Information Pyramids

Compactly Visualising Large Hierarchies

Josef Wolte

Information Pyramids

Compactly Visualising Large Hierarchies

Master's Thesis

at

Graz University of Technology

submitted by

Josef Wolte

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

October 1998

Advisor: o.Univ.-Prof. Dr. Dr.h.c. Hermann Maurer
Supervisor: Univ.Ass. Dr. Keith Andrews

Informations Pyramiden

Kompakte Darstellung Großer Hierarchien

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Josef Wolte

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

October 1998

Diese Arbeit ist in englischer Sprache verfaßt.

Betreuer:	o.Univ.-Prof. Dr. Dr.h.c. Hermann Maurer
Mitbetreuender Assistent:	Univ.Ass. Dr. Keith Andrews

Abstract

Visualising and exploring large hierarchies may look like a simple task, but there is currently no single tool available which addresses this task sufficiently.

This thesis describes Information Pyramids which are a new technique to visualise large hierarchies. It utilises three-dimensional graphics to display large hierarchies in an compact and appealing way. Square-shaped plateaus arranged on successive planes represent nodes of a hierarchy. The plateaus representing child nodes of a hierarchy node are laid out atop of the parent node plateau. The overall impression is that of pyramids growing on a plane.

To develop and test this new technique a file system browser using Information Pyramids was created. It is written in Java and uses the native code interface to existing 3D libraries such as OpenGL and Mesa.

Kurzfassung

Hierarchien graphisch darzustellen scheint eine einfache Aufgabe, aber bis jetzt gibt es kein Programm, daß diese Aufgabe zufriedenstellend löst.

Diese Diplomarbeit beschreibt die “Informations Pyramiden”, eine neue Methode zur Darstellung von Hierarchien. Dazu wird drei-dimensionale Graphik verwendet, um große Hierarchien in einer kompakten und anschrechenden Form zu visualisieren. Auf einer Ebene representieren quadratische Plateaus Knoten der Hierarchie. Plateaus die Kinder eines Kontens der Hierarchie sind, werden auf dem Plateau des Elternkontens verteilt. Diese Layoutschema führt zu Pyramiden verteilt auf einer Ebene.

Um diese neue Technik zu entwickeln und zu testen, wurde ein Filesystem-Browser programmiert. Dieser ist in Java geschrieben und verwendet das Java Nativecode Interface um auf 3D Bibliotheken wie OpenGL oder Mesa zuzugreifen.

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

This thesis could not have been written without the help and feedback of many friends and colleagues at the IICM. In particular I want to thank my advisor Keith Andrews for his support, suggestions, and not to forget for correcting the draft versions of this thesis. Michael Pichler, who helped me whith my problems and questions starting the 3D Explorer project. And Helmut Heidegger for many discussions about usability and the user interface of the 3D Explorer.

I also want to thank my friends Arnold Graschl, Siegfried Rossmann and Robert Wenko for their constructive criticism, which helped me a lot to improve the design and development of the 3D Explorer.

Josef Wolte
Graz, Austria, October 1998

Credits

- Figure 2.3, Figure 2.4 and Figure 2.5 were taken from the web site`http://www.crim.ca/hci/cheops/paper.html` ©Centre de recherche Informatique de Montreal
- Figure 2.6 was taken from [LRP95]
- Figure 2.8 was taken from the web site `http://www.cs.umd.edu/projects/hcil/treeviz.html` ©University of Maryland
- Figure 2.9 was taken from the web site `http://www-cui.darmstadt.gmd.de/visit/Activities/Lyberworld` of the German National Research Center for Computer Sience (GMD)
- Figure 2.10 and Figure images/fsnmain.ps were taken from the web site `http://www.sgi.com/Fun/free/fsn.map2.jpg` ©Silicon Graphics Inc.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this theses I will describe a new technique for visualising and exploring hierarchies in 3D called Information Pyramids. The technique was introduced first at IEEE Visualisation 97 [AWP97], and it is patent pending by Hyperwave Inc.

The first chapter, Chapter 2, describes related techniques for visualising hierarchies. Conventional 2D graphics representations such as tree views are described, as well as relatively new 2D technologies like Cheops, tree maps, and the hyperbolic browser. Several 3D visualisations are explored in detail, for instance the cone tree visualisation, which uses cones to display a hierarchy. FSN and the Harmony Information Landscape are also introduced, both using a landscape metaphor.

Chapter 3 gives an overview of the different 3D libraries available. OpenGL and GE3D are described in more detail, since these libraries are used in this thesis. A section is included on the new Java3D API, which is a platform independent 3D API for the Java programming language. This project was developed in Java, but unfortunately this library was not available at the time. The Java language itself and its advantages and disadvantages are described in the next section of Chapter 3. Special attention was taken to explain the native code interface of Java, since this was used to access the GE3D/OpenGL/Mesa libraries. Finally, the details of accessing GE3D functions within Java are discussed.

The Information Pyramids technique is introduced in Chapter 4. Firstly, the visual representation of hierarchy nodes and their layout in the 3D landscape are explained. The possibilities for mapping data attributes to attributes of the 3D object representation are also described. Secondly, navigation in the Information Pyramid is discussed. Some well-known navigation techniques are described, such as a free flying mode, as well as techniques focusing on browsing information, such as an automatic fly mode, which flies the user automatically to the selected object. Furthermore, the excellent features of Information Pyramids to provide the user with a global navigation context are explained. Finally, techniques to focus the user on a particular subtree, such as *pruning*, are described.

To test and improve the Information Pyramids technique a sample implementation was developed, the 3D Explorer. The design of this file browser is discussed in detail in Chapter 5. Since the application was written in Java, it is an object-oriented design. Firstly, the classes to start and initialise the application are described. Secondly, the user interface classes are shown in detail. The application uses many standard widgets of the Java windowing API, but the navigation bar is a custom-designed widget. It was designed to provide an appealing

and usable interface, especially designed for the 3D Explorer needs. Thirdly, the classes to display the 3D graphics are described. This section also deals with the handling of the user input, such as picking a 3D object. The next section gives an introduction to the internal data model. It shows how the hierarchical data and their corresponding 3D objects are represented in memory. At the end of the chapter the classes to access the file system are described. These classes use threads to load data while the user can continue navigating through the landscape. The handling of threads is therefore also described here.

Information Pyramids are a new technique so it is under continuous development. Chapter 6 briefly describes new ideas and suggested improvements to the technique.

In the Appendix A a user guide to the 3D Explorer application can be found. It describes how to install and start this Java application. All user interface widgets and their function are explained. It also gives a short introduction to the navigation methods implemented, in particular how to use the browse mode for navigation and the handling of viewpoints.

Chapter 2

Visualising Large Hierarchies

2.1 Coping with Hierarchies

Due to recent developments in hardware and software, the average computer user has to deal with ever larger amounts of data. The hardware industry provides storage devices with higher capacities and faster access, such as DVD, CD-ROM, and improved hard disks. On the software side, programs are becoming larger and more complex. The fast growth of the Internet also makes it easy to access overwhelmingly large information sources. To cope with huge amount of data, some structuring and navigational support has to be provided. As an example, the WWW provides links and anchors to structure information, resulting in a web of related pieces of information. This structural paradigm is very flexible, but it can also be very confusing. People often become "lost in hyperspace": neither knowing where they are or how to get to a particular document.

A popular way to structure data is in a hierarchy. It is a very intuitive and easy structure, but it is not particularly flexible. Hierarchic structures are widely used in both technical and non-technical applications. A well-known application is a file system. Increasingly, WWW sites use a hierarchy to structure the content. In this case the links are primarily used to build a hierarchy. Some companies like Yahoo categorise WWW pages to fit them into a hierarchy. A non-technical application would be an organisation chart, to display the relation of employees to divisions or departments.

The concept of a hierarchy is intuitive and simple, but it is surprisingly difficult to visualise large hierarchies. Small hierarchies are visualised well by the traditional 2D scrolling browsers with horizontal tree layout (a tree view such as the Windows Explorer). However, this approach breaks down with larger hierarchies. The user continually has to scroll the tree, because it is too big to fit on the computer screen. Only a small part of the hierarchy is visible at any one time. Numerous techniques have been developed to better display larger hierarchies, but so far none of them has replaced the tree view. Nevertheless, even on a simple home PC the directory structure is quite large, and will grow in the future, to store all the files of new and more complex operating systems and applications. Eventually, some visualisation technique will need to supercede the tree view. Some possible candidates will be discussed in this chapter. Many of them use 2D graphics to display the data, but there are also several visualisations which use 3D graphics. The 3D space has some advantages over 2D.

Using the third dimension, more information can be visualised within the same screen space. Many computer systems provide hardware accelerated 3D graphics, so 3D visualisations like Cone Trees or Information Pyramids will gain more attention in the future.

To cope with large hierarchies, a visualisation has to satisfy certain design goals. Of course, it is very difficult, or almost impossible, to satisfy all of them. Due to the many applications of hierarchies, visualisation techniques often meet only a subset of the following goals:

- The visualisation has to fit on the screen. The user gets an overview and can easily see which parts of the hierarchy are deeper.
- After frequent use of the visualisation, users should easy recognise where they can find a particular piece of information.
- The visualisation should be consistent over time. When some data can be found in a particular part of the visualisation, it should be there next time.
- Easy navigation within the visualisation. The navigation should be intuitive, so users do not loose context.
- Not only the structure of the data should displayed, also other interesting attributes should be visualised.

2.2 Tree Views

Tree views are a common way to visualise hierarchies. There are numerous incarnations such as Microsoft's Explorer for Windows or the Macintosh File Finder. The visualisation displays the hierarchy as a horizontal tree, which is a typical result when drawing a hierarchy on a sheet of paper. With some simple operations (usually mouse clicks) subtrees can be opened or closed. When the visualisation is too large to fit on the screen, scrollbars are used to navigate through the visualisation. To see the items in a category of a hierarchy most of the applications provide a second window. In this window the items are arranged in a simple list.

Tree views provide an easy and intuitive visualisation. Standard components of an GUI system can be used, so if the user knows the GUI he knows almost how to work with the visualisation. The big disadvantage of a tree view is that large hierarchies are hard to handle, because the user has to scroll through the visualisation all the time. Due to this scrolling of the viewing window it is almost impossible to get an overview of a large hierarchy.

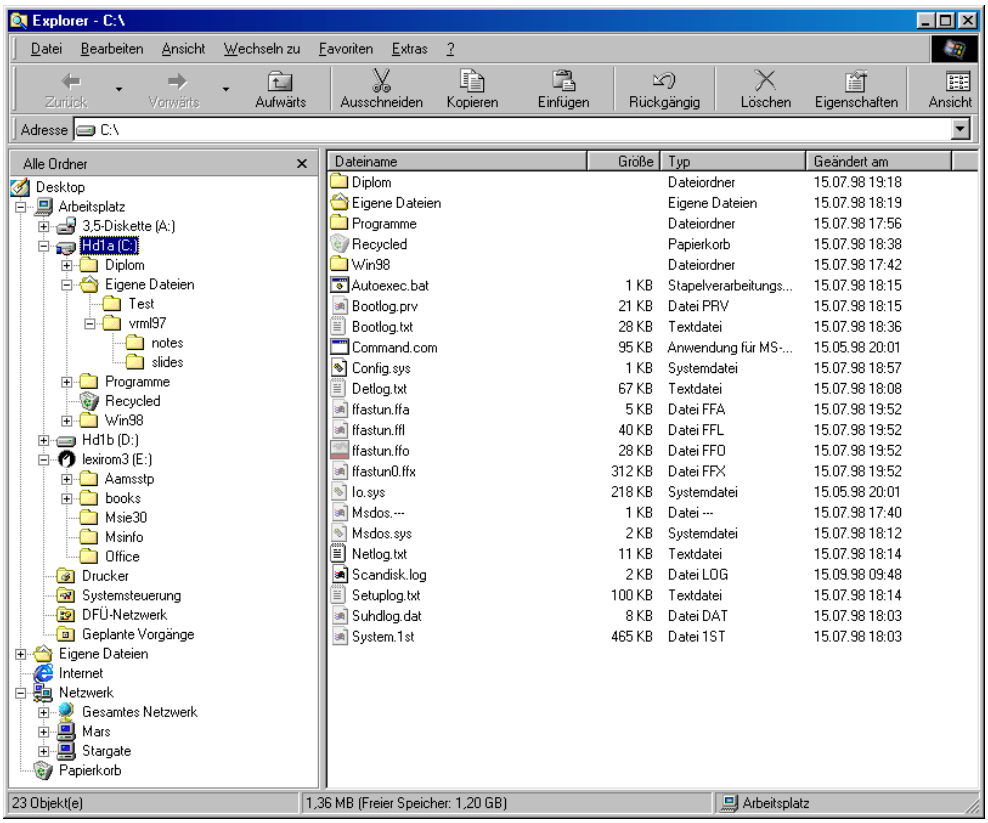


Figure 2.1: The Explorer of Microsoft Windows.

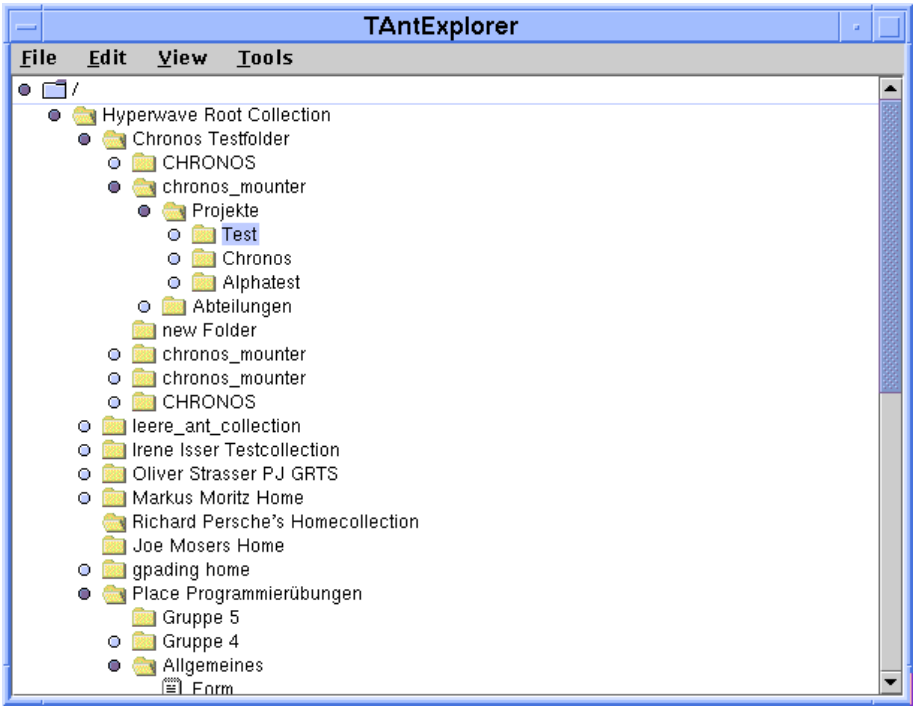


Figure 2.2: The AntExplorer - an example for a tree view.

2.3 Cheops

The Cheops [BPV96] method tries to compress the simple tree view visualisation. A triangle is used as a visual representation of a node of the hierarchy. The root node of the hierarchy is displayed as a triangle at the top of the screen. Starting from this node, the levels of the hierarchy are visualised as levels of triangles (see Figure 2.5). To gain a pyramidic shape for the whole visualisation, triangles overlap at each level. The actual meaning of a triangle is determined by the selection of a triangle in the layer above. As an example, a hierarchy of depth three and branch factor of three looks like Figure 2.3.

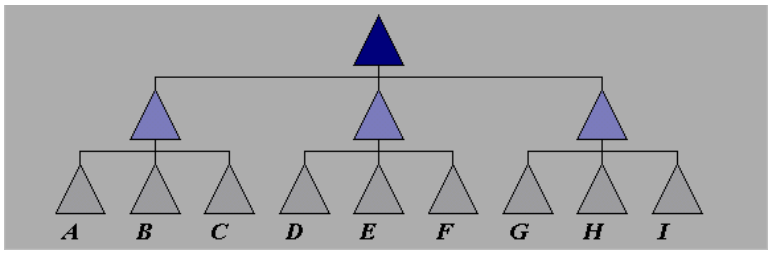


Figure 2.3: A hierarchy of depth 3 and branch factor of 3.

The Cheops visualisation of this hierarchy looks like Figure 2.4. The third level in the Cheops visualisation has only five nodes, whereas the conventional visualisation requires nine nodes. The compression is done by overloading the nodes, triangles in the last level represent more than one logical node. This overload becomes unambiguous when a node in the laver above is selected.

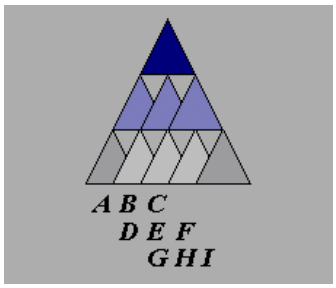


Figure 2.4: The Cheops representation of the hierarchy in Figure 2.3.

For instance, if the leftmost triangle of the second level is selected, the three leftmost triangles of the last level represent nodes E, F, and G. The other two triangles of the last level have no meaning in this case.

If a node is chosen by the user, then the triangle changes its colour (see Figure 2.5). Also all the nodes at this level become uncle nodes and change colour. The children of the selected node also change colour. Of course different colours are used to distinguish the different node types. Due to this colour coding, each branch in the hierarchy has a unique profile in the Cheops representation. Highlighting is used to aid browsing. As the cursor enters a visual component, the entire branch associated with this node is highlighted. This

allows the user to access nodes within the hierarchy without finding the right branches in the levels above. Additional navigation is provided by choice boxes and navigation buttons. There is one choice box for every level of the hierarchy at the right of the pyramid. When selecting a node in the visualisation, the name of the node is displayed in the choice box. Using the choice box to select an node of the hierarchy, highlights the associated triangle in the visualisation. Depending on screen size and number of nodes in the hierarchy, navigation with the mouse can be difficult. The triangles on the screen can become very small, so selecting one is difficult. Navigation buttons allow the user to browse in the hierarchy: there are up, down, left, and right buttons to change the selected triangle.

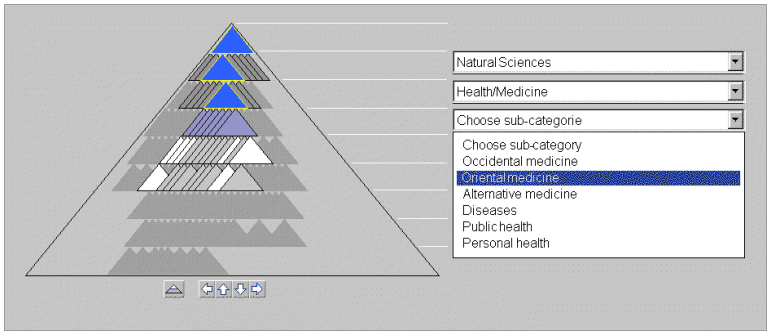


Figure 2.5: The Cheops visualisation with colour coding and choice boxes.

Navigation in large hierarchies is easy with Cheops, but it does not provide a good global overview. Another disadvantage is that the technique was designed to visualise static hierarchies, because the whole hierarchy has to be known in advance to calculate the visualisation.

2.4 Hyperbolic Browser

The hyperbolic browser [LRP95] uses a focus-and-context (fish-eye) technique based on hyperbolic geometry. It can be seen as a conventional tree view, laid out on a hyperbolic plane. This is an easy task in the hyperbolic plane, which has infinite space in each direction. A node of the hierarchy is allocated a wedge of the hyperbolic plane. All children are placed along an arc in that wedge. Sub-wedges angling out from every child are calculated to hold its descendants. This layout schema is performed recursively for all nodes. Once the tree is laid out on the hyperbolic plane, it is mapped to the 2D plane for display. A conformal mapping, or Poincaré model, is used to map the hyperbolic plane to the unit disk. This mapping preserves angles but distorts lines to arcs on the unit disk. The result is a visualisation where nodes at the centre are largest and nodes near the edge get gradually smaller, as shown in Figure 2.6.

Text labels are attached to the branches of the tree as space allows. The user can change the focus by clicking on a label to bring it into focus. Another way to browse through the hierarchy is to drag a point interactively to another position. In both cases the transformation is smoothly animated.

A hyperbolic browser provides a good global overview of the hierarchic structure. A problem of the visualisation is that if the text of the node labels is very long, labels will

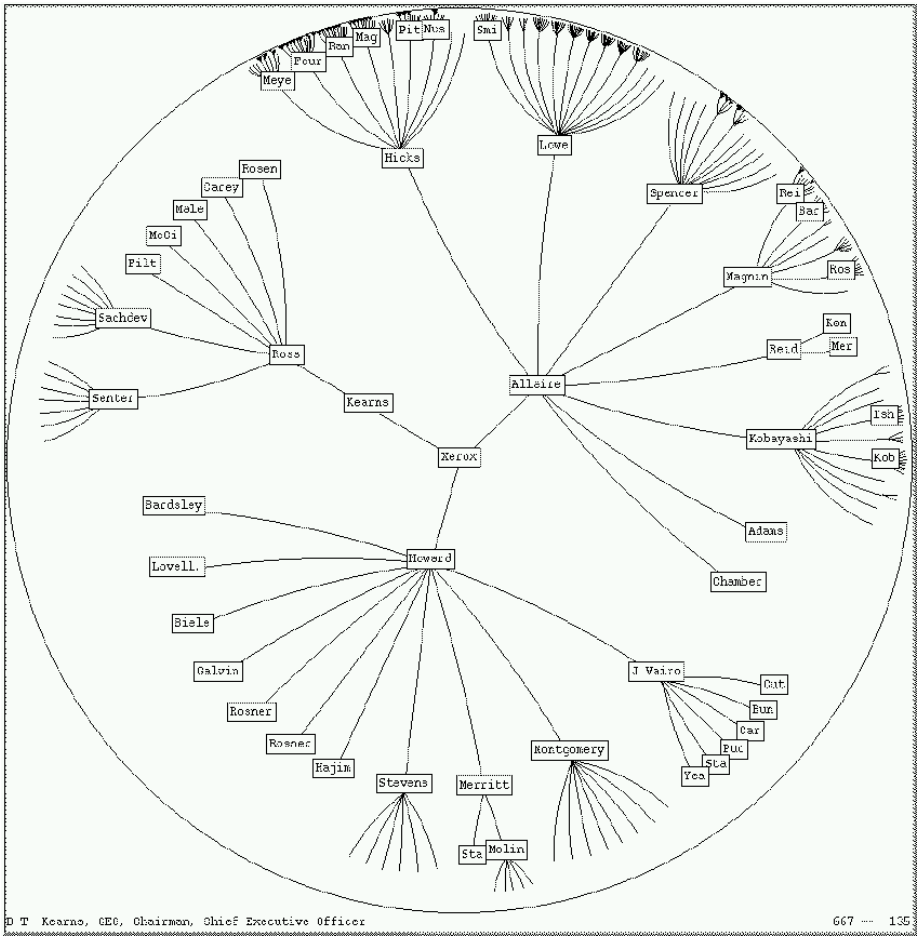


Figure 2.6: The Hyperbolic Browser.

overlap. An option to cut of the text to given length leads to unreadable labels. To read the whole text the mouse cursor has to be moved over the label.

