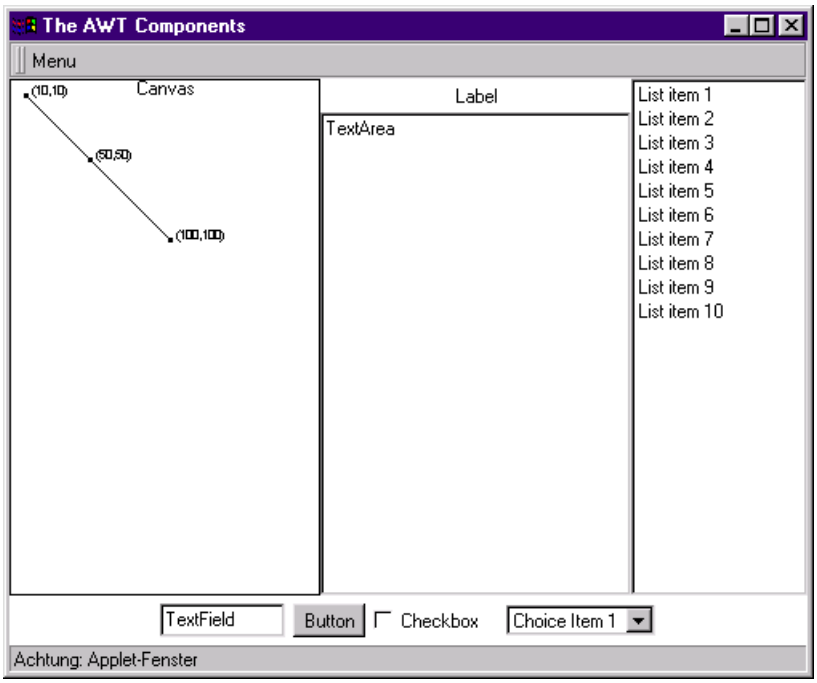


Figure 3.1: The basic components of the AWT.

Figure 3.1 shows these components under the Windows environment. It should be mentioned that the components, by default, provide basic features only. For example, a TextField does not provide mechanisms to check the user input. This has to be done by the programmer. Each of these components, except Menus, are derived from the *Component* class. The *Component* class contains the functionality needed for drawing, layout and event handling of a component. The *Canvas* class can be used to write custom components. For example, a class that is a subclass of *Canvas* could draw images or an alternative button with any kind of event handling.

Furthermore there exist container classes in which the components can be grouped and drawn. These containers are subclasses of the class *Container*. *Container* is derived from *Component* and therefore every *Container* object can also be used as a *Component* object. The container classes are:

- **Subclasses of Window**
  Frame, *Dialog* and *FileDialog* are subclasses of *Window*. *Frame* objects are used to create full-fledged windows. They are independent on other windows. *Dialog* objects in contrast to *Frame* objects do have a parent window and can be modal or non-modal.

- **Panel**
  Panel objects group components within a container. Components can be added or removed from a panel. A component that is part of a panel is displayed within the boundaries of it.

- **ScrollPanes**
  A *ScrollPane* is a special kind of panel. The purpose of a *ScrollPane* is to provide a scrolling possibility for a part of a window. The user can scroll in a *ScrollPane* using scrollbars. The way information is visualised and scrolled depends on the components that are inserted into a *ScrollPane*.

Figure 3.2: A panel that uses *FlowLayout*.

The Layout of Components

If components are added into a container and the container is drawn, the positions and sizes of the components are calculated by a layout manager. Every container has its layout manager and even if no layout manager is explicitely assigned a default layout manager is used. The sizes and positions of the components depend on the dimensions of the drawing area of their container. If the dimension of the container changes, e.g. if a window if resized, the components are laid out again. The sizes and positions can be different after the new layout. Next, some of the most important layout managers that are part of the AWT are described.

FlowLayout

FlowLayout uses a simple algorithm to arrange the components. The components are positioned line by line from top left to bottom right and are compacted to their smallest size. The order in which the components are positioned depend on the order in which they are added to the container. The following example adds three buttons to a panel that uses *FlowLayout*. Figure 3.2 shows the resulting screenshot.

```
Button button1 = new Button("Button 1");
Button button2 = new Button("Button 2");
Button button3 = new Button("Button 3");

Panel p = new Panel();   //uses FlowLayout by default

p.add(button1);
p.add(button2);
p.add(button3);
```

BorderLayout

BorderLayout is the default layout manager for all *Window* objects. A container that uses *BorderLayout* can hold up to five components. The five regions in a *BorderLayout* container are north, south, east, west and center. The center region gets the most space available. The north and the south part have the same width as the complete drawing area but they are only as high as their minimum height. The east and the west part are filled vertically but are only as wide as their minimum width. If a component is added to a *BorderLayout* container the name of the region is the first argument of the *add()* method. *BorderLayout* suits well for applications with a menu, a status bar and a center region, e.g. an editor. Following example uses a *BorderLayout* layout manager and the resulting screenshot can be seen in Figure 3.3.

```
Button buttonNorth = new Button("North");
```

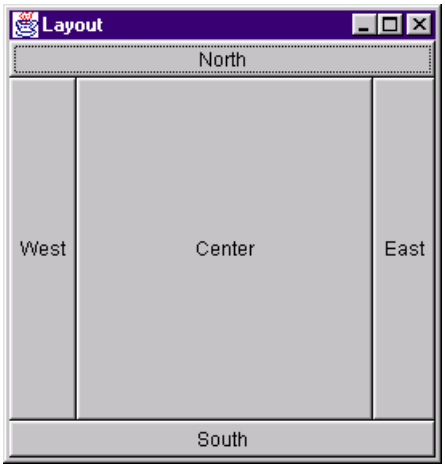


Figure 3.3: A panel that uses *BorderLayout*.

```
Button buttonSouth = new Button("South");
Button buttonEast = new Button("East");
Button buttonWest = new Button("West");
Button buttonCenter = new Button("Center");

Panel p = new Panel();
p.setLayout(new BorderLayout());

p.add("North", buttonNorth);
p.add("South", buttonSouth);
p.add("East", buttonEast);
p.add("West", buttonWest);
p.add("Center", buttonCenter);
```

CardLayout

The functionality of the *CardLayout* layout manager can be best imagined like a pile of cards. From a pile of cards only the face of the topmost card can be seen. If another card is taken and laid on top of the pile, only the face of this new card is visible. Using the same technique, a panel using *CardLayout* can hold more than one component like a pile of cards. Only the topmost component can be seen. *CardLayout* provides functionality to change the component that lies on the top of the pile.

GridLayout

With the *GridLayout* layout manager components can be positioned in a tabular manner. Arguments to the constructor of a *GridLayout* object are the number of rows and columns that should be used. Every component in the table has the same size. The position of a component in the table depends on the order in which the components are added into the container. Like *FlowLayout*, the components are added from top left to bottom right. The last row of the table does not have to be fully filled. The remaining part of the table is simply empty.

GridBagLayout

GridBagLayout positions the added components also in a tabular manner. The difference to *GridLayout* is, that the components do not have to have the same size and there do not have to be the same number of columns per row. Each component is referenced by a GridBagConstraints object. The GridBagConstraints object is responsible for the layout of the component within the table. *GridBagLayout* is the most sophisticated layout manager that comes with the AWT. As it is very sophisticated it is also very difficult to understand. A complete description would go far beyond this brief overview of the AWT.

Absolute positioning

Absolute positioning of components within containers is also possible. It should be avoided because of the different platforms a Java application can run on. Absolute positioning is useful if the target platform is known but differences in the sizes of e.g. fonts make it very hard to position components for more than one target system.

Combination of Layouts

For most applications one layout manager will not be enough to fulfil all layout problems, If more than one layout manager has to be used within a window, *Panel* objects can be used. As mentioned earlier panels are used to group components. As a *Panel* object is a subclass of *Container* it has a layout manager. Of course, different panels can have different types of layout managers. Thus the combination of layout managers within one window is no problem at all. The next example shows the usage of *BorderLayout* and *GridLayout* in one window and Figure 3.4 shows the resulting screenshot.

```
Button buttonNorth = new Button("North");
Button buttonSouth = new Button("South");
Button buttonEast = new Button("East");
Button buttonWest = new Button("West");
Button buttonCenter1 = new Button("Center 1");
Button buttonCenter2 = new Button("Center 2");
Button buttonCenter3 = new Button("Center 3");

Panel p = new Panel();
p.setLayout(new BorderLayout());

Panel p1 = new Panel();
p1.setLayout(new GridLayout(3,1));
p1.add(buttonCenter1);
p1.add(buttonCenter2);
p1.add(buttonCenter3);

p.add("North", buttonNorth);
p.add("South", buttonSouth);
p.add("East", buttonEast);
p.add("West", buttonWest);
p.add("Center", p1);
```

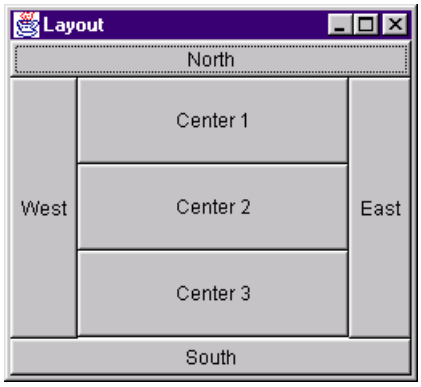


Figure 3.4: A panel that uses a combination of *BorderLayout* and *GridLayout*.

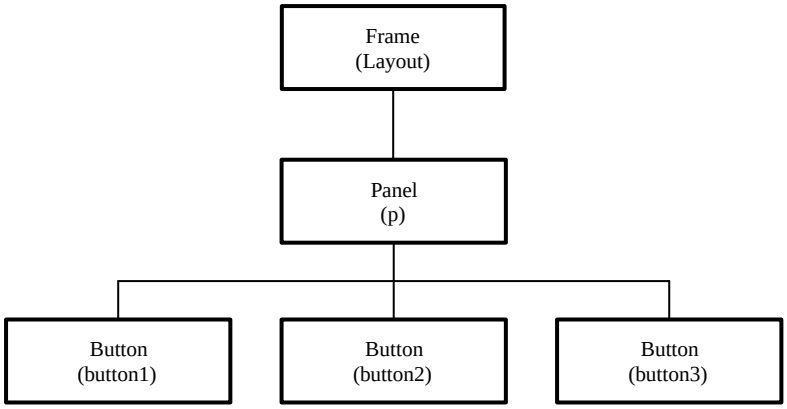


Figure 3.5: The drawing hierarchy of the *FlowLayout* window.

The Drawing of Components

After the components are laid out within their containers they have to be drawn. Drawing requests occur, for example, if the user interface appears for the first time or if a window has been resized. The drawing begins with the components at the back and ends with those at the front. Thinking of a container, the container is drawn before its components are drawn. The order in which components are drawn looks like a hierarchy (see Figure 3.5). The drawing of a component in one level has to be finished before components of the next level are drawn. Otherwise it could happen, that components are not displayed properly. The correct order is guaranteed by the AWT drawing system.

A component only draws itself if the AWT drawing system tells it to do so. The AWT creates threads to order drawing requests. There are three important methods that are used for drawing operations. All of them are methods of the class *Component*.

- ***update()***
  The *update()* method is called by the AWT drawing system to tell the component that it should draw itself. If not overwritten, *update()* clears the background of the component and then invokes the *paint()* method which is responsible for the drawing.

- ***repaint()***
  When the *repaint()* method of a component is called, the component is scheduled for an *up-*

date() by the AWT drawing system.

- ***paint()***
  The *paint()* method draws the component. By default it does nothing. Most often the *paint()* method is the only drawing method that is overwritten by subclasses of *Component*.

The *repaint()* method has no arguments and the *update()* and *paint()* methods have only one argument, the *Graphics* object. The *Graphics* object represents the drawing context of the application. If the *paint()* method wants to draw something it uses the functionality of the *Graphics* object. Therefore the *Graphics* object hides the platform dependent operations from the application. The graphical primitives that can be drawn are:

- Lines
- Polygons
- Rectangles
- Arcs
- Ovals
- Text
- Images

The *Graphics* object also provides methods for filling these primitives with a specified colour. The colours and fonts can be changed as well as the drawing mode (XOR or paint).

Event Handling

The event model of the Java 1.1 AWT uses a kind of observer design pattern [GHJ+95]. If some kind of action happens to a component, e.g. the user clicks with the mouse on a button, the component initiates an event. It is said that the component "fires" an event. One or more objects, called listeners, can register to be notified if a component fires an event. This model has the advantage that the place where an event is created and the place where the event is handled can be separated.

There are several different types of event objects. The type of the event object depends on the component that has fired it and the type of action that occured. Custom components can have custom event classes. The event listeners have to implement a particular type of listener interface depending on the kind of events they want to listen for. To register a listener to events of a particular component the listener has to be added to a list of listeners using the *addXXXListener()* method of the component. XXX represents the type of event the listener wants to listen for. Examples of *adddXXXListener()* methods are *addKeyListener()* or *addMouseListener()*.

The next example shows the usage of the *addActionListener()* method. Event listeners can be of any class but inner classes are very often used for event handling. Inner classes have the advantage that they have access to the variables and methods of their parent class.

```
...
Button button = new Button("Hi folks, let's beep");
button.addActionListener(new MyButtonListener());
...
```

```
class MyButtonListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      Toolkit.getDefaultToolkit().beep();
   }
}
```

For listener interfaces that have more than one method, adapter classes are provided to simplify the usage of the interfaces. The adapters implement the interface methods using default methods. Most of these default methods are empty and do nothing. Therefore the programmer has only to derive the listener class from the corresponding adapter class. Only the necessary methods have to be overwritten and not all of them have to be implemented. An example of a listener interface and the corresponding adapter is the *WindowListener* interface and the *WindowAdapter* class.

3.3 Swing

Swing is the newest graphical user interface toolkit for Java. It is a class library that is available for Java 1.1 and Java 1.2. While it is an extra package for Java 1.1 it is already included in Java 1.2 and therefore also part of the Java Development Kit 1.2. Mark Andrews wrote several very good articles about an introduction to Swing and that is the reason why parts of this section are based on his work [Anda] [Andb].As he noted in [Anda]:

> "The Swing component set is a new graphical user-interface (GUI) toolkit that simplifies and streamlines the development of windowing components – that is, the visual components (such as menus, tool bars, dialogs and the like) that are used in graphically based programs.
>
> With Swing, you can develop powerful and scalable GUIs that have precisely the "look and feel" that you specify. That means that when you write a program using Swing components, a user can execute your program without modification on any kind of computer, and it will always look and feel just like a program written specifically for the particular computer on which it is running.
>
> Furthermore, Swing components are said to be lightweight because they don't rely on user-interface code that's native to whatever operating system they're running on. Consequently, they can usually be implemented using less code than earlier "heavyweight" components require."

Swing and AWT

When the new version of the AWT 1.1 replaced the older version 1.0 many things were changed and the event handling architecture has completely been replaced by the new listener event model. Although Swing is newer than the AWT 1.1 it does not replace it but extends it. With Swing, the original design goal, the ability to write user interfaces that look good and consistent on all supported platforms, at last became reality.

The Swing set contains more components than the AWT and many enhancements make it easier to write modern graphical user interfaces. Swing components are lightweight, that means they do not rely on native code. Many of the old AWT components are replaced by new, 100% pure Java, versions. Examples of old AWT components that are replaced by newer, lightweight components are:

- Buttons

- Labels
- Menus
- Textfields etc.

Components that do not exist in the AWT and are therefore new in the Swing set are:

- Trees
- Tables
- Tabbed-Panes etc.

Figure 3.6 shows some of the new components from Swing. For the programmer it is very easy to recognise these new components because most of their names start with a *J* and the rest of the names are equal to the old AWT names. For example the name of a button in Swing is *JButton* and the old AWT name is only *Button*. In spite of the new versions of the components the old components can still be used together with the new versions within one application. The event handling mechanism is still the same. Swing components are also much more sophisticated than their AWT counterparts. Three examples of why this is true are:

- Swing buttons can display both text and images.
- Most of the Swing components provide borders that can be drawn.
- A Swing component does not have to be rectangular. For example a button could also be round or triangular.

Java Foundation Classes

The Java Foundation Classes (JFC) [Gea99] are a set of class libraries that are used to develop user interfaces for enterprise-ready applications. As Figure 3.7 shows is Swing a part of the JFC library set. It may be a little bit confusing that the Java 2D API and the Drag and Drop API are not part of the Swing package. This is because both APIs use native code to perform some of their operations and Swing components never use native calls because otherwise they would not be 100% pure Java. As noted in [ELW98]:

> "In April of 1997, JavaSoft announced the Java Foundation Classes, or JFC, which supersedes (and includes) AWT. A major part of the JFC is a new set of user interface components that is much more complete, flexible, and portable. These new components are called "Swing." (The JFC also includes a comprehensive facility for 2D graphics, printing, and "drag-and-drop.") With Swing, you can design interfaces with tree components, tables, tabbed dialogs, tooltips, and many other features that computer users have grown accustomed to."

As described in the previous section, Figure 3.7 shows that the Swing library really does not replace the AWT but sits atop parts of the AWT set. Beside Swing there are three major parts of the JFC:

- **Accessibility API**
  The Accessibility library can be used to develop software for handicapped people. For example an application could use other fonts and colours for people with vision defects.

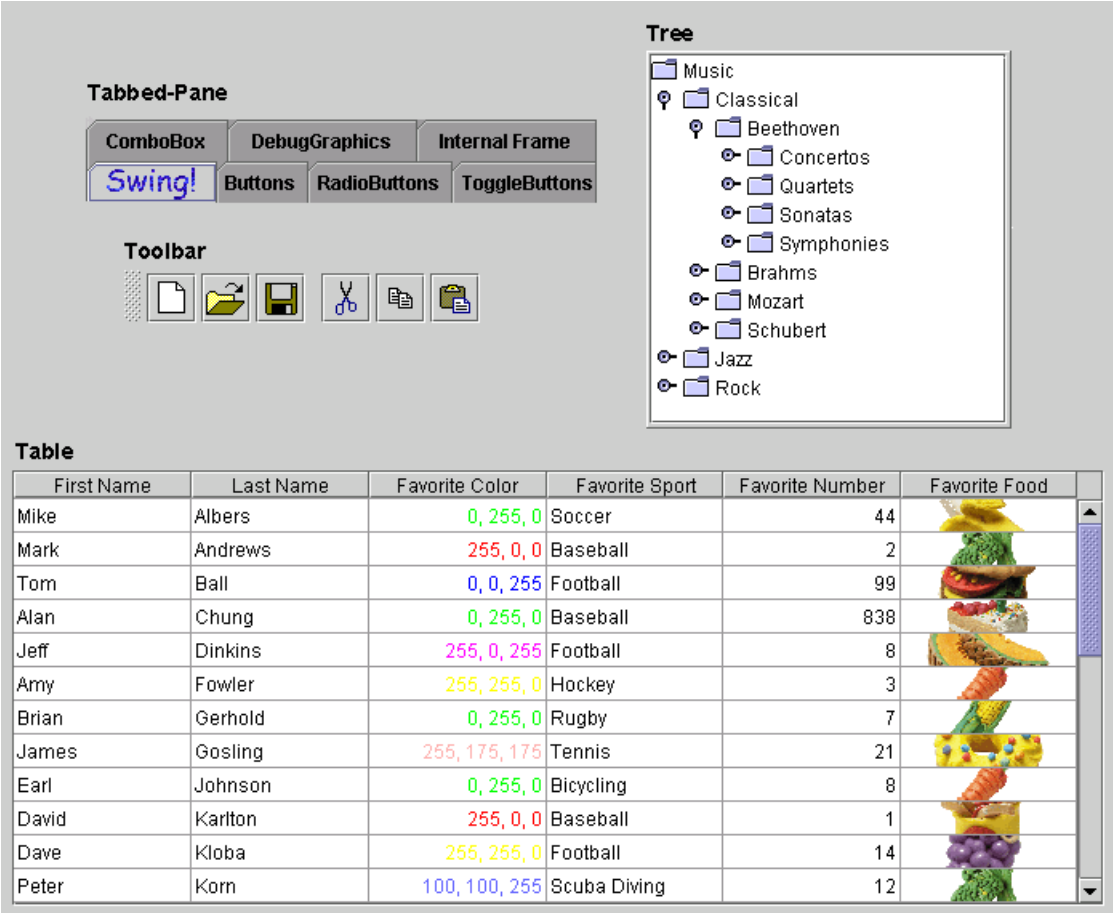


Figure 3.6: Some of the new components of the Swing package.

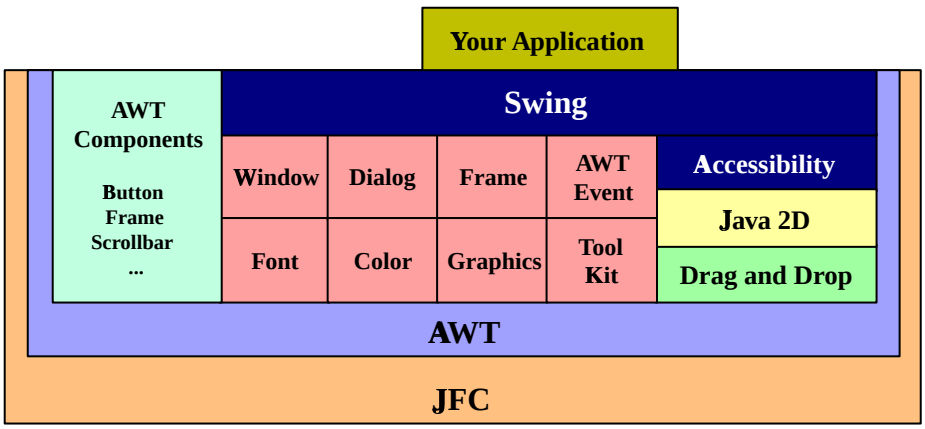


Figure 3.7: The different parts of the Java Foundation Classes.