2.5 Tree Maps

Tree maps [Shn92] use a 2D space-filling approach to visualise hierarchies. A rectangular area is partitioned into rectangular subareas whose size corresponds to an arbitrary attribute. For example when visualising a file system, the partition size might represent cumulative file size or number of files. A simple recursive algorithm is used. The input for this algorithm is a node and a rectangular area. The number of children of this node determines the the number of partitions. The size of each partition is proportional to Size(child)/Size(node). That means the area is divided into pieces which are sized proportional to the size of the child nodes. For every child the algorithm is invoked recursively using the depth-first strategy (see Figure 2.7). For better visual representation the partitioning alternates horizontally and vertically. Even levels of the hierarchy are vertically partitioned and odd levels horizontally, for instance. Colour coding of the different regions is provided to represent an attribute. It could be used

to represent the file type, age of a file, owner of a file, or the frequency of use for example.

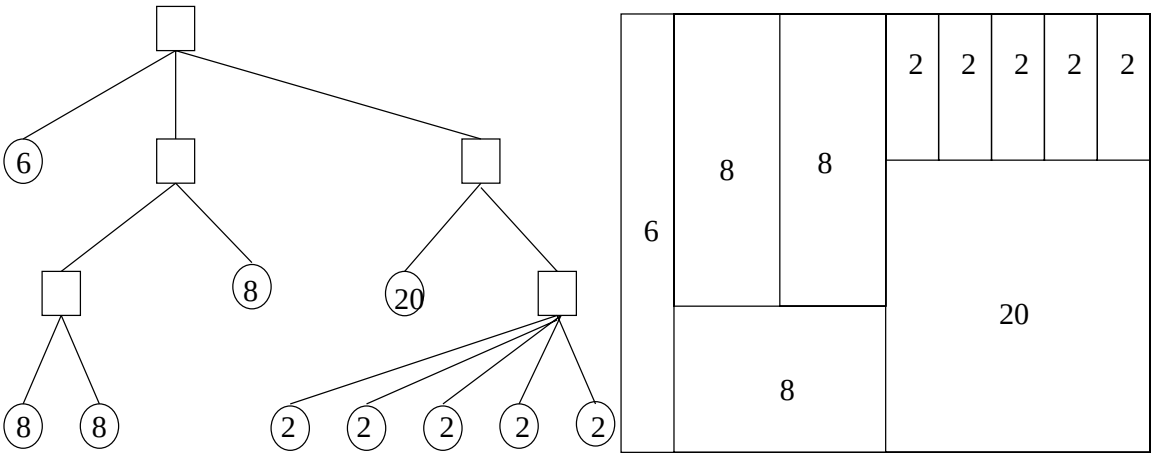


Figure 2.7: A tree structure with numbers indicating the size of each node and the corresponding tree map representation.

The algorithm runs linearly with the number of nodes in the tree and can display one to two thousand nodes on a standard VGA display (640 x 480 pixels). Of course, small nodes which become too small in the visualisation are eliminated. To see such nodes a zoom operation has to be implemented.

The tree maps provides a good overview, but in applications with many leaf nodes (e.g. files), the whole visualisation is very confusing (see Figure 2.8).

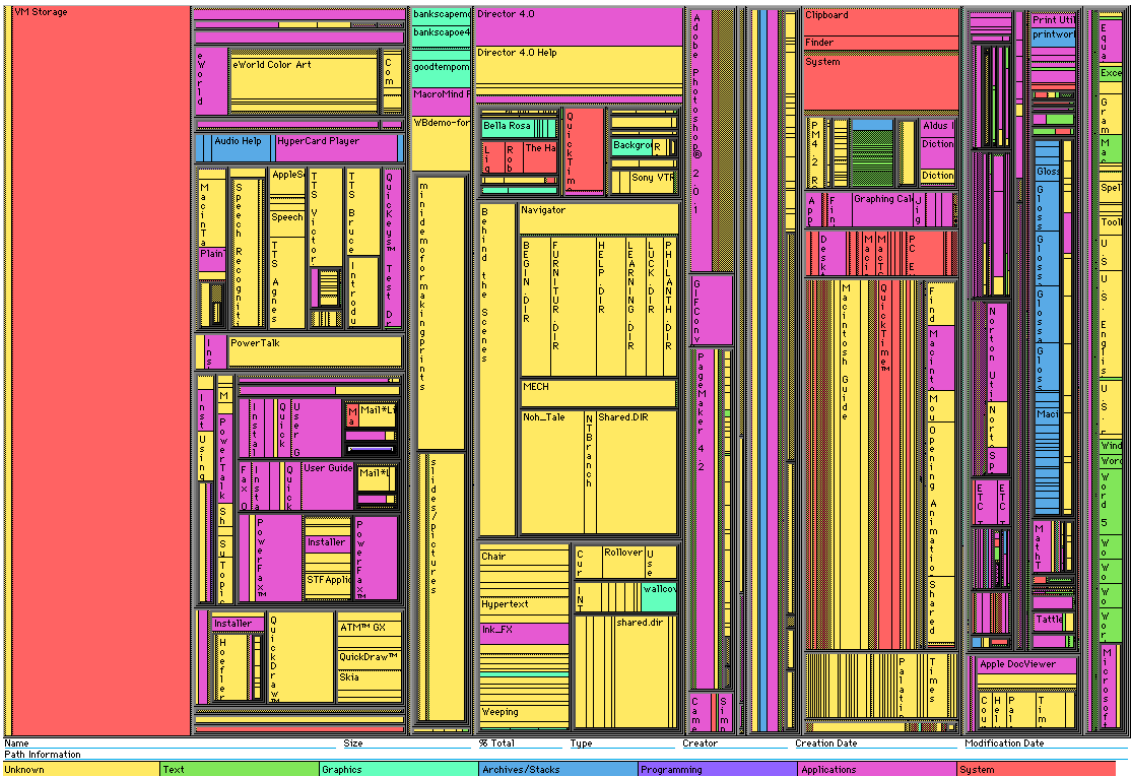


Figure 2.8: The TreeViz application displaying a file system with tree maps.

2.6 Cone Trees

The traditional way to visualise a hierarchy is to use a horizontal tree layout. Cone Trees [RMC91] use the same concept, but utilise three-dimensional space to display the tree.

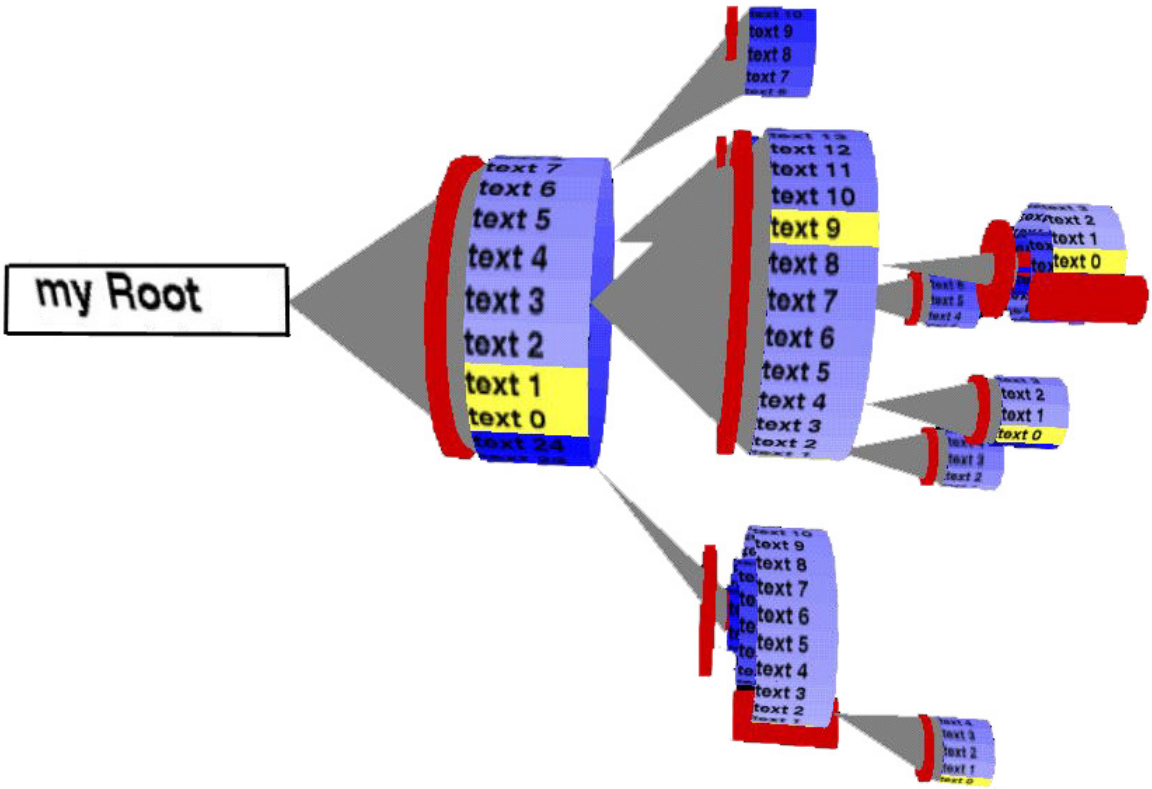


Figure 2.9: The cam tree visualisation in Lyberworld.

A node of the hierarchy (e.g. a directory in a file system) is visualised as a cone in 3D. All the items in a node (e.g. files and subdirectories in a file system) are represented as text labels along the base of the cone. The base of the hierarchy is placed near the ceiling of the screen. The next layer of nodes is drawn below the first, also as cones. The top of these children cones is connected to the appropriate label. To ensure that the whole structure is visible all the time, the height and the base diameter of the cones is calculated to fit into the screen. Also each layer has cones of the same height. All cones are shaded transparently, so that a cone does not block the view of cones behind it. A variation of this approach are *Cam Trees*. Here the cones are arranged horizontally, and the tree of cones grows from left to right, as shown in Figure 2.9. The advantage of this rotation is that the text labels are more readable and can contain larger text.

When a node is selected, the cone trees rotates so that the selected node and each node in the path are brought to the front. Furthermore every node in the path is highlighted. The rotations in each level from the selected node up to the top are performed in parallel and are smoothly animated. The whole animation is deliberately designed to take about one second, so that the human perceptual system can track it and the user always knows what is

happening, without having to re-assimilate.

Cone trees uses the available screen space more effectively than conventional 2D tree views due to the three dimensional visualisation. Screen space used by 2D tree views grows almost exponentially. Cone Trees uses depth to fill the screen with more information. The 3D perspective view provides a fish-eye view of the information. The selected path is larger, closer and highlighted. All other nodes are displayed distorted because of the 3D perspective. So we get a fish-eye effect just like the fish-eye camera lens. This effect presents the selected path prominently. Also the information related to the path is visualised well. All other information is hidden or just a small object in the background. The big advantage is that only potentionally interesting information is presented to the user. All other information is visualised in a way that does not distract the user. The user also gets extra information about the structure by the shadows of the cones on the floor. It helps the user to understand the hierarchic structure. The whole visualisation reduces the cognitive load by exploiting the human perceptual system. The interface helps the user to understand the structure just by exploring it. When the user understands the structure, navigation becomes faster and easier.

2.7 File System Navigator

The File System Navigator (also FSN or Fusion) is a 3D file browser for UNIX file systems. The technology was developed and patented by Silicon Graphics [ST96a] [ST96b]. FSN uses a landscape metaphor to display the hierarchy. Directories are displayed as 3D square pedestals on a plane. The height of the pedestal depends on the number of files in this directory. The hierarchical structure is visualised with connecting lines between the pedestals on the ground plane. Looking from a top view, the directory pedestals are laid out just like in a conventional 2D Tree View (see Figure 2.10).

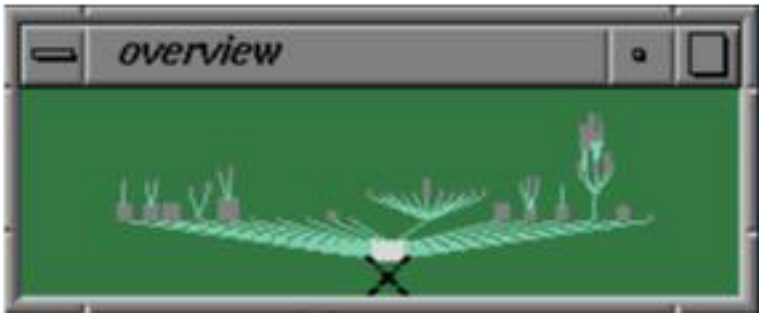


Figure 2.10: The overview window of the FSN.

All the files in a directory are displayed as 3D columns on the directory's pedestal (see Figure 2.11). The height of a file column is proportional to the file's size, the colour represents the file's age. An icon is mapped on top of the column to show the file type. Selecting a file column with the mouse highlights it with a spotlight. Double clicking on a file opens it with an appropriate viewer or editor.

To navigate through the landscape the user can choose among four different modes:

Free Flight: By clicking and moving the mouse you can move through the 3D landscape. When clicking on a connecting line between two directories, the user is automatically move to the subdirectory.

Zoom Navigation: By clicking on an object in the 3D space, the camera zooms to this object. This navigation method allows very fast navigation in large hierarchies.

Marker Navigation: The user can mark interesting or often used places. By choosing a marker from a list, the user is immediately moved to its viewpoint.

Warp Navigation: In this mode only the selected directory and its files are displayed. To move to subdirectories, labels on the floor have to be selected. These labels act as hyperlinks to the subdirectories.

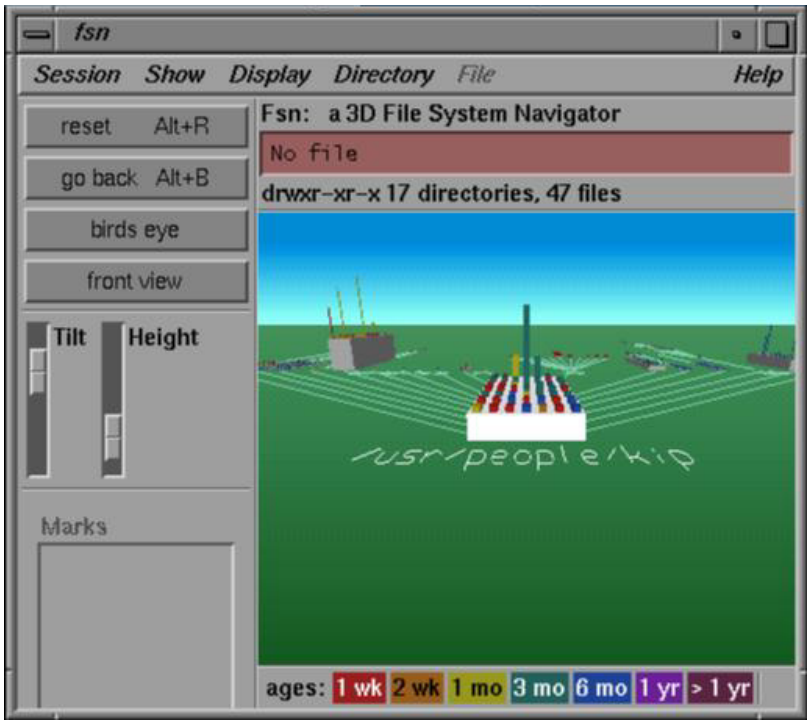


Figure 2.11: The FSN main window.

The mapping of attributes like size and type to visual representations simplifies navigation. Since each directory gets its specific look which is easy to recognise by the user. For instance, large files can act as landmarks, so the users easily know which part of the hierarchy they are. Due to the 3D perspective, the user's view is focused on the selected directory and its subdirectories. All other, probably less interesting directories are smaller objects towards the horizon or are invisible. So the user is not distracted by uninteresting objects. To focus on a directory is easy, but for an good structural overview, a separate overview window is needed (see Figure 2.10).

2.8 Harmony Information Landscape

The Harmony Information Landscape [Eyl95] [Wol96] was a part of the Harmony project at the IICM. This tool has some similarities with FSN (see Section 2.7), but it was designed to display the hierarchy of a Hyperwave server. It also uses a landscape metaphor, i.e. 3D objects are laid out and connected by lines on the floor. The directories (collections) are also displayed as pedestals. Files (documents) are laid out atop these pedestals, but they have different shapes according to their type (see Figure 2.12). The size of the 3D icon corresponds to the size of the document. The mapping of document attributes to visual representations are user configurable. For instance, the document type can be represented by a texture applied to the 3D object.

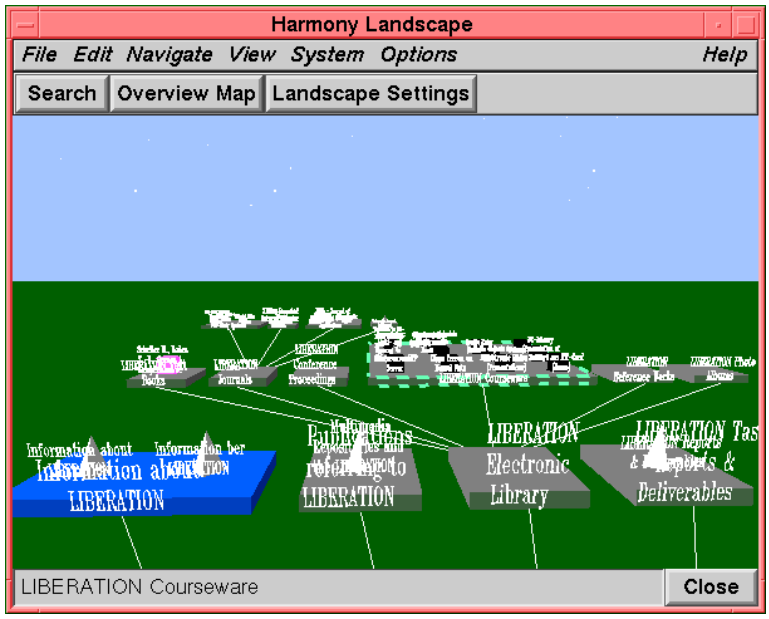


Figure 2.12: The Harmony Information Landscape.

Due to the fact that the information source for this visualisation is a Hyperwave server, which stores hyperlinked documents, the Harmony Information Landscape has some features for visualising the hyperlinks. Objects with a link to a specific object are displayed in a plane beneath the object. Objects which are reachable through links from this specific object are displayed a plane above the object (see Figure 2.13).

Lines connect the specified object in the landscape with the objects above and beneath the landscape. If the original position of a linked object is visible in the landscape, then there is also a connecting line to this object so the user can see where linked objects are located in the information hierarchy. The user can also open paths to all collections with the original objects in the landscape.

The user can move around the landscape freely with the mouse. With the different mouse buttons the user can control the direction and velocity of motion. Collections can be selected with a single mouse click, and opened by a double mouse click. Clicking on an object with the middle mouse button moves the user in front of this object. By clicking with the right

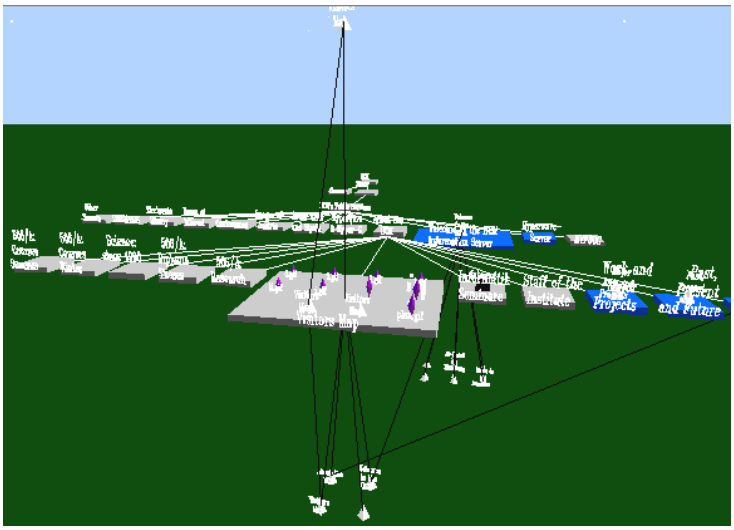


Figure 2.13: Displaying hyperlinks in the Harmony Information Landscape.

mouse button, the document attributes are shown. An overview map is also available to help navigation in large hierarchies. Also the hyperlinks can be used to navigate in the 3D landscape. Clicking on a connecting line between an object above or beneath the landscape and the original object in the hierarchy, moves the user to original object.

Large hierarchies are clearly laid out in the Harmony landscape. The visualisation of the hyperlinks is solved not as good, for example if many hyperlink visualisations are opened, the whole visualisations becomes distracting. The text labels of the each object also distract the user. Sometimes text labels overlap or hide objects behind them.

2.9 Gopher VR

Gopher was one of the first systems to access multimedia documents easily on the Internet. Now it is more or less obsolete, because of the success of the World Wide Web. To visualise the hierarchical structure of a Gopher server, the GopherVR client was developed. This program uses a slightly different landscape metaphor. Only the current hierarchy level is visualised. This level is displayed as a plane, with the documents in it as 3D objects. The shape of the objects depends on the document type. At the centre of the plane is a special 3D object. Clicking on this object takes the user to the parent of the current level. All other documents are arranged in circles around the central object (see Figure 2.14). With the Gopher+ protocol the server administrator can lay out the documents freely, for instance related documents could be grouped in clusters.

Clicking with the right mouse button on an object opens this document in an external viewer. If the object represents a subdirectory, GopherVR moves to the new directory. By pushing the buttons at the bottom of GopherVR window, the user can zoom or rotate around the centre. Selecting the menu item Overview moves the camera automatically to a position high above the plane to get an overview and back again. By selecting Up or Down the camera moves a fixed amount up or down. An interesting navigation idea is to jump. By pushing the

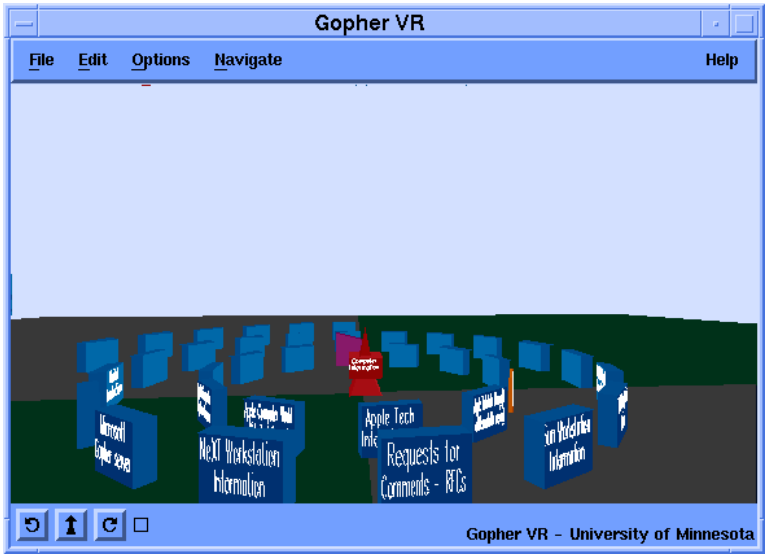


Figure 2.14: GopherVR - the red, pyramidic shaped object in the middle is the “parent link”.

middle mouse button the user jumps in the air, to gain a better overview.

The visualisation is very simple and clear. For instance objects are displayed with an text label only when they are near the camera. The ability of GopherVR to cluster objects structures the information very good, but the clusters have to build manually. The major disadvantage of GopherVR is that it only displays one level of the hierarchy at a time. The user does not get an overview over the whole hierarchy. Also the unconventional navigation methods are not very intuitive and do not help to reduce the cognitive load.

Chapter 3

3D Graphics and Java

This thesis describes a Java implementation of the Information Pyramids technique. The Java language was chosen for its cross-platform appeal and the ability to convert the application into a Java applet running in a web browser. Unfortunately there was at the time no 3D library available for Java. Therefore an existing 3D library was utilised for the application. To access the library the Java native interface has to be used. The advantages and disadvantages of this approach are discussed here.

3.1 3D Graphics

3D computer graphics has found its way into numerous of applications. From simple graphic programs and games on personal computers to modelling and visualisation software on supercomputers. Writing software for this variety of platforms leads to the need for a widely accepted 3D graphics API. There are several APIs in use, which provide such an interface.

A well known one is PHIGS [ANS88] [ISO87] and its descendant PHIGS+. The acronym PHIGS means *Programmers Hierarchical Interactive Graphics System*. It provides a means to draw and manipulate 3D objects by encapsulating object descriptions and attributes into a display list, which is referenced when the objects are displayed or manipulated.

PEX extends the X Window System [SG86] to manipulate and draw 3D objects. It originally is based on PHIGS, but it allows immediate mode rendering. This means objects can be displayed as they are described, rather than having first to complete a display list. The main disadvantage of PEX is that it is based on the X protocol and so it is only available on X Window Systems.

Renderman is an API for rendering 3D objects in a photo-realistic manner. It provides a programming language to describe how objects appear when drawn. This allows very realistic looking images, but is impractical to implement on most graphics hardware in real time. So Renderman is the API of choice for realistic images, but a bad choice for interactive 3D graphics.

The current industrial standard API for 3D graphics is OpenGL [Ope92]. It is a simple, direct interface to operations of 3D graphics rendering. OpenGL is supported on almost every platform. Since the OpenGL library is used for this thesis it is described in more detail here.

3.1.1 OpenGL

OpenGL is the successor to the Silicon Graphics IRIS GL library [Sil90] which made SGI workstations so popular. IRIS GL was an API specially designed for SGI workstations. But SGI realized the importance of open standards. Several software and hardware makers took part in specifying an open version of IRIS GL. The Architectural Review Board (ARB) oversees the OpenGL specification, accepts or rejects changes, and proposes conformance tests.

In contrast to the IRIS GL, the OpenGL library was designed to be platform and operating system independent. To achieve this independence, all functions for windowing tasks as well as functions for user input were excluded. Special care was taken to easily combine OpenGL with other, platform-dependent, programming libraries, which handle windowing and user input.

Furthermore, the OpenGL library was designed as a streamlined, high performance graphics rendering library. The design is also very hardware near to achieve a reasonable performance. OpenGL is in fact a rendering pipeline, so parts of the pipeline can be implemented either in software or hardware. Many OpenGL primitives can be easily implemented at the hardware level, which of course improves the display speed. Therefore, OpenGL also defines only very primitive geometric objects (points, lines, and polygons). There are some higher level APIs which support methods to describe higher-level graphical objects, for instance Open Inventor.

OpenGL is designed as a *state machine*, which draws primitives to a frame buffer according to the state of the machine. More than 150 selectable modes can change the state of the machine. Each mode is set independently, setting one does not affect others, although modes interact to determine what is drawn into the frame buffer. Almost all state of the art rendering parameters can be set by different modes. This includes the options for the virtual camera, the light sources, and antialiasing. A primitive is a point, line segment, polygon, pixel rectangle, or bitmap, which are the input to the OpenGL machine as shown in Figure 3.1. Of course there are some functions to perform other OpenGL operations, which do not change the machine state. The output of the OpenGL machine is stored in a frame buffer, which is a memory area to be displayed on the screen. OpenGL supports double buffering, for smooth animations: while OpenGL draws in one frame buffer a second one is displayed, when the drawing process ends the buffers are switched.

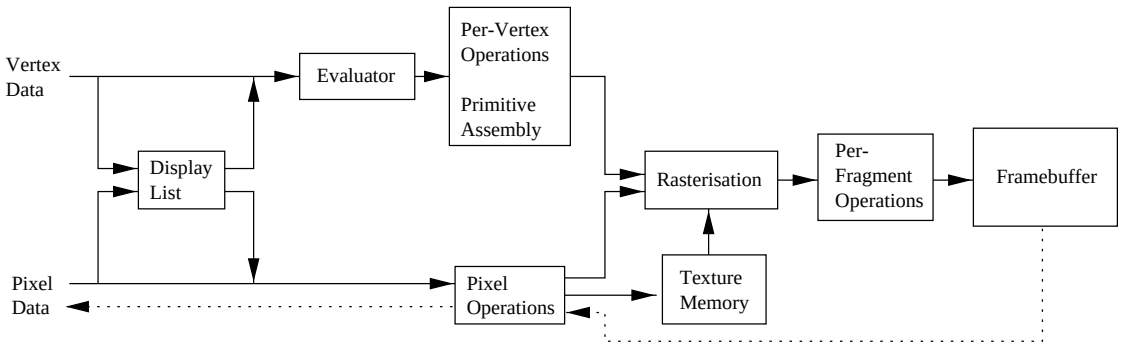


Figure 3.1: Block diagram of the OpenGL pipeline.

As shown in the block diagram in Figure 3.1, commands enter the OpenGL pipeline on the left. It is possible to accumulate commands in a display list to process them later. Otherwise, the commands are sent directly through the processing pipeline. The first block provides a means for approximating curve and surface geometry. The next stage processes primitives described by points, line segments, and polygons. Vertices are transformed and lit. Primitives are also clipped here to the viewing volume. After that the rasterisation begins. This block produces a series of frame buffer addresses and values. These are called fragments, which represents a portion of a primitive that corresponds to a pixel in the frame buffer. The fragment encapsulates the coordinates of the pixel, the colour, depth and if used the texture coordinates. Texture coordinates are calculated using the texture memory. After building the fragments, they are processed in the next stage. Operations on a fragment could be blending of fragment colours as wall as masking or other logical operations (see Figure 3.2). The most important operation is a conditional update of the frame buffer based on the incoming and stored depth values of the fragments.

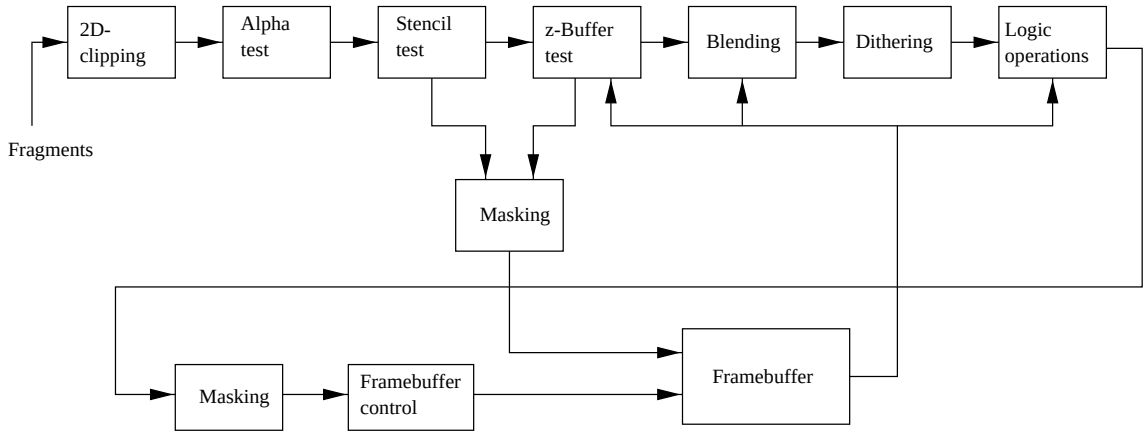


Figure 3.2: Detailed block diagram of the fragment operation block in Figure 3.1.

Pixel rectangles and bitmaps (i.e. ordinary 2D images) bypass the vertex processing. They are fed directly to the pixel operations block. Then they are sent through the rasterisation to the fragment operation block. Pixel data is thus also rasterised to fragments. Fragments are treated the same no matter if they come from a geometric primitive or an image.

It is also possible to read back values from the frame buffer. So frame buffers or portions of them can be copied. These transfers could also include some type of encoding or decoding.

3.1.2 GE3D

The GE3D library - Graphics Engine 3D - is a machine-independent, immediate mode, 3D graphics interface. It was developed at the IICM[1] as a higher level 3D graphics API atop of low level APIs like OpenGL or Mesa (see Figure 3.3). Thus it improves the portability of

[1]Institute for Information Processing and Computer Supported New Media (IICM), Graz University of Technology

applications. Since the header files of the library are completely independent of any other header file of graphic interfaces another implementation of the GE3D library can be linked at any time without changes.

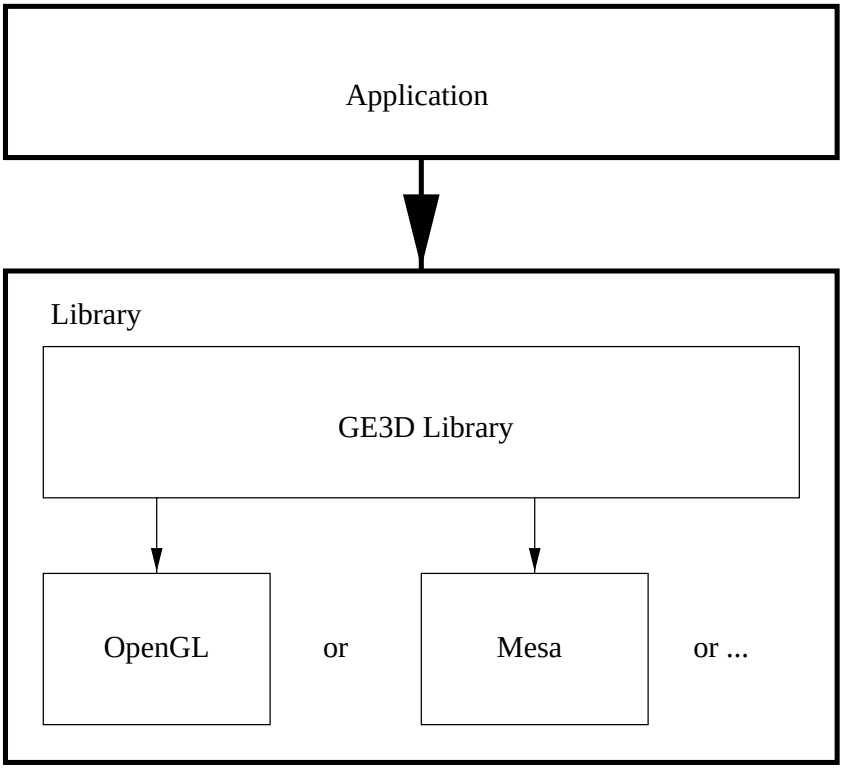


Figure 3.3: The structure of the GE3D library.

The functionality of GE3D includes:

- manipulation of vectors and matrices
- camera definition
- definition of light sources
- double buffering
- drawing in wire frame, hidden line, flat shaded and smooth shaded
- drawing of 3D faces
- drawing of 2D primitives

As it can be seen from the list above, GE3D provides higher level functionality and also some low level functions. These are functions such as drawing lines and rectangles and manipulating the transformation matrix stack. These functions have a corresponding counterpart in the lower level graphic libraries.

Currently only OpenGL and Mesa [Pau] are used as low level libraries. While OpenGL is the industrial standard for 3D graphics, the Mesa library is a free implementation of OpenGL available for almost any platform. The latest releases of Mesa now also support some 3D hardware acceleration.

3.1.3 Java3D

Java is a powerful and platform-independent programming language. Java includes APIs for windowing (awt package), for input and output (io package) and for other basic tasks. There is a strong effort to create standard APIs for every imaginable task, to preserve the "write once - run everywhere" nature of Java. To address the need of full integration of multimedia the Java Media API was created. It integrates the following technologies into Java:

Java2D: Extends the JDK AWT package to include line art, images, color, transforms, and compositing.

Java Media Framework: Specifies an architecture for media players, media capture and conferencing.

Java Collaboration: Allows multi-party communications over a network.

Java Telephony: Integrates telephones with computers.

Java Speech: Provides Java based speech recognition and speech synthesis.

Java Animation: Provides for motion and transforms of 2D objects.

Java3D: Provides an abstract, interactive imaging model of 3D objects.

The recent release of Java3D API provides a programming interface for standalone 3D graphics applications such as 3D modellers and viewers, or web-based 3D applets. It is the result of a collaboration between Sun Microsystems, Apple Computer, Intel Corporation and Silicon Graphics (SGI). Each of these companies is developing low-level 3D APIs. Java 3D is meant to be a high-level, cross-platform 3D API atop a native low-level API.

Of course Java 3D has to perform well to compete with other APIs. Many design decisions were made so that Java 3D can deliver the highest performance to application users. Internal optimisations of the API allow tuning the Java3D implementation for specific hardware. The application programmer uses the high-level methods to build this application, without concern about optimising the code for different platforms. The API follows a high-level, object-oriented paradigm. It is fairly easy to write sophisticated applications with the rich set of 3D features provided by Java3D.

Java3D assumes a double-buffered, true-colour, and Z-buffered rendering model. The underlying low-level API is responsible for providing these minimum requirements. How the low-level API implements these features, whether by software or in hardware, is unimportant to Java3D. In addition to these basic requirements Java3D defines three rendering modes:

Immediate Mode: This mode is almost equivalent to a low-level API. Drawing methods with sets of points, lines, or triangles are used here. This approach leaves only very little room for internal optimisation of the rendering.

Retained Mode: Using this mode the application must define a scene graph and specify which elements of the graph may change during rendering.

Compiled-Retained Mode: Like in retained mode, a scene graph is required and the elements which may change during rendering have to be specified. Additionally, the application can compile parts of the graph. Compiling means, that Java3D transforms the graph into an internal representation, which is functionally equivalent to the scene graph. These compiled representations are highly optimised.

It is recommended to use the retained and compiled retained modes to take advantage of the convenience and the performance benefits these modes provide.

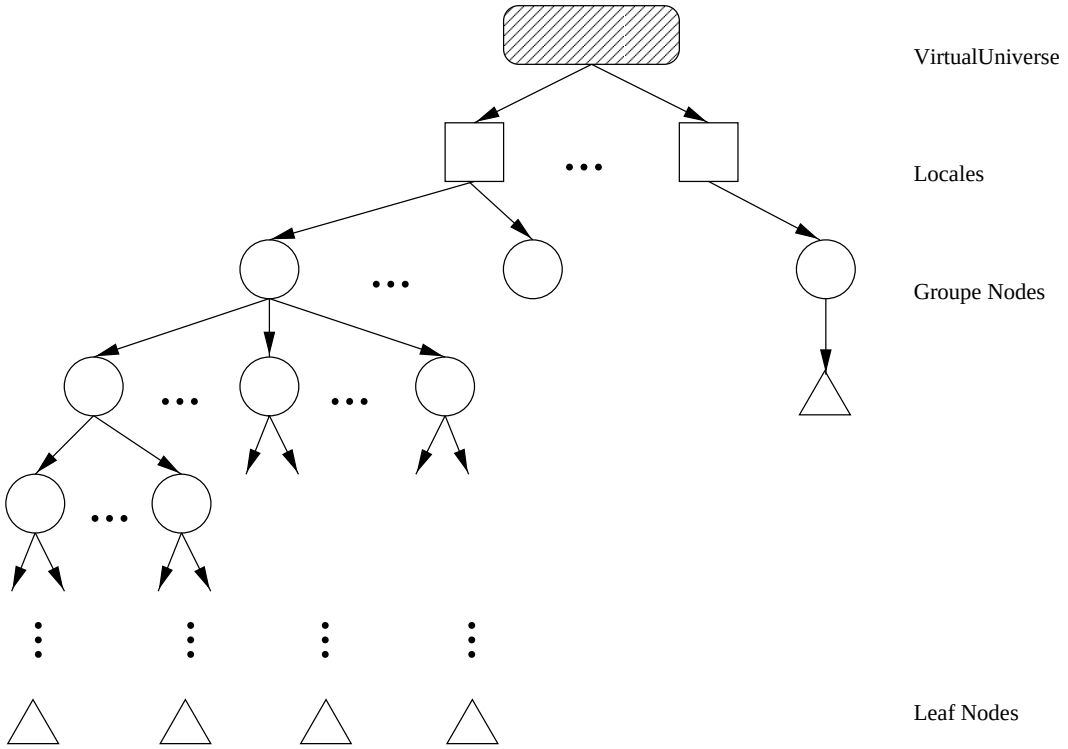


Figure 3.4: A Java3D API scene graph.

A scene graph is a simple and flexible way to represent and render complex 3D environments. It contains a complete description of the virtual 3D universe. As can be seen in Figure 3.4, the scene graph is a directed acyclic graph, whose root node is a *VirtualUniverse* object. It is possible to create multiple universes, but most applications will use only one. This node provides the base of the scene graph. Every scene graph must be connected to a *VirtualUniverse*. The children of a *VirtualUniverse* are *Locale* objects. These objects define the origin of the the attached subgraphs in high resolution coordinates. A *VirtualUniverse* can hold as many *Locale* objects as needed. The scene graphs itself start with *Group* nodes. *Group*

nodes have a variable number of child nodes including other *Group* nodes, but exactly one parent node. Many operations on *Group* nodes are possible, such as adding, removing and enumerating the children of the node. Many different *Group* nodes for special purposes are available, like the *BranchGroup* node. The *BranchGroup* node is the root of a subgraph of a scene, attached to a *Locale* or another *Group* node. Another *Group* node is the *LOD* node, which contains an ordered list of children and a level-of-detail value for every child. At the end of every branch there are *Leaf* nodes attached. These nodes are simply nodes which have no children. *Leaf* nodes specify things like lights, geometry, sound, and a view platform for positioning and orienting a view in the virtual world. Very important *Leaf* nodes include the *Behavior* nodes, which provide the means for animating objects, processing keyboard and mouse inputs, reacting to movement, and enabling and processing pick events. These node contain Java code to interact with Java objects; values within a Java3D scene graph can be changed, and other computations performed.

3.2 The Java Programming Language

3.2.1 History

The Java programming language was developed by Sun Microsystems in 1991. Back then Sun did a research project to develop software for consumer electronics, like TVs, VCRs and other household appliances. For this purpose a small, fast, efficient and easily portable language was needed. This language was also to use the modern object-oriented programming paradigm. Java is not only a language, it also has a “write once, run anywhere” concept for writing programs which can be executed on many different hardware platforms.

Java was used by Sun in many different projects, but did not gain much public attention. In 1994 Sun made the HotJava WWW browser publicly available. This browser was the first one which was able to execute Java applets. Applets are Java programs embedded in HTML pages. With applets it was possible to include interactive elements in an ordinary WWW page the first time. Also the entire HotJava browser was written in Java, to show that also serious applications can be written in Java. All these features made Java a hot topic on the Internet.

The next step was the release of the Java Developers Kit (JDK). The JDK contains the Java compiler, the Java runtime environment, the basic Java classes and some other development tools. The JDK is also the reference implementation of the Java language. With the JDK it was possible to develop applets and applications in Java. After the release of the JDK other software companies developed higher level development environments for Java, like Visual Cafe from Symantec or JBuilder from Borland.

Today, Java is accepted as a very flexible and portable programming language. However, until today there is no “killer application” which has established Java as a language for large software projects. To reach this goal, Java is still under heavy development. That does not mean that the language is being changed, but rather that many different APIs (Application Programming Interface) are currently being developed, to achieve standardised interfaces for almost every programming task. For example the “Internet Foundation Classes” (also known as “Swing”) were developed to provide an API for building user interfaces. Also many other

APIs are still under development, like the Java3D API for 3D graphics in Java, or the Java Media Framework for displaying different types of multimedia data.

There are also some efforts to bring Java back to its roots. Special versions of Java are planned for consumer electronics. There is also a project to use Java on smart cards (the JavaCard project). Sun is trying to establish Java as the language for almost every imaginable task, but still has a way to go to reach this goal.

3.2.2 Java's Platform Independence

The biggest advantage of Java over other languages is its platform-independence. That means that a Java program can be executed on every type of computer hardware and operating system which supports Java. A Java application can be developed using a standard IBM-PC and Windows NT. The application can than be executed on an Apple Macintosh or a Sun SPARC Workstation without any changes. Java is thus platform-independent at the source code and binary level. This ability of Java is also known as the "write once - run everywhere" feature of Java.

At the source code level no special code for different platforms has to be provided. In C or C++, programmers use macros and define statements to make to source code platform independent, for instance when defining new primitive data types. The types have the same size on every supported platform. As an example the type int32 is defined as a 32 bit integer. So for every platform a define statement maps the int32 to the appropriate 32 bit integer type. In Java all primitive data types have the same size on every platform. Also the standard Java class libraries are alway the same on every platform.

Platform-independency does not stop at the source code level. Java binaries are completely platform independent. Consider the compile process in ordinary languages like C or C++ (see Figure 3.5). A C++ complier translates the source code to machine code. This code is specific for a certain CPU type. For instance, compiling a source code on a Pentium system produces a binary file which is only executable on Pentium systems. To create an executable file for a different platform the source has to be recompiled for this new platform. Of course, a special compiler for this platform is needed.

In Java a different concept is used (see Figure 3.6). The Java source code is compiled by a Java compiler which produces *bytecode*. This bytecode is machine code for the Java Virtual Machine (also JVM). This JVM is not a real processor. Until now there is only a written specification for this machine. Although some companies have announced their intention to build a real processor upon this specification. To execute the generated bytecode an interpreter is used. The bytecode interpreter transforms the bytecode into machine code for the CPU of the platform, and then executes it. Today, a JVM is available for almost every hardware platform.