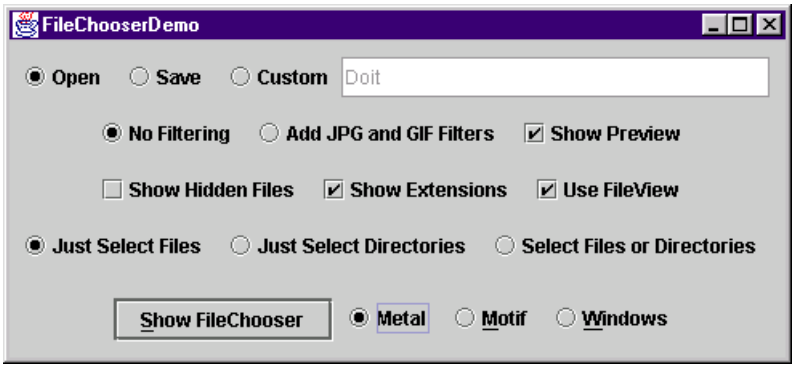


Figure 3.8: A window using Java look-and-feel.

- **Java 2D API**
  The Java 2D API is used for advanced drawing purposes. The next section is going to describe the Java 2D API in detail.

- **Drag and Drop API**
  The Drag and Drop API provides standardized mechanisms to exchange data between a developed application and other, native or Java, applications.

Pluggable Look-and-Feel

What is the look-and-feel of a graphical user interface? This question can easily be answered if user interfaces of different operating systems are compared. Users of Microsoft Windows know how a textfield, a button or any other component looks alike and how they have to be used. The same is valid for a user of a UNIX operating system. If a user of Microsoft Windows has to work with the user interface of an application under the UNIX operating system the components will look and behave different. Thus the look-and-feel of a user interface is how components look alike and how they behave on user actions.

Applications using Swing have a pluggable look-and-feel. An application uses a look-and-feel module that is responsible for the look and behaviour of its components. The look-and-feel module is selected during initialization of the application but can also be exchanged during runtime. All Swing components are designed for a pluggable look-and-feel. The ability of the components to use a pluggable look-and-feel is only possible because they are lightweighted and have no native counterpart. If a Swing component would rely on a native component it would neither be possible to change the look of a component nor to change the behaviour of it.

Therefore the Swing components do not have to look like their theoretical native counterparts. There are several look-and-feel modules that come with the Swing package. For most operating systems a native look-and-feel is available. This makes sense because users might want to work with the look-and-feel they already know. The default look-and-feel that is used if no other one is explicitely selected is the Java look-and-feel. The Java look-and-feel is a new look-and-feel, developed by Sun Microsystems. A programmer can also create custom look-and-feels and there are already many custom look-and-feel modules available.

Figure 3.8 is a screenshot of a window using the Java look-and-feel. Figure 3.9 is the same window but the Windows look-and-feel is used.

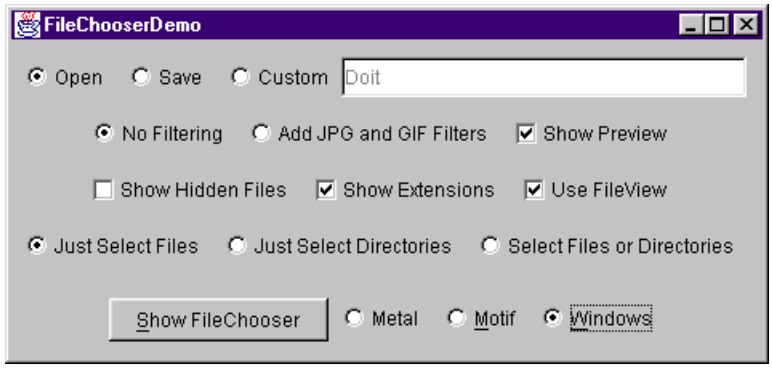


Figure 3.9: A window using Windows look-and-feel.

Lightweight Components

In the last few sections the name lightweight component has very often been used and it has been said that lightweight components are an advantage of the Swing library. So what are lightweight components? The explanation is very simple.

A lightweight component is a component written in 100% pure Java and it does not use any native methods for its operations. In contrast to a lightweight component, heavyweight components use native methods and are therefore not platform-independent. In the AWT every component has a native peer component and thus the functionality of an AWT component is limited by the functionality of the native component. Lightweight components do not have such limitations. Therefore functionality like a pluggable look-and-feel is possible. As it is possible to mix lightweight components and heavyweight components within one Java application this does not mean any limitations.

Seperable Model Architecture

The components of the Swing library use a modified version of the model-view-controller design pattern (MCV) [GHJ+95]. The MVC architecture divides a component into three collaborating parts. The parts are the model, the view and (what a surprise) the controller.

- **Model**
  The model holds the data items that are used by the component.

- **View**
  The view is responsible to draw the component and that the visualised items always correspond to the items in the model.

- **Controller**
  The controller reacts on interactions of the user with the component.

If the model changes, it informs the view that a change occured. Then the view is responsible to refresh the visualisation of the component. This technique has the advantage, that more than one view can be bound to one model. If the model changes, the views are automatically changed, too.

The Swing components use a somehow different version of the MVC architecture. As Mark Andrews wrote in [Andb]:

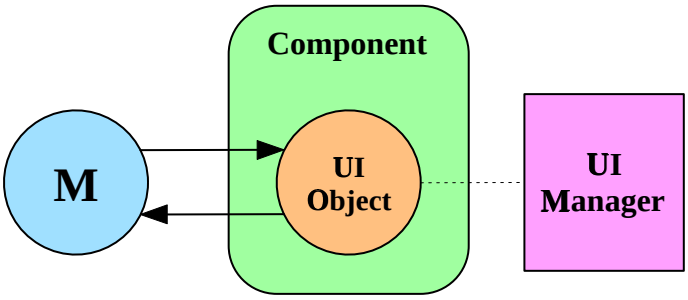


Figure 3.10: The separable model design of Swing.

> "Classic MVC architecture is simple and elegant, and the Swing team started designing the Swing component set, they experimented with using a pure MVC design. But they soon discovered that classic MVC architecture didnt work very well in real-world Swing application. So they wound up basing Swing architecture on a modified adaptation of the traditional MVC design. To implement Swing components, the Swing team created a modified MVC design. The component design that the Swing team eventually settled on is sometimes referred to a separable model architecture.
>
> In Swing's separable model design, the model part of a component is treated as a separate element, just as the MVC design does. But Swing collapses the view and controller parts of each component into a single UI (user-interface) object. The result is an architecture that looks like Figure 3.10.
>
> Swings separable model architecture was developed because the creators of Swing found that the traditional MVC design didn't work well in practical terms in Swing-style components. Why not? Because the view and controller parts of a traditional MVC-based component require a tight coupling that is sometimes difficult to achieve in practical terms. For example, traditional MVC architecture makes it very hard to create a generic controller that doesn't know at design time what kind of view will eventually be used to display it."

The UIManager in Figure 3.10 is responsible to handle the look-and-feel of the given component.

3.4 Java 2D

The Java 2D API [Knu99] [Java] is an enhancement to the Abstract Window Toolkit (AWT). It was designed to improve the 2D graphics and print capabilities of Java programs. Although the AWT provides drawing methods, they do not really fulfil the needs of today's programmers. As the Java 2D API is an enhancement to the AWT the drawing methods of the AWT can still be used. In fact the Java 2D API is closely integrated with the AWT.

The Java 2D API provides methods to:

- draw lines with different thickness, colours, endcap styles, join styles and stroke styles.
- fill arbitrary shapes with solid colour, textures or gradients.
- apply transformations to objects
- perform filter operations on images.

- draw text with different fonts and styles.

The Rendering Mechanism

The AWT drawing system calls the *update()* method of a component to tell it that it should draw itself. As the Java 2D API is integrated with the AWT, the same drawing system is used. In the AWT a *Graphics* object is used to represent the drawing context. The Java 2D API uses a *Graphics2D* object for this purpose. The class *Graphics2D* is a subclass of *Graphics*. *Graphics2D* contains methods to access the enhanced drawing capabilities of Java 2D. A simple *paint()* method that uses a *Graphics2D* object could look like the following example:

```
public void paint(Graphics g) {
   ...
   Graphics2D g2D = (Graphics2D) g;
   ...
}
```

A programmer simple typecasts the *Graphics* object to *Graphics2D* and has access to the advanced Java 2D methods yet.

Thus the *Graphics2D* object provides methods to gain access to the advanced drawing capabilities of Java 2D. It also holds a set of state attributes that are used to define the way how drawing and rendering operations are performed. The set of state attributes is called the *Graphics2D* rendering context. This rendering context consists of the following attributes:

- **The stroke attribute**
  It is used to define the drawing styles of lines. The drawing styles that can be defined are the line thickness, the dashing pattern, the join style and the endcap style.
- **The paint attribute**
  This attribute defines how shapes are filled with solid colours, gradients or textures.
- **The composite style**
  The composite style can be used to group several objects into one object.
- **The font**
  This attribute represents the font that is being used if text is drawn.
- **The transformation**
  This attribute can be used to perform affine transformations. For example objects can be rotated, scaled and translated.
- **The clipping path**
  The clipping path defines which parts of the drawn objects are visible to the user.
- **The rendering hints**
  With this attribute, the programmer can specify preferences in the trade-offs between speed and quality of the drawings, e.g. it can be specified if antialiasing should be used or not.

The methods used to set the attributes of the rendering style are part of the *Graphics2D* object. The names of these methods are:

- *setStroke()*

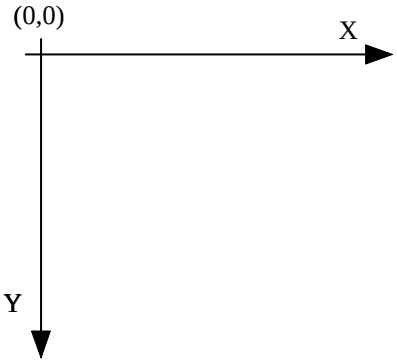


Figure 3.11: The user space coordinate system.

- *setPaint()*
- *setComposite()*
- *setFont()*
- *setTransform()*
- *setClip()*
- *setRenderingHints()*

As the *Graphics2D* class enhances the *Graphics* class, there are some new methods that can be used for drawing. These methods are:

- *draw()*
- *fill()*
- *drawString()*
- *drawImage()*

Coordinate Systems

The Java 2D API distinguishes between two different coordinate systems:

- User space
- Device space

The user space is a logical coordinate system. Because of the user space, which is device-independent, the programmer does not has to write different programs or methods for different devices. The origin of the user space coordinate system is in the upper left corner of the drawing area of a component. The values of the x-coordinates increase to the right and the values of the y-coordinates increase downwards. Figure 3.11 illustrates the user space and the two axes of this coordinate system.

The device space is a device-dependent coordinate system. Each device has its own coordinate system, e.g. a screen has another coordinate system than a printer. The mapping between user space

and device space is done automatically during rendering. Transformations that are applied to objects are also computed during the conversion of the coordinates from user space to device space.

The Java 2D API does the conversion from user space to device space automatically. Three levels of configuration information are used to accomplish this and these levels are encapsulated in three classes:

- **GraphicsEnvironment**
  This object is used to provide a collection of rendering devices that can be used by a Java application on the actual platform.

- **GraphicsDevice**
  Each rendering device in the collection of the *GraphicsEnvironment* object is represented by a *GraphicsDevice* object.

- **GraphicsConfiguration**
  Every possible configuration of a *GraphicsDevice* is stored in a separate *GraphicsConfiguration* object.

Drawing Shapes

With the Java 2D API virtually any geometric shape can be rendered. To do this the *draw()* and *fill()* methods of *Graphics2D* are used. *draw()* is used to draw lines, arcs etc. and *fill()* is used to draw filled shapes. To define the shapes that should be rendered the java.awt.geom package defines common graphics primitives. The available graphics primitives are:

- Arc2D
- Area
- CubicCurce2D
- Dimension2D
- Ellipse2D
- GeneralPath
- Line2D
- Point2D
- QuadCurve2D
- Rectangle2D
- RectangularShape
- RoundRectangle2D

Transforming Objects

The rendering context of the *Graphics2D* object can be modified in a way that a transformation is applied to the drawn objects. Possible transformations are translations, rotations, scaling and shearing operations. All used transformations are affine transformations. An affine transformation is a transformation in which straight lines are always transformed into straight lines and parallel lines into parallel lines. An affine transformation can simply be performed by a matrix multiplication of the coordinates of an object with a transformation matrix.

The attribute of the rendering context for the transformation is an instance of the *AffineTransform* class. There are several possibilities to change the transform attribute. One possibility would be to call the *setTransform()* method of the *Graphics2D* object directly. Another possibility is to concatenate the active transformation with another transformation. The *Graphics2D* object has some methods to directly concatenate the current transformation with a new one. These methods are:

- *rotate()*
- *scale()*
- *shear()*
- *translate()*

If the programmer wants to define own *AffineTransform* objects, the *AffineTransform* class provides several methods that make this easy. The following methods return *AffineTransform* objects that can be used:

- *getRotateInstance()*
- *getScaleInstance()*
- *getShearInstance()*
- *getTranslateInstance()*

Rendering Quality

The rendering quality of drawings has a big influence on the speed of the rendering process. If the rendering quality has to be high the rendering speed gets slow and the other way round. On a high-performance graphical workstation this may be no problem but using an ordinary computer the speed may be slow. Therefore the rendering quality can be adjusted.

This is accomplished by setting the rendering hints attribute of the *Graphics2D* object. Examples of rendering hints are antialiasing, alpha interpolation and dithering. It is possible that not all rendering hints are supported by the platform the program runs on. Therefore it is not guaranteed by the Java 2D API that each rendering hint is going to be used on every platform.

The rendering hints attribute can be set in two different ways. Either a *RenderingHints* object is instantiated and passed over to the *Graphics2D* object using its *setRenderingHints()* method, or the *setRenderingHint()* method of *Graphics2D* can be used.

3.5 Java 3D

The Java 3D API [Javb] is a three-dimensional, platform-independent graphics library developed by Sun Microsystems. It is written in 100% pure Java using object-oriented techniques and it draws its ideas from existing graphics libraries and from new technologies. As noted in [Javb]:

> "The Java 3D API is an application programming interface used for writing three-dimensional graphics applications and applets. It gives the developers high-level constructs for creating and manipulating 3D geometry and for constructing the structures used in rendering that geometry. Application developers can describe very large virtual worlds using these constructs, which provide Java 3D with enough information to render these worlds efficiently.
>
> Java 3D delivers Java's "write once, run anywhere" benefit to developers of 3D graphics applications. Java 3D is part of the JavaMedia suite of APIs, making it available on a wide range of platforms. It also integrates well with the Internet because applications and applets written using the Java 3D API have access to the entire set of Java classes.
>
> The Java 3D API draws its ideas from existing graphics APIs and from new technologies. Java 3D's low-level graphics constructs synthesize the best ideas found in low-level APIs such as Direct3D, OpenGL, QuickDraw3D, and XGL. Similarly, its higher-level constructs synthesize the best ideas found in several scene graph-based systems. Java 3D introduces some concepts not commonly considered part of the graphics environment, such as 3D spatial sound. Java 3D's sound capabilities help to provide a more immersive experience for the user."

The Java 3D API was designed to provide a sophisticated easy to use three-dimensional graphics library to the programmers and to deliver the highest level of performance to application users. Since the rendering of three-dimensional scenes is a complex operation, poor implementation or design of the graphics library leads to slow runtime behaviour of developed applications. Therefore, high performance was always the main design goal of the Java 3D API developers. The most important Java 3D goals are to:

- deliver the highest level of performance to application users.
- provide a rich set of features for creating interesting 3D worlds to the programmers.
- provide a high-level object-oriented programming paradigm that enables developers to develop sophisticated applications.

The Scene Graph Programming Model

The developers of the Java 3D API developed a scene graph-based programming model which is simple to use and flexible enough to build complex three-dimensional scenes. An advantage of this programming model is that the programmer can concentrate on the objects in the scene and the combination of these objects instead of thinking about triangles which build the objects in the scene or about rendering code for displaying the scene. The graphics elements in the scene are separate objects and can be manipulated by the predefined accessor, mutator, and node-linking methods. The separate objects are connected together in a treelike sructure called the scene graph. The scene graph is a description of the entire scene and stores all graphical elements that are within the given scene. It is build by the application during runtime. In Java 3D the entire scene is called the virtual universe. Every virtual universe has its own scene graph. The scene graph contains the necessary information to render the given scene from any particular point of view (see Figure 3.12).

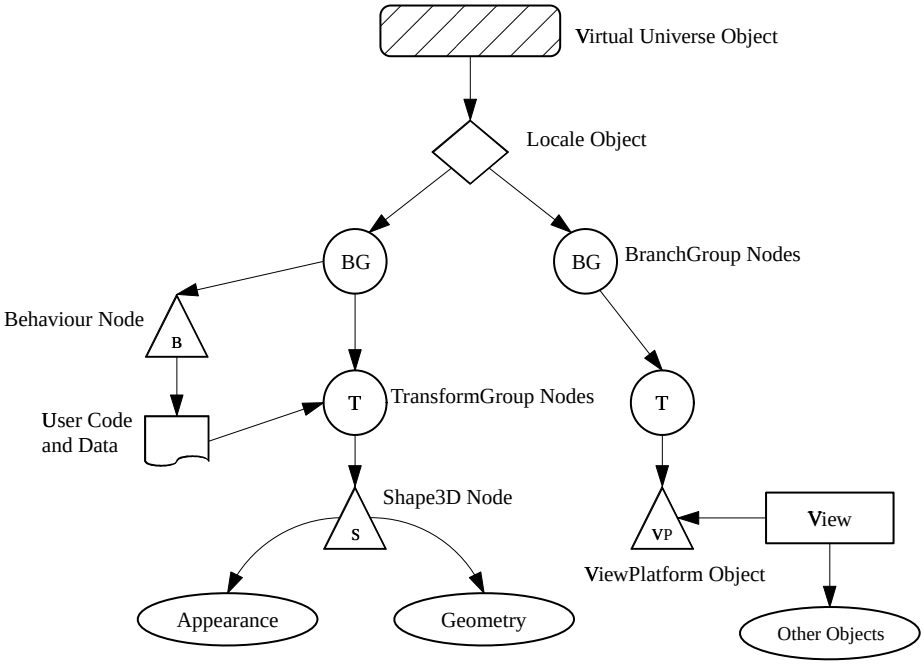


Figure 3.12: An example of a scene graph.

The Rendering Modes

The Java 3D API provides three different modes to render the given virtual universe: immediate mode, retained mode, and compiled-retained mode. The difference between these modes is the possibility to optimise the execution of the rendering of the actual scene. Each successive mode provides more freedom for optimisations. The three modes are in more detail:

- **Immediate Mode**
  Immediate mode provides the lowest level for optimisations. The application does not build a scene graph but provides a Java 3D draw method with all points, lines, and triangles which should be displayed in the selected scene. These graphical objects are then rendered by the Java 3D renderer.

- **Retained Mode**
  Within retained mode the application has to build a scene graph which describes the complete scene. To optimise the execution of the rendering the application has to specify which elements of the scene may change during runtime.

- **Compiled-Retained Mode**
  Compiled-retained mode provides the highest level for optimisations at the scene graph level. Like retained mode, the application has to build a scene graph and has to specify which elements of the scene may change. To optimise the execution of the rendering process some parts of the scene graph can be compiled into a Java 3D internal format.

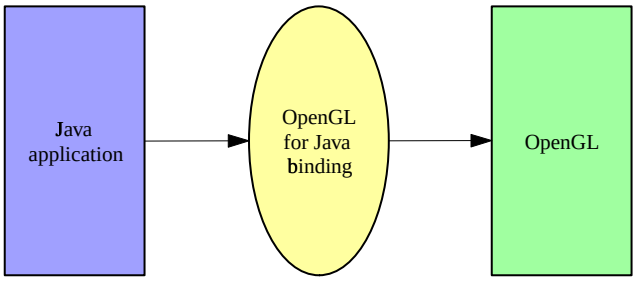


Figure 3.13: The binding between an application written in Java and the OpenGL graphics library.

3.6 Java and OpenGL

OpenGL [NDW93] [Ope], introduced in 1992, is the most widely adapted graphics library for developing interactive, portable two-dimensional and three-dimensional graphics applications. The high-quality, high-performance graphical capabilities of OpenGL provides a broad set of rendering, texture mapping, and other powerful visualisation functions. OpenGL has been ported to all popular platforms and enables developers to develop portable graphical applications.