The major disadvantage of this concept is the issue of performance. Programs which are compiled for a specific system run faster than Java code on the same system. The reason for this is that platform-specific programs can be directly executed on the machine. Java code must first be interpreted and only then can it be executed. Lack of performance is the biggest obstacle to Java establishing itself as a language for every purpose. To combat this performance disadvantage some techniques have been developed to speed up Java code. Many

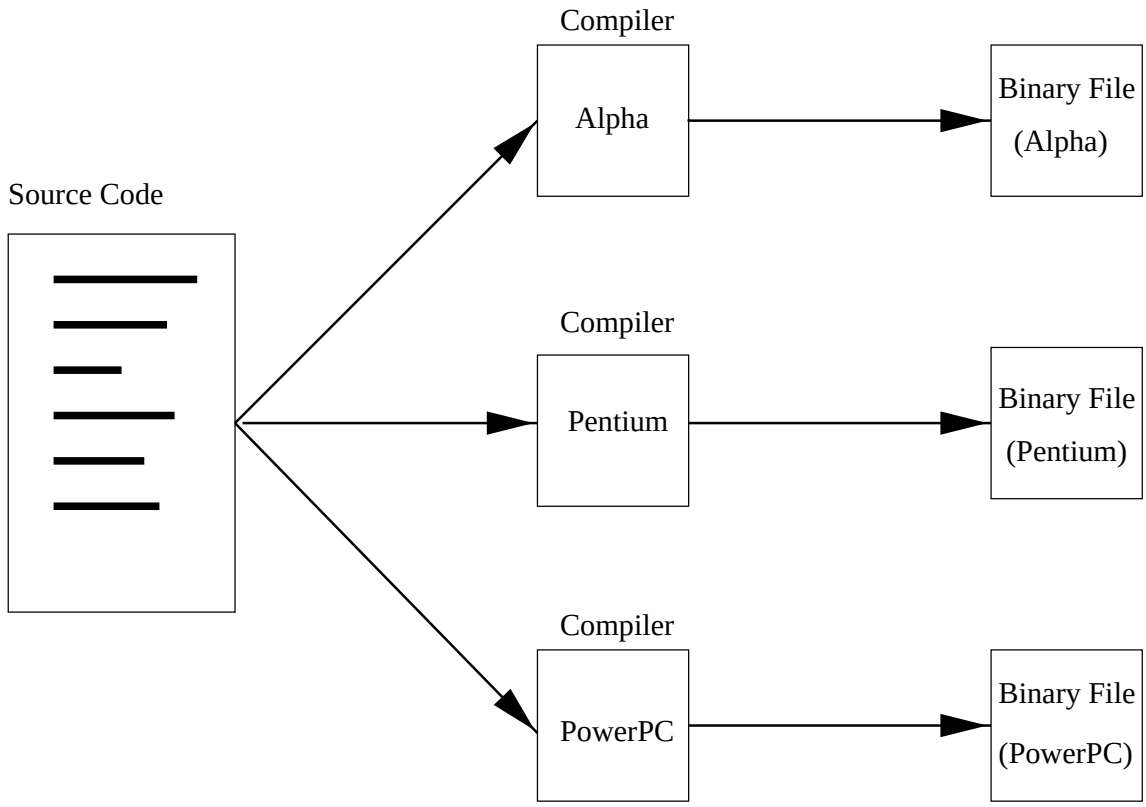


Figure 3.5: The steps to compile programmes for different platforms with ordinary languages like C.

different software companies have developed *Just In Time compilers* (or JIT compilers). A JIT compiles the the Java bytecode in advance during interpretation. For example, with this technology it is not necessary to interpret a loop in the program every time. The loop is compiled by the JIT compiler once and thereafter can be executed directly. A further improved JIT compiler is the recently announced *Hot Spot* technology from Sun, which will speed up Java even more. There are also direct compilers from Java to machine code, which create a platform-dependent binary directly.

3.2.3 The Java Native Code Interface

This interface is a standardised way to access non-Java code. This feature is not special to Java. For instance, the qualifier `pascal` in C signals use of the Pascal calling convention, rather than the C calling convention. Most other languages provide a mechanism to call methods written in other languages, for instance Pascal or FORTRAN. In Java native methods are used for this purpose. This methods act like ordinary Java methods, but they are implemented in non-Java code.

Native methods are called by other classes just like any other 100% Java code. The caller does not even realise it is calling a native method. The calling convention is the same as for any other method. Modifiers such as `public, private, protected`

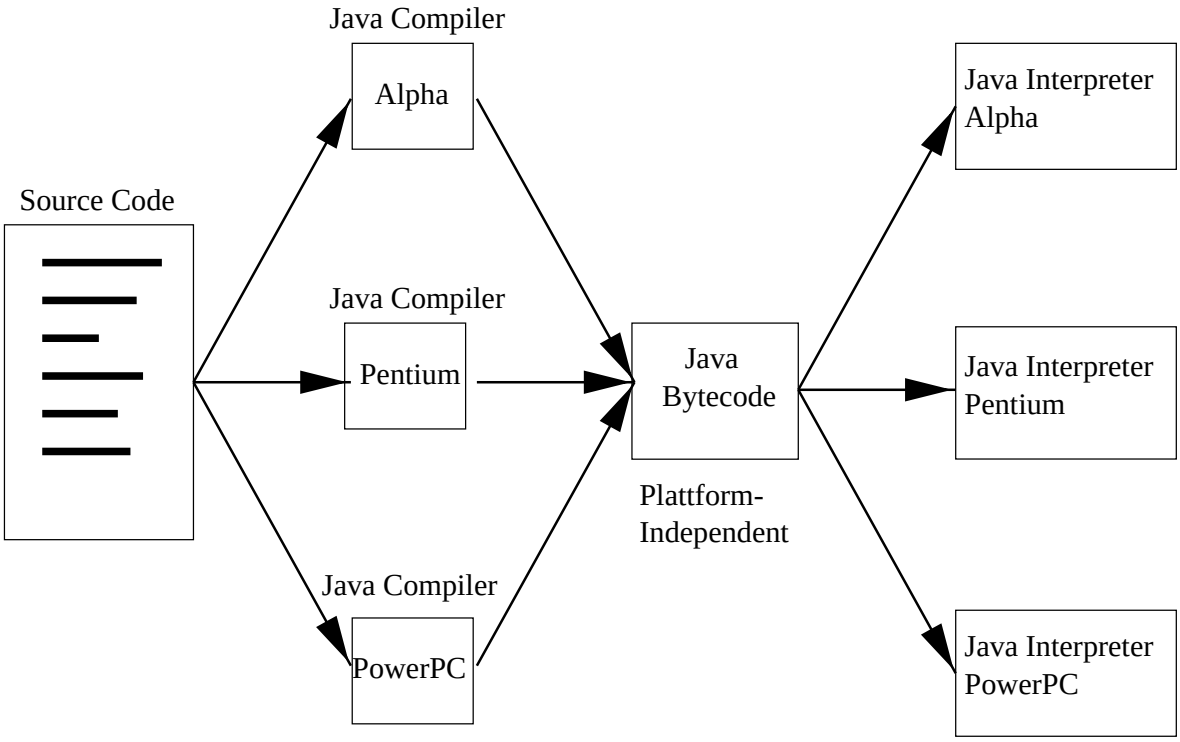


Figure 3.6: Compiling and running Java programs.

and `synchronized` act as they do for a normal Java method. The Java Virtual Machine handles all the visibility and synchronisation details when a native method is called. In the case of a `synchronized` method, the JVM will perform the monitor locking prior to entering the native code. Type checking of the method's parameters is also done by the JVM.

Native methods are a very powerful, elegant, and effective way to extend the Java virtual machine. In fact, current implementations of the JVM use native methods in many places. For instance, the Abstract Windowing Toolkit (AWT) uses native methods to call the operating system functions to manipulate windows and their components.

The well-defined Java virtual machine is a system on which every Java program can rely. The JVM supports the Java language and its runtime library. The JVM is not a complete system, it relies on existing system resources such as a web browser or an operating system. The JVM accesses the operating system using native methods. For instance, the Abstract Windowing Toolkit (AWT) accesses operating system functions to handle windows, so it gets the look and feel of the window manager a particular platform.

Native methods allow access to custom hardware. Any imaginable hardware such as hardware accelerated 3D graphics cards can be accessed by Java through native methods. Also Java virtual machines can be ported to Set-Top-Boxes or PDAs. To access such special-purpose hardware, native methods are the best solution.

As discussed previously, Java programs dose not perform as well as native programs written in C or another ordinary language. This fact has two reasons. First, the performance trade-off caused by garbage collection, the security model, and the dynamic nature of Java. Second, the typical implementation as an interpreter. The performance issue is becoming

less pronounced as the implementation of Java technology improves. There will be alway a technology overhead, which will force programmers to use native methods in some circumstances. Native methods can be implemented at a very low system level and languages like C or even assembler can be used to achieve maximum performance. So an application can be written mostly in Java, with the time consuming computations written with native methods.

Java is targeted as a platform neutral language. This is an big advantage. But it implies also the disadvantage of not having access to the system specific features some applications might need. Many of these features are encapsulated in libraries. With the native code interface these libraries can be accessed by Java. As an example such libraries could be 3D graphics libraries like OpenGL or GE3D. It is unlikely that such existing libraries have functions and parameter types Java expects, thus it is recommended to write a *glue library* which sits between the Java code and the existing library (see Figure 3.7). This glue library has the library functions with the name and parameters Java expects. The functions in this library just cast parameters to the right type, if necessary, and the call the functions in the existing native library.

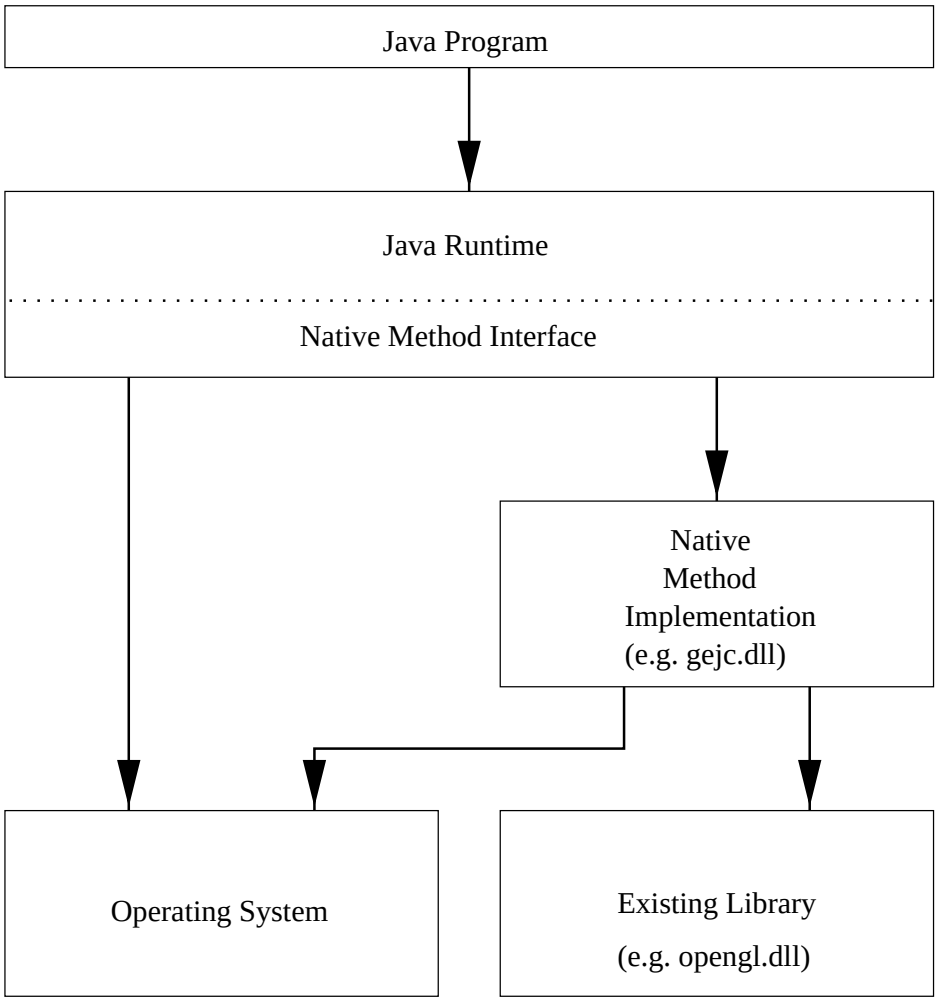


Figure 3.7: A Java application using native methods of an existing library.

In the Java code native methods are marked by the `native` modifier, for example:

```
protected native boolean setContext (boolean shading);
```

With this statement the method `setContext` is defined. Instead of the method body a semicolon is in its place. The method body is implemented in a separate C file. To implement the method in C, the Java Development Kit provides header files and C libraries. In this files some types, macros and functions are defined to access the Java classes and types easily in C. Unfortunately this interface was not very flexible and well-defined in the 1.0 release of the JDK. Some other major companies like Netscape and Microsoft implemented a different interface to the original Sun JDK. To standardise also the native interface of Java the "Java Native Code Interface" (JNI) was introduced with JDK 1.1. The JNI is more flexible and easier to use than the old JDK 1.0 interface.

To access the native method, the dynamic linking capabilities of the underlying operating system are used. That means, that the native code has to be compiled as a dynamic link library. On Unix platforms such libraries are called *shared objects* or *shared libraries* (with file extension *.so*). On Win32 platforms, they are called *dynamic link libraries* (with file extension *.dll*).

Before a native method is invoked, the Java virtual machine has to know which library it should load and link. This is done by calling the static method

```
java.lang.System.loadLibrary(''mylib'');
```

The parameter of this method is the name of the library to load. Java maps the name to the expected filename. On a Win32 system this would be `mylib.dll` and on Unix systems `mylib.so`. After that Java loads the library, but it will not resolve the symbols of a method until its point of use. If a symbol is resolved correctly, a standard C call to this resolved function is made. If the resolution fails a `java.lang.UnsatisfiedLinkError` exception will be thrown.

3.3 3D Graphics with Java

When work started on this thesis, there was no standard way or a standard library for 3D graphics in Java. There were, however, some solutions for this problem. There are three possibilities to write 3D graphics applications in Java, which will be briefly discussed. Further the selected solution will be described in detail.

3.3.1 3D Libraries in Java

To do 3D graphics in Java there are following possibilities:

Java3D library

The lack of an standardised 3D library in Java has brought some major software companies together to create such a library. The goal is to build flexible, fast and portable 3D library called Java3D. The library is designed to lay atop a hardware accelerated layer, like OpenGL

or Direct3D. So the performance eventually will be good. Flexibility and portability should also be quite reasonable, due to the long period of creating a specification for Java3D. Unfortunately this library was not available when the work on the thesis started.

Write a 3D library in Java

Writing a 3D library in "100% pure Java" is a big job. Such a library can not take advantage of 3D hardware acceleration, so it will have poor performance, but would be portable. For almost every 3D graphics application performance is a major issue. Due to the performance of the Java language itself, a software-only solution is too slow for smooth animated 3D graphics.

Using a native 3D library in Java

Using a standard library like OpenGL or GE3D is much easier. The Java native code interface can be used to access the native graphics engine. This solution can also take advantage of any hardware acceleration. The major disadvantage is obviously that the native code library has to be provided for every platform. So the big advantage of platform-independency is lost. Another disadvantage is that a 3D application using this solution can not easily be used as an applet. The native library has to be installed manually before starting such an applet and the applet has to be authorised to allow access to a locally installed native library.

3.3.2 Using GE3D with Java

For this thesis a native 3D library was used with Java, the in-house GE3D library. GE3D is a stable and reliable high level 3D library atop OpenGL or Mesa. It is available and well tested for many different platforms. So the amount of work to support many different platforms was minimised. Another reason to choose GE3D was performance. When calling a native method in Java, a time-consuming context switch to the native library has to be done. To do this for every OpneGL command, the performance decreases rapidly. GE3D provides higher level commands than OpenGL or Mesa. So fewer context switches have to be done, and the performance goes only slightly down.

A simple 3D application with GE3D needs the Java package `iicm.ge3d` and the compiled C library `gejc`. Of course the `gejc` library has to be compiled for every supported platform. So there is a `gejc.dll` file for Microsoft Windows and `gejc.so` libraries for the different Unix platforms like Linux, Solaris, and Irix. Every 3D application using the GE3D native library has the structure shown in Figure 3.8.

To integrate GE3D into Java the package iicm.ge3d was created. This package consists of two classes, one to create a 3D window and one to execute the GE3D commands. Most of the methods of these two classes are implemented as C functions, which are part of the `gejc` library.

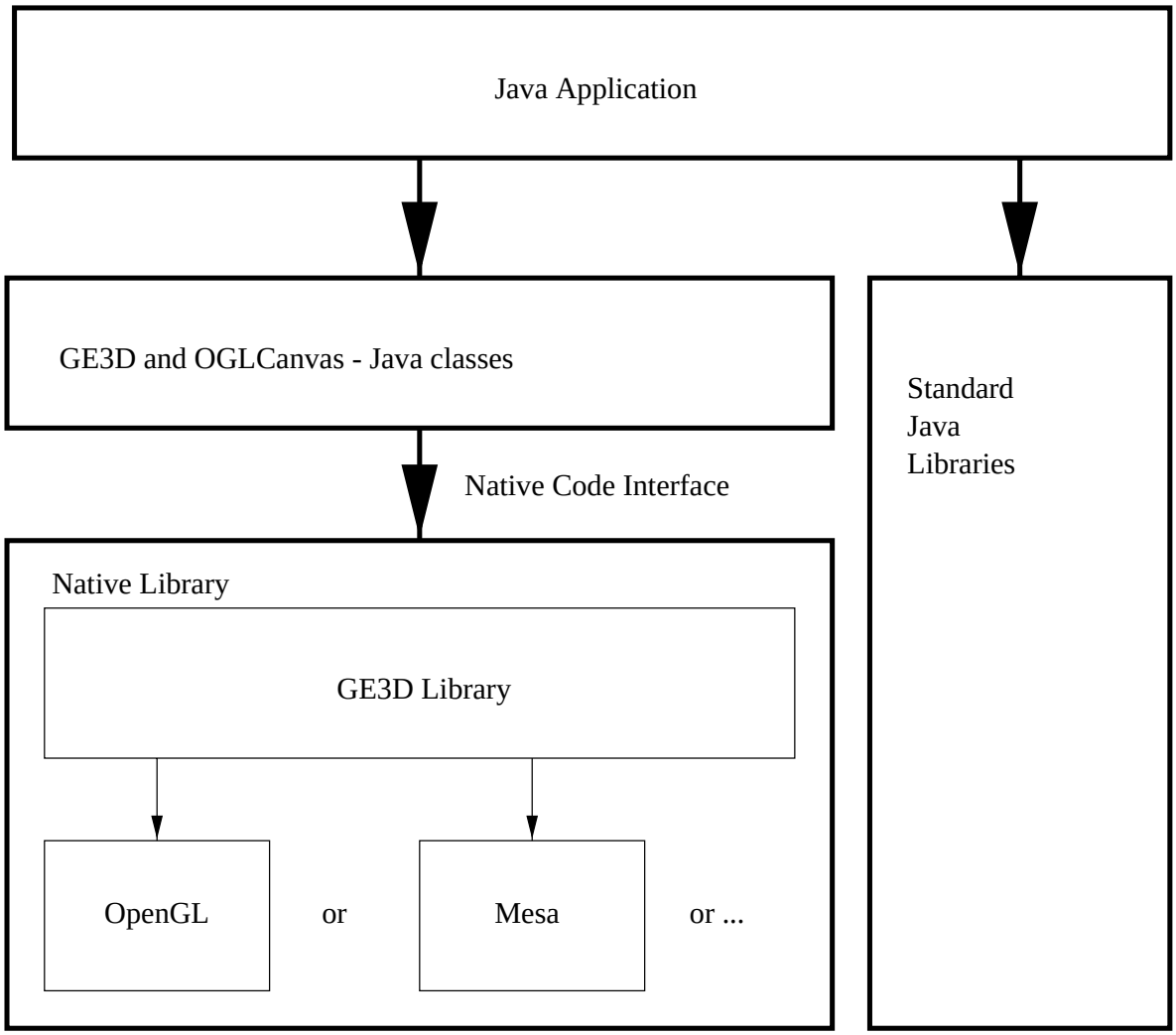


Figure 3.8: Structure of a Java application using the GE3D interface.

The `iicm.ge3d.OGLCanvas` class

This class extends the `Canvas` class of the AWT. It provides a window, into which the OpenGL/Mesa output is directed. In fact this is a bit more complicated. `OGLCanvas` creates a normal `Canvas` window. To initialise this class for 3D graphics, the `createContext` method has to be called. `createContext` is implemented in C. It uses operating system calls to get a pointer to the application window. After that it searches for the `Canvas` created by `OGLCanvas`. Now a new window with the same size and position of the `Canvas` window is opened. This window contains the 3D output and is positioned right atop the original `Canvas` window. If the original `Canvas` is resized or moved, the 3D window has also to be resized or moved. The `paint` method of the `OGLCanvas` class or a derived class has to call the `haveContext` and `setContext` methods. This is usually done like this:

```
public void paint (Graphics gc)
{
super.paint(gc);
```

```
if (!hasContext () || !setContext (true))
{
  ... print an error message, because no 3D
      context available
}
.... paint the 3D objects ....
swapBuffers ();
}
```

The method `hasContext` returns `true` if a 3D context (or window) has been created, `false` otherwise. If it was true the method `setContext` sets this 3D window as the current context for the 3D output. Calling this method every time the `paint` method is called makes it possible to have more than one 3D output window. `setContext` checks the size and position of the original `Canvas` window. If this window has changed size and position `setContext` changes also the size and position of the 3D window to put it exactly atop the original `Canvas` window. `setContext` returns `true` if the context switch was successful, `false` otherwise. If both, `hasContext` and `setContext` return `true`, 3D objects can be painted. If one of these methods returned `false`, an error message has to be displayed.

At the end of the `paint` method, the `swapBuffers` method of the `OGLCanvas` class is called. The 3D context uses double buffering. This means, while displaying one graphic buffer, painting is done in the second buffer. At the end of the painting the `swapBuffers` method changes the buffers. This is done to avoid flicker effects when directly painting on a graphic buffer being displayed.

The `iicm.ge3d.GE3D` class

All the methods of this class are defined as native methods. These methods are simply the Java wrapper methods of the GE3D methods. The native C code of the Java methods simply converts variable types if necessary and calls the appropriate GE3D method. All methods are executed in the current 3D context. Which means that the methods draw into the window which was set current by the `setContext` method of the `OGLCanvas` class as described above. Not every GE3D method is accessible in Java (see Table 3.1), because the implementation of the methods is done on a “is used” strategy. So only methods which are used by applications right now are implemented to keep the maintenance work for the native interface as low as possible.

Category	Methods Implemented in the Java Interface
Basic Functions	initGE3D setDrawMode setBackgroundcolor clearScreen hint antialiasingSupport antialiasing
Transformations	rotate3f translatefff
Matrix Operations	loadIdentity pushMatrix pushThisMatrix popMatrix getMatrix
Camera Setting	setPerspectiveCamera simpleOrthoCamera
Lighting	deactivateLights setHeadLight activateLightSource
Color and Material	fillColor3f lineColorRGBi lineColor3f material defaultMaterial lineStyle
Texturing	createPixelTexture craeteImageTexture doTexturing applyTexture textureRepeat freeTexture setTextureMipmapping loadTextureIdentity loadTextureMatrix alphaTest getTextureAlpha
Geometry	drawCube drawWireCube drawCylinder drawFaceSet drawLineSet drawPointSet drawSphere
Other Functions	drawAsciiString drawLine2D drawRect2D drawPolyLines2D drawCircle

Table 3.1: Currently supported GE3D methods in the Java interface.

Chapter 4

Information Pyramids

4.1 Introduction

As described in Chapter 2, there are numerous techniques for visualising hierarchies. All these visualisations have advantages and disadvantages, which affect their usability. Some of them focus on the currently selected subhierarchy and therefore loose the global context. Others do not scale well for large hierarchies. Until now there is no solution which serves all the needs of a perfect visualisation. *Information Pyramids* are a new approach to the problems of visualising large hierarchies. 3D graphics were chosen for the following reasons:

- Compared to 2D graphics more information can be displayed in 3D graphics using the same screen space. When the third dimension is used in a proper way, the visualisation can be compact and nevertheless very clear.
- These days great efforts are made to supply every computer with fast 3D graphics. Graphics card vendors enhance their graphics cards with 3D graphics features, and software companies are supporting these hardware features. 3D graphics will soon become a common feature of every PC, so 3D visualisation tools will not have the performance problems which they have had upto now.
- A 3D landscape is a very intuitive visualisation and is not as abstract as some 2D visualisations. A three dimensional landscape is something we see every day. The visualisation is easy to understand and users need less time to learn to work with the interface.

Information Pyramids are a very simple and intuitive three dimensional visualisation of a hierarchy. A hierarchy is displayed as an pyramidical object in a 3D landscape as shown in Figure 4.1. It provides mechanisms to focus on particular subhierarchies, without loosing context. It also scales very well for large hierarchies. Its simplicity and elegancy makes it very easy for novice users to use.

In describing hierarchies in this chapter, a tree metaphor (see Figure 4.2) will be adopted. Due to the many application of hierarchies, this is useful to keep the description as general as possible. Items of the hierarchy are called *nodes*. The first node of the hierarchy is called

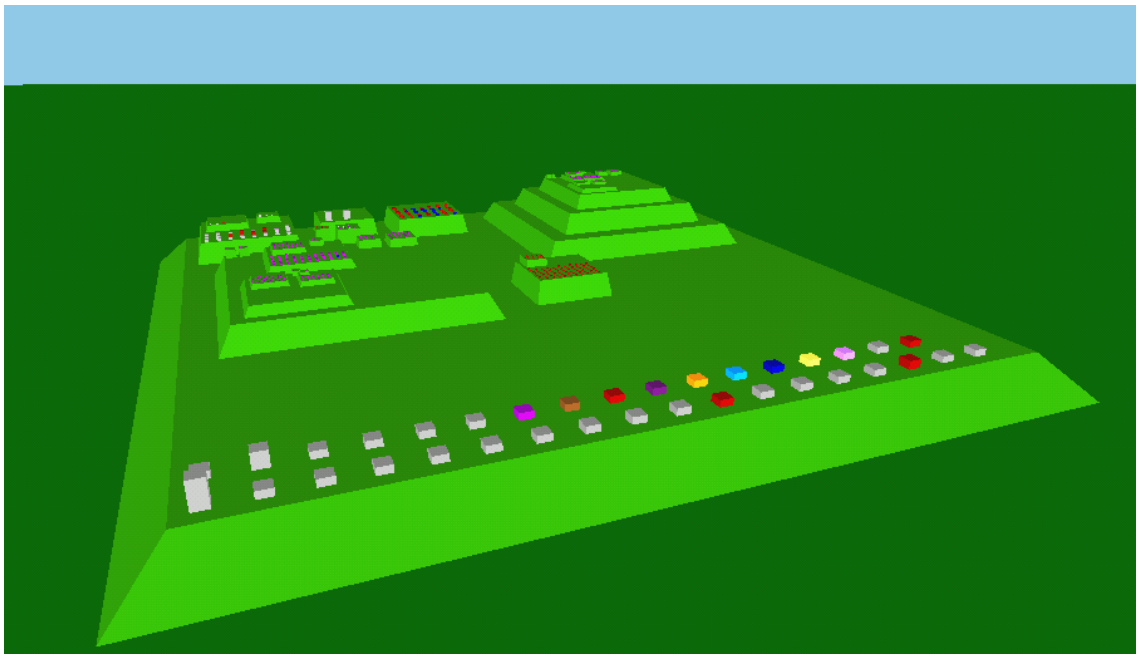


Figure 4.1: Information Pyramids.

the *root node*. *Inner nodes* are nodes which have children, in a file system these would be directories. The subhierarchy starting at an inner node is called subtree. Nodes without children are called *leaf nodes*. In a file system leaf nodes would be files. A *level* of the hierarchy corresponds to a level in a tree.

4.2 Concept

The Information Pyramids approach utilises three dimensions to compactly visualise large hierarchies as a landscape. On a plane a square-shaped plateau represents the top of the hierarchy. Atop of this plateau, smaller plateaus represent the subtrees of the hierarchy. Leaf nodes of the hierarchy are arranged at the bottom left corner of their base plateau. Subtrees are placed starting from the top left corner of their base plateau (see Figure 4.3). While subtrees are represented as plateaus, leaf nodes are represented by small three-dimensional objects.

This layout schema for the base of the hierarchy is used recursively for every plateau representing a subtree. Going down the hierarchy the visualisation of a subtree becomes smaller and smaller. A limit of this approach is of course the range of the floating point numbers used to calculate the 3D objects. The overall impression of the three dimensional layout is that of pyramids or mountains (see Figure 4.1) growing from the base plateau.

Information Pyramids appear at first to be very similar to FSN (Chapter 2.7) or the Harmony 3D Landscape (Chapter 2.8). These visualisations use also a 3D landscape to visualise hierarchies, but they do not utilise the three dimensions as efficiently as Information Pyramids. Large hierarchies visualised by FSN or Harmony result in a very large landscape. To

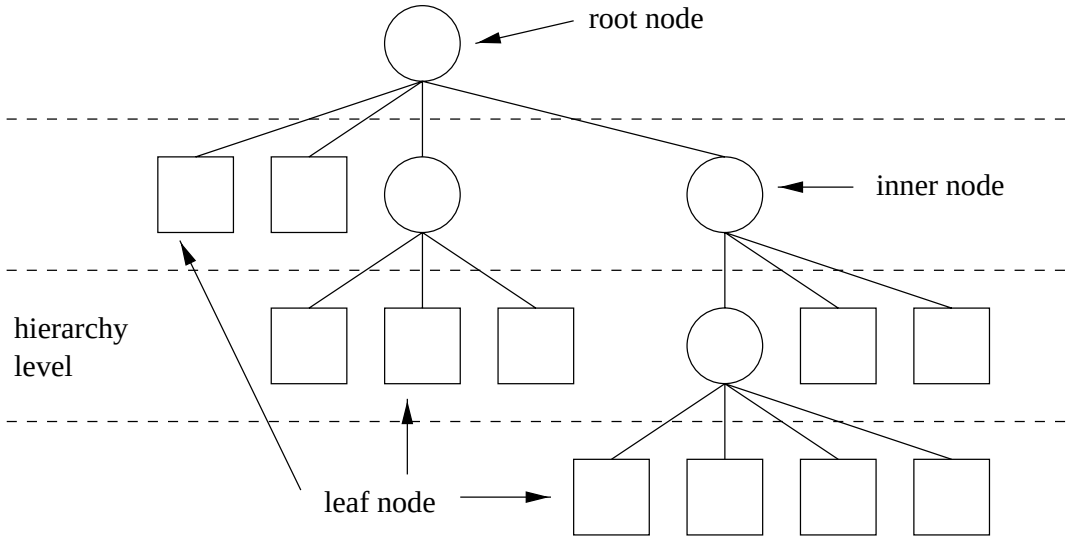


Figure 4.2: A hierarchy an the terms describing it.

navigate in such a big landscape is very time consuming. Furthermore, the user tends to loose global context in such a wide landscape. Moving through such large landscapes implies also moving through large empty areas, which are needed to lay out the hierarchy correctly. FSN and the Harmony Landscape provide an extra overview window, displaying the tree as a rough overview, to show the global context. A good visualisation should inherently provide the user with a better global context visualisation.

In the case of a large hierarchy, the visualisation produced with Information Pyramids will be very compact. Since only the height of the pyramids will grow, that means the third dimension of the 3D space is better used. Of course deeper levels of the hierarchy will be very small, but with a zooming operation the user should be able to reach any specific subtree. Due to the compactness of the visualisation, the user will not lose the global context while zooming. With an animated zoom the user will always perceive the global structure while zooming to a particular subtree, so an extra overview window is not necessary.

4.2.1 Leaf Nodes in the Visualisation

All the leaf nodes of a level of the hierarchy are visualised as small objects. The size of these objects depends on the size of the plateau on which they should be placed. Of course, a leaf object on the first level of the hierarchy is bigger than a leaf object deeper in the hierarchy, since the size of the plateaus decreases. The absolute size of a leaf object decreases the deeper it is in the hierarchy, but the relative size size remains constant. The relative size in this context means the size of the leaf object compared to the size of the plateau on which it is placed. This relative size should be selected in a way, that the resulting visualisation *looks good*. In practice this means they should be smaller compared to the subtree plateaus on a particular level, but they should be large enough to recognise every single object. The space between the objects should be selected to easily distinguish between the different leaf objects. These considerations result in a constant, defining the relation between plateaus and

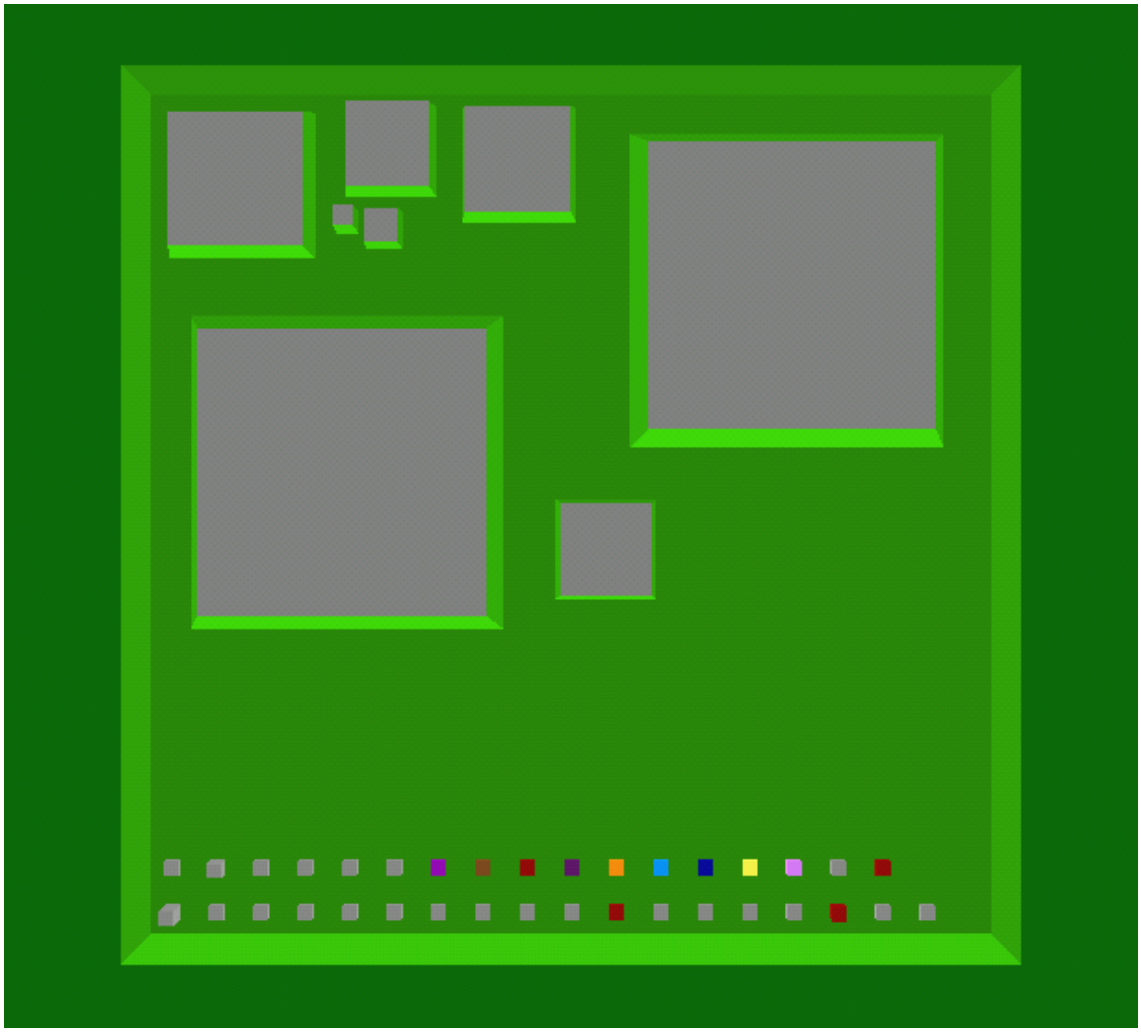


Figure 4.3: The Layout of Objects in the Information Pyramids. Looking from above the big gray plateaus represent subtrees, and the small object at the bottom leaf nodes.

their leaf node objects. In the case of a hierarchy level with unusually many leaf nodes, this constant leads to unusable results. For instance, the leaf object could use all the space on a plateau, so the sub plateaus could not be placed on the plateau. In such a case, the leaf objects has to be made smaller to make room for the subplateaus. The perception of the leaf objects will not be as good as with the constant factor, but should remain useful.

The leaf object could also be used to visualise some attributes of the leaf node. For instance in a file system the colour of an leaf object could represent the file type. Or the object height could be scaled to represent the file size as shown in Figure 4.4. Several other attribute mappings can be done, depending on the attributes of the hierarchically organised information.

The placement of the leaf objects on a plateau can also be done in many different ways, depending on the task. The simplest way is to arrange the objects in a matrix. The objects could also be sorted in different ways and then placed in rows and columns. If there is also some meta information available, which describes the relationship between the leaf nodes, a

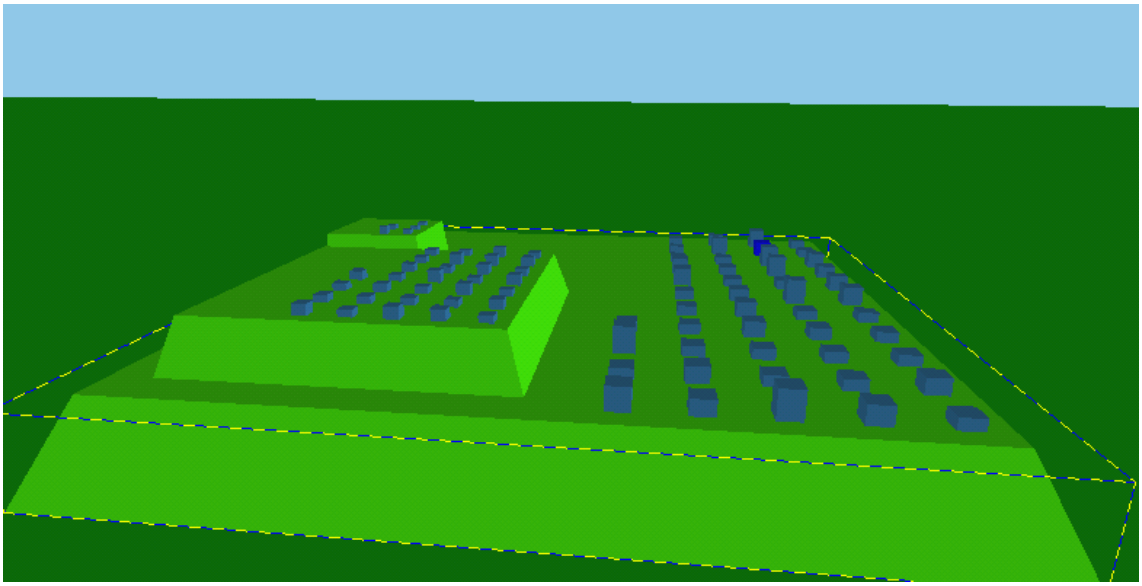


Figure 4.4: The leaf nodes in the Information Pyramids. Notice the different height to show different file sizes.

more complex layout could be done. For example, a *simulated annealing* algorithm [Cha96b] could be used to place related nodes nearer than unrelated ones.

In conclusion, the leaf objects can visualise many attributes and relations in the Information Pyramids approach, so the user acquires rich information about the hierarchy by viewing the objects and their positions. This is a huge advantage to other visualisations, which focus on visualising only the structure of the hierarchy.

4.2.2 Subtrees in the Visualisation

On every plateau there is an area dedicated to be used by subplateaus representing subtrees: the whole plateau area minus the area used by the leaf node objects. This area must be divided in a reasonable way to contain the subplateaus. Of course, it is not useful to represent the subtrees by equally sized plateaus. A subtree consisting of just one leaf node would require the same space as a subtree with many nodes. Obviously, a subtree with just one leaf node should use less space so the plateau of the deeper subtrees can be larger. Using plateaus with different sizes has some advantages:

- The nodes of large subtrees are represented by larger three dimensional objects than in the equal size case. So the overview over the entire landscape is much better.
- The user can easily determine which subtrees are larger just by comparing the sizes of the plateaus. Of course it is only possible to compare plateaus on the same base plateau, because the calculation of plateau sizes are done locally for every base plateau. That is necessary because every base plateau has different sized free areas for its subplateaus.

- In the equal size case, a subtree of only few nodes looks awful, since a relatively large plateau holds only few 3D objects, which is a waste of space. Also navigation becomes inconvenient, since users have to move through empty space containing no useful visual information.

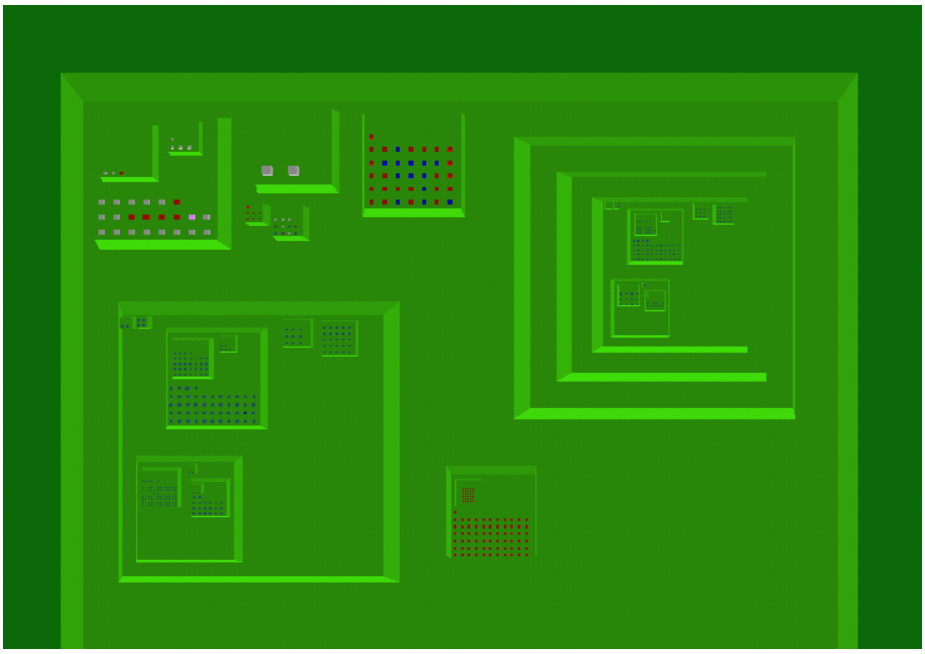


Figure 4.5: The layout of the subplateaus seen from above.

For every subplateau a value is obtained which describes the information contents of the associated subtree. For instance, this could be the number of nodes contained in the subtree. With these values the relative sizes of the plateaus can be easily calculated. Usually, the free area for the subplateaus of the base plateau is a rectangular area. By calculating the relative size, this rectangular area is divided into areas proportional to the information content of the subtrees. We use square plateaus for the visualisation, since given the area of the shape the square size can be calculated easily. However, we now have the non-trivial problem of packing square shapes into a rectangular area. To solve this problem different layout algorithms can be used. There are two criterion for choosing an algorithm:

- The performance of the algorithm should be sufficient to lay out large plateaus in a reasonable time.
- The result should use be almost optimal. That means the free area for the subplateaus should be filled up almost entirely by the subplateaus.

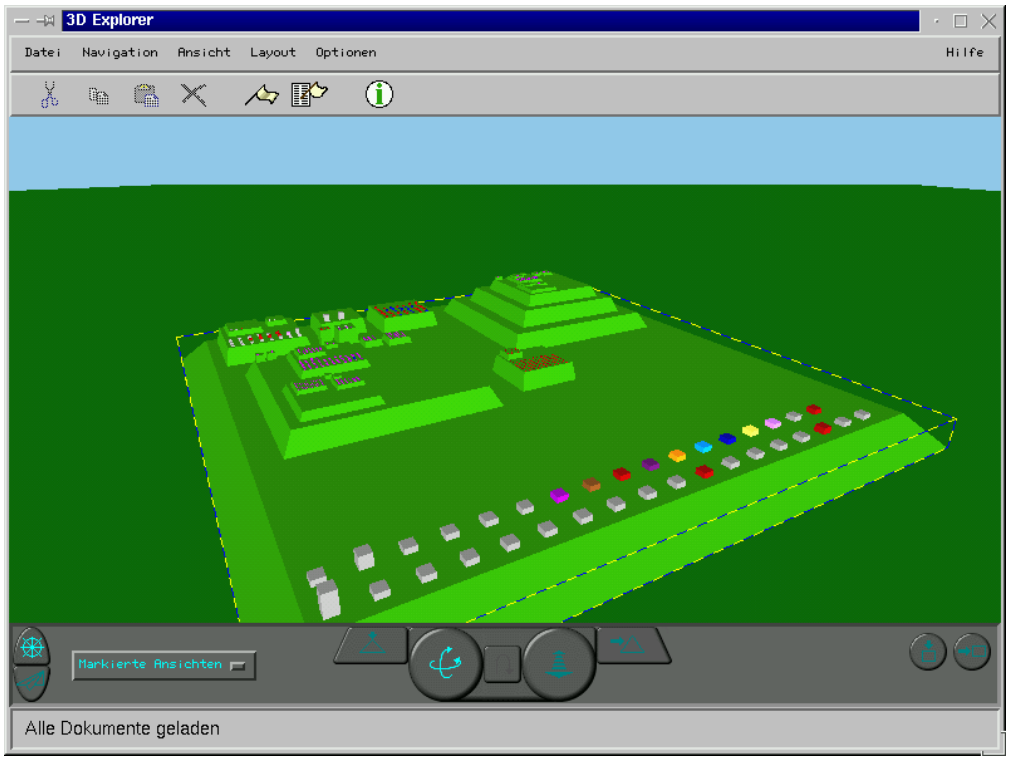


Figure 4.6: The *3D Explorer* - a file browser using the Information Pyramids. Note the navigational aids at the top and the bottom of the window.

4.3 Navigation

4.3.1 Navigation Methods

The main purpose of a visualisation is to facilitate the exploration of information. Visualisations therefore have to provide navigation 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 a 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 mouse and keyboard. Possible navigation aids include:

- A navigation mode to move freely through the 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.
- 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 mange 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 the 3D space. It would be useful to be able to undo or redo navigational actions.

4.3.2 Global Context

It is very important for the users to quickly gain a rough overview of the structure of the hierarchy. This helps users to build a representation of this structure in their mind. Such a generated global context gives users a *feeling* for the structure, so users know approximately where to find some specific information. Or which parts of the hierarchy are very large and therefore contain much information. This contextual information obviously speeds up navigation through the information space.

The 3D representation of the Information Pyramids provides the user with rich contextual information. Just by looking at the pyramids, the user assimilates context information. Of course it must be ensured that the visualisation is consistent over time. Starting the Information Pyramids application with the same hierarchy should always produce the same visual representation.

Navigating through the visualisation helps also to build contextual information, but it must alway be apparent to the users where they move. That means moving from one position to another should be smoothly animated. This is usual in interactive navigation modes, where the users determine which direction and which speed they want. Animation should also used when the user is transfered automatically from one position to another. If this would be simply done by switching from one position to another, the user will become confused. Probably users would completely loose global context and become *lost in space*. If that happens users have to move around to find out which part of the hierarchy they are exploring. Of course, this is very annoying and should be avoided.