Since Java became a popular programming language, bindings between the platform-independent Java programming language and the platform-dependent OpenGL library emerged. An example of such a binding between Java and OpenGL is Jogl [Jog]. OpenGL bindings for Java try to provide a complete set of Java bindings to the OpenGL graphics library. As Figure 3.13 shows forwards the OpenGL for Java binding the graphical methods from the Java application to native methods of the OpenGL graphics library.

Chapter 4

Information Pyramids

4.1 Introduction

The visualisation of information is an important part of the field of human computer interaction. Modern computer systems have to deal with huge amounts of information and this information must somehow be visualised. The human brain is not capable of storing large amounts of information in its short term memory [Mil56]. Therefore the designers of user interfaces have to find ways to visualise information so that the user can deal with it. As hierarchies are very often used to structure information, the intuitive visualisation of hierarchies is a very important question in human computer interaction. Chapter 2 gave a brief introduction to some of the numerous techniques to visualise hierarchies. As it is easy to visualise small, compact hierarchies, most of the described techniques consider the question of visualising very large hierarchies. Although there are existing techniques to visualise large hierarchies, it is interesting that even today a tree representation is most often used.

Information Pyramids, as described in [AWP97] [Wol98], are a new approach for visualising and manipulating very large hierarchies. The main design goal of the Information Pyramids technique was to provide a simple and intuitive method to deal with large hierarchies. Hierarchies are displayed as pyramidical buildings in a three-dimensional landscape (see Figure 4.1). As mentioned earlier small hierarchies are easy to visualise. Information Pyramids were designed to efficiently visualise even very large hierarchies. Hierarchical information structures contain two kinds of information. First there is the content information that is associated with each node. Second there is the structural information that is associated with the hierarchy. Information Pyramids are capable of visualising both the content information as well as the structural information. In order to efficiently use the visualisation of a hierarchy it is necessary that a user can focus on different parts of the hierarchy. While doing this, the global context must not be lost. The Information Pyramids technique provides mechanisms to focus on particular parts of the hierarchy without losing the global context.

Information Pyramids have some similarities to the File System Navigator and the Tree-Maps described in Chapter 2. Like the File System Navigator, Information Pyramids use a three-dimensional landscape to visualise a hierarchy. The advantage is that there is better usage of the third dimension. Unlike the File System Navigator, which displays only content information as three-dimensional objects, Information Pyramids visualise the content information and the structural information in three dimensions. The similarity to Tree-Maps is that the available display space for children depends on the position and size of their parent (see Figure 4.2). Children are not visualised within the boundary of their parents but on top of them.

Information Pyramids use the three-dimensional space to display a hierarchy. There are several reasons why 3D graphics were chosen. One reason was that, compared to two-dimensional visuali-

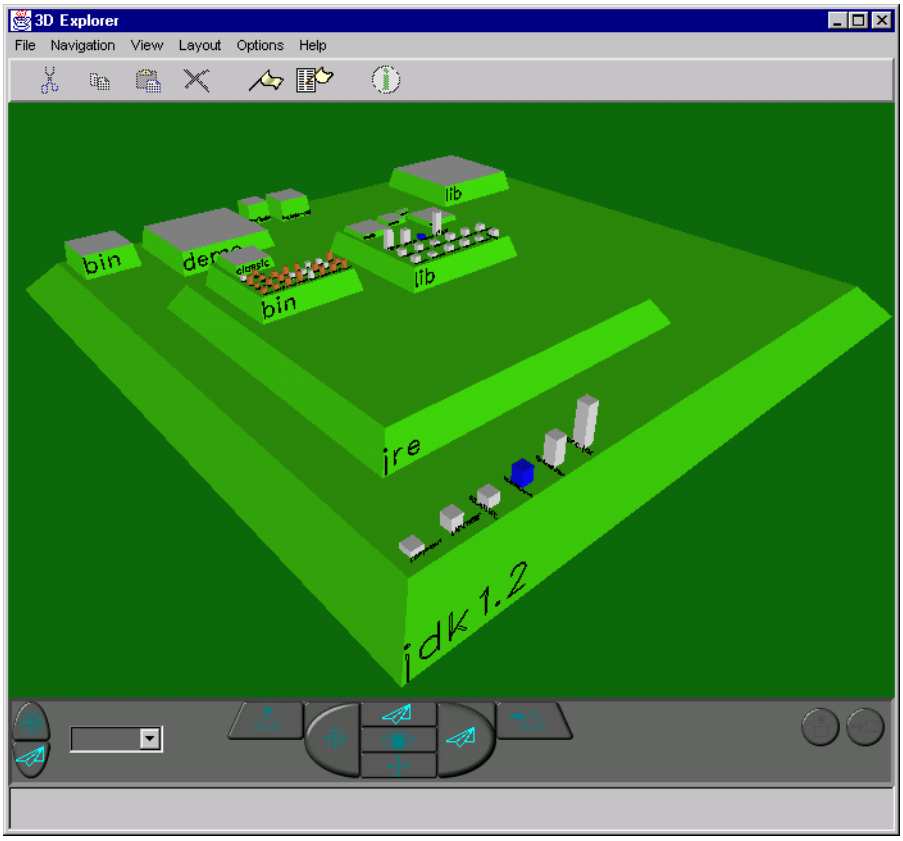


Figure 4.1: The main window of the 3D Explorer application.

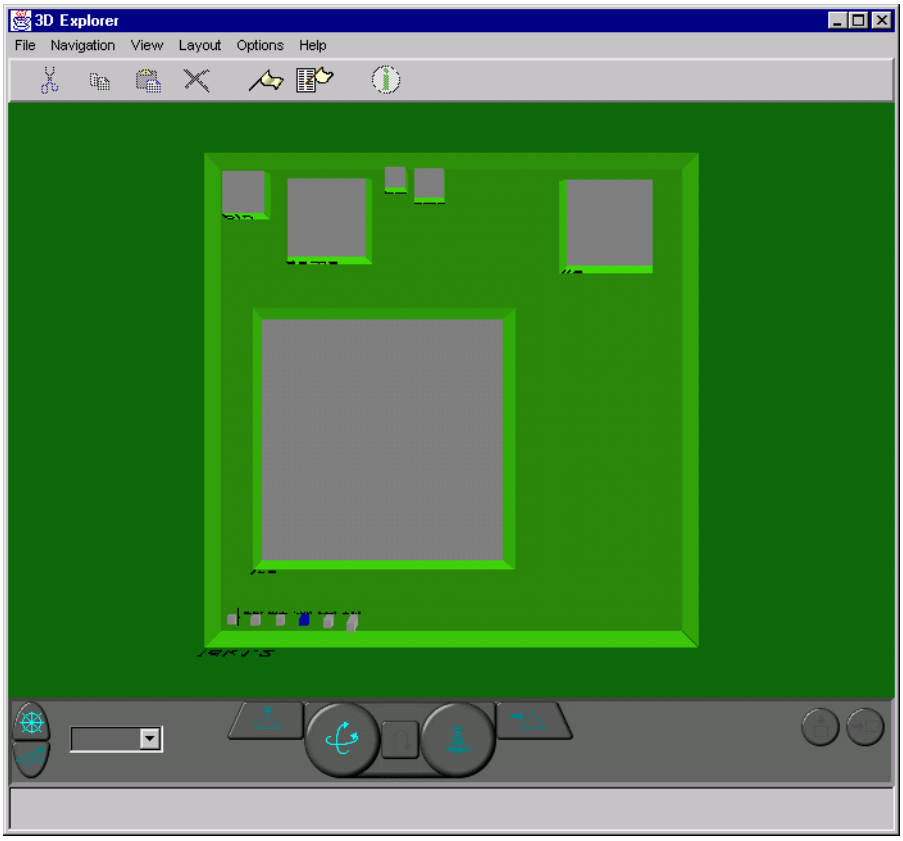


Figure 4.2: The viewpoint above the pyramid looking down.

sations, the usage of the available display space is much better. As the hierarchy can be displayed as a whole the user gets a better overview of the structure of the hierarchy. Another advantage is that it is easier to display not only the hierarchy but also other type of content information, e.g. the size of a file on a hard disk. Three-dimensional visualisations are more intuitive for the user and do not seem to be as abstract as some two-dimensional approaches.

The remaining part of this chapter is going to describe the Information Pyramids technique in more detail. The example hierarchies are directory trees of a hard disk.

4.2 The Pyramid

The Information Pyramids technique can be used to draw hierarchies as pyramidical objects. The main advantage is that even very large hierarchies can be compactly visualised as a landscape. This section describes the layout of the different nodes within the pyramid. The name Information Pyramids comes from the overall visual impression of the visualisation which is that of a pyramid.

The root of the hierarchy is positioned at the bottom of the pyramid. Descending the hierarchy causes the pyramid to grow upwards. The root node is represented by a square-shaped plateau. The children of the root node are positioned atop of this plateau. If a child is an inner node (an empty directory of a file system is also an inner node!), it is also represented as a plateau. Leaf nodes are represented by smaller three-dimensional objects. Therefore leaf nodes can easily be distinguished from inner nodes. This supports the user to gain a better overview of the hierarchy and to find information faster. Inner nodes are placed starting from the top left corner of their parent's plateau while leaf nodes are arranged at the bottom left corner of it. Starting at the root node and going down the hierarchy the same layout method is used recursively.

The size of the base plateau is the same for all hierarchies. That means it is independent of the size of the hierarchy. As only the height of the pyramid grows, the total size of it depends on the number of levels in the hierarchy only. With this technique two hierarchies having the same number of levels but a completely different number of nodes have approximately the same size in the three-dimensional space.

Going down the hierarchy the representations of the nodes become smaller and smaller. The size of children relative to the size of their parents is more or less always the same. As the number of children is not the same for all nodes, the real size of a node is calculated such that the resulting visualisation is visually appealing. As nodes become smaller if they are deeper in the hierarchy, a zooming mechanism is provided.

There are many possibilities for arranging the children of a node atop their base plateau. The order of their arrangement can be used to visualise some other properties of their information content. For example, the children could be sorted and the arrangement could visualise this sorting. Another possibility to visualise more information than only the hierarchy is to use different colours and sizes of the nodes. Different colours could visualise different types of information. A node that has more information content than another node could be visualised larger. More information content could for example mean that a node has more children than another node. There are two advantages if a node with more information content is visualised larger than a node with less one. First the user easily recognises the different sizes and second there is more space available to display the children of a larger node.

4.3 Visualisation of an Inner Node

The root node and every inner node of a hierarchy is represented by a square-shaped plateau. Inner nodes are positioned atop the plateau of their parents, starting from the top left corner of the plateau. The area on the top of a plateau which is available for the placement of inner node children depends on the number of children that are leaf nodes. If there exist inner nodes, the maximum area which is available for leaf nodes is the bottom half of the complete area. If the leaf nodes do not fill this half area completely the remaining part is also available for the placement of inner nodes. That means the available area for inner nodes is the complete top area of the plateau of their parent minus the area available for leaf nodes that are their siblings.

The area which is available for the placement of inner nodes must be divided in a reasonable way to contain all inner node children. It is not necessarily useful to represent all those nodes by equally sized plateaus. The size of a plateau can be used to visualise more information than just the structure of the hierarchy. If plateaus with different sizes should be used, every node has to be assigned a value that represents its size. The available information which could be used for this purpose is for example:

- the number of children,
- the number of all leaf nodes of a subhierarchy,
- or the size of a directory if a directory tree is visualised.

There are some advantages if different sizes are used for the plateaus. The user gets a better overview of the structure of the complete hierarchy if different sizes are used. The parts of the hierarchy that contain more information are visualised larger than those where only some information can be found. It is therefore easier for the user to find information. Of course, it is only possible to compare the absolute sizes of siblings because the size of a plateau is calculated locally and depends on the size of its base plateau. It is not guaranteed, that those parts of the hierarchy contain the needed information but there is another advantage of this approach. The more nodes are within one part of the hierarchy the more space is needed to draw them. If equally sized plateaus would be used a node with just one child would have the same size as a node with many children. One design goal of the Information Pyramids was to maximise the usage of the available display space. Different sizes make this possible.

To lay out the plateaus different algorithms can be used. The available area for the children of a node is either a square, if no children of a node are leaf nodes, or a rectangle. The problem of placing the square-shaped plateaus into a rectangular area is non-trivial. The layout of the plateaus can be used to visualise some kind of sorting of the children. A layout algorithm should fulfil the following criteria:

- As even very large hierarchies should be visualised the performance of the algorithm should be sufficient enough to lay out the hierarchies in a reasonable time.
- The available display space should be used as optimal as possible.
- The layout of the plateaus should help the user to find information faster.

4.4 Visualisation of a Leaf Node

All the leaf nodes of the hierarchy are visualised as small objects. The screenshots in this chapter show the leaf nodes as small square pedestals but other visualisations are also possible. The reason

why leaf nodes have another visual appearance than inner nodes is that a user can easily distinguish them.

The size of a leaf node is calculated relative to the size of its parent. Inner nodes are visualised as square-shaped plateaus. The size of a plateau depends on its position in the hierarchy. The deeper an inner node is in the hierarchy the smaller is its visual representation. The same is true for leaf nodes. The size of a leaf node is always calculated in a way that it still can be recognised and it still looks good relative to the size of its base plateau. The space between two neighbouring leaf nodes is selected in a way that they can easily be distinguished.

If there are no siblings that are inner nodes the whole top area of the plateau can be used to arrange the leaf nodes. If there exists at least one sibling that is an inner node only half of this area is going to be used. The number of leaf node siblings is not known in advance. Therefore, their size has to be calculated in a way that all of them can be positioned in the available area. There are many possibilities to arrange the leaf nodes in the available area. The simplest way is to arrange the in a matrix. Another possibility would be to sort the leaf nodes and arrange them in rows and columns.

The representation of the leaf nodes can be used to visualise different attributes of them. This should help the user to find information and to get a better overview of the hierarchy during navigation. The different attributes of the leaf nodes that can be displayed depend on the content of the visualised hierarchy. Figure 4.3 shows a screenshot of a pyramid that uses colour coding to visualise different file types. Attributes of the small objects that could be used to visualise attributes of the leaf nodes are:

- The colour
- The height or absolute size
- Usage of an icon instead of a simple object

4.5 Navigation

Not only the visual presentation is important, but also the navigation and exploration of an information hierarchy is very important. The compact three-dimensional visualisation of the Information Pyramids technique enables the designer of a user interface to provide many different navigation methods. Figure 4.4 shows the screenshot of a pyramid where the user zoomed into the visualisation. As Josef Wolte noted in [Wol98]:

> "The main purpose of a visualisation is to facilitate the exploration of information. Visualisations therefore have to provide navigational tools. The 3D landscape of the Information Pyramids is very flexible, so many different navigation methods can be implemented. However, the input devices of an average PC are not designed for use with three-dimensional interface. So great efforts have to be made to design a user interface for the 3D visualisation, which can be easily used with the mouse and keyboard. Possible navigation aids include:
>
> - A navigation mode to move freely through 3D space. Such a navigation mode ensures the user gains an overview over the hierarchic structure.
> - The user should be able to easily navigate to the interesting information. This means providing mechanisms to move easily to child nodes or to the parent node, so the user navigates rapidly to the desired level of the hierarchy.

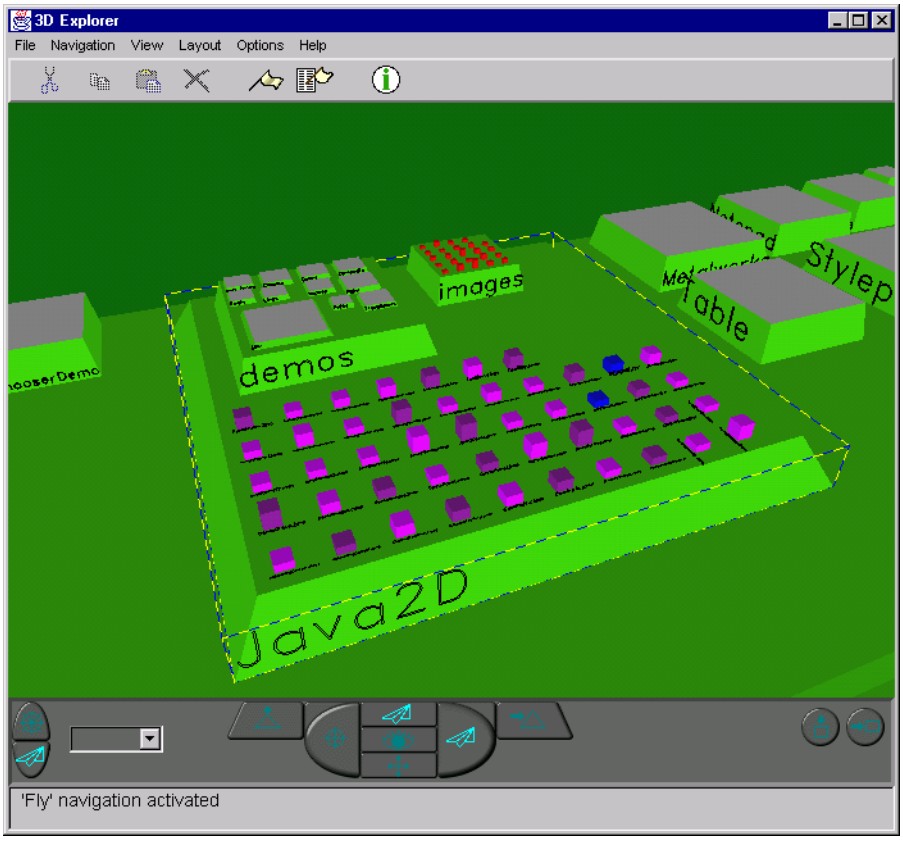


Figure 4.3: A screenshot of a pyramid that uses colour coding to visualise different file types.

- Special views of the landscape such as a top view should be reachable quickly. A top view displays the ground plan of a plateau. This should not only work for the root plateau, but also for every other plateau.
- Often used views of the hierarchy should be reached without much navigation. A simple mouse click or a key shortcut should be enough to move to such viewpoints of the hierarchy. Also, facilities to define and manage such viewpoints should be provided.
- Search functionality can be used to navigate quickly through the information space. The search results could be highlighted and some easy to use facility should move the user between these objects.
- A navigation history which tracks the navigation through 3D space. It would be useful to be able to undo or redo navigational actions."

During navigation the motion from one position to another should be smoothly animated. As the user builds contextual information during navigation simply switching from one position to another would distract the user. It is even possible that the user looses the global context completely and has to start exploring the information space again. It would also prevent the user from getting a good overview of the whole hierarchy. The speed of the animations has to be considered, too. If the animation is too fast it has the same negative effect than simply switching from one position to another. If it is too slow it is annoying to the user and prevents continuous work. A suggestion of human computer interface research is that an animation should take about one second [RCM93].

It is important that the user is able to recognise hierarchies or parts of hierarchies even if not seen for a long time. If the visualisations were not consistent over time, it would be impossible to get an overview of the hierarchical structure. This does not mean that different methods to lay out or sort nodes should not be implemented. It is even easier for the user to find specific information if different layout and sorting possibilities are provided.

It is not always useful to display the hierarchy as a whole, but sometimes preferable let the user decide which parts of the hierarchy should be displayed. The user interface should therefore provide navigational aids to expand or collapse different parts of the hierarchy. If the user expands a subtree of the hierarchy only several levels should be expanded and not the whole subtree. This helps the user to understand what happened and to get a better feeling for the structure of the hierarchy.

Sometimes the user finds a specific part of the hierarchy and decides that only this part is of further interest. Thus a pruning mechanism should be provided. The selected plateau is going to be the new base plateau of the visualisation. Only its children are displayed and all other parts of the hierarchy disappear.

4.6 3D Explorer

The 3D Explorer [Wol98] was the first application to implement Information Pyramids. It is completely written in Java but it uses OpenGL, a platform-dependent graphics library. The intention of the developers of the 3D Explorer was to develop an application that can be used as a "proof of concept" for the Information Pyramids technique. As the 3D Explorer has been developed in a period of time where not all of the possibilities of the Information Pyramids technique were known, not all of them were implemented. Nevertheless many very interesting experimental features and options have been implemented. The application has been used for usability testing purposes and many important features of the Information Pyramids have been found.