When implementing animation the speed of animation has to be considered. Too fast animation has almost the same effect as switching from one position to the other. If the animation is too slow, users have to wait all the time. This is also very annoying and prevents continuous work. Human computer interface research suggest that animation should take about one second. This value is long enough for the user to track the motion in the landscape, but it is not so long that users will be hindered in their work. When visualising large hierarchies with slow computer hardware it is very difficult to do the animation in one second. Therefore, rendering has to be done in a more intelligent way. Some sort of *level of detail (LOD)* mechanism could be used, to not draw small objects far away from the camera position. Decreased levels of detail for objects far away speeds up the animation. Small objects sometimes only consists of a few pixels of the display. Such objects do not give users any information, but will possibly distract them. Only large objects which are far away should be displayed to maintain global context.

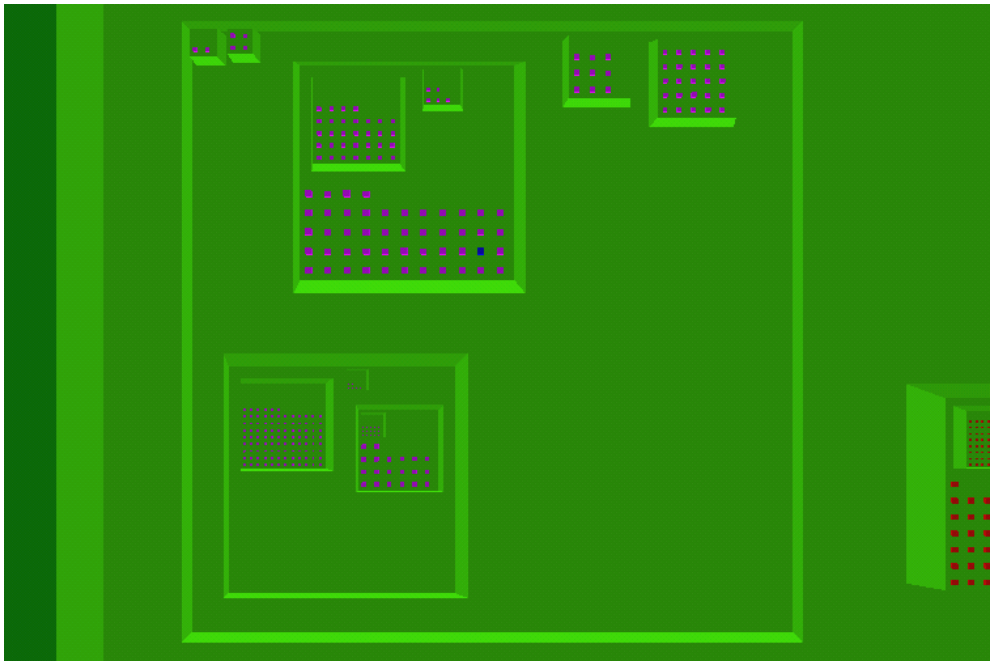


Figure 4.7: A top view of a plateau.

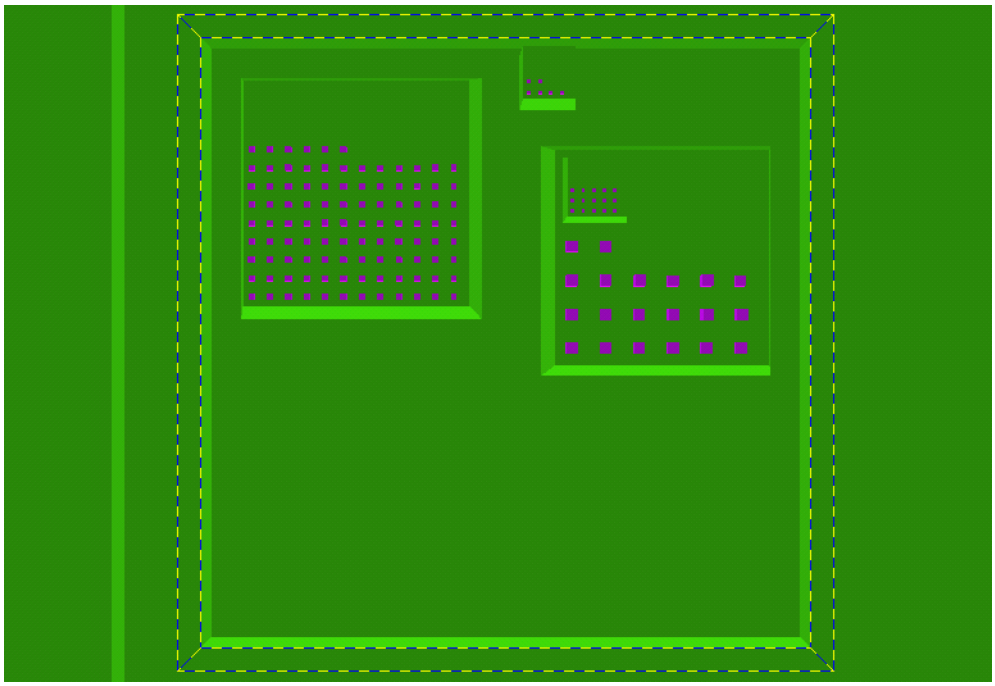


Figure 4.8: Focusing in on the bottom left subtree of the Figure 4.7.

Information Pyramids provides another way to see a global overview. Looking down on the landscape from a position right above the Information Pyramids, results in a top view of the pyramids. This view is very meaningful, because it easily can be seen how the information is structured and distributed over the hierarchy. Due to the rapidly decreasing size of the plateaus it should also be possible to get a top view at any sub-plateau. The camera should zoom to the selected plateau so that the plateau uses the whole screen space. The whole subtree can then be seen from above.

4.3.3 Local Focus

It is not only important to get a feeling for the overall structure of the hierarchy. Usually users want to quickly find the subtree of interest. After that they usually want to focus on that particular subtree to do some work. This could be simply exploring the hierarchy, or some editing of the hierarchy, or some other operation which depends on the visualised information. For instance, when exploring a file system the user might want to explore a directory or copy, delete, or move some files.

To focus on a particular subtree, some of the navigation methods discussed in Section 4.3.1 could be used. These methods should be designed not only to help the user to gain an overview, they should also take the users quickly to the subtree they want to explore. Also some navigation method should be implemented specially designed for exploration of a specific plateau.

Even when the user focuses on a specific subtree the other pyramids can be seen in the background and thus providing contextual information. This is a big advantage of the three dimensional landscape. Even if the user always focuses on some subtrees, contextual information about the hierarchy is provided. Sometimes it is not desirable to display all this surrounding information. For instance when visualising very large hierarchies the frame rate of animations may decrease below a usable minimum. Or sometimes this extra information in the background might disturb the user. In such cases a *pruning* operation can be useful. This operation declares some subplateau of the visualisation to be the new base plateau for the visualisation (see Figures 4.9 and 4.10). Only this plateau and its subtrees will be displayed. Thus only a fraction of the hierarchy is displayed and therefore the frame rate will be easier to maintain at interactive rates.

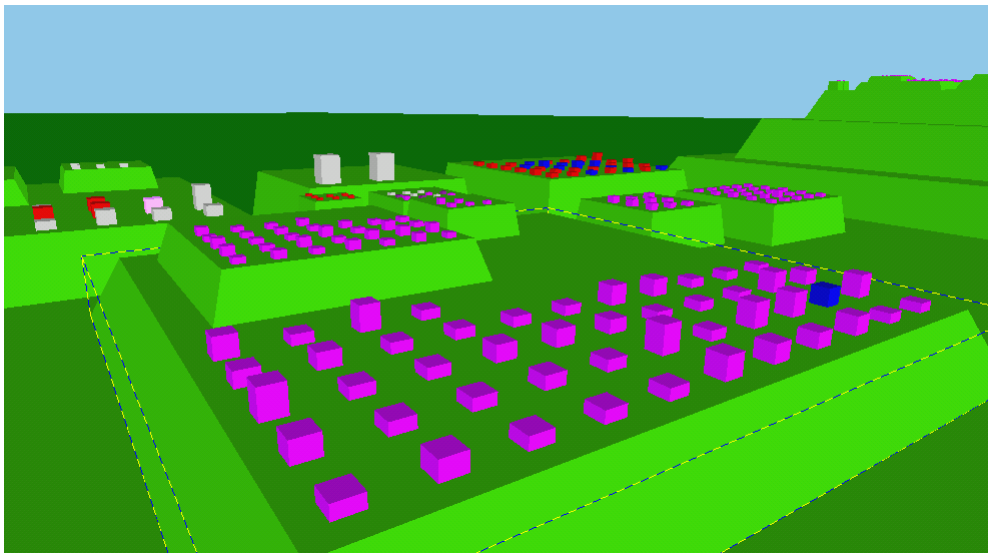


Figure 4.9: A focused subtree with other objects in the background.

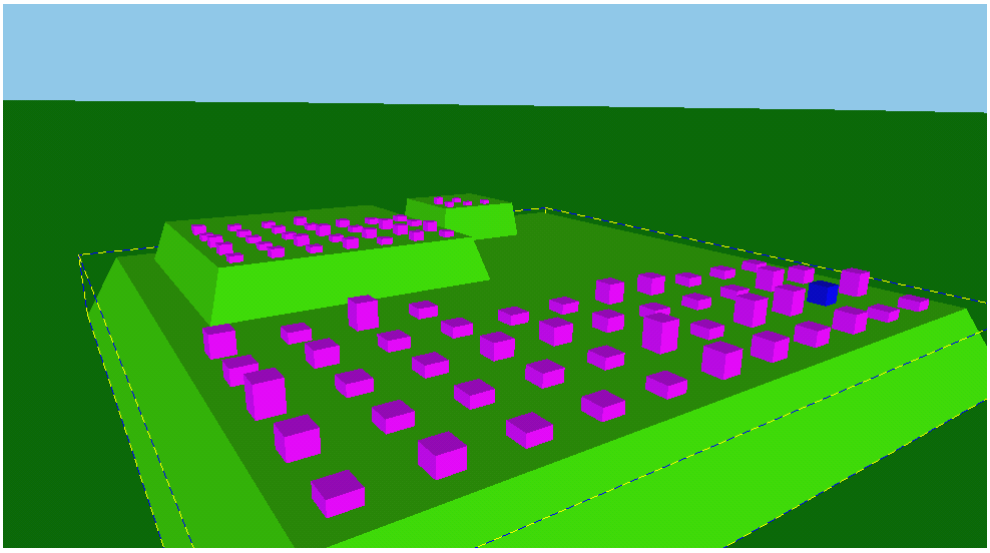


Figure 4.10: The *pruned* view of the subtree of Figure 4.9.

Chapter 5

3D Explorer

The *3D Explorer* is the first implementation of the Information Pyramids concept. It visualises the hierarchy of a file system. The application was completely written in Java, but uses the GE3D library for 3D graphics output as described in Section 3.3. GE3D is, however, platform-dependent and has to be compiled for every target platform (see Section 3.2.3).

This application is not a fully featured file system browser, but was intended as a "proof of concept" for the Information Pyramids technique. Hence it provides only the basic functionality of a file browser. Due to its intention, it has also numerous experimental features and options, which were implemented during the design of the Information Pyramids to test them. For instance a layout mode for the directories can be enabled where all directories are made equally sized. Enabling this feature is not useful, but it shows the evolution of the application and the progress the concept made over time.

5.1 Architecture of the 3D Explorer

The 3D Explorer was implemented in Java, and is structured in objects. Figure 5.1 shows the global structure of the application, in terms of function blocks and the classes belonging to each block.

Starting with the *3D Explorer* block at the top, where the initialisation work is done, first the *Command Line Parser* processes the command line parameters. After that different resources are loaded by the *Resource Loader*, such as images for buttons or the 3D font files. Then the *3D Explorer* block initialises the *Application Frame*. This frame or window contains three main parts. Firstly, the *Icon Bar* at the top of the window displays icons for functions like editing or handling viewpoints. Secondly the *Bottom Panel* is initialised at the bottom of the window. This panel contains the *Navigation Bar* which contains all the buttons for navigational purposes and the *Status Line* which displays status information as text or shows a progress bar for some functions. Thirdly, the *3D Canvas* is placed into the centre of the window. This canvas actually displays the three-dimensional landscape and handles user input. It is closely related to the *3D World*, which is the internal representation of the landscape. User input is directed from *3D Canvas* to *3D World*, initiating painting the landscape through the *3D Painter* or loading new file system structures with the *Data Server*. All these functions have access to the internal representation of the hierarchy, the

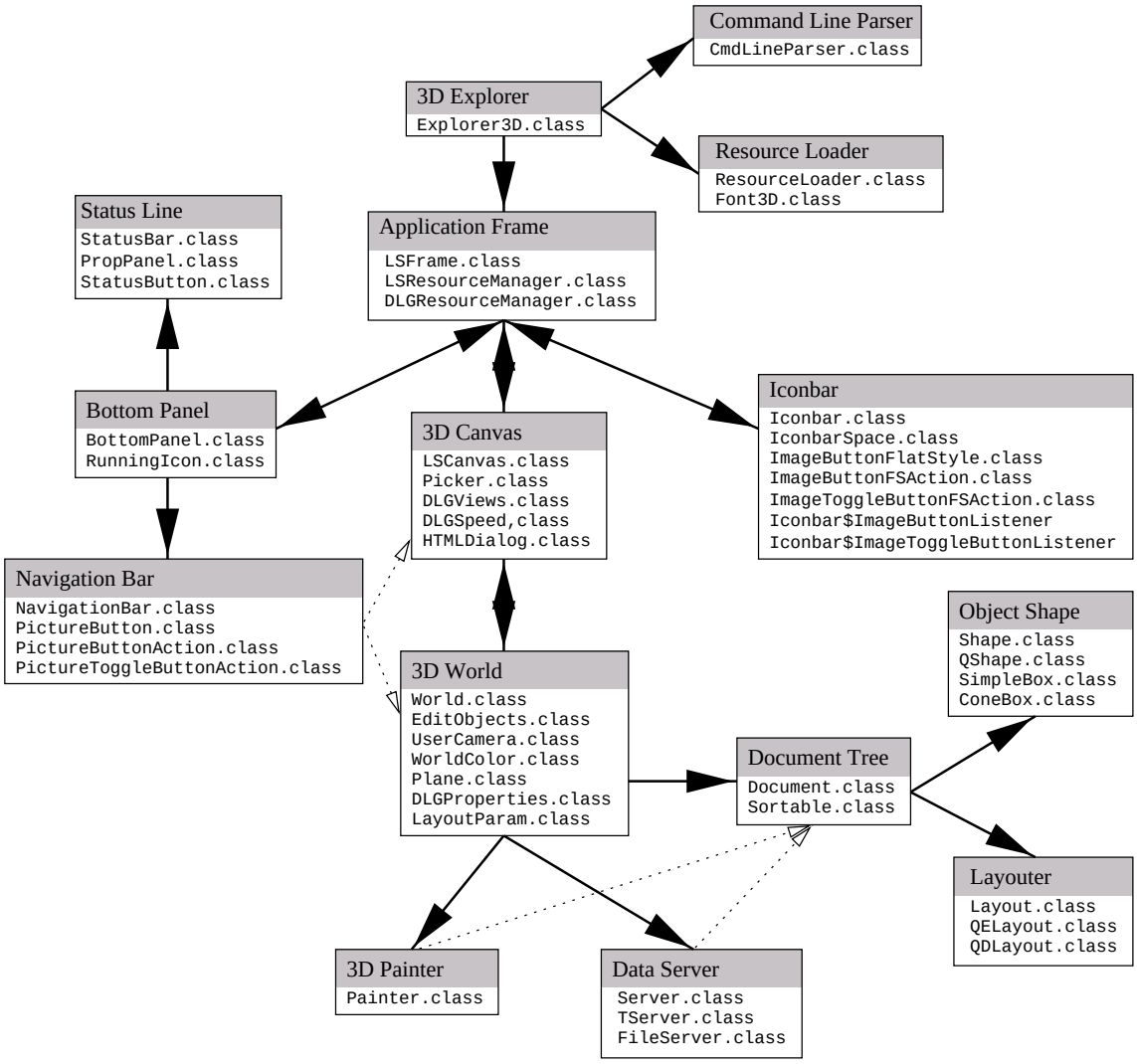


Figure 5.1: The structure of the 3D Explorer application.

Document Tree. The tree is of course a part of the *3D World*, but the *3D Painter* and the *Data Server* must have access to this data structure, as indicated by dotted lines. The nodes in the document tree all have a visual representation by an *Object Shape*. This block defines the shape, size, colour and position of the 3D object representing a node. If new objects are added to the *Document Tree*, the tree invokes a *Layouter* to generate the 3D objects and their position in the landscape.

5.2 Application Start

The application is started by invoking the class `iicm.lsape.Explorer3D` with the Java interpreter. This class just initialises the application: It reads the command line parameters and the resources used by the application. After that the application frame is generated, which handles any further operations.

5.2.1 Command Line

The usual command line parameter is a starting directory. This directory is used as the base directory of the visualisation. If no directory is specified as a command line parameter the current directory is used as the start directory. There are also some other parameters which can be specified at start-up. To parse the command line parameters the `CmdLineParser.class` is used. This class takes the command line arguments, a list of option strings and a list of the associated action tags as parameters. The option strings of the 3D Explorer are:

```
private final static String[] ARGOPTIONS =
{
  "-h","-help",
  "-verbose",
  "-home"
};
```

And the associated action tags are:

```
private final static int[] ARGTAGS =
{
  ARG_HELP, ARG_HELP,
  ARG_VERBOSE,
  ARG_HOME
};
```

The parser has four public methods, which are the `currentArgString()`, `more()`, `getNext()`, and the `nextArgString()` method. `more()` returns true if there is another command line argument. With `getNext` the action tag for a parameter is returned or an error code if it is not valid. The error codes are:

```
public final static int UNKNOWN_OPTION = -1;
public final static int OTHER_ARGUMENT = -2;
public final static int NO_MORE_ARGS = -3;
```

With the `currentArgString()` and `nextArgString()` the current or next argument Strings can be obtained. The `Explorer3D` class uses these methods to get the arguments or to print an error message.

5.2.2 Resource Loader

The `ResourceLoader` class loads all external resources like images and font data. Images are used for the icons, the "running" symbol, or the graphic buttons of the navigation bar. Font data is used by the 3D fonts of the landscape. The font data are public domain Hershey fonts which were converted into a format for use with the GE3D library.

When the constructor of the class is invoked, a window with a progress bar is displayed. This bar shows how many resources have been loaded. After all resources have been successfully loaded, the window is closed. In case of errors, some error messages will be printed.

Two public methods are defined in the `ResourceLoader` class, which are the `getIcons` and the `getFont3D` methods. The `getIcons` returns a vector of image vectors. The image vectors are defined for different purposes. For instance there is a vector for the *Icon Bar* icons or one for the *Navigation Bar* icons. The `getFont3D` returns an array of `iicm.utils3d.Font3D` objects which represents the three dimensional fonts.

5.3 Application Frame

The main class of the application frame is the `LSFrame` class. The constructor of this class initialises the three main parts of the window. These are the *Bottom Panel*, the *Icon Bar* and the *3D Canvas*, which will be described later in this chapter.

The main task of the class is to initialise and handle the menu bar of the application. The menu bar event handler invokes the appropriate methods of the *3D Canvas* or the *3D World*. There are also methods to enable and disable menu items and to synchronise these actions with the buttons in the icon bar or the navigation bar.

5.3.1 Icon Bar

The icon bar shown in Figure 5.2 provides some icons at the top of the window to give the user shortcuts to often used commands. These are the edit commands like cut and copy. There are also some icons to define and mange view points and an icon to view the properties of the selected 3D object. Notice that no navigation related actions are available in this icon bar, they are all placed in a separate bar at the bottom.

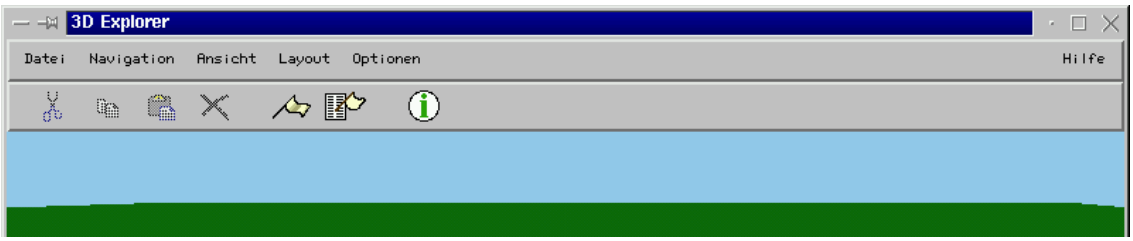


Figure 5.2: The icon bar of the 3D Explorer.

All the icon buttons in this bar are instances of the `ImageButtonFlatStyle` class

(see Figure 5.3). This class is derived from the `iicm.widgets.ImageButton` class and defines an image button with a three dimensional border.

The class `ImageButtonFlatStyle` adds the fields:

```
protected boolean focused_ = false;
protected boolean enabled_ = true;
```

In addition, the methods `setFocused(boolean)` and `setVisible(boolean)` were added to set the two values. The `focused_` flag is true if the mouse cursor is over this icon. The new `paint` method checks this flag, when it is set a 3D frame around the icon is painted, otherwise no frame is painted. So the image button is only a flat icon unless the mouse moves over it, than it is recognisable as a button. The `enabled_` flag can be set by `setVisible` to enable or disable the button. When the button is disabled, the icon is painted with a gray layer above it to show its state.

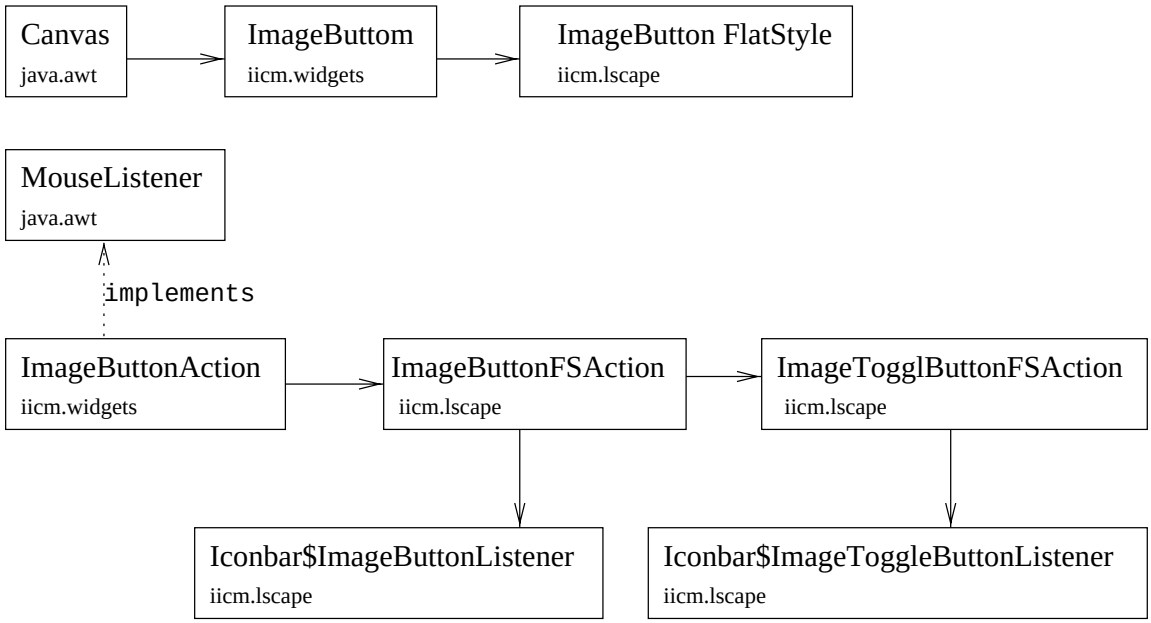


Figure 5.3: The tree of the classes for the image buttons used by the icon bar.

To set the `focused_` flag, the `ImageButtonFSAction` class or an class inherited from it like `Iconbar$ImageButtonListener` must be registered as a `MouseListener` for the `ImageButtonFlatStyle` class. The `ImageButtonFSAction` class is inherited from `ImageButtonAction` which implements the `MouseListerner` interface. `ImageButtonAction` handles only the `mousePressed` event. In this case it checks if the button is one of a button group. If that is true, it sets the state of the last pressed button to normal. So a button group acts like well-known radio buttons. `ImageButtonFSAction` adds to this behaviour only the handling of the `mouseEntered` and the `mouseExited` events. If one of this events occurs the `setFocused` method is used to set the flag accordingly.

All of the image buttons described above have the behaviour to stay in the “pushed” state after a mouse click. A following mouse click takes it back to the “unpushed” state.

This is used for options to switch on or off, or for the mentioned radio button feature. For buttons with normal behaviour the `ImageToggleButtonFSAction` class, which is inherited from `ImageButtonFSAction`, can be used. It adds the `mouseReleased` event handling. After the button was pushed the release of the mouse button is detected and the button is immediately set back to the "unpushed" state. So the `ImageButtonFlatStyle` can be used in two different modes depending on which `MouseListener` is added.

With the described `MouseListener` classes only the behaviour of the buttons is defined. To process a `mousePressed` event the classes `Iconbar$ImageButtonListener` or `Iconbar$ImageToggleButtonListener` are used. These classes are inherited from `ImageButtonFSAction` or `ImageToggleButtonFSAction` as can be seen in Figure 5.3. They add only code to invoke some actions on the `mousePressed` event.

5.3.2 Bottom Panel

The *Bottom Panel* block contains only the `BottomPanel` class which extends `java.awt.Panel` and is only a container for the *Navigation Bar* and the *Status Line*. On startup it initialises the navigation bar and the status bar.

Status Line

The status line is initialised and managed through the `BottomPanel` class. All access methods for the status line are defined in this class. Basically, the status line displays helpful information. In fact, the status line is a container which uses the `CardLayout` to display different types of status line as requested. The private method `makeStatus()` creates three types of status line:

- A simple status line with a nice 3D frame to display some information, such as a help text when the mouse is over an button in the icon bar.
- A special bar which is displayed during the loading of the file system structure. A small animated icon is displayed at the left side to show that the loading process is active. Also a *STOP* button at the right side is displayed, so the user can stop a loading process.
- A bar which is displayed while a re-layout of the visualisation is done, for instance after changing the sort order. Here a progress bar is shown at the right side to display the progress of the re-layout.

Loading and re-layout operations are performed in separate threads, hence the public method `setBar` is defined `synchronized` so the threads can switch to the appropriate status line. The `BottomPanel` class has also some public methods to write in the currently displayed status line, and to set the value of the progress bar or the animation frame of the animated loading icon.

Navigation Bar

The main class of the *Navigation Bar* block is the `NavigationBar` class. It is a container for graphically well designed navigation buttons. It is closely related to the *3D Canvas* and *3D World* function blocks, since pushing buttons in the navigation bar alters the *3D World* or leads to some action in the *3D Canvas*.

Figure 5.4: The navigation bar of the 3D Explorer.

As shown in Figure 5.4, the navigation bar has many custom-designed graphic buttons. These buttons are defined in the `PictureButton` class. Figure 5.5 shows that the class hierarchy is almost equal to that of the `ImageButton`.

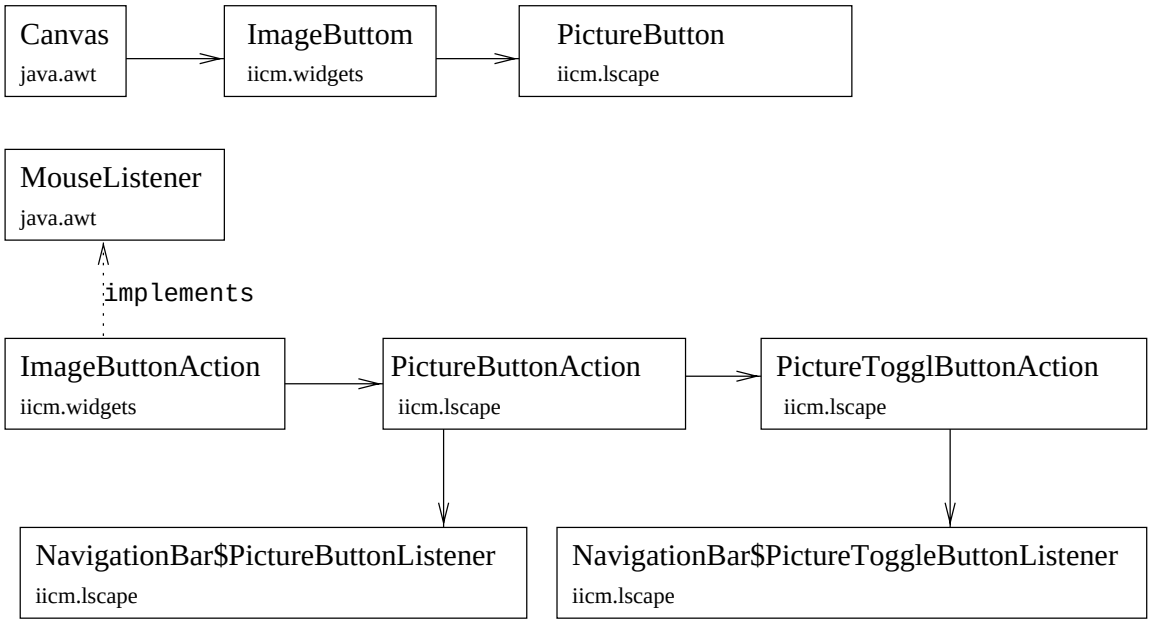


Figure 5.5: The tree of the classes for the picture buttons used by the navigation bar.

In fact they are very similar, but the `ImageButton` displays an icon with an rectangular equal sized 3D frame. However, the `PictureButton` displays a custom shaped graphic as a button. For instance this would be one of the round buttons on the right of Figure 5.4. When the mouse enters the button and it is enabled another button image is displayed. In the navigation bar an image with a larger 3D effect and a lighter gray is displayed. For instance the two left buttons of the figure 5.6 are in normal state, the right one is focused by the mouse. Another difference to the `ImageButton` is that the icon which is painted atop a `PictureButton` can have three different states.

disabled: The icon is drawn in dark gray, so it is almost invisible on the gray button (left picture of Figure 5.6).

enabled: The icon colour is now dark cyan. The icon is now clearly visible (middle picture of Figure 5.6).

active: Now the light cyan colour of the icon shows that this button is active (right picture of Figure 5.6).

Figure 5.6: The different states of a picture button.

The interface of the classes related to the `PictureButton` are equal to the `ImageButton` classes. Of course the implementation is slightly different. The different images have to be managed and displayed, but the flags and methods are used in the same way as in the `ImageButton` class.

The `NavigationBar` class basically handles the events from the picture buttons it contains. Usually, it invokes some action in the *3D Canvas* or the *3D World* blocks.

5.3.3 3D Canvas

The central class of the *3D Canvas* block is the `LSCanvas` class. It extends `OGLCanvas` (see Section 3.3.2) which is the basic 3D graphics canvas. The main task of this class is to handle the mouse and keyboard events applied to the 3D window.

There is a huge `keyPressed` method which starts actions depending on the key pressed. Most of the time this would be to initialise some action on the internal document tree, or begin an animated translation to a new position. The mouse events handled by this class usually lead to moving the camera. For instance rotating, panning or zooming the camera controlled by the user using the mouse.

Of course the user can select some objects by pointing and clicking. The selected objects are highlighted by a coloured frame. Another special mouse event is to click on an object with the right mouse button. This brings up a dialog to show the object properties such as file name, size, and creation time. To detect which object was selected by the user in the three dimensional space the `Picker` class is used. This class calculates the hit point of the picking ray with objects in the landscape as shown in Figure 5.7. The picking ray starts from the camera position and goes through the point of the mouse click on the camera plane. The object with the nearest hit point is returned.

Another important task is to start the painting of the 3D landscape. The `paint` method of the canvas first sets the GE3D/OpenGL context to this canvas. After that the landscape is drawn by calling a method of the `World` class. Then some user interface components are drawn in the 3D canvas if necessary. Many navigation operation like zooming are done by clicking on the canvas and moving the mouse. To give the user visual feedback a cross is

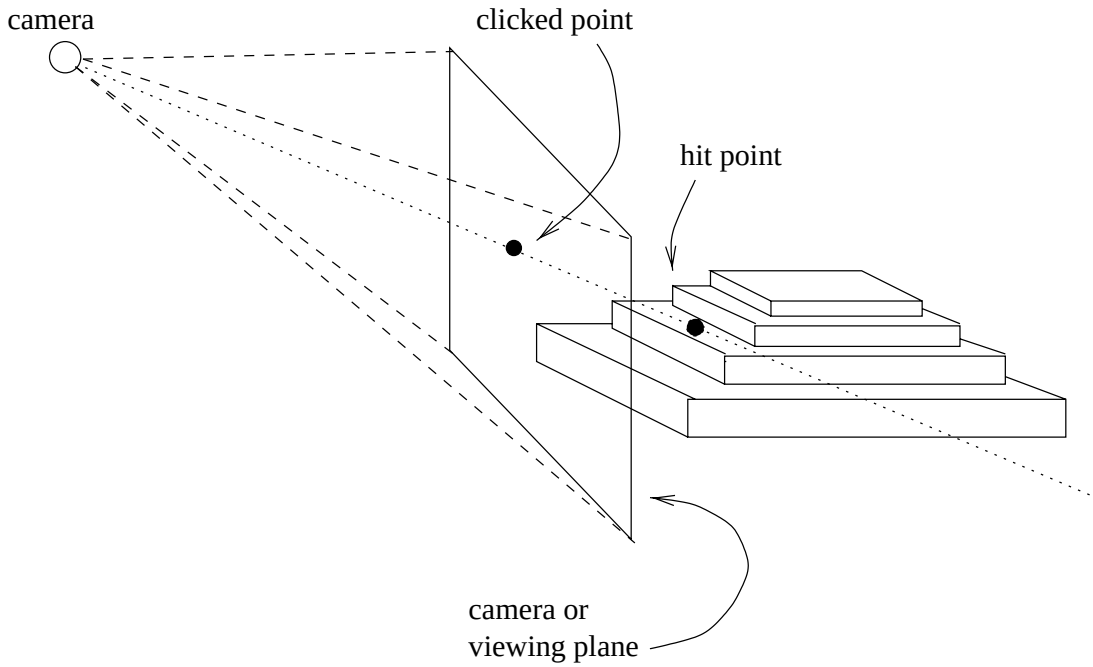


Figure 5.7: Picking a 3D object by clicking a point on the view plane.

painted at the point where the user pushed the mouse button, and a line from this point to the current mouse position is painted, as shown in Figure 5.8. After that it is checked whether an animation step has to be done. If so, the next animation step is calculated and a repaint of the canvas is scheduled.

Since all the animated navigation is done in this class, all methods to move the camera in some way are placed here. For instance, handling of user-defined viewpoints is also done in the `LSCanvas` class. The `DLGViews` class is a member of the `LSCanvas` to show a dialog to manage the viewpoints.

5.4 3D World

All the classes discussed in Section 5.3 deal with visual components. The *3D World* block with its central `World` class is the internal representation of the three dimensional landscape. It has many methods to alter this representation, for instance to add a new subtree to the visualisation or to change the layout.

The `World` class encapsulates all properties to display the three dimensional landscape. The most important is the *Document Tree* which represents the file system hierarchy. This tree is the central data structure of the application and is described in detail in Section 5.5. To add documents to the *Document Tree* the *Data Server* block is used. The *Data Server* uses threads, which are managed in the `World` class. How this is done is described in detail in Section 5.6.

Also there are some methods to change the GE3D/OpenGL drawing modes, like the method `setDrawMode`. This method sets the drawing mode of the 3D engine to wireframe,

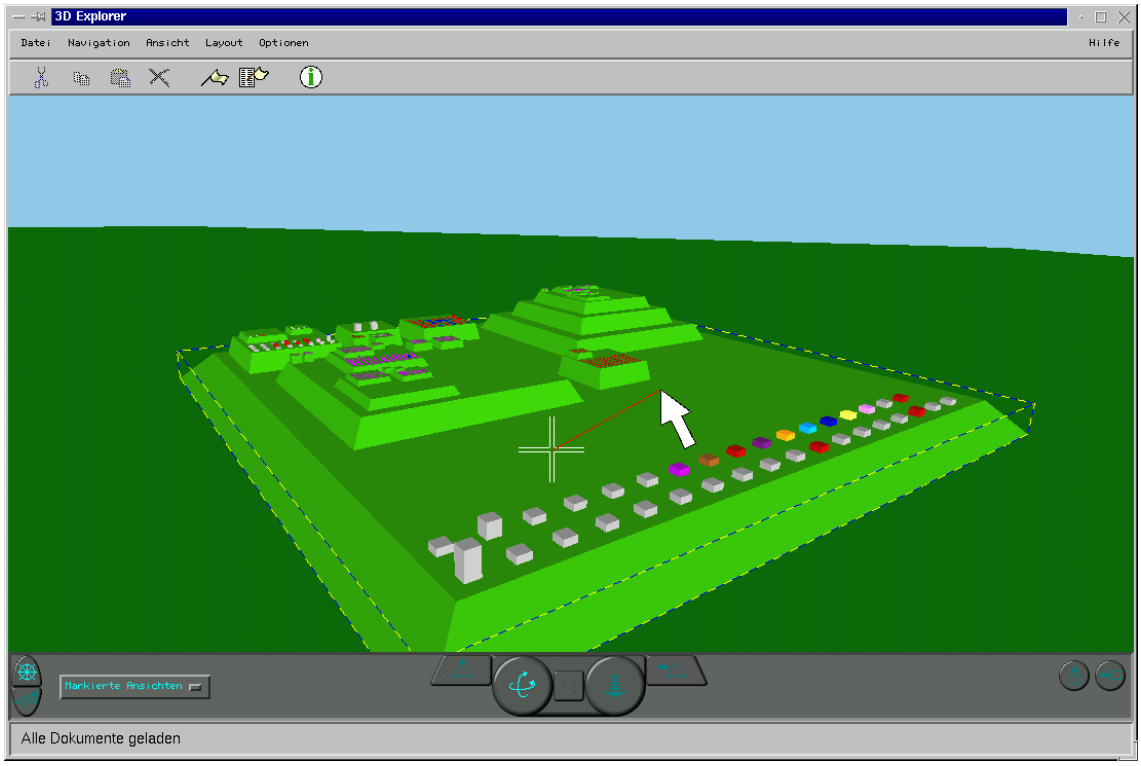


Figure 5.8: The visual elements during rotation.

hidden line, or flat shaded mode.

The only 3D object which is not a part of the *Document Tree* is the ground plane of the landscape, which is handled by the `World` class. If the user prunes or adds a new base, the ground plane is moved up or down to the new base. The objects are than painted beginning from this base.

Another important object which is a member of the `World` class is the camera. It is modelled by the `UserCamera` class which extends the `iicm.utils3d.Camera` class. Basically, this class encapsulates the position and the orientation of the camera. There are many methods to manipulate the camera position and orientation. Method `zoomOut` moves the camera away form the viewing plane, `translateVec` translates the camera along a given vector, and `translateVP` moves it parallel to the view plane. There are also some methods to rotate the camera, like `rotateXYposition` which rotates the camera in the current position, and `rotateXYcenter` which rotates the camera around a given rotation centre. Other methods to move the camera are the `approachNormal` and `approachPosition` methods. These methods move the camera a fraction of the current distance to the given point, `approachNormal` additionally corrects the orientation of the camera to the negative given vector so that the camera approaches a surface with a camera orientation normal to the surface. By calling these methods with fractions from 0.0 to 1.0 and repainting the landscape between every call, the camera can be move animated from one position to another. All these methods are defined in the `Camera` class. The `UserCamera` class extends these methods and adds some new methods. One added feature of the `UserCamera` is that it can move the camera with collision detection, so the camera can not be moved through

3D objects. For every moving method mentioned above a new method was created with the same name, but with the suffix "CD" for collision detection. So there is for instance a `zoomOutCD` or a `approachPositionCD`. The "CD" methods call the normal methods and after checking if the new position collides with a surface in the landscape. If not the move will be done, otherwise it is cancelled.

The `Camera` object manages the view of the 3D landscape. The `World` class contains the data for the entire 3D world and has a `draw` method to paint it. This method initialises the 3D engine: for example it sets the draw mode for the 3D engine, the background colour, and the camera. After that the method `paint` of the class `Painter` is called to do the actual painting of the 3D landscape (see Section 5.4.2). At the end of the `draw` method the `selectedDocs_` vector is checked. This is the vector where all the user selected objects are stored. Notice that there can be more than one, because while holding down the control key and selecting objects with the mouse, multiple objects can be selected. For every object in the `selectedDocs_` vector a coloured (red and green) wire frame is painted around it to show its selected state (see Figure 5.9). The last selection is painted in a different colour (blue and yellow), because some operations like showing the properties, can only be applied to one object. In Figure 5.9 it is the bottom centre plateau, which has many file objects.

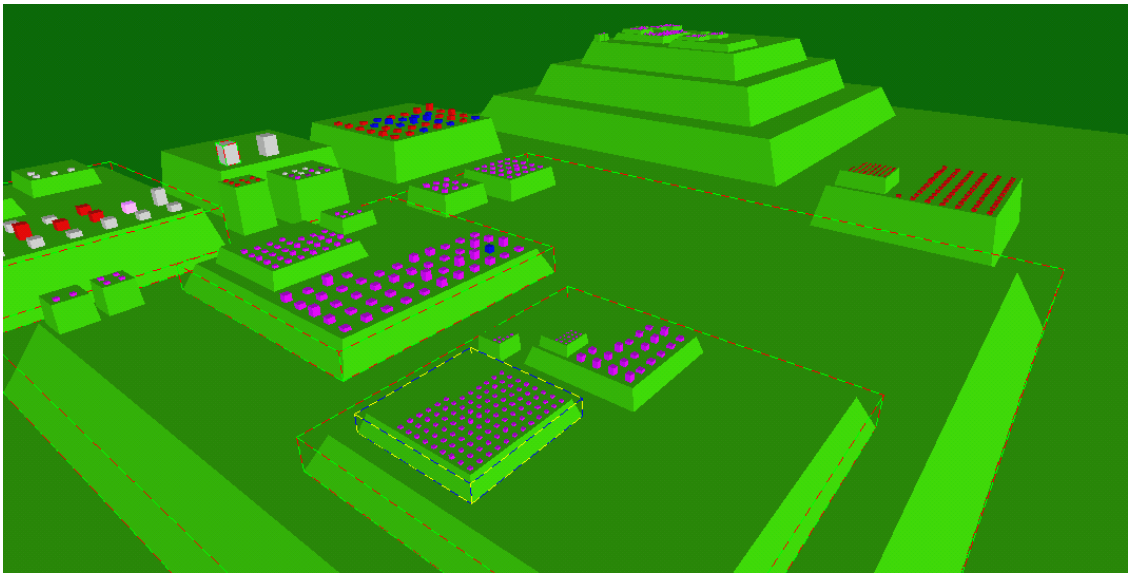


Figure 5.9: The landscape with multiple selected documents.

5.4.1 Layout Parameters

The user can interactively change the parameters of the 3D object layout. Changing these parameters results in a different visual representation. The parameters are:

Sort Order: The files and directories can be sorted by different criteria like size, type, age, or name.

Layout Scheme: The Layout of the objects can be done by any layout scheme. Currently, two are implemented: one which calculates every subplateau with the same size, and one in which the size of the subplateau is proportional to the contained data.

Directory Size Estimation: To estimate the amount of data contained in a directory two different strategies can be used: the number of files in the directories, or the cumulative size of these files.

Scan Depth: When estimating the amount of data contained in a directory, the files upto a specific depth of the hierarchy are taken into account.

Shape of Objects: The 3D objects can have different shapes. Currently only the shapes of the directory plateaus can be changed from an ordinary box to a cone shaped box.

After changing one of these parameters by selecting a menu item or in a dialog box, a `set` method for this parameter is called in the `World` class. If the displayed hierarchy is large, changing the parameter can take a while. To ensure that the user can continue to navigate while the layout parameter is changed, this is done in a separate thread. While the relayouting is done, each plateau is not visible. After the layout is done it is displayed, so it looks like the pyramids grow up from the base.

To implement this threaded changing of the layout parameter the class `TLayoutParam` was created, which extends the `Thread` class. It has only one important method, the `run` method. This method calls the `setLayoutParam` of the `LayoutParam` class in an infinite loop.

The `setLayoutParam` method starts with a wait loop, which puts the calling thread in a wait state by using the `wait` method. The threaded `TLayoutParam` class changes into the wait mode. To wake up this thread, the other methods of the `LayouParam` class are used. These methods are the `set` methods for the mentioned layout parameters. If the user changes one of the parameters the appropriate `set` method will be called. These methods sets some internal flags and parameters for the relayout and then the `notify()` method is used to wake up the waiting thread. The thread which called the `set` method just goes back to normal work and the `TLayoutParam` thread leaves the wait loop. After the wait loop the thread sets the status line for the parameter change. Then the method `recPlace` is called to relayout the landscape. This method does a relayout of the entire landscape using the parameters which were set by the `set` method.

It is important to mention that there is only one thread to change the layout parameters. While the thread is running the parameters can not be changed. Furthermore, a directory can not be opened by a user during the relayout. Opening a directory is also done in a thread, and this loading thread and the layout parameter thread can not be running at the same time. A message box is displayed if the user attempts to do this.

5.4.2 3D Painter

The *3D Painter* block contains only one class, the `Painter`. This class is responsible for drawing the 3D object of the landscape. It accesses the *Document Tree* to obtain the shape for every document, but simply drawing the entire landscape on every redraw would be

unnecessarily time consuming. For a smarter `Painter` class, the following considerations were made:

- For smooth animation a constant high frame rate is necessary. The user should set a frame rate according to the used hardware. This frame rate should than be used for the animations.
- Due to the variety of computer hardware, the speed of animation varies on different platforms. A mechanism should be provided to keep a frame rate constant independent of the hardware in use.
- The larger the hierarchy the more time it takes to paint the 3D representation. Not every part of the hierarchy has to be painted, some parts might be too far away or not even in the user's view.
- It is not necessary to paint every tiny detail, because small or distant objects will become only one pixel on the screen.