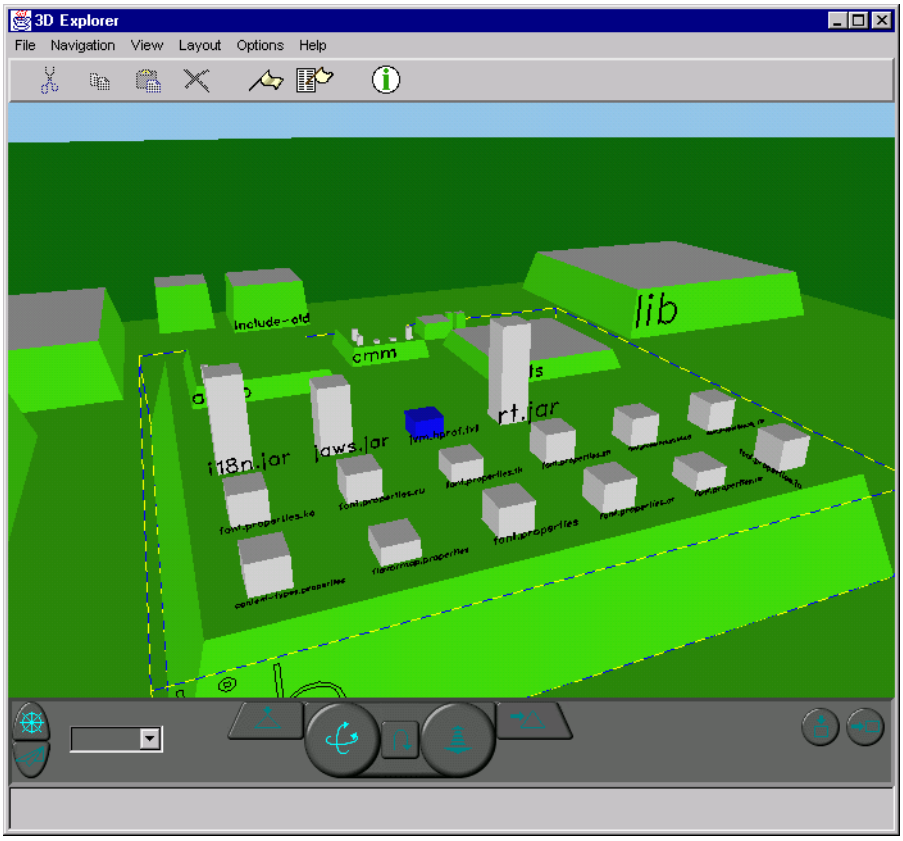


Figure 4.4: A screenshot of a pyramid where the user zoomed into the visualisation.

The 3D Explorer is a visual exploration tool for file systems. This has the advantage that not many prerequisites have to be fulfilled to run the program. The most important features that have been implemented are:

- The user can select between different types of layout algorithms. There exists a layout algorithm to lay out equally-sized plateaus and another one to lay out plateaus that have different sizes. The layout algorithms try to maximise the usage of the available display space.
- The visual representations of the files display some attributes of them. The colour is used to visualise the type of the file and the height is proportional to its size.
- The user can select between different types of sorting methods for the files and directories.
- There are sophisticated facilities to enable a simple navigation through the visualised landscape.
- Great efforts have been made to find values and algorithms to calculate the absolute sizes of the plateaus, that represent the inner nodes, and small objects, that represent the leaf nodes.
- The object-oriented approach of the application makes it easy to develop further versions that visualise hierarchies other than a file system.

4.7 3D Explorer for VRML

The 3D Explorer for VRML[1] was the second application that implemented the Information Pyramids technique. The programming language that has been used was Java. The main difference between the 3D Explorer and the 3D Explorer for VRML is that the version for VRML does not use OpenGL. Instead, a VRML browser is used to visualise the Information Pyramids. A standardized programming interface, the External Authoring Interface (EAI), is used for the communication between the 3D Explorer for Java application and the VRML browser (see Figure 4.6). The application produces VRML scene graphs [CB97] which are transported through the EAI to the VRML browser for display (see Figure 4.5).

All user interaction is handled by event listeners written in Java. The event listeners are informed by the VRML browser of user activities. Then the event listeners react to the user interaction and produce new scene graphs or parts of them to send to the VRML browser over the EAI. Thus the possibilities to react on user interaction depend on the implementation of the used VRML browser. This approach has the disadvantage that the visualisation and interaction with the user cannot be implemented very flexibly. By using a graphics library, such as Java 2D or Java 3D, it is possible both to write applications that produce more sophisticated visualisations and to make the user interface more convenient for the user.

[1] VRML stands for Virtual Reality Modelling Language

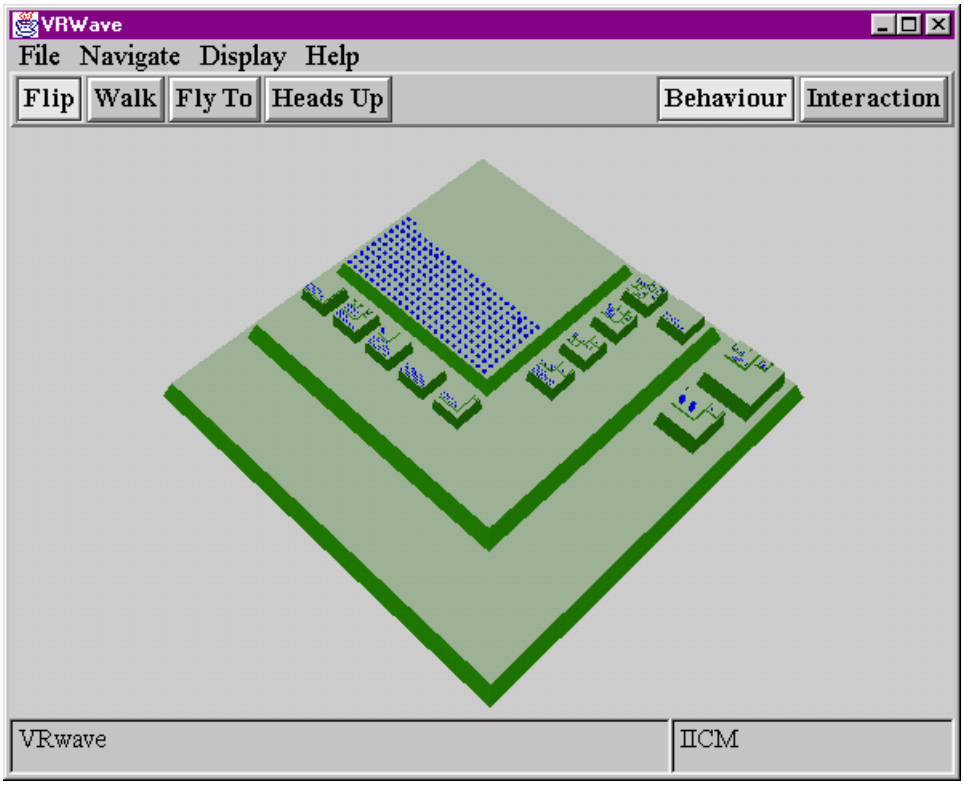


Figure 4.5: The VRwave VRML browser displaying the Information Pyramid produced by the 3D Explorer for VRML application.

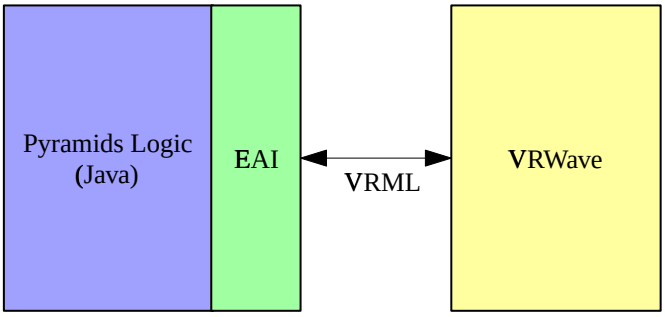


Figure 4.6: The 3DExplorer for Java application uses the EAI for the communication between the application and the VRwave VRML browser.

Chapter 5

The Java Pyramids Explorer

5.1 Introduction

The Information Pyramids technique seems to be a good solution to visualise hierarchies. Even very large hierarchies with thousands of nodes can be visualised compactly and manageable for the user. The first applications that used the Information Pyramids technique were the 3D Explorer and the 3D Explorer for VRML. They have been developed to be used as a proof of concept for this technique. In spite of the advantages of the Information Pyramids technique, these applications have still some disadvantages. The main problems are:

- The visualisation gives a good overview of the hierarchical structure but it is still not easy to find information fast and efficiently.
- Most users are not really familiar with navigation through three-dimensional landscapes. In spite of the navigational aids, they often loose the global context of their current position. This means also that it takes longer to get an overview of the whole hierarchy.
- There is still no good solution on how to display textual information within the visualisation of the hierarchy. Using different arrangements, sizes, shapes and colours many attributes of the hierarchical information can be displayed. Nevertheless, to find information, a textual description of the nodes is important to the user. The compact visualisation of the Information Pyramids makes it impossible to display a textual description of every node of the hierarchy.
- The 3D Explorer and 3D Explorer for VRML were developed using Java but their implementations are not 100% pure Java. 3D Explorer uses OpnGL and the 3D Explorer for VRML assumes the availability of a VRML browser. As 100% pure Java graphics libraries exist, it seems more portable to develop applications using these libraries.

The main purpose behind the Java Pyramids Explorer was to find solutions for these problems. The Java Pyramids Explorer (JPE) is an application which visualises the hierarchical structure of a file system. It was developed using 100% pure Java and the techniques described in Chapter 3. The Java Development Kit used is the JDK 1.2, also called Java 2. The Swing package and the Java 2D API are already included in this version of the Development Kit. In order to visualise hierarchies the Information Pyramids technique is used in combination with traditional approaches like a tree and a list. The directories of the file system are visualised through the Information Pyramid and the tree. The files contained in the selected directory are displayed in the list.

The decision which visualisation is used for navigation depends on the user only. The navigation within one visualisation method affects the other views, too. The three views are synchronised all the

time. There are several reasons why the usage of Information Pyramids together with a tree and a list are an advantage to the user, including:

- Users are familiar with working with trees and lists. Thus the user can decide to navigate through the hierarchy by using these views but has also all the advantages of the visualisation of an Information Pyramid. If it is needed, there is always the possibility to switch between navigation methods.

- A textual description of the nodes is visible in the tree and the list. Thus it is easier and faster to find the desired information.

- The Information Pyramid does not display the files of the file system but only the directory tree. Therefore there is more space available to display the hierarchical structure. To find files it is essential to see their names. As it is not possible to draw the names of the files in the Information Pyramid, the files are not drawn at all. Instead, a simple list is used for this purpose. If a user selects a directory in the Information Pyramid or the tree, the list displays the files contained in it.

Information Pyramids are a three-dimensional visualisation technique. The JPE does not produce a full three-dimensional but a simulated three-dimensional visualisation of the pyramids. Not all sides of the pyramid are drawn and the user cannot see the pyramid from below. Therefore, this simulated three-dimensional visualisation also limits the navigational possibilities for the user. This seems to be a disadvantage but the purpose is that the navigation can be simplified. It is harder to get lost in space and to loose the global context of the current position in the landscape.

The JPE uses the Java 2D API to display the pyramids. There are two reasons why a two-dimensional graphics library has been used instead of a three-dimensional one. First, the JPE produces only a simulated three-dimensional visualisation of the pyramid and even with a two-dimensional graphics library this can easily be accomplished. Second, the Java 2D API is already part of the JDK 1.2 and therefore no other package is needed to run the application.

5.2 The User Interface of the Application

The user interface of the JPE was designed to fulfil the needs of modern applications. To give the user easy access to the functionality of the application a menu bar and a tool bar are used. The user interface contains three different views to visualise the hierarchical information. Furthermore one panel is used to display additional attributes of selected nodes. Figure 5.1 displays the different parts of the application window. The purpose of each part of the visualisation is:

- **The Information Pyramid**
  One panel is used to visualise the directory tree of the selected file system as an Information Pyramid. Only the directories are displayed and not the directories and the files as a whole.

- **The Tree View**
  The tree view visualises the same directory tree as the Information Pyramid. The files are not displayed in the tree view either.

- **The Children List**
  Both the Information Pyramid and the tree view do not display the files of the file system. This is the reason why the list is used. It displays all children of a selected directory.

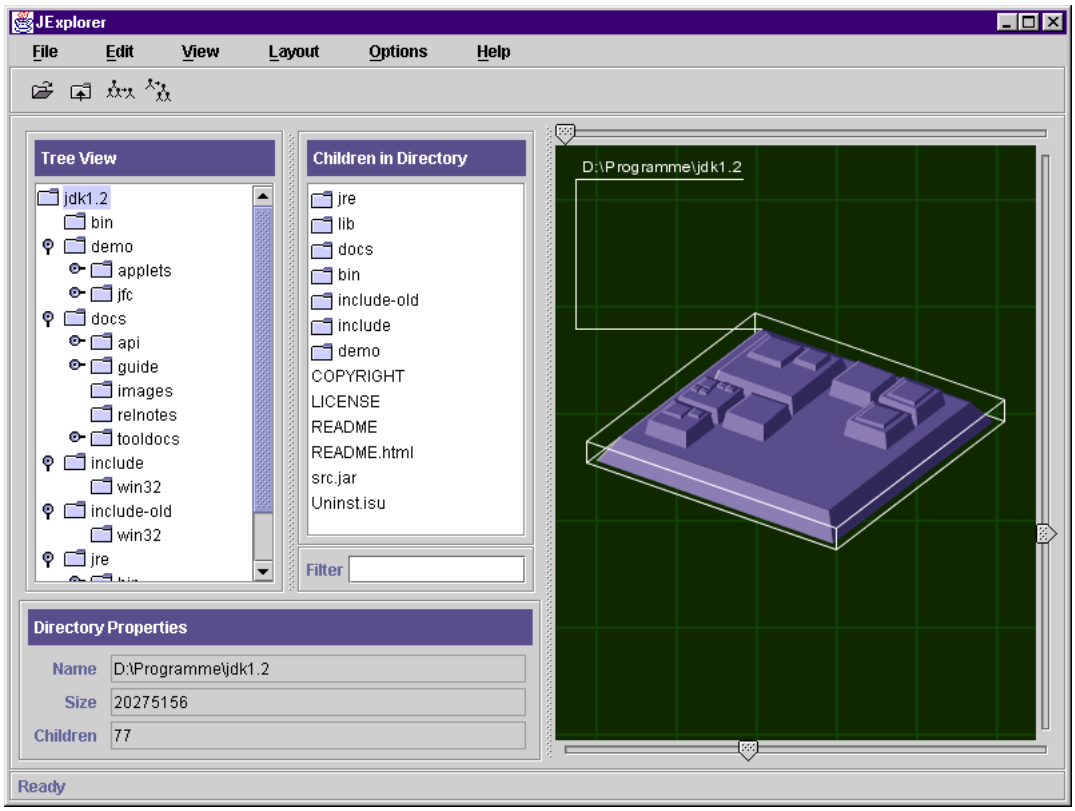


Figure 5.1: The main window of the JPE application.

- **The Properties Panel**
  The properties panel is used to display additional information such as the absolute path and the size of selected nodes.

The next sections describe these four panels in more detail.

5.3 The Information Pyramid

The Information Pyramid view (see Figure 5.2) visualises the directory tree of the selected file system. This view visualises only the directories and does not display the files, so that more space is available to display the hierarchical structure of the file system. To display the graphics of this view the Java 2D graphics library is used. The visualisation is a simulated three-dimensional model of the Information Pyramid. Thus the pyramid cannot be viewed from below. This visualisation technique simplifies the navigation through the landscape for the user.

To help the user in exploring the hierarchy and finding information some attributes of the pyramid can be adjusted. There are different ways to sort the directories before they are arranged and displayed. The user can select between the following sorting algorithms:

- **Sort by Name**
  The directories are sorted by the name of their path.

- **Sort by Size**

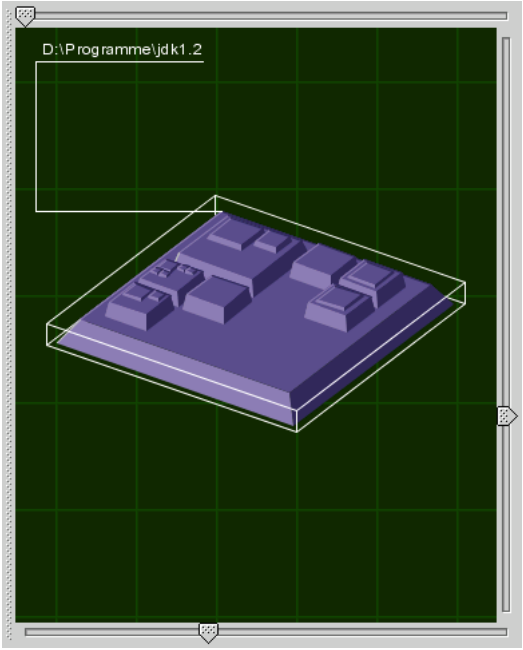


Figure 5.2: The pyramid view using the Information Pyramids technique.

The size of a directory is the total number of bytes occupied by its children. The children of a directory are the files and the subdirectories contained within it (recursively).

- **Sort by Number of Children**
  The number of children of a directory are the total number of files plus the number of children of the subdirectories (recursively).

The user can decide if the directories should be in ascending or descending order. The size and the number of children are only estimated values. The algorithm to compute these values typically looks a few levels into the hierarchy but not all levels. The reason for this estimation is to gain performance, because if a very large directory tree has to be visualised, the computation of the correct values could take too long. The number of levels can be defined by the user, the default value being 2 levels.

Another adjustable attribute of the visualisation is the layout of the plateaus in the pyramid. The object-oriented design of the JPE application makes it easy to develop different layout algorithms and to use them for the arrangement of the directories. By default, the JPE has two different layout managers which can be used:

- **Box Layout**
  The Box Layout layout manager displays the directories using equally-sized plateaus. The plateaus are arranged in rows and columns and the order in which they are drawn depends on the used sorting method. The arrangement of the directories starts in the upper left corner of the corresponding base plateau.

- **Size Sorted Layout**
  This layout manager computes different sizes for the plateaus. The arrangement of the plateaus depends on the sorting algorithm used. The Size Sorted Layout is described in more detail in Chapter 6.

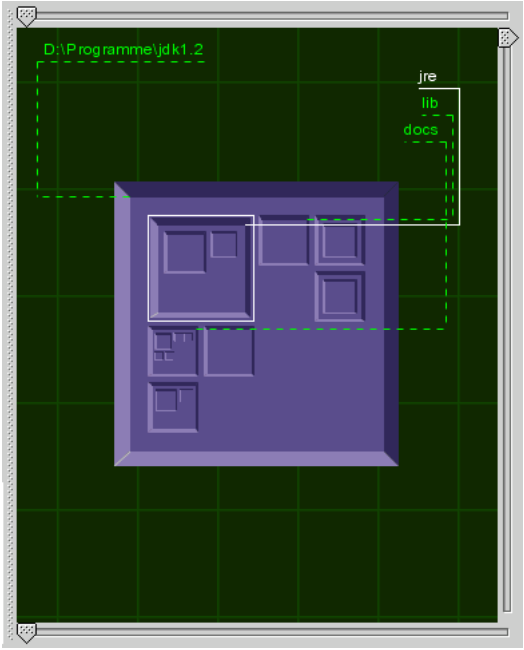


Figure 5.3: The pyramid view displays the names of some of the drawn directories.

During exploration it is important to display a textual description of the nodes. Therefore the names of the directories can be displayed if needed (see Figure 5.3). In the visualisation of the pyramid, there is too little space available to draw the names of all visible nodes. Thus only a textual description of some nodes, those seemingly important, are displayed. Of course, it is not easy to decide which nodes are important at the moment. Therefore the names of the selected node, its parent and two of its siblings are drawn. This seems to be a good compromise. Which siblings are taken depends on the selected node and the used sorting algorithm. If the selected node is the first one compared to the sorting order, the next two siblings are taken. If it is the last one, the two nodes before it are taken. Otherwise one node before it and one node after it is taken.

5.4 The Tree View

The tree view (see Figure 5.4) visualises the same hierarchical structure as the Information Pyramid view. The graphical component used for the tree is the *JTree* class of the Swing package. The tree view and the pyramid view are synchronised on user interaction. If a subdirectory of the hierarchy is expanded or collapsed in one view it is reflected in both the tree view and the pyramid view. A selection of a node leads to a selection in both views, too.

There are two main reasons why the tree view is used in the JPE application. Firstly, users are familiar with working with such outline visualisations of hierarchies. If a user does not want to use the navigational facilities of the pyramid it is still possible to use the tree view for navigation. Secondly, the textual description of the nodes are displayed in the tree view. It is not possible to display all these names in the pyramid view but in the tree view it is possible.

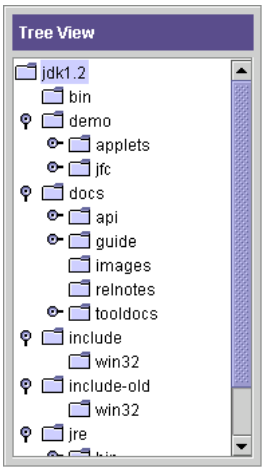


Figure 5.4: The tree view.

5.5 The Children List

If a user selects a directory in the pyramid view or the tree view the children list displays its children. Since both pyramid and tree views do not display the files of a directory, this is the only possibility for the user to find files. As Figure 5.5 shows, both the subdirectories and the files of a selected directory are displayed. To be able to distinguish the subdirectories from the files, an icon is drawn in front of the name of a directory.

The class that is used for the list component is the *JList* class of the Swing package. The list can be used for navigation through the hierarchy, too. If a subdirectory is opened in the children list, it is also opened in the pyramid view and the tree view. The files in the list can be sorted in different ways, either ascending or descending. The different sorting methods that can be used are:

- Sort by Name
- Sort by Size

The *Filter* field of the children list can be used to apply a filter for the files in the children list. The user can use the * wildcard to apply useful filters. Possible filters are:

```
filename    display only files that exactly match "filename"
*filename   display all files that end with "filename"
filename*   display all files that start with "filename"
*filename*  display all files that contain the word "filename"
```

5.6 The Properties Panel

The properties panel (see Figure 5.6) displays several attributes of a selected directory. The attributes that are displayed in the current version of the JPE are:

- **Name**
  The *Name* field displays the absolute path of the selected directory.

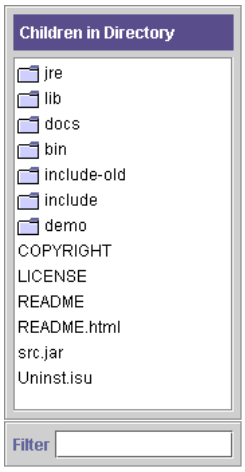


Figure 5.5: The children list, both the subdirectories and the files of the selected directory are displayed.

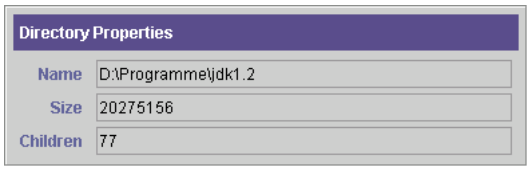


Figure 5.6: The properties panel.

- **Size**
  The *Size* field displays the size of the selected directory.
- **Children**
  The *Children* field displays the number of children of the selected directory.

5.7 Navigation

As the JPE application provides three different views of the hierarchically structured information, there are numerous ways to explore the information space. As most users are not familiar with working with three-dimensional landscapes, the tree view and the children list can be used like an ordinary tree-based file system browser, e.g. like the Windows Explorer, to explore the hierarchy. Nevertheless, the navigational possibilities of JPE go far beyond the possibilities of an ordinary tree-based browser.

The pyramid view of the JPE application provides the most important navigational facilities to the user. The simulated three-dimensional visualisation of the pyramid makes it easier to provide simple navigational aids to explore the three-dimensional landscape than a complete three-dimensional one. The user cannot move through the landscape as freely as in the 3D-Explorer applications. Instead the user can change the viewpoint within the visualisation by using the three sliders that are positioned at three sides of the pyramid view panel. These three sliders can be used to rotate the axes of the pyramid and to zoom into the visualisation. Figure 5.7 shows the axes of the scenery and labels the three sliders. The slider at the bottom of the panel is used to rotate the scenery around its Z-axis. The

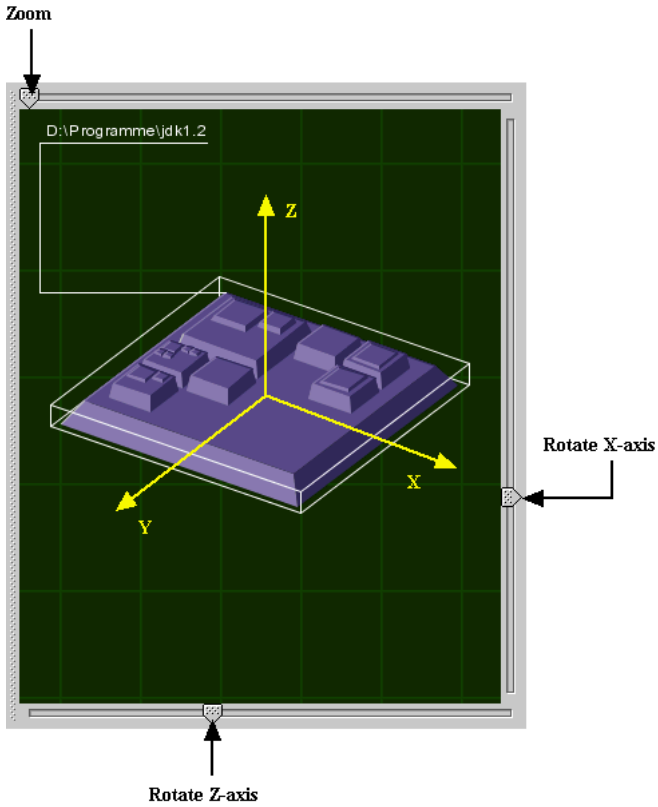


Figure 5.7: The axes of the pyramid in the pyramid view.

slider to the right of the panel rotates the scenery around its X-axis and the one at the top of the panel can be used to zoom to a selected node. The scenery can be rotated around the X-axis and the Z-axis between 0 and 90 degrees.

The user also has the possibility to select between three predefined viewpoints. These three viewpoints rotate the axes of the scenery to values that seem to be useful as starting points for exploration. The predefined viewpoints are:

- **Top View**
  The view from above the pyramid looking downwards can be used to get an overview of the different sizes of parts of the hierarchy.

- **Front View**
  This view is useful if a user wants to know how deep different parts of the visualisation are.

- **3D View**
  The 3D View gives a good overview of the whole visualised landscape.

Figure 5.8, Figure 5.9 and Figure 5.10 show these three different viewpoints for one scenery. To be more flexible, a user defined viewpoint can be selected, too. The values of the rotation angles are stored and can be recalled even if the user closes the application and starts it again.

The pyramid view also provides methods to select nodes and to hide or open parts of the hierarchy. The user can select a directory by clicking on it with the mouse. A white wireframe box is drawn around a selected plateau. A selected directory can be expanded or collapsed by clicking on it with

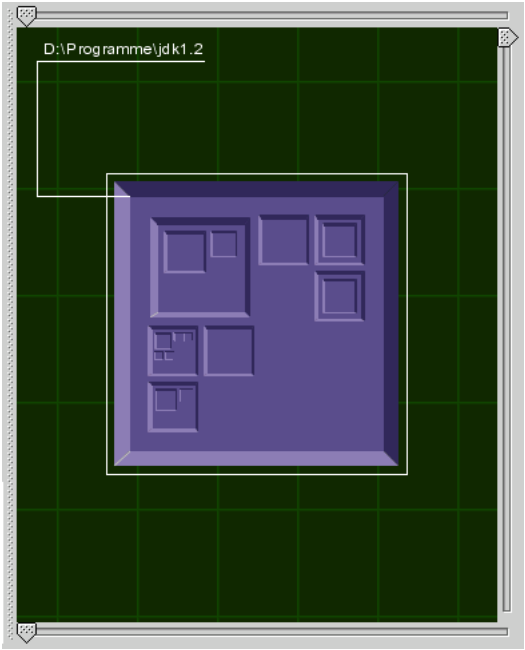


Figure 5.8: The top view.

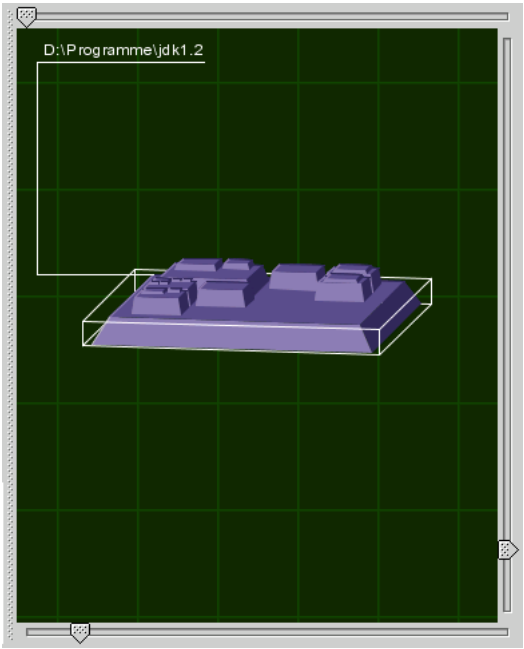


Figure 5.9: The front view.

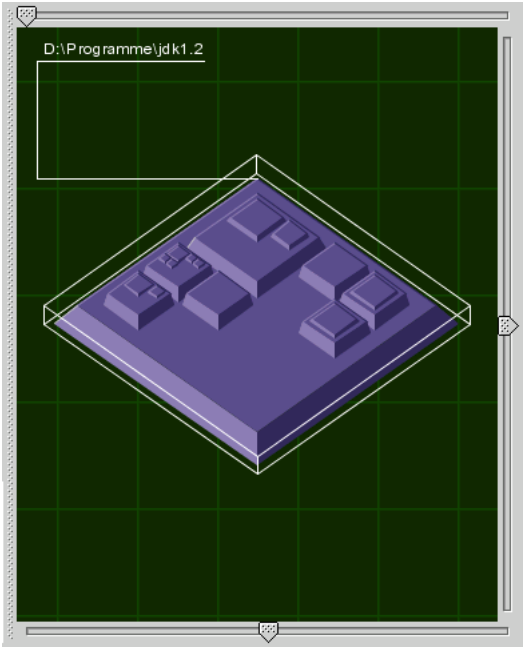


Figure 5.10: The 3D view.

the mouse. If the directory was already expanded, it is collapsed and its children are no longer drawn until it is expanded again. If the user expands a node, not only one level of children is opened but typically several levels. The number of levels expanded can be defined by the user.

There is also the possibility to select the parent of the currently selected directory. This navigational aid helps the user to go deep into the hierarchy and to go back again, level by level, if needed. There are two more navigational facilities that should be mentioned. Firstly, there is the possibility to expand all nodes that are deeper in the hierarchy than a selected node. This facility is called *Open All Children*. Secondly, the user can select a node and tell the application that the selected node should be the new root directory of the visualisation. This navigational aid is called *Make Root*.

Chapter 6

Selected Details of the Java Pyramids Explorer

6.1 Introduction

This chapter describes some of the implementation details of the Java Pyramids Explorer (JPE) application. Some of the following sections are useful to understand the techniques that were developed to increase the usability of the Information Pyramids technique. Other sections can be used if a deeper knowledge of the used programming methods and design patterns is needed.

The programming language that has been used to develop the JPE application was Java, more specifically the Java Development Kit (JDK) 1.2, also called Java 2. As Java is an object-oriented programming language the application was designed and developed using object-oriented techniques. Most of the used design patterns can be found in [GHJ+95]. The JPE application contains 75 classes but more than half of them are inner classes that are used for event handling purposes of the graphical user interface. Only 23 of the 75 classes are not used for event handling purposes, but not all of them are important for understanding of the implementation details. The following sections describe the class hierarchy of the application, the purpose of the important classes and the interaction between these different classes.

The class diagram of the JPE application is shown in Figure 6.1. The figure shows only those classes that are necessary for the understanding of the implementation. The used notation for the class diagram is the Object Modelling Technique [RBP+90]. All classes of the application are members of the *iicm.JExplorer* package.

6.2 JExplorer

The class *JExplorer* is the main class of the implementation. It is only used to start the application and to bring up a welcome window. It implements the *main()* method like following:

```
public static void main(String[] args) {
   new WelcomeWindow();
} //endmain
```

The welcome window is an instance of the *WelcomeWindow* class. This class displays a welcome message, opens the main window of the application and closes itself again.

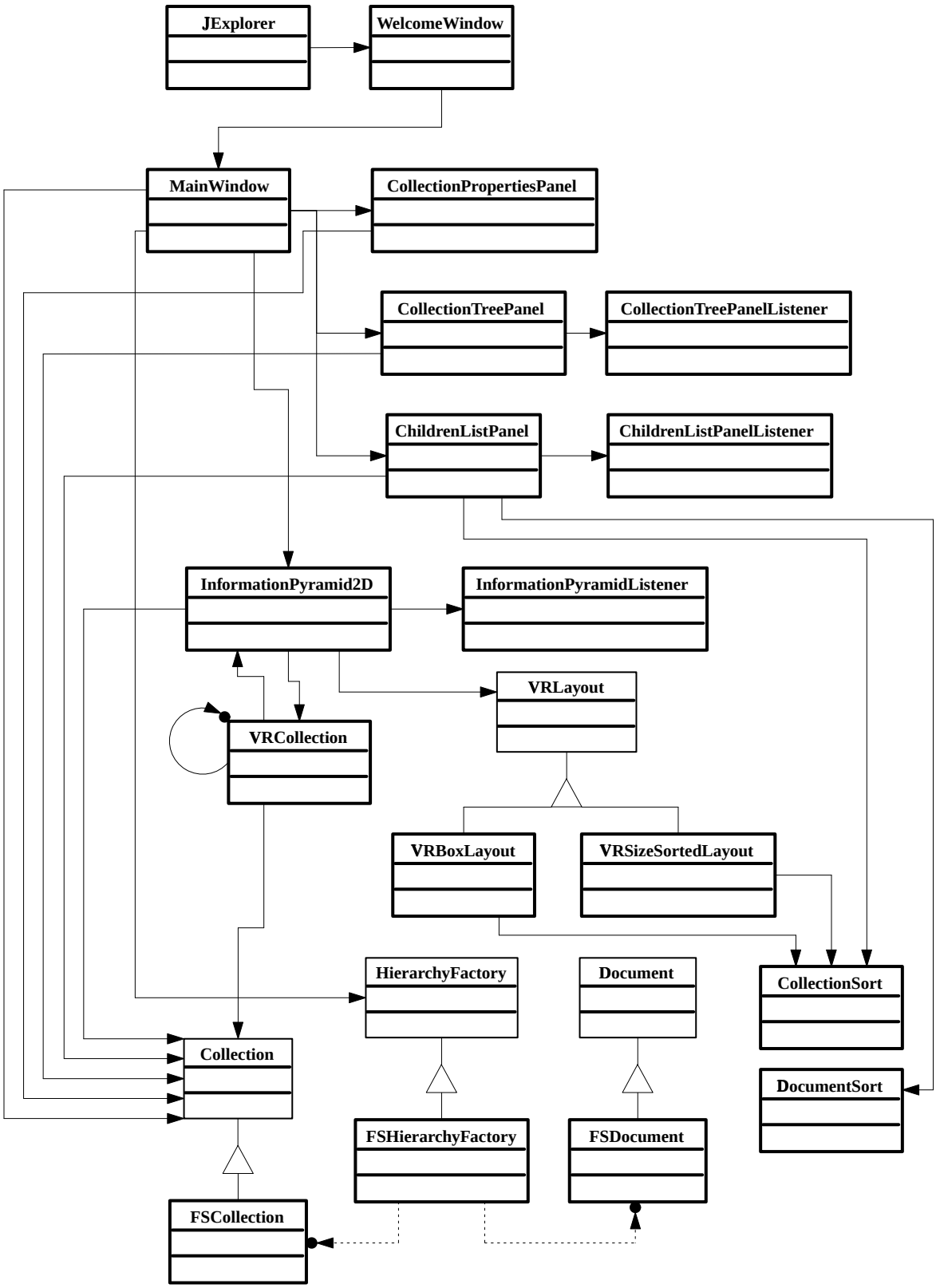


Figure 6.1: The class hierarchy of the JPE application.

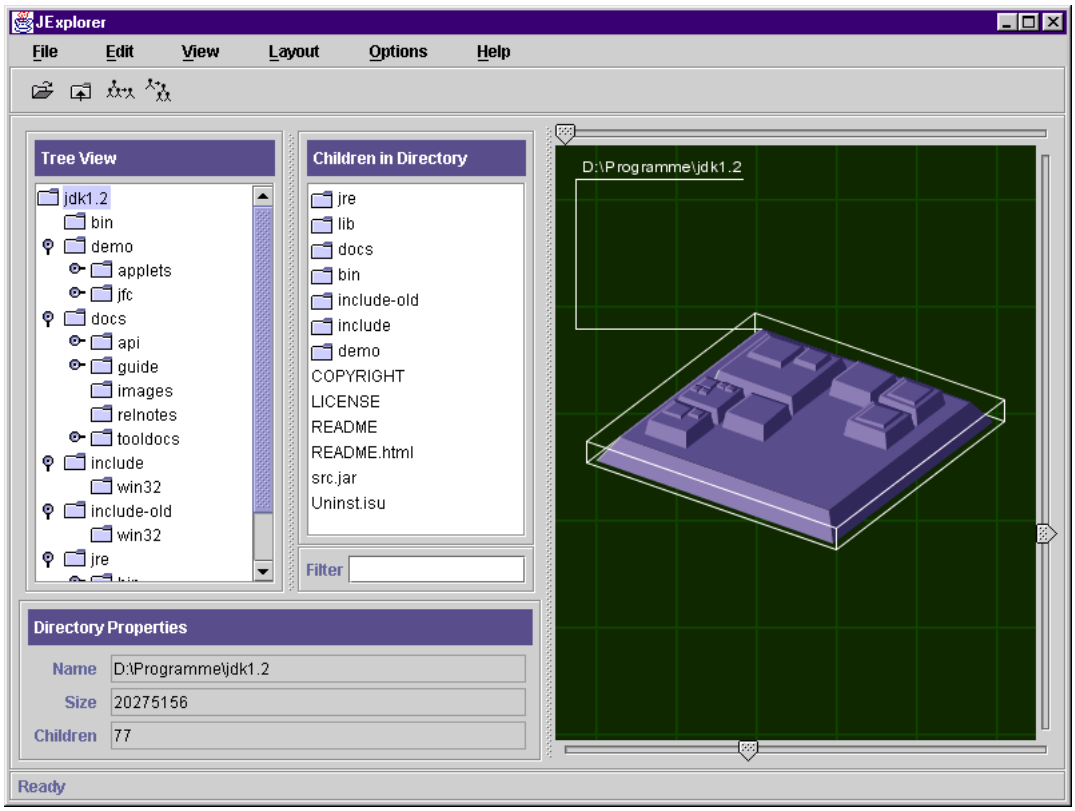


Figure 6.2: The main window of the JPE application.

6.3 MainWindow

The *MainWindow* class of the *iicm.JExplorer* package has several functions within the application. It is a window that contains all the components of the main window of JPE. Figure 6.2 shows the user interface of this class. It contains:

- The menu bar
- The tool bar
- The three visualisations of the hierarchical information
- The properties panel
- The status bar

Further the *MainWindow* class is used for event handling purposes and the synchronisation between the three different views. Event handling is done using inner classes which are listeners to the graphical components. The advantage of using inner classes is, that an inner class has access to the variables of its parent class. Thus, if an event listener class that is an inner class wants to read or set values of the components that are referenced by its parent class, this can simply be done by using the references of the parent. The *MainWindow* class handles the synchronisation between the three different views, too. It is a kind of listener that is informed by the views on user interactions and it is also a kind of controller that forwards these interactions to the other views. If needed, the interactions are forwarded to the properties panel, too.

Initial values for JPE options, such as the sorting order, the user defined view, etc., are stored in property files on the hard disk. These values are loaded when the application is restarted. The options are saved if the user selects the corresponding menu item. The *MainWindow* class provides methods to read or write the options from or to the property files. If an object of the *MainWindow* class is instantiated, the constructor loads the options and sets the values of the menus and state variables.

6.4 HierarchyFactory

One design goal of the JPE application was to make it easy to visualise different types of hierarchies. The current implementation is used to display the hierarchical structure of the directory tree of a hard disk. Further implementations could be used to visualise hierarchies with completely different types of information contents. An example would be that the JPE application visualises the hierarchical structure of the pages of a web server. Therefore a technique had to be found to make it easy for the programmer to change the classes responsible for the internal representation of the hierarchical stored information. A good way to accomplish this is to use an abstract factory design pattern [GHJ+95].

The class *HierarchyFactory* is such an abstract factory. An object of type *HierarchyFactory* is responsible for creating objects that are used to represent the hierarchy. To create these objects the following methods are provided:

- *getCollection()*
- *getDocument()*

The *getCollection()* method returns an object of type *Collection.* The abstract class *Collection* represents an inner node of the hierarchy. The method *getDocument()* returns an object of type *Document* which represents a leaf object of the hierarchy. The *Document* class is an abstract class, too. Figure 6.3 shows an example of the structure of the internal representation of a directory tree. In order to build this internal representation, the *Collection* class provides two methods which are used to get the children of such an inner node. These methods are:

- *getSubCollections()*
  This method returns all children of an inner node that are inner nodes by themselves. The returned objects are instances of *Collection.*
- *getSubDocuments()*
  The method *getSubDocuments()* is used to return the children of an inner node that are leaf nodes. The type of class for them is *Document.*

The JPE application uses the methods provided by the abstract classes *HierarchyFactory*, *Collection* and *Document* to access the hierarchical structured information. As all three classes are abstract, they have to be subclassed and their abstract methods have to be implemented to get access to the hierarchical information. It depends on the subclasses which kind of information can be accessed. One implementation could access a file system while another one could get its information from a web server. Only the methods of the abstract classes are used by the other classes of the JPE application and therefore the implementations can easily be exchanged without changing the other classes.

The current version of the JPE application is used to display the hierarchically structured information of a directory tree. Thus the subclasses are designed to access this kind of information. The names of the subclasses start with *FS* which means *File System.* The implemented classes are:

- *FSHierarchyFactory*

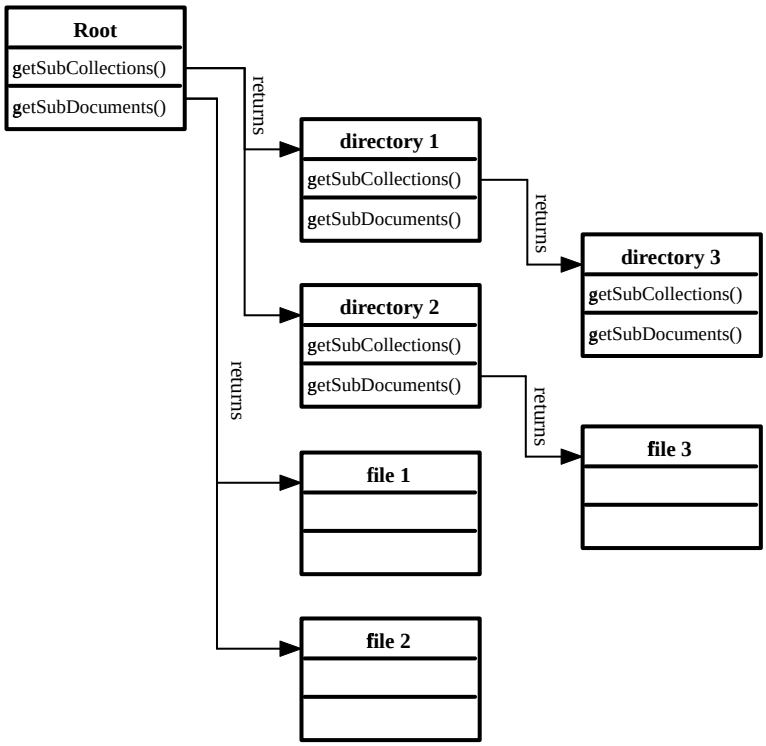


Figure 6.3: An example of the structure of the internal representation of a directory tree.

- *FSCollection*
- *FSDocument*

6.5 ChildrenListPanel

The *ChildrenListPanel* class is a subclass of *javax.swing.JPanel*. It is used to display the children list view of the application. The list of the children is an object which is an instance of the *javax.swing.JList* class. To display scrollbars if needed the list object is added to a *javax.swing.JScrollpane* object.

The default implementation of the *JList* does not display icons or graphics in the cells of the list but only text. Nevertheless it is possible to display any graphical component in a *JList*. The children list view displays the subdirectories and files of a given directory. To be able to distinguish the subdirectories from the files an icon is drawn in front of the names of the directories. To display these icons an own cell renderer class had to be implemented. The implemented cell renderer is an inner class of *ChildrenListPanel*. The name of the cell renderer class is *MyCellRenderer* and it is a subclass of *javax.swing.DefaultListCellRenderer*. The cell renderer has to extend this class because otherwise it could not be used by the *JList*. The only method that has to be implemented by *MyCellRenderer* is *getListCellRendererComponent()*. This method returns the component that has to be displayed in the list at a given index. The implementation of this method looks like following:

```
public Component getListCellRendererComponent(JList list,
   Object value, int modelIndex, boolean isSelected,
   boolean cellHasFocus)
```

```
{
   int index = ((Integer)value).intValue();
   String text;

   if (index < collections_.size()) {
      text = (String) collections_.elementAt(index);
      setIcon(new FolderIcon16());
   } //endif
   else {
      text = (String) documents_.elementAt(index -
         collections_.size());
      setIcon(null);
   } //endif

   return super.getListCellRendererComponent(list,
      text, index, isSelected, cellHasFocus);
} //endfunc
```

To be able to react on user interactions a listener can be defined that is informed by interactions with an object of type *ChildrenListPanel*. The listener has to implement the *ChildrenListPanelListener* interface. The method that is declared in this interface is *collectionSelected()*. This method is called if the user opens a directory in the children list view.

6.6 CollectionTreePanel

The tree view of the application visualises the hierarchy as a tree. The class *CollectionTreePanel* is used for this purpose. *CollectionTreePanel* is a subclass of *javax.swing.JPanel*. It contains three graphical components:

- **A label**
  The label displays the name of the panel.
- **A tree component**
  The tree component is an object of the class *javax.swing.JTree*. The tree displays the hierarchical structured information.
- **A scroll panel**
  The tree component is added to this scroll panel object to provide scrollbars if the tree does not fit into the available display area. The class of the scroll panel object is *javax.swing.JScrollPane*.

The *CollectionTreePanel* object reacts on three different types of user interactions. Two inner classes are used to listen to these events and to react to them. The two inner classes and the three methods which react to user events are:

- ***ExpansionListener***
  This class reacts to events that occur if the user expands or collapses tree nodes. The method *treeCollapsed()* is called if a node is collapsed and *treeExpanded()* is called if a tree node is expanded by the user. As mentioned in Chapter 4, if a node is expanded, not only one level of the hierarchy is expanded but several levels. If the *treeExpanded()* method is called for a node, the remaining nodes have to be opened, too. The method *openNode()* of the *CollectionTreePanel* class is used for this purpose and is called by the *treeExpanded()* method.

- ***SelectionListener***
  The *SelectionListener* class is a subclass of *javax.swing.TreeSelectionListener*. It listens to events that are created if the user selects a node in the tree view. If the user selects a node the *valueChanged()* method of this listener class is called.

In order to synchronise the different views in the application, *CollectionTreePanel* sends events to a listener if one of these user interactions occur. The listener class has to implement the methods of the *CollectionTreePanelListener* interface. The methods that are declared in this interface are:

- *collectionTreeOpened()*
  This method is called if the user expands a node in the tree view.

- *collectionTreeClosed()*
  collectionTreeClosed() is called if a tree node is collapsed.

- *collectionTreeSelected()*
  A selection of a node in the tree view leads to a call of this method.

6.7 CollectionPropertiesPanel

The *CollectionPropertiesPanel* class extends the *javax.swing.JPanel* class and it displays several attributes of a selected directory. The attributes are the name, the number of children and the size of the directory. If a directory is selected in the tree view or the pyramid view the method *setCollection()* of this class is executed and sets the new values of the components that display the three attributes.

6.8 InformationPyramid2D

InformationPyramid2D is a panel that displays the simulated three-dimensional visualisation of the Information Pyramid. The superclass of *InformationPyramid2D* is *javax.swing.JPanel*. *InformationPyramid2D* does not paint the plateaus of the pyramid nor does it compute the positions or the sizes of them. It is used to hold status information of the pyramid view, to react to user interactions and to forward painting requests to the responsible objects. The status information that is managed by the *InformationPyramid2D* class contains the following items:

- *rootCollection_*
  This object is the base plateau of the visualised pyramid. The class of this object is *VRCollection*.

- *clickedCollection_*
  If the user clicks with the mouse on the pyramid, the *clickedCollection_* reference is used to remember the corresponding plateau.

- *layoutManager_*
  To compute the sizes and positions of the plateaus a layout manager is used. This object is such a layout manager. All layout managers are subclasses of *VRLayout*. This class is described later in this chapter. If the layout manager is exchanged by another one during runtime, the visualisation is computed again and refreshed.

- *listener_*
  This is a reference to a listener object that is informed by events that occur within the pyramid view. The listener class has to implement the *InformationPyramidListener* interface. The interface contains only the *collectionSelected()* method. This method is called if the user selects a directory in the pyramid view.

- *shapesFromVRCollections_*
  If the user clicks with the mouse on the pyramid the *shapesFromVRCollections_* vector is used to find the corresponding plateau. The used technique is described later in this chapter.

- *openDepth_*
  This value defines the number of levels that are opened below a selected directory if the user collapses it.

- *sortType_*
  The variable *sortType_* defines the way how the directories are sorted before they are laid out. Possible values for this variable are:

  – *CollectionSort.SIZE*
  – *CollectionSort.CHILDRENCOUNT*
  – *CollectionSort.NAME*

- *sortAscDsc_*
  It is possible to sort the directories ascending or descending, independent which sorting method is used. The order depends on the *SortAscDsc_* variable. Possible values for *sortAscDsc_* are:

  – *CollectionSort.ASC*
  – *CollectionSort.DSC*

- *paintText_*
  If a textual description of the hierarchies should be displayed or not depends on the value of *paintText_*. It is true if the text should be drawn and false otherwise.

- *antialiasing_*
  The *antialiasing_* value is used to switch antialiasing on or off. If antialiasing is really used or not depends on the used hardware and operating system. Therefore it is not guaranteed that antialiasing is really done if it is turned on.

- *fadeInOut_*
  If this value is true and the user selects a new hierarchy or the view of the pyramid is refreshed, the transitions are animated.

- *transparent_*
  This value defines if the whole pyramid should be drawn transparently. This is an effect that does not make much sense but looks freakly good.

- *zoom_*
  This value sets the zooming factor of the display. If a directory is selected the zoom slider can be used to zoom to this directory. If no directory is selected it is zoomed into the center of the pyramid.

- *rotateX_*
  *rotateX_* defines the angle of rotation of the pyramid view around the X-axis.

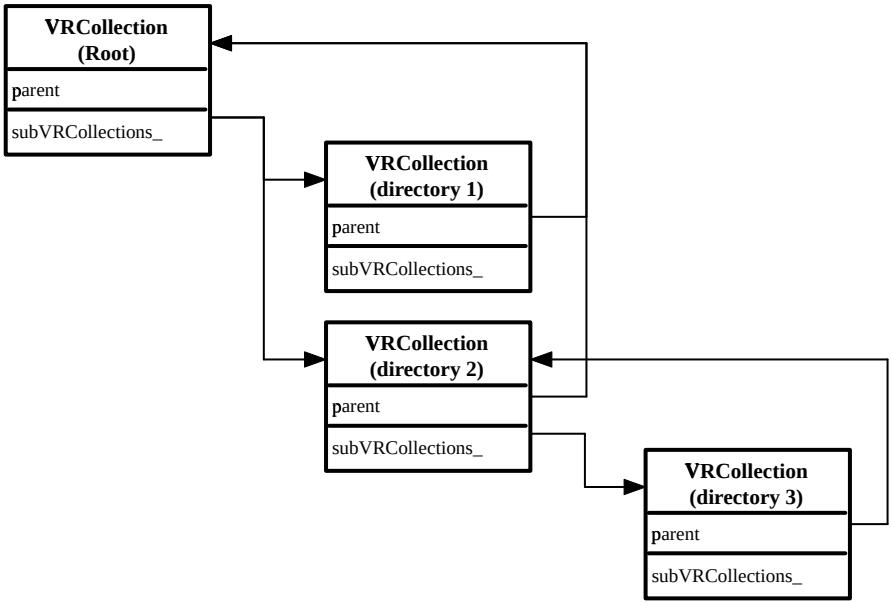


Figure 6.4: An example of the hierarchical relationships between different *VRCollection* objects.

- *rotateZ_*
  *rotateZ_* defines the angle of rotation of the pyramid view around the Z-axis.

The *InformationPyramid2D* class overwrites the *paint()* method of *JPanel*. The paint method draws the whole visualisation of the pyramid as well as the textual descriptions of the directories. The order in which this is managed is as follows. First the *paint()* method draws a background image. It uses a cascading layout to fill the complete background area. After that it sets some of the possible rendering attributes like antialiasing and transparency. Then it paints the pyramid. To draw the pyramid it forwards the paint request to the visual representation (VR) objects. These objects are hierarchically structured and build the complete pyramid. Finally, the textual description of the directories are drawn if the *paintText_* value is *true*.

The VR objects are used to draw the pyramid. Like the internal representation of the hierarchy they are hierarchically structured. There are two kinds of VR objects. First there are objects of type *VRCollection* that are the visual representations of the nodes in the hierarchy. Every *VRCollection* object knows its children VR objects and its parent VR object. This builds the hierarchical structure of the VR objects. Second there are the layout manager objects that are subclasses of the *VRLayout* class. These layout managers are responsible for calculating the sizes and positions of the plateaus of the pyramid. For one pyramid there exists only one layout manager at a time. Nevertheless it is possible to exchange the layout manager during runtime.

VRCollection

The nodes of the hierarchy are visualised as plateaus in the pyramid. Every plateau is represented by an object of type *VRCollection*. If a plateau has to be drawn, the *paint()* method of the corresponding *VRCollection* object is called. The hierarchical structure of the pyramid is embedded in the relationships between the *VRCollection* objects. Every *VRCollection* object has references to its parent and its children nodes. Figure 6.4 shows an example of the hierarchical relationships between the different *VRCollection* objects.

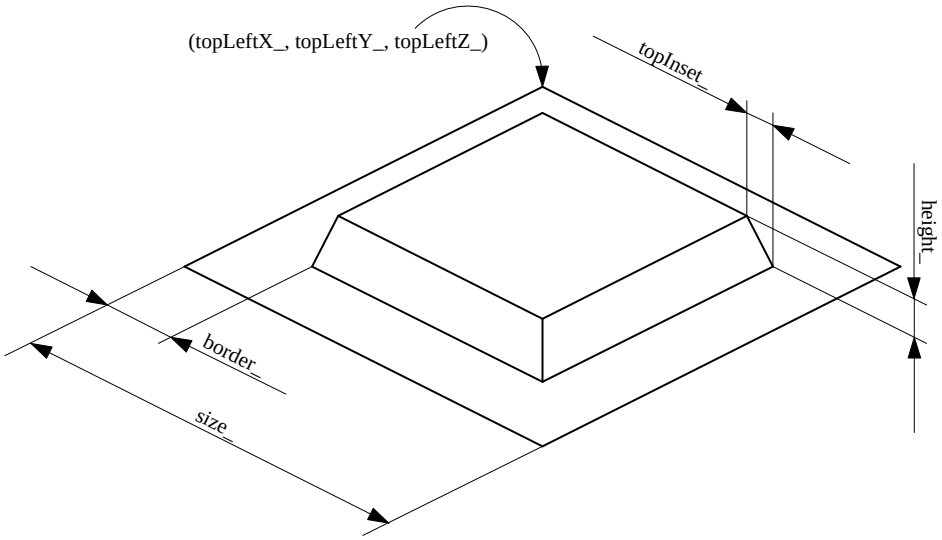


Figure 6.5: The position and dimension of a plateau in the pyramid.

The Dimensions of the Plateaus

The sizes and positions of the plateaus is computed by the used layout manager. Only the base plateau has fixed size. All other plateaus are positioned relative to the position and size of the base plateau. The variables that are used to set the position and dimension of a *VRCollection* object are (see Figure 6.5):

- *topLeftX_*
- *topLeftY_*
- *topLeftZ_*
- *height_*
- *size_*
- *topInset_*
- *border_*

The layout manager calculates the values of the first five items in the list. The remaining two values are calculated by the *VRCollection* object itself. The source code that calculates *topInset_* and *border_* looks like following:

```
topInset_ = (double) (size_ / 100d * COLLECTION_INSETS);
border_ = (double) (size_ / 100d * BORDER);
```

The two constants *VRCollection.COLLECTION_INSETS* and *VRCollection.BORDER* influence the shape of the plateau. The default value for both of them is 5.

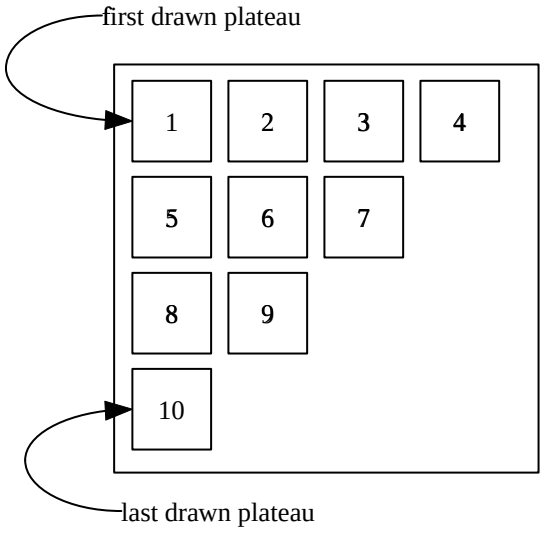


Figure 6.6: The drawing order of the plateaus in the pyramid view.

Hidden Surfaces

In order to hide the surfaces that should not be visible in the current viewpoint the order in which the plateaus are drawn is from bottom to top and from back to front. The directories that are positioned on one base plateau are drawn from top left to bottom right (see Figure 6.6). Thus the plateaus have to be sorted in this way before they are drawn. This sorting is handled by the layout manager.

The order in which the surfaces of a plateau are drawn depends on the angle of the rotation around the Z-axis of the current viewpoint (see Figure 6.7). If the angle is less or equal 45 degrees the order is:

1. surface 1
2. surface 4
3. surface 2
4. surface 3
5. surface 5

If the angle is greater than 45 degrees surface 4 is drawn before surface 1. The order of the other surfaces remains the same.

The Lighting Model

The used lighting model depends on the navigational possibilities of the pyramid view. The view can be rotated clockwise around the Z-axis. Therefore the most useful position of the light source seems to be at the front to the left of the pyramid. This means that surface 4 and surface 3 are lighter than surface 2 and surface 1. This is different to the commonly used position of the light source. The commonly used position of the light source is at the back to the left of the object. If this position would be used the lateral surfaces that are visible if the pyramid is rotated would both be dark. Figure 6.8 shows the pyramid if it is rotated 45 degrees around the X-axis and the Z-axis. As the light source is

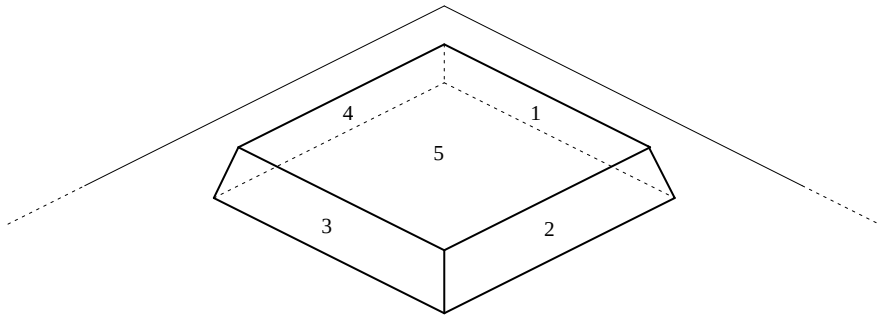


Figure 6.7: The surfaces of a plateau in the pyramid.

at the front to the left, one lateral surface is light and one is dark. With the light source at the back both surfaces would be dark.

VRLayout

Layout managers are used to compute the positions and dimensions of the plateaus in the pyramid. The position and dimension of the base plateau of the pyramid is always fixed. The other plateaus are laid out on the top of this base plateau. The layout manager classes are subclasses of the abstract class *VRLayout*. Every *InformationPyramid2D* object holds a reference to the used layout manager. If the layout manager of such an *InformationPyramid2D* object is exchanged the visualisation of the pyramid is refreshed and the plateaus are laid out again. The abstract class *VRLayout* declares two abstract methods. These two methods have to be implemented by the layout manager classes. The two abstract methods are:

- *layoutSubCollections()*
  This method calculates the position and sizes of the children of a given node. The algorithm that is used to lay out the children plateaus depends on the used layout manager. The method returns an object of type *Vector*. The elements of that vector are the *VRCollection* objects that represent the children nodes of the given node. The elements are sorted by their position in the three-dimensional space. Hidden surfaces are not drawn by the used drawing algorithm if the plateaus are sorted from back to front and from left to right prior to drawing. This sorting has to be done within the *layoutSubCollections()* method of the layout manager.

- *calculateSubCollectionsHeight()*
  The *calculateSubCollectionsHeight()* method computes the height of a plateau.

The JPE application has two different layout managers that can be used. The *VRBoxLayout* class is a layout manager that uses equally-sized plateaus to build the pyramid. *VRSizeSortedLayout* is a layout manager that sorts the directories by their size or the number of children before they are laid out. To build the pyramid it uses differently-sized plateaus. These two layout managers are described in the next two subsections.

VRBoxLayout

The *VRBoxLayout* layout manager arranges the children of a node in rows and columns within their available display area. The sibling nodes of the pyramid are represented by equally-sized plateaus. The height of a plateau is 8 percent of the width of the parent plateau. Prior to layout the directories are

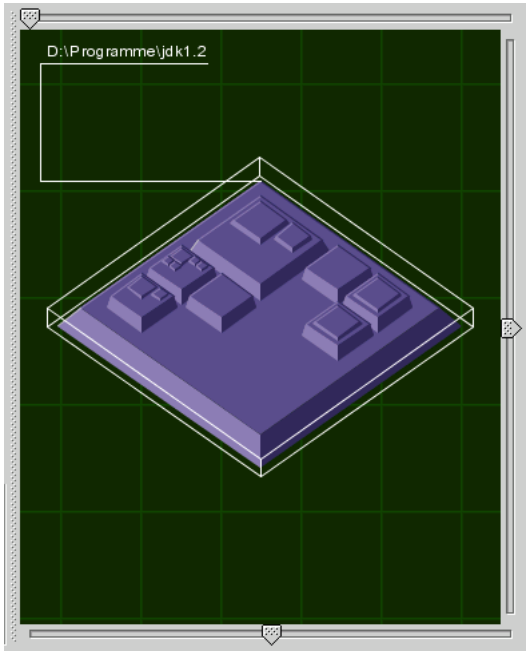


Figure 6.8: The rotated pyramid view. One lateral surface of the plateaus is dark and the other one is light.

sorted using an object of type *CollectionSort*. Figure 6.9 shows a pyramid that uses the *VRBoxLayout* layout manager. The source code of the *layoutSubCollection()* method that does the layout of the plateaus looks like following:

```
if (collCount > 0) {
   int divider = (int) Math.ceil(Math.sqrt(collCount));
   double subSize = size / divider;

   double x = topLeftX;
   double y = topLeftY;
   double z = topLeftZ;
   double height = calculateSubCollectionsHeight(size);

   int cellCount = 1;

   for (int i = 0; i < collCount; i++) {
      Collection subCollection = (Collection)
         subCollections.elementAt(i);

      subVRCollections.addElement(
         new VRCollection(rootPanel,
         subCollection, vrCollection, x, y, z,
         height, subSize));

      cellCount++;
      x += subSize;
```

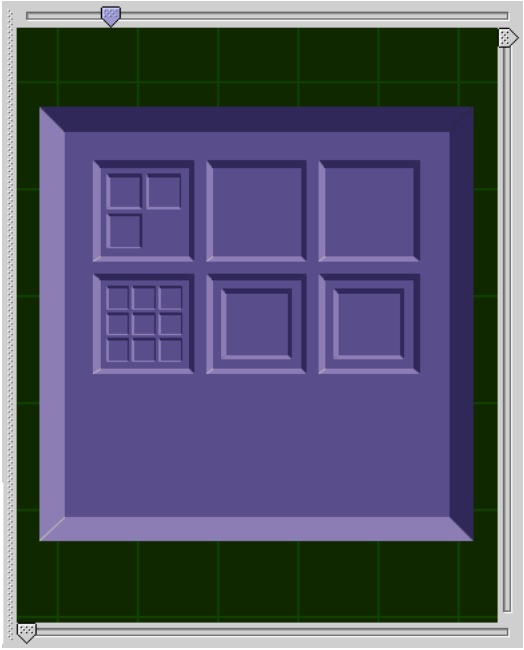


Figure 6.9: A pyramid view that uses the *VRBoxLayout* layout manager.

```
        if (cellCount > divider) {
           x = topLeftX;
           y += subSize;
           cellCount = 1;
        } //endif
     } //endfor
  } //endif
```

VRSizeSortedLayout

The *VRSizeSortedLayout* class is a layout manager that uses plateaus with different sizes to build the hierarchy. The dimension of a plateau is proportional to the size or the number of children of its corresponding directory. The user can select if the dimensions of the plateaus are proportional to the size or the number of children. Figure 6.10 shows a pyramid that uses the *VRSizeSortedLayout* layout manager. The *VRSizeSortedLayout* layout manager tries to maximise the usage of the available display space. As the children are arranged on the top of the plateau of their parent the available space to lay them out is a square. The layout manager uses the following algorithm to compute the sizes and positions of the plateaus:

1. Sort the directories descending by their size or the number of children.
2. Compute the total size of all children and the percentage of each directory compared to this total size. The percentage of a directory is used as the size of its plateau. If the percentage of a directory is smaller than a given minimum size, increase its percentage to the minimum size.
3. Place the biggest plateau in the top left corner of the square shaped display area.

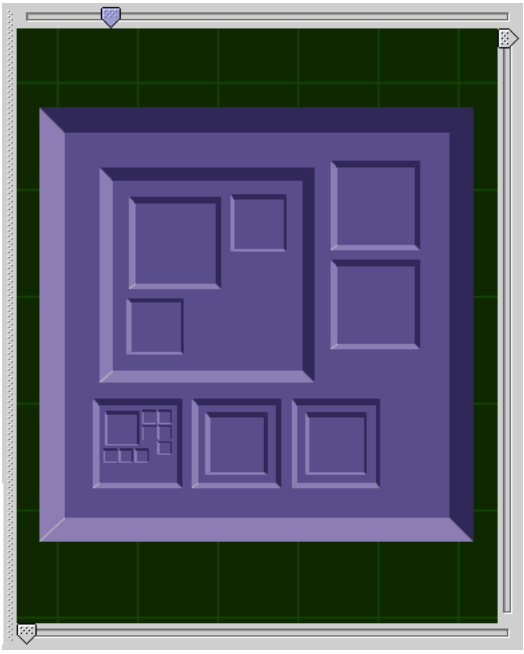


Figure 6.10: A pyramid view that uses the *VRSizeSortedLayout* layout manager.

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 directories are sorted descending, the sizes of the following plateaus are equal or smaller than the size of the second one. 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.

Hit Detection

If the user clicks with the mouse on the pyramid, the two-dimensional coordinates of the position of the mouse are available to find a selected directory. The *shapesFromVRCollections_* vector is used for this purpose. Each surface of every plateau that is drawn is represented by a *java.awt.geom.GeneralPath* object. The *GeneralPath* class represents a geometric path constructed

from straight lines, and quadratic and cubic curves. Every *GeneralPath* object is added to the *shapesFromVRCollections_* vector if the corresponding surface is drawn. That means the first object in this vector is the backmost surface of the pyramid and the last one is the frontmost one.

The advantage of the *GeneralPath* object is that it has a method called *contains()* which returns true if a given coordinate lies within its area. The algorithm to find the clicked plateau passes the vector from back to front and calls the *contains()* method for every *GeneralPath* object. The first object that returns true is a surface of the selected plateau. The two-dimensional *shapesFromVRCollections_* vector stores a reference to the corresponding *VRCollection* object for every *GeneralPath* object. Thus it is easy to find the directory on which the user clicked. The source code that looks for the corresponding *VRCollection* object in the vector looks like following:

```
protected VRCollection getClickedCollection(int mouseX,
   int mouseY)
{
   for (int i = (shapesFromVRCollections_.size() - 1);
      i >= 0; i--)
   {
      GeneralPath path = (GeneralPath) ((Vector)
        shapesFromVRCollections_.elementAt(i)).elementAt(0);

      if (path.contains(mouseX, mouseY)) {
         return (VRCollection) ((Vector)
            shapesFromVRCollections_.elementAt(i)).
               elementAt(1);
      } //endif
   } //endfor

   return null;
} //endfunc
```

6.9 CollectionSort and DocumentSort

The classes *CollectionSort* and *DocumentSort* are two helper classes that are used to sort the directories and files of the visualisation. The *CollectionSort* class is used to sort a vector of *Collection* objects. The objects can be sorted by name, size and number of children ascending or descending. A vector of *Document* objects can be sorted using the *DocumentSort* class. Documents can be sorted ascending or descending by name or size. In both cases the used sorting method is a simple Quicksort algorithm.

Chapter 7

Outlook and Further Work

7.1 Introduction

The intentions during the design and the development of the Java Pyramids Explorer (JPE) application were to improve the Information Pyramids technique and to find solutions to some problems of the previous applications that implemented this technique. Due to lack of time some of the found ideas could not be implemented in the current version of the application. Nevertheless these ideas should be implemented in future versions. Of course, many new possibilities to improve the application and the technique have been found during the development process. It had to be weighed up which improvements should be made in the current version because of the limited time available to finish the application. Therefore many new ideas and improvements are still not implemented and should be part of the further work, too. The purpose of this chapter is to explain these ideas and improvements to support the development process of the future versions.

7.2 Search Possibilities

The current implementation of the JPE provides many navigational possibilities and aids to explore the hierarchical structure of the selected file system. The user can use either the tree view or the pyramid view to navigate through the hierarchy. This helps the user to gain an overview of the structure of the hierarchy. If the user is looking for a specific file or directory this has to be done by exploring the hierarchy and searching for the items manually. Of course, this approach is not very efficient. Thus a search facility would be very important and should be implemented to make the application more useful.

The search function should provide the possibility to search for files and directories by their attributes like the name, the size, etc. It would also be desirable to search for the content of the files. Therefore a full text search engine could be used to extend the possibilities of the search function. One important task during the design of the search functionality is to find techniques how the search results should be visualised in the JPE application. There are many different possibilities how this could be accomplished.

One possibility to display the search result would be to open a new window and display the found files and directories in a simple list like the children list of JPE. This seems to be appropriate if the search result list contains only some few entries. Nevertheless this solution does not use the possibilities of the Information Pyramids technique to display the search result. A solution has to be found where the search result is displayed within the pyramid view.

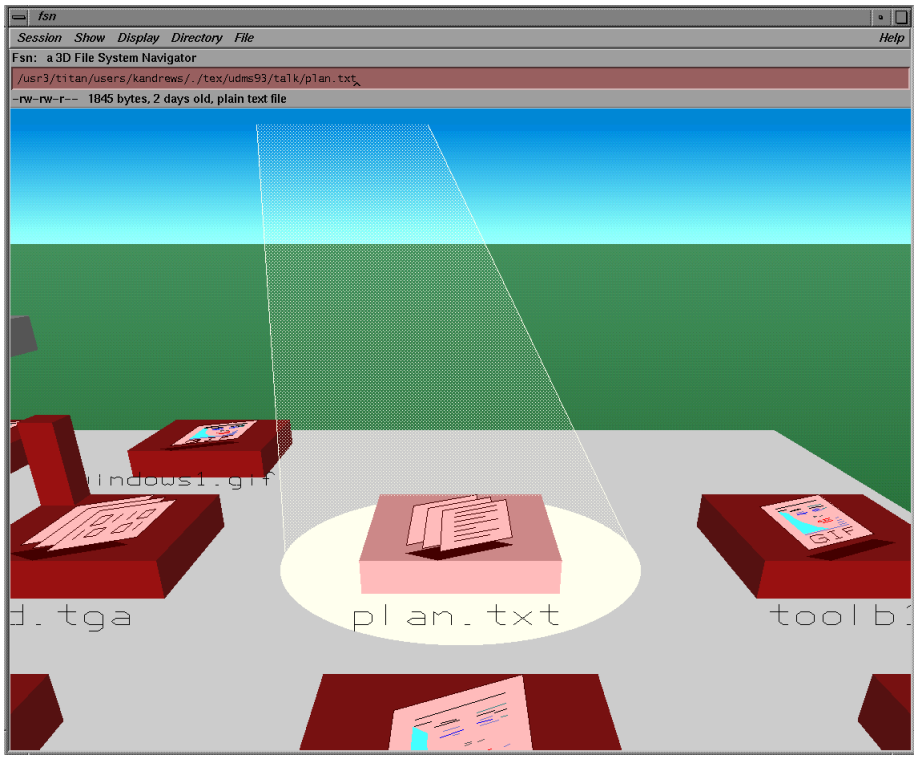


Figure 7.1: A screenshot of the FSN application with a highlighted search result object.

To display the result set in the pyramid view the directories containing the found files could be highlighted in some way. Maybe the plateaus could be coloured or highlighted using a wireframe box. It would also be possible to highlight the plateaus using a technique similar to the one used by the File System Navigator. The File System Navigator uses spotlights to highlight the search results (see Figure 7.1). As the search result list can contain many different entries a possibility should be provided to go through it step by step. At each step the directory that contains the current file is highlighted in the pyramid view and the file is displayed in the children list.

7.3 Improved Layout Managers

To build the hierarchy and to lay out the objects in the landscape the JPE application uses layout manager objects. The object-oriented design of the program makes it easy to develop different types of layout managers and to use them in the application. Currently two different layout managers have been implemented. The *VRBoxLayout* layout manager uses equally-sized plateaus arranged in rows and columns to build the hierarchy. The *VRSizeSortedLayout* layout manager uses plateaus with different sizes for this purpose. These two layout managers are simple to implement and they have good runtime behaviour. Nevertheless there are many possibilities to arrange the plateaus that build the pyramid. The layout of the pyramid is a very time critical operation because if it takes too long to visualise the pyramid the user gets bored. Thus the layout managers must have good runtime behaviour.

One layout technique that could be implemented is the one used by the 3D Explorer. The 3D Explorer uses differently-sized plateaus to build the hierarchy. The resulting layout makes it easy to find information and to give a good overview of the structure of the whole hierarchy. Before the

directories are laid out they can be sorted in any possible way. A disadvantage of this layout manager is its bad runtime behaviour that can lead to inacceptable delays.

Another method to arrange the plateaus is to use clusters. As Josef Wolte noted in [Wol98]:

> "The layout algorithm simply arranges the file 3D objects in rows and columns on a plateau. Another method would be to arrange the objects in clusters. For instance, files created in the same month could be build a cluster or files of the same type. Obviously there must be some kind of a relation between the objects which could be used to form clusters. In a file system there are only few relations which can be used for this purpose. This feature would be very useful when visualising hyperlinked hierarchical data. For instance when visualising a Hyperwave server, the links between the documents can be used to build clusters. Documents which have many links to each other should be placed closer together than documents with no links to each other."

7.4 Display Text in the Pyramid View

To explore the visualisation of the hierarchy and to find information it is very important to display a textual description of the nodes in the pyramid view. The compact visualisation of the Information Pyramids technique makes it impossible to draw the names of every node in the hierarchy within the display area, particularly for hierarchies with hundreds or thousands of nodes. The JPE application displays the textual descriptions of some of the displayed nodes. It depends on the selected node which nodes are labeled. Currently, the selected node, its parent and two siblings of the selected node are labeled.

A internal heuristic evaluation showed that it would be useful during navigation to display the textual descriptions of some of the children of the selected node, too. If the children were labeled it would be easier for the user to decide which parts of the hierarchy should be explored next. JPE displays the names of the nodes at the top of the display area in the pyramid view. The textual descriptions of the children of the selected node could be drawn at the bottom of the pyramid view. The labels of the children of the selected node should have different colours than the other labels. This should make it easier to distinguish them from the other labels. Figure 7.2 shows an example of a modified pyramid view. The children of the selected node are labeled using yellow colours and labels.

7.5 Improved Children List

The JPE application displays the children of a selected node in a simple list. The list contains the subdirectories and files that are in the selected directory. To distinguish the directories from the files an icon is drawn in front of them. Instead of using a simple list an improved visualisation technique could be used. Like other file browsers, e.g. the Windows Explorer, a table could be used to display more information than just the names of the files and directories. The table could display attributes of the entries like:

- Name
- Size
- File type
- Creation date

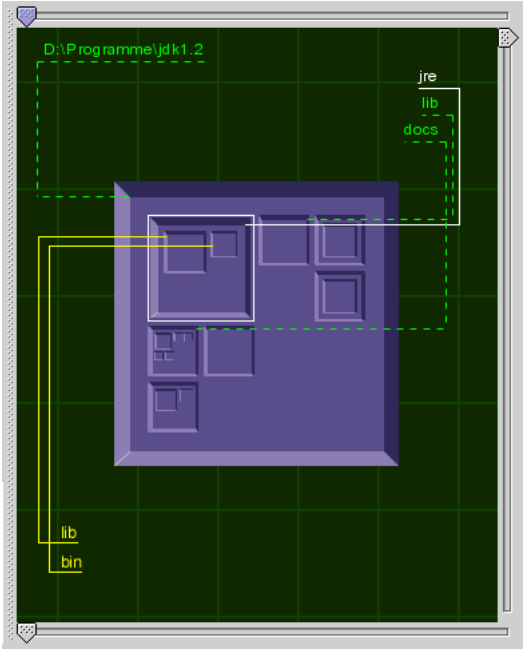


Figure 7.2: A modified pyramid view. The yellow labels are the names of the children nodes of the selected node.

In order to visualise the different types of the files, icons could be drawn in front of the names of the files in the table, too. The columns of the table could be resizable and the order of the displayed columns could be changeable. It should be possible to sort the rows of the table by the displayed attributes. It would also be possible to let the user decide how the children of a directory should be displayed. Like the folders of the Microsoft Windows operating system different visualisations such as a list, a table or a matrix could be possible.

7.6 File Preview

During navigation and to find specific information it is important to see the content of the displayed files. The name of a file is very often not enough information to find the files the user is looking for. Therefore a possibility to display the content of the files would be very useful for the user. A file preview that visualises the content of a selected file could be implemented in an own panel or a separate window. For example, if the file is an image a small preview of this image could be displayed.

7.7 Edit Functions

The current implementation of the JPE application can be used to view the hierarchical structure of a file system. Nevertheless the user can only navigate through the visualisation but it is not possible to change the content of it. Like other file browsers the application should provide the functionality to edit the files and directories that are visualised. Typical editing functions that could be implemented include:

- Cut

- Copy
- Paste
- Delete
- Rename
- Undo

7.8 Display Files in the Pyramid View

The applications 3D Explorer and 3D Explorer for VRML use the Information Pyramids technique to visualise the hierarchical structure of a file system. Both applications visualise the directories and the files of the file system in the pyramid. The directories are visualised as plateaus and the files are visualised as small three-dimensional objects. The JPE application displays only the directories that are represented as plateaus but it does not display the files in the pyramid view. Nevertheless both opportunities should be possible in the JPE application because both techniques have advantages that depend on the usage of the program. The user should have the possibility to select if the directories alone or both the directories and the files should be displayed in the pyramid view. As the pyramid view and the tree view are synchronised on user interaction there remains the open question if the files should be displayed in the tree view, too.

Chapter 8

Concluding Remarks

This master's thesis presented the Java Pyramids Explorer, an application using the Information Pyramids technique to visualise hierarchically structured information.

Chapter 2 discussed related techniques for visualising hierarchies. Some of the techniques use 2D graphics to draw the visualisation some of them 3D graphics.

The Java Pyramids Explorer application was implemented using the Java programming language. The Abstract Window Toolkit and the Swing package were used to develop the graphical user interface. The Java 2D API was used to draw the graphics of the Information Pyramid. These libraries and the Java programming language were explained in Chapter 3.

The Information Pyramids technique was discussed in Chapter 4. Chapter 5 and Chapter 6 introduced the Java Pyramids Explorer and selected details of the implementation of the application. The improvements of the new approach were described and the new possibilities to explore the hierarchically structured information were presented. The main improvements were the combination of different visualisation techniques, the improved display of the textual descriptions of the nodes in the hierarchy, the usage of a platform-independent graphics library, and the introduction of exchangeable layout managers due to the improved object-oriented design of the application.

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

Appendix A

User Guide

A.1 Introduction

The Java Pyramids Explorer (JPE) application is used to visualise the hierarchical structure of a file system. The hierarchy is displayed simultaneously in both a tree view and a pyramid view. The pyramid view uses the Information Pyramids technique to visualise the hierarchy. The pyramid is visualised as a simulated three-dimensional object. The JPE application can be used to explore the hierarchy and to efficiently find information within the hierarchy.

The programming language for the application is Java, and more precisely the Java Development Kit 1.2 from Sun Microsystems. In order to run the application either the Java Development Kit 1.2, the Java Runtime Environment 1.2 or a compatible runtime environment is required. To draw the graphics the Java 2D API is used. The Java 2D API is part of the Java 1.2 compatible runtime environments from Sun Microsystems and does not have to be installed as a separate package.

A.2 Installation

Before running the JPE application, a Java 1.2 compatible runtime environment has to be installed. The JPE application itself is available as a jar-file. The *jexplore.jar* file contains all classes of the JPE application. To start the application the *JExplore* class is used. The command to start the JPE is:

```
java iicm.JExplore.JExplore
```

The *jexplore.jar* file has to be part of the *CLASSPATH* environment variable to find the necessary classes. All images needed by the application are in the *images* directory. Two property files are provided to configure the application. The *JExplorer.properties* file is used to set all labels and menu items that are visible in the main window and all other windows. It is also used to set the path to the images. The second property file *Settings.properties* is used to store the options that can be set within the application.

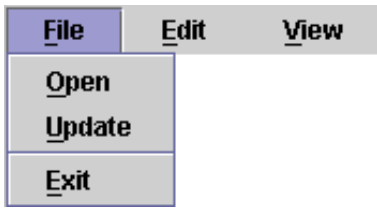
Two settings are important to start the application correctly. First, the path of the application has to be set in the *JExplorer.properties* file. Second, the initial directory of the application has to be set in the *Settings.properties* file. If one of them is invalid the application refuses to start.

A.3 Menu Bar

The menu bar provides easy access to all functions available within the JPE application. Some of the menu items depend on the used options and are only accessible if the appropriate options are selected.

Figure A.1: The menu bar.

File Menu



Open Opens a dialog box where a directory can be selected (see Figure A.2). The selected directory is the new root directory of the visualisation.

Update Reloads the directory structure from the hard disk and refreshes the visualisations of the hierarchy. This is useful if the content on the hard disk has changed.

Exit Closes the application.

Edit Menu

Make Root Makes the selected directory the new root directory of the visualisations. Expanded directories below the selected directory remain expanded.

Open All Children Expands all levels of children below a selected directory.

Select Parent Selects the parent of a selected node. This is useful to go up in the hierarchy. If the selected node is the root node of the visualisation the parent directory is opened and visualised.

View Menu

Top Changes the current viewpoint of the pyramid view to a viewpoint above the pyramid looking down. The transition between the current viewpoint and the Top viewpoint is animated.

Front The viewpoint of the pyramid view is set to a viewpoint in front of the pyramid.

3D Rotates the pyramid 45 degrees around the X-axis and the Z-axis.

User View Changes the viewpoint to a user defined position. The position can be set using the *Set User View* menu item.

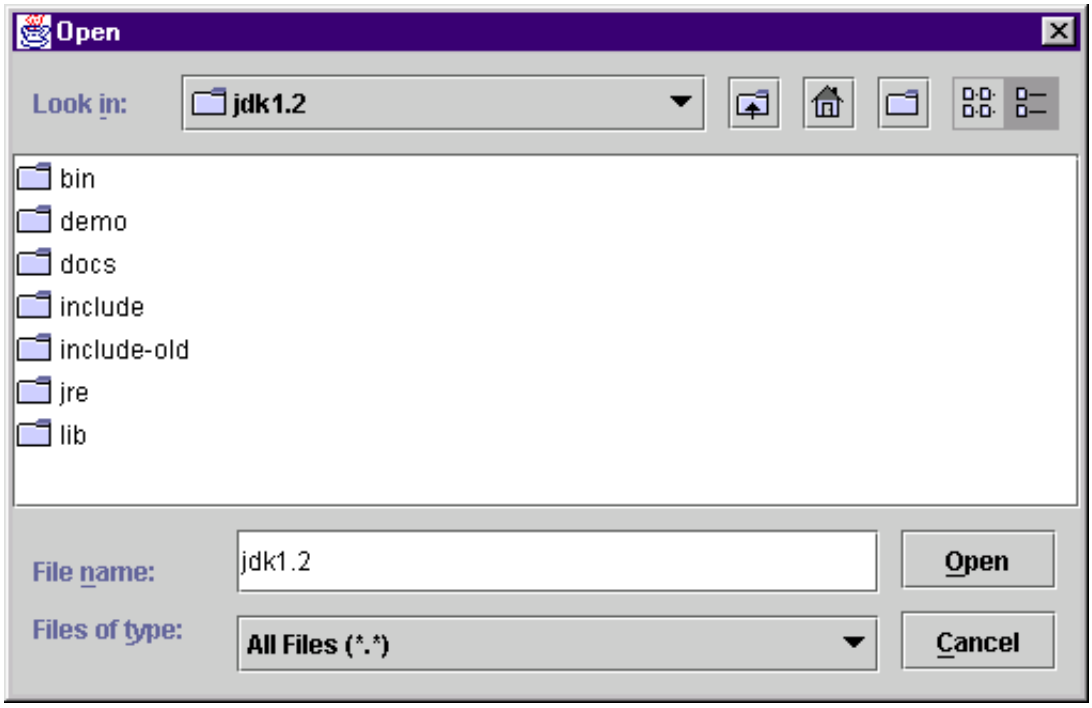
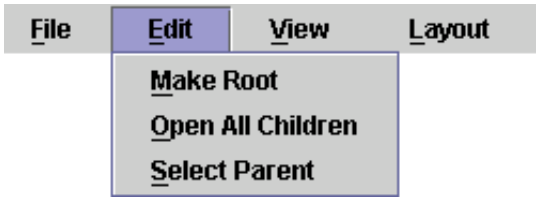


Figure A.2: The directory selection dialog box.



Zoom In Zooms to a selected directory.

Zoom Out Undoes a zoom operation and displays the whole pyramid.

Set User View The current point of view is taken as the user defined viewpoint.

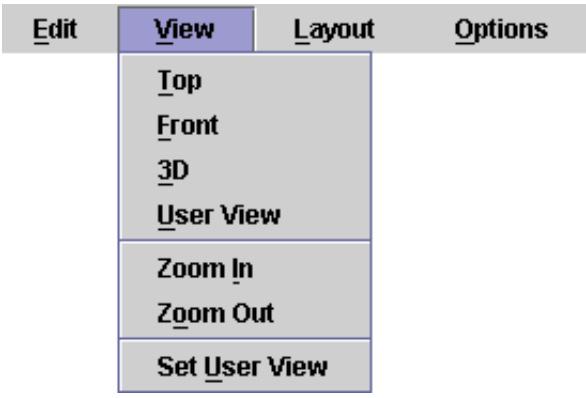
Layout Menu

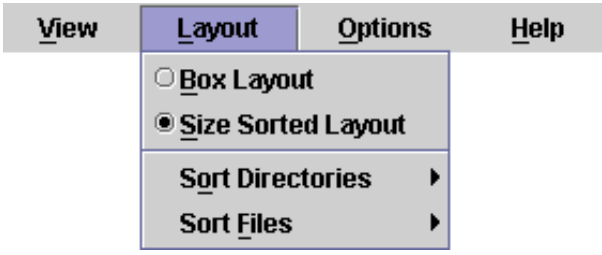
Box Layout Changes the layout manager to the *Box Layout* layout manager. *BoxLayout* uses equally-sized plateaus to build the hierarchy.

Size Sorted Layout The *Size Sorted Layout* layout manager uses plateaus with different sizes to build the hierarchy. This menu item sets the used layout manager to *Size Sorted Layout*.

Sort Directories Sorts the visualised directories in different ways. The available sub-menu items are:

- **by Name** Sorts the directories by their name.
- **by Size** Sorts the directories by their size.
- **by Children** Sorts the directories by the number of their children.
- **Ascending** The sorting of the directories is done ascending.





Descending The sorting of the directories is done descending.

Sort Files The files in the children list can be sorted in some ways. The available submenu items are:

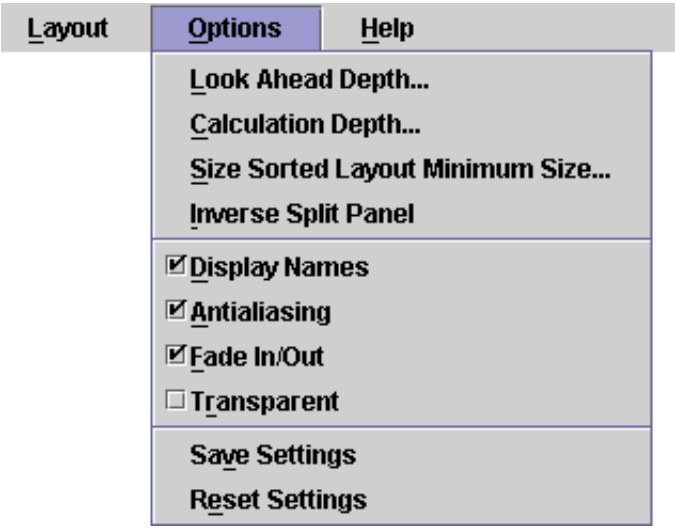
by Name Sorts the files by their name.

by Size Sorts the files by their size.

Ascending The sorting of the files is done ascending.

Descending The sorting of the files is done descending.

Options Menu



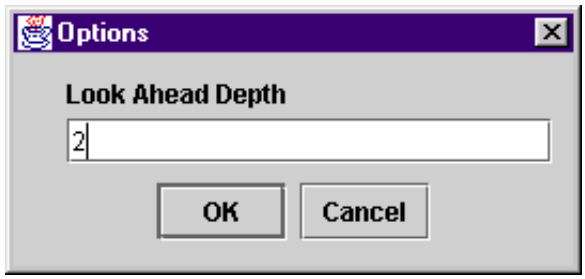


Figure A.3: The Look Ahead Depth window.

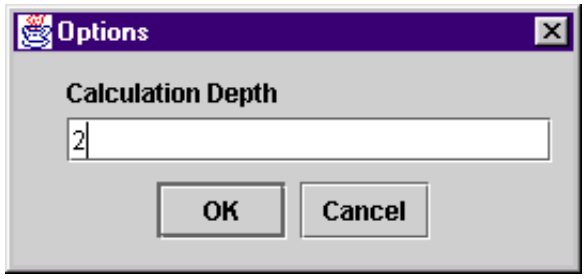


Figure A.4: The Calculation Depth window.

Look Ahead Depth... Opens a window where the number of levels that are expanded if a directory is opened in the visualisations can be set (see Figure A.3).

Calculation Depth... This menu item opens a window where the number of levels can be set, that are used to calculate the size and the number of children of a directory (see Figure A.4).

Size Sorted Layout Minimum Size... By using this menu item a window is opened where the minimum relative size of a plateau of the *Size Sorted Layout* layout manager can be set.

Inverse Split Panel The pyramid view can be either to the right or to the left of the tree view. This menu item changes the position of the pyramid view (see Figure A.5).

Display Names If this option is set, the names of the directories are displayed in the pyramid view, too.

Antialiasing This option defines if antialiasing is used to draw the graphics in the pyramid view or not.

Fade In/Out If this option is set, the transition between different views is animated.

Transparent By selecting this menu item the pyramid is drawn transparently.

Save Settings This menu item stores the options on the hard disk. If the application is restarted the stored options are read from the hard disk and used again.

Reset Settings Sets the options to default values.

Help Menu

About Opens a window that displays some information about the Java Pyramids Explorer application.

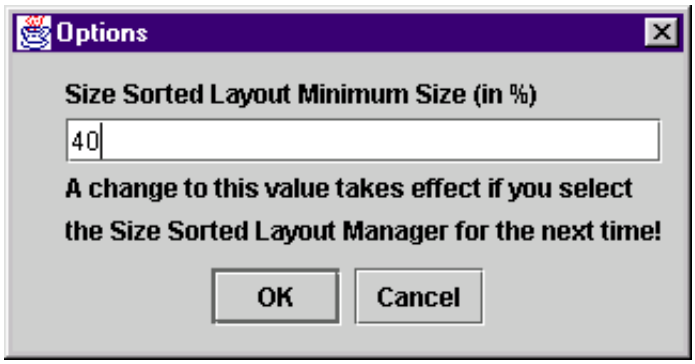


Figure A.5: The Size Sorted Layout Minimum Size window.

A.4 Tool Bar

The tool bar provides quick access to some of the most important functions of the application. All functions that are accessible through the tool bar are available as menu items in the menu bar, too. The available buttons are from left to right:

Open File Opens a dialog box where a directory can be selected. The selected directory is the new root directory of the visualisation.

Select Parent Selects the parent of a selected node. This is useful to go up the hierarchy. If the selected node is the root node of the visualisation the parent directory is opened.

Make Root Makes the selected directory the new root directory of the visualisations. Directories below the selected directory that are expanded remain expanded.

Open All Children Expands all levels of children of a selected directory.

A.5 Pyramid View

The pyramid view uses the Information Pyramids technique to visualise the hierarchical structure of the directory tree. The pyramid is visualised as a simulated three-dimensional object. It can be rotated around the X-axis and the Z-axis and it can be zoomed into the visualisation to enlarge parts of the pyramid that are important at the moment. Three sliders are used to change the viewpoint. Figure A.6 shows a screenshot of the main window of the application.

A single click with the mouse on an unselected directory selects it. The directories can be expanded or collapseed to explore the hierarchy. Interaction is synchronised between the pyramid view,

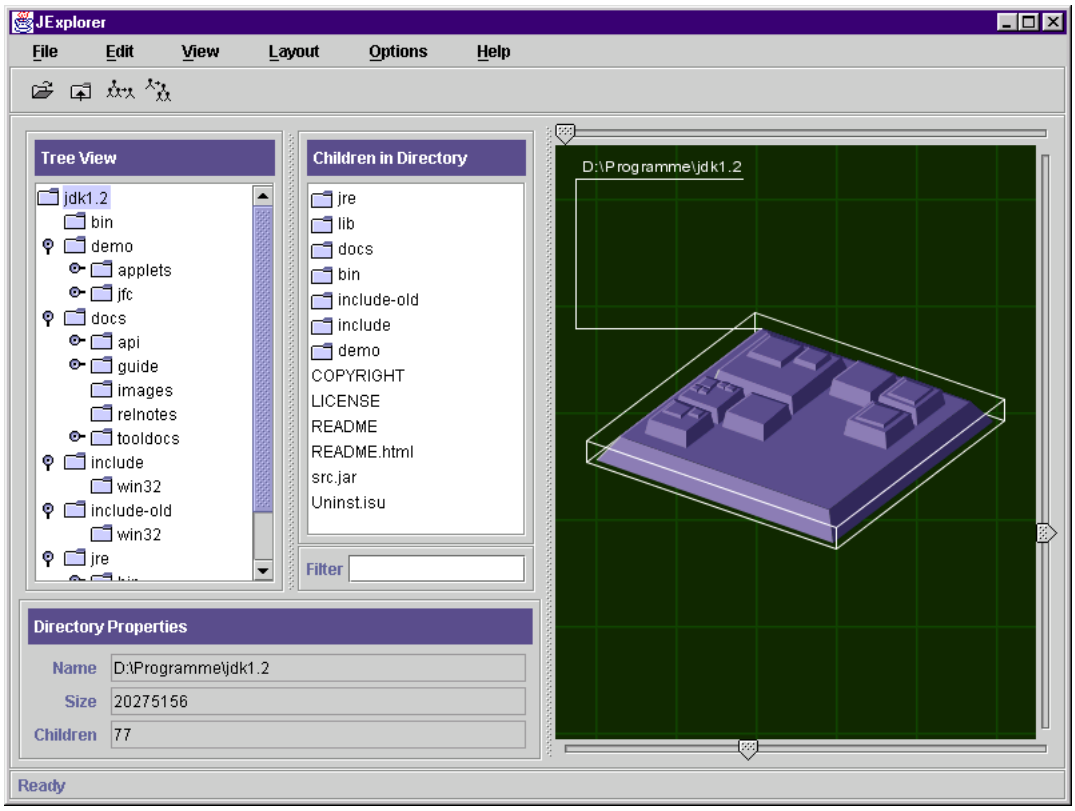


Figure A.6: The main window of the JPE application.

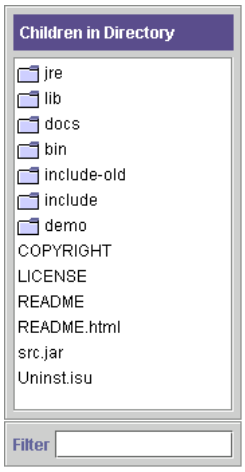


Figure A.7: The children list panel.

the tree view and the children list. A single click with the mouse on a selected directory expands it if was collapsed and vice versa.

A.6 Tree View

The tree view visualises the same hierarchical structure as the Information Pyramid view. The tree view and the pyramid view are synchronised on interaction. If a subdirectory of the hierarchy is expanded or collapsed in one view it is executed in the tree view and the pyramid view. A selection of a node leads to a selection in both views, too.

A.7 The Children List

If a directory is selected in the pyramid view or the tree view the children list displays the children of it. As both, the pyramid view and the tree view, do not display the files of a directory this is the only possibility to find files. As Figure A.7 shows, both the subdirectories and the files of a selected directory are displayed. To be able to distinguish the subdirectories from the files, an icon is drawn in front of the name of a directory.

The *Filter* field of the children list can be used to apply a filter for the files in the children list. The * wildcard can be used to apply useful filters. Possible filters are:

```
filename    display only files that exactly match "filename"
*filename   display all files that end with "filename"
filename*   display all files that start with "filename"
*filename*  display all files that contain the word "filename"
```

A.8 The Properties Panel

The properties panel displays several attributes of a selected directory. The attributes that are displayed in the current version of the JPE are:

- **Name**
  The *Name* field displays the absolute path of the selected directory.
- **Size**
  The *Size* field displays the size of the selected directory.
- **Children**
  The *Children* field displays the number of children of the selected directory.

Bibliography

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, May 1996.

[AH98] Keith Andrews and Helmut Heidegger. Information Slices: Visualising and Exploring Large Hierarchies using Cascading Semi-Circular Discs. In *IEEE Visualization'98, Late Breaking Hot Topic Paper*, Research Triangle Park, North Carolina, October 1998. `ftp://ftp.iicm.edu/pub/papers/ivis98.pdf`.

[Anda] Mark Andrews. Introducing Swing. `http://java.sun.com/products/jfc/tsc/archive/what_is_arch/intro/intro.html`.

[Andb] Mark Andrews. Introducing Swing Architecture. `http://java.sun.com/products/jfc/tsc/what_is_swing/getting_started_2/getting_started_2.html`.

[AWP97] Keith Andrews, Josef Wolte, and Michael Pichler. Information pyramids: A new approach to visualising large hierarchies. In *IEEE Visualization'97, Late Breaking Hot Topics Proc.*, pages 49–52, Phoenix, Arizona, October 1997. `ftp://ftp.iicm.edu/pub/papers/vis97.pdf`.

[BPV96] Luc Beaudoin, Marc-Antoine Parent, and Loius C. Vroomen. Cheops: A Compact Explorer for Complex Hierarchies. In *Proc. Visualization'96*, pages 87–92, San Francisco, California, October 1996. IEEE Computer Society. `http://www.crim.ca/hci/cheops/paper.html`.

[CB97] Rikk Carey and Gavin Bell. *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley, Reading, MA, April 1997.

[CMS99] Stuart K. Card, Jock D. MacKinlay, and Ben Shneiderman. *Readings in Information Visualization : Using Vision to Think*. Morgan Kaufmann, San Francisco, CA, January 1999.

[DFAB98] Alan Dix, Janet Finlay, Gregory Abowd, and Russel Beale. *Human Computer Interaction*. Prentice Hall Europe, Hemel Hempstead, second edition, 1998.

[DK98] David Durand and Paul Kahn. MAPA: a system for inducing and visualizing hierarchy in Websites. In *Proc. of the ninth ACM conference on Hypertext and hypermedia: links, objects, time and space structure in hypermedia systems*, pages 66–76, 1998. `http://www.acm.org/pubs/citations/proceedings/hypertext/276627/p66-durand`.

[ELW98] Robert Eckstein, Marc Loy, and Dave Wood. *Java Swing*. O'Reilly & Associates, September 1998.

[Eyl95] Martin Eyl. The Harmony Information Landscape: Interactive, Three-Dimensional Navigation Through an Information Space. Master's thesis, Graz University of Technology, Austria, October 1995. `ftp://ftp.iicm.edu/pub/theses/meyl.pdf`.

[FvDF+] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, and Richard L. Phillips. *Computer Graphics : Principles and Practice.* Addison-Wesley, Reading, MA, second edition.

[Gea99] David M. Geary. *Graphic Java 2, Mastering the JFC: Swing*, volume 2. Prentice Hall, Englewood Cliffs, NJ, third edition, March 1999.

[GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides+, and Grady Booch. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, first edition, October 1995.

[Hyp] Hyperwave. `http://www.hyperwave.com/`.

[Java] Programmer's Guide to the Java 2D API: Enhanced Graphics and Imaging for Java. `http://java.sun.com/products/jdk/1.2/docs/guide/2d/spec/j2d-title.fm.html`.

[Javb] Java 3D API Specification. `http://java.sun.com/products/java-media/3D/1_2_api/j3dguide/j3dTOC.doc.html`.

[Javc] The Java Tutorial. `http://java.sun.com/docs/books/tutorial/index.html`.

[Jog] Jogl Home Page. `http://copa.pajato.com/jogl/jogl.html`.

[JS91] Brian Johnson and Ben Shneiderman. Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures. In *Proc. IEEE Visualization '91*, pages 284–291, San Diego, California, October 1991. IEEE Computer Society. `ftp://ftp.cs.umd.edu/pub/papers/papers/2657/2657.ps.Z`.

[Knu99] Jonathan B. Knudsen. *Java 2D Graphics.* O'Reilly & Associates, May 1999.

[LR94] John Lamping and Ramana Rao. Laying out and Visualizing Large Trees Using a Hyperbolic Space. In *Proc. UIST'94*, pages 13–14, Marina del Rey, California, November 1994. ACM.

[LR97] John O. Lamping and Ramana B. Rao. Displaying Node-Link Structure with Region of Greater Spacings and Peripheral Branches. US Patent 5619632, Xerox Corporation, April 1997. Filed 14th Sept. 1994, issued 8th April 1997.

[LRP95] John Lamping, Ramana Rao, and Peter Pirolli. A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies. In *Proc. CHI'95*, pages 401–408, Denver, Colorado, May 1995. ACM. `http://www.acm.org/sigchi/chi95/Electronic/documnts/papers/jl_bdy.htm`.

[Mau96] Hermann Maurer. *Hyper-G now Hyperwave : The Next Generation Web Solution.* Pearson Education UK, May 1996.

[Mil56] George A. Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information, 1956. `http://www.well.com/user/smalin/miller.html`.

[NDW93] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, 1993.

[Ope] OpenGL - High Performance 2D/3D Graphics. `http://www.opengl.org`.

[RBP+90] James Rumbaugh, Michael Blaha, William Premerlini, Frederick Eddy, and William Lorenson. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, NJ, October 1990.

[RCM93] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay. Information visualization using 3D interactive animation. *Communications of the ACM*, 36(4):56–71, April 1993.

[RMC91] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone Trees: Animated 3D Visualizations of Hierarchical Information. In *Proc. CHI'91*, pages 189–194, New Orleans, Louisiana, May 1991. ACM.

[Shn96] Ben Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proc. 1996 IEEE Symposium on Visual Languages*, pages 336–343, Boulder, Colorado, September 1996. IEEE Computer Society. `ftp://ftp.cs.umd.edu/pub/papers/papers/3665/3665.ps.Z`.

[Shn97] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, Reading, MA, third edition, July 1997. `http://heg-school.awl.com/cseng/authors/shneiderman/dui3/dui3.html`.

[ST96a] Steven L. Strasnick and Joel D. Tesler. Method and Apparatus for Displaying Data within a Three-Dimensional Information Landscape. US Patent 5528735, Silicon Graphics, Inc., June 1996. Filed 23rd March 1993, issued 18th June 1996.

[ST96b] Steven L. Strasnick and Joel D. Tesler. Method and Apparatus for Navigation within Three-Dimensional Information Landscape. US Patent 5555354, Silicon Graphics, Inc., September 1996. Filed 23rd March 1993, issued 10th Sept. 1996.

[Wol96] Peter Wolf. Three-Dimensional Information Visualisation: The Harmony Information Landscape. Master's thesis, Graz University of Technology, Austria, May 1996. `ftp://ftp.iicm.edu/pub/theses/pwolf.pdf`.

[Wol98] Josef Wolte. Information Pyramids: Compactly Visualising Large Hierarchies. Master's thesis, Graz University of Technology, Austria, October 1998. `ftp://ftp.iicm.edu/pub/theses/jwolte.pdf`.