First, a mechanism was implemented to track the frame rate. The frame rate expresses how often the landscape should be painted in one second. The unit of the frame rate is "frames per second" (fps). To calculate this value the start time of painting the landscape is determined. The painting of the landscape is done in the `paint` method of the `Painter` class. The end time of the painting is not measured at the end of the method, because the actual painting is done in a buffer. This technique is called "double-buffered painting". While one buffer is displayed, the application can paint in the other to ensure that no flicker effects occur. This effect is otherwise visible when displaying and painting is done in the same buffer. So the `painter` method draws the objects in a buffer, which is displayed later. This is done with the method `swapBuffers` in the class `LSCanvas`. Remember that this class extends the `iicm.ge3d.OGLCanvas` class, where `swapBuffers` is defined. If the end time would be measured at the end of the paint method the time-consuming painting has not yet been done. To get the right value the time must be measured after swapping the buffers. So the steps of painting the 3D landscape are:

- `iicm.lscape.LSCanvas` sets the GE3D/OpenGL context to this canvas. Now all 3D methods affect this window.
- `iicm.lscape.LSCanvas` calls `iicm.lscape.World.draw()`.
  - `iicm.lscape.World.draw()` does the basic initialisation for the landscape, like setting the camera and colours.
  - `iicm.lscape.Painter.paint()` is called.
    - Start time of painting is measured.
    - Painting the 3D objects. After that the `paint` method ends.
  - `iicm.lscape.World.draw()` paints a wire frame around the selected objects. Than the `draw` method ends.

- `iicm.lscape.LSCanvas` draws some user interface components in the 3D window.
- Method `swapBuffers` is called.
- `iicm.lscape.LSCanvas` called the method `endPaint` of the `Painter` class to measure the end time of the painting.

The `endPaint` method of the `iicm.lscape.Painter` class can now easily calculate the time for painting this frame and also the current frame rate.

After determining the current frame rate, a mechanism was implemented to keep the frame rate at a user defined level. To choose which objects are to be painted and which not, a *relative distance* value for every object is calculated:

$$relative\ distance = \frac{width\ of\ object}{distance\ from\ camera\ to\ object}$$

With this value it is now possible to determine which objects are very small, or displayed very small because they are far away from the camera. Now we can determine which objects should be painted by calculating the *relative distance* and comparing it to given value. If the *relative distance* is bigger, it is painted, otherwise not. This is the basic approach to minimise the painted objects, but to get to a usable algorithm some improvements have to be made.

First the algorithm decides only if an object should be painted or not. It would be better to gradually decrease the details of the displayed objects. Since the objects in the landscape are very simple now, only two steps of intermediate detail were introduced.

- Paint the full 3D object.
- The objects is painted in wire frame mode.
- Only the coloured square at the “floor” is painted.
- No object is painted.

If more complex or textured objects are implemented, more detail levels can be introduced. To decide which detail level should be used these values are used to compare with the *relative distance* of an object:

```
private float draw_doc_full_ = -1.0e-7f;
private float draw_doc_wire_ = -5.001e-4f;
private float draw_doc_rect_ = -9.001e-4f;
```

As shown in Figure 5.10, if the *relative distance* is less than `draw_doc_rect_` than the object is not painted. If it is between `draw_doc_wire_` and `draw_doc_rect_` a rectangle is painted. Between `draw_doc_full_` and `draw_doc_wire_` the object is painted in wire frame mode. If it is less than `draw_doc_full_` the normal 3D object is painted. The values of these variables are experimental starting values, which are changed dynamically after every redraw.

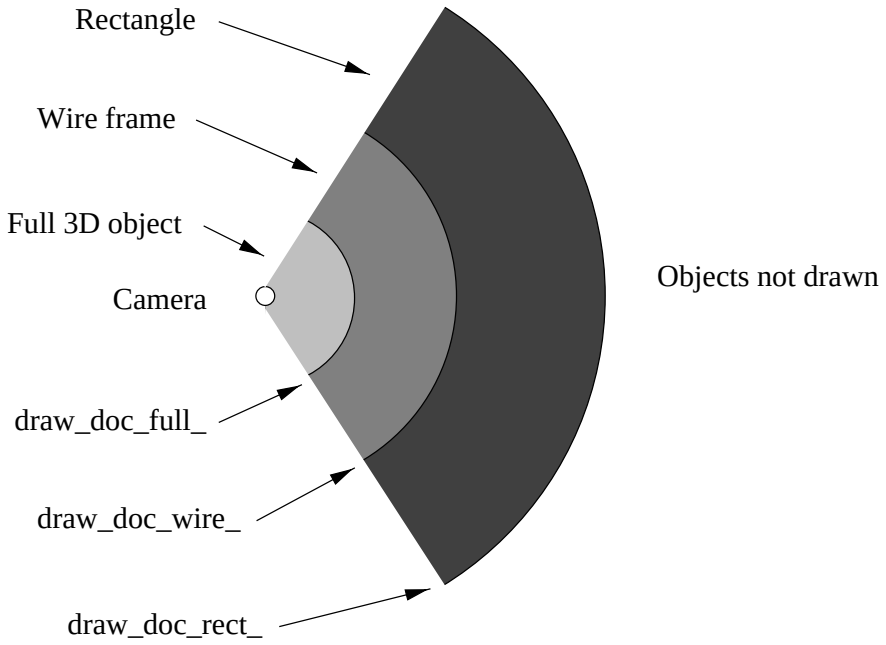


Figure 5.10: The camera and the regions where different object details are painted.

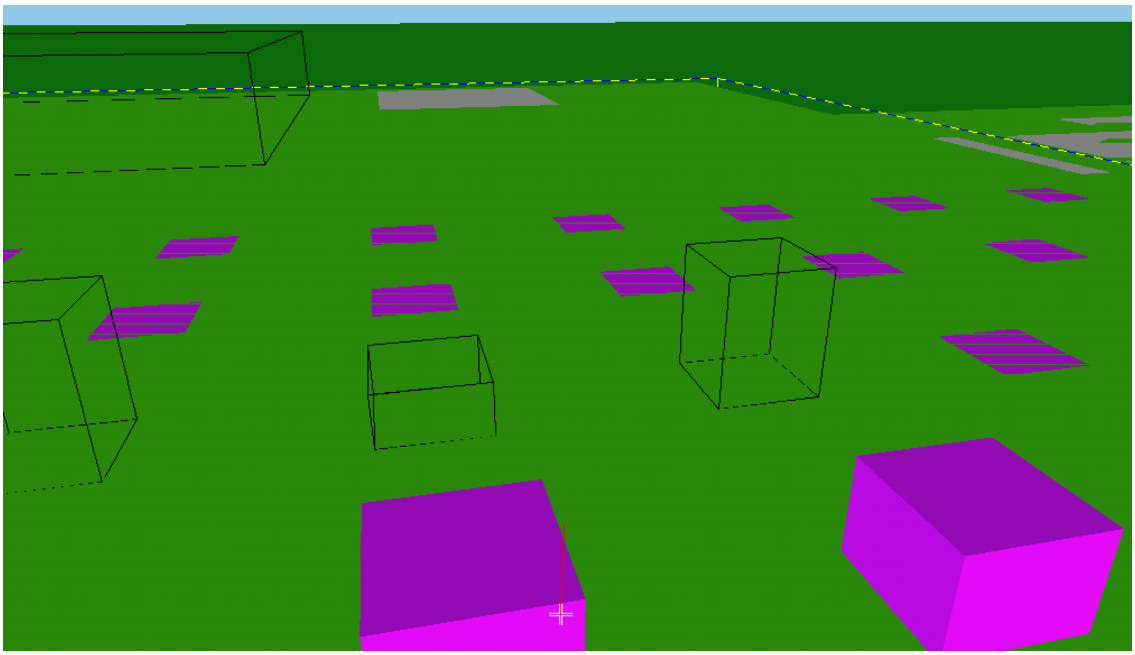


Figure 5.11: Objects are displayed with different details while navigating.

Figure 5.11 shows all the different detail modes. The file object nearest to the camera is painted normally. The three objects in the next row are painted only as wire frame models. Finally, the objects in the last two rows are only displayed as rectangles on the floor. On the left side there is also a plateau in wire frame mode visible. If a plateau is not painted as a full 3D object, its child objects will not be painted. This speeds up the calculation, because if a plateau is not near or large enough to paint with all details, then its child objects will not be displayed.

If there is no animation to be done, the user is motionless and the whole landscape with all details can be painted. For instance, when the user navigates to a specific directory it should be fast with smooth animation, so details are hidden during animation. When doing some operations like copying or moving some files the whole landscape is painted.

The primary goal of the algorithm is to keep the frame rate constant. To reach this goal, not every object has to be painted, but since we do not not know how fast the machine is on which the application is running, the values of the above variables are different on every machine. Due to the dynamic nature of the visualisation, we have to display sometimes very large or very small pyramids. So we have to change the values of these variables dynamically after every redraw. This is done in the `iicm.lscape.Painter.endPaint()` method. As mentioned above this method calculates first the current frame rate. After that it is checked if the current frame rate is equal to the user requested frame rate. If it is not equal a multiplication factor (`diff`) is calculated using the difference between the requested and the actual frame rate. This multiplication factor is than used to change the value of the variables.

```
draw_doc_full_ += diff;
draw_doc_wire_ += 0.5f*diff;
draw_doc_rect_ += 0.1f*diff;
```

If the actual frame rate is too high the `diff` value will be negative and so more details will be painted. If the frame rate is too low it is positive and fewer details will be painted. The constants for the calculation of `draw_doc_wire_` and `draw_doc_rect_` were inserted to change the size of the region where less details are displayed slower. So we have following paint algorithm:

```
paint ground plane
push root document on stack

do
  get document form stack
  if camera moves then
    calculate detail level
  else
    set detail level to paint everything
  paint the document shape
  if the children of the document are visible
  and the detail level is at least wire frame then
    put all sub directory documents on stack
    for all file objects do
      if camera moves then
        calculate detail level
      else
        set detail level to paint everything
      paint the document shape

while the stack is not empty
```

The overall impression when moving through the landscape is that of moving through fog. Only when the camera comes close to an object does it become visible. First only as a rectangle on a floor and suddenly popping up to a wire frame object. If the camera is very close, it is displayed in full detail. However, large objects in the background are displayed at a higher level of detail. Standing still displays the entire landscape in full detail. The displayed details strongly depend on the hardware used and on the size of the landscape. Thus moving through the same landscape on two different machines will usually result in different detail levels.

5.5 Document Tree

The *Document Tree* block represents the internal representation of the hierarchy. The central class is the `Document` class. This class is the abstraction of a tree node. In the case of the 3D Explorer, it represents a file or directory. Files and directories are not handled in a special way. To distinguish between directories and different file types the `type` field is used. It can have following values:

```
public final static int collection = 3;
public final static int text = 5;
public final static int java = 6;
public final static int classf = 7;
public final static int picture = 8;
public final static int sound = 9;
public final static int movie = 10;
public final static int compressed = 11;
public final static int postscript = 12;
public final static int library  = 13;
public final static int other = 14;
```

When an object of the `Document` class is instantiated, the type of the document is passed as a parameter. The constructor of the object sets the type field and also creates a new `Shape` object depending on the type. This `Shape` object is stored in the `shape` variable of every document. For instance, a directory is assigned a green box as the default shape, and a Java "class" file a magenta coloured cube. For more on the visual representation of a `Document` object see Section 5.5.1.

The constructor of the `Document` class has also some other parameters, like the file or directory name, the size of the file, and the creation date. All these values are stored for later use.

For the internal tree representation the two fields:

```
public Vector collections;
public Vector documents;
```

are used. The `Vector documents` holds the files and the `Vector collections` the subdirectories. So we have the a tree like the one shown in Figure 5.12. To build this structure the method `addChildren` is used. When new nodes of the hierarchy are loaded (see Section 5.6), the `addChildren` method of the object which is the parent of these new objects is called. The new object are passed as a `Vector` of `Document` objects. `addChildren` just adds the objects to the `documents` `Vector` if it is a file, or to the

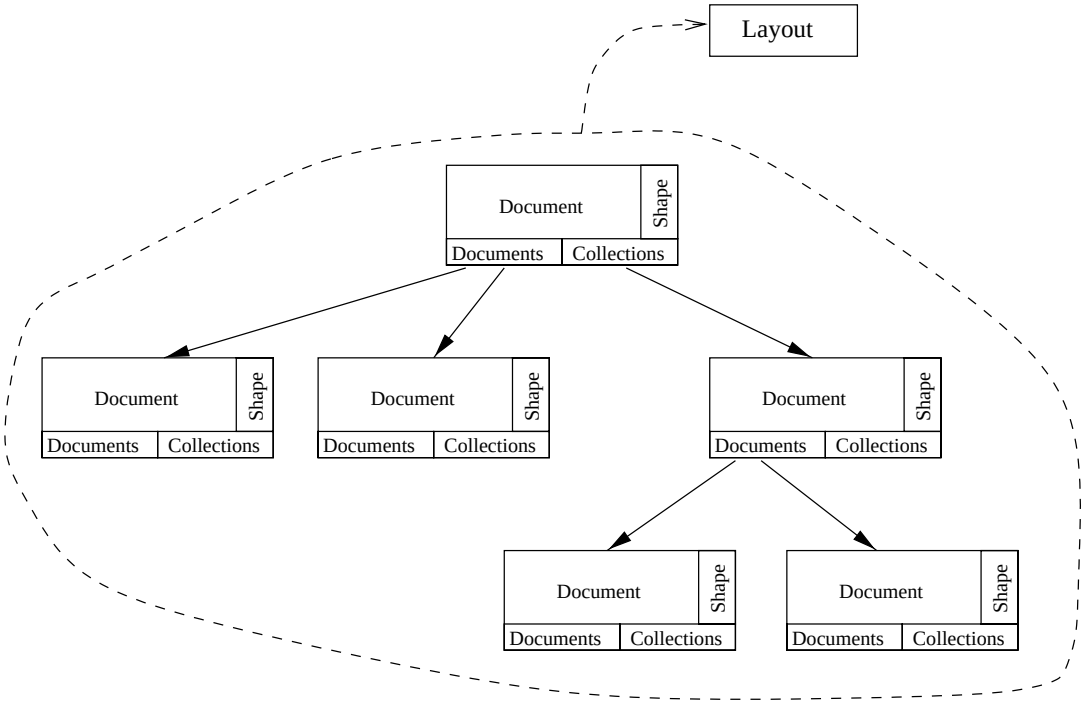


Figure 5.12: A small document tree.

`collections` `Vector` if it is a directory. Additionally it is checked that there are no duplicated objects in one of the `Vectors`.

After the child objects have been added, they are not immediately visible in the landscape, since their 3D shapes have not yet been laid out. So the `loaded` flag of the parent document is set, but not the `childrenVis` flag. The `childrenVis` is used by the `Painter` to check if the next level of the hierarchy should be painted. To make the children visible the variables:

```
private long sizeCount=1;
private long childrenSizeCount=1;
```

have to be calculated. As described in Section 4.2.2 the space of the parent plateau is distributed according the size of the subplateaus. The variable `sizeCount` stores the size of a document. If this document is a file this is just the file size. When it is a directory further calculations have to be done. These calculations are done by the *Data Server* (see Section 5.6). The `childrenSizeCount` just holds the added `sizeCount` values of the child documents. While calculating, the *Data Server* uses following methods to set the variables:

```
public void setSizeCount(long sc) { sizeCount = sc; }
public void setChildrenSizeCount(long sc) {
  childrenSizeCount = sc; }
public void addSizeCount(long sc) { sizeCount += sc; }
public void resetSizeCount() { sizeCount = 1; }
```

After the correct values for the size are set, the method `placeChildren` can be called. This method is also called by the *Data Server* after it has loaded the new documents and

calculated the directory sizes. First `placeChildren` starts to sort the elements in the `collections` and `documents` `Vector`. The `Vectors` can be sorted by different criteria, including:

```
public static final int BY_NAME = 1;
public static final int BY_SIZE = 2;
public static final int BY_TYPE = 3;
public static final int BY_DATE = 4;
```

To sort the `Vectors` easily by different criterion the `Document` class implements the `Sortable` interface. This interface defines just sort criterion above and three methods:

```
public Object getValue(int type);
public boolean gt(Sortable s, int sort_by);
public boolean lt(Sortable s, int sort_by);
```

`getValue` returns the value according the given `type` of sort criterion. For instance, calling this method with the `BY_SIZE` constant returns the size of the `Document` as a `Long` object. The methods `gt` and `lt` return `true` if the object is greater than, or less than the given object of the type `Sortable`. The comparison is done by the `sort_by` criterion, which can have the values of one of the above defined constants. In the case of using `gt` and `lt` with the `BY_TYPE` criterion it is usual that there are more than one documents of the same type. So the methods compare first the type, an if it is equal the names are compared. Thus we get the documents sorted by type, and in the different types the documents are sorted by name. The same is done with equal sized documents.

To actually sort the documents the `placeChildren` method calls the method `QSort.quicksort()`, which is an implementation of the Quicksort algorithm. The parameters of this method are just the `Vector` which should be sorted and the sort criterion. The sort criterion is a global value for the landscape, which can be set by the `setSortCriterion` method. After that the entire landscape has to be laid out anew, as described in Section 5.4.1.

After the sorting is done, `placeChildren` starts to layout the 3D representations of the documents. This is done by using the object `layout` which is a class member of the `Document` class. So every instance of `Document` can access this variable (see Figure 5.12). The layout is actually done by calling

```
layout.placeChildren(this,frame);
```

For more details on the layout of the 3D object see Section 5.5.2. After the layout is done, the flag `children_placed_` of the parent object is set. Finally, the 3D representation of the documents can now be set visible by setting the flag `childrenVis` with the method `setChildrenVis`.

5.5.1 Object Shape

The 3D shapes which are used by the 3D Explorer are bundled in an extra package, the `iicm.lscape.shapes` package. As can be seen in Figure 5.13 the base class of this package is the abstract `Shape` class, which defines basic properties of a shape such as the

coordinates of the bounding box and the data structures for the colours and vertices of the shape. Since we use the `GE3D.drawFaceSet()` method of the GE3D engine to draw every object, we use data structures which this method uses as parameters. Such as

```
/**
 * the vertices of this shape
 */
protected float[] verts;
/**
 * the definition of the coorinates of this shape
 */
protected int[] coordinds;
/**
 * the normals for every vertex of this shape
 */
protected float[] fnormals;
/**
 * the numer of vertices of this shape
 */
protected int numcoordinds;
```

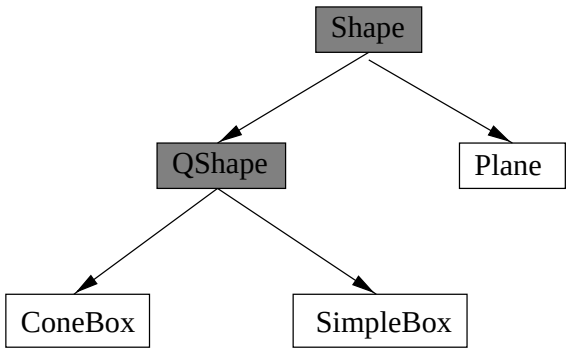


Figure 5.13: The class hierarchy of the 3D shapes.

The central method of the `Shape` class is the `paint` method. This is a generic paint method for painting a shape with the `GE3D.drawFaceSet()` method. The `paint` method has to be provided with the detail mode of the shape. The detail mode is used to paint a shape in less detail as described in Section5.4.2. For instance only the wire frame object or a rectangle on the floor has to be painted.

Another method for painting is the `paintFocus` method. This method is used by the `World` class to paint a coloured wire frame box around a selected object (see Section 5.4).

Of course, there are some methods to change the object's size and position, like `setSize` to set the object size, or `move` and `moveto` to move the object to a new position in the 3D space. An important method is also the `setSizeTentative` method. It sets the size of the object only temporarily. This method is used by the layout algorithm, so the size can easily be changed until it is fixed by `setSize`. An interesting feature is the ability of the `setSize` method to scale the object in height. To do this an extra parameter is passed to this method, which defines the size, in our case the file size. To avoid very high objects for large files, the height is scaled logarithmically. Files which are smaller than 2 kilobytes have a constant height.

There are also some methods which are defined abstract. These methods have to be implemented by an inherited class. These methods are:

```
public abstract void setColor(int t);
public abstract float getHeight();
```

The class `Plane` extends the `Shape` class. It describes the ground plane of the landscape. It just defines the abstract methods, and replaces the `paint` method by a more simple one. The ground plane is painted every time, no code for detail levels is needed.

More important is the abstract `QShape` class, which also extends the `Shape` class. The "Q" stands for quadratic, which describes the base of the shape. It basically adds some new abstract methods:

```
public abstract void setTopSize(float[] newMin, float[] newMax, long f);
public abstract float[] getTopMin();
public abstract float[] getTopMax();
public abstract float getFree(Document root, int pos,float maxdepth);
```

The `QShape` class represents square boxes, which may be bevelled. Due to this fact, the `getTopMin` and `getTopMax` were added to get the coordinates of the corners of the top area. This information is necessary for the layout methods to place the child objects on the top of a directory plateau. The `getFree` method is also used by the layout methods to determine the area on the plateau which is not occupied by another object. For details on doing the layout of objects refer to Section 5.5.2.

`QShape` implements another important method for quadratic shaped objects, the `interQuad` method. This method returns true if the shape intersects the shape given as an parameter, false otherwise. This method is intended to be used by the implementations of the `getFree` method.

The abstract class `QShape` is extended by the `SimpleBox` and `ConeBox` classes. These classes are the two object shapes which are currently used by the visualisation. While `SimpleBox` is used for the files the `ConeBox` objects are used for directories by default. The two classes implement all the abstract classes defined by their parent classes. The only difference in the two classes is the shape. The `ConeBox` has the same bottom size as an equivalent `SimpleBox`, but the top square is slightly smaller. This results in a slightly bevelled box.

5.5.2 Object Layout

The *Layouter* block is closely related to the *Document Tree*. The `Document` class contains a `layout` class variable, so every instance of the `Document` class can access this object. `layout` is an instance of the `Layout` class. As can be seen in Figure 5.14, the `Layout` class is defined as abstract. Only classes derived from `Layout` can be used as the `layout` variable. This construction was chosen to implement different layout strategies. These strategies are encapsulated in an object which has to extend the `Layout` class. Thus they have to implement the abstract methods defined in the `Layout` class.

The `Layout` class defines only two abstract methods, which are

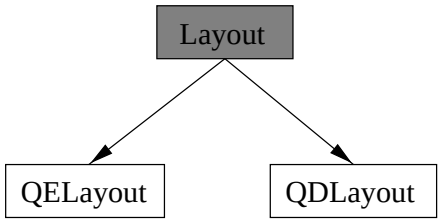


Figure 5.14: The class tree of the *Layout* classes.

```
public synchronized abstract void placeChildren(Document root, LSFrame frame);
public synchronized abstract boolean placeSubDocs(Document doc, LSFrame frame);
```

The method `placeChildren` should lay out or place the child documents of the given document. How this is done depends on the implementation of this method.
The `placeSubDocs` method also has to lay out the child documents and all documents in the subdirectories. That means that this method simply uses `placeChildren` for the given document and calls this method for every subdirectory.

Some methods are implemented directly in the `Layout` class.

initStart: This method generates the base plateau when the application is started. This plateau is given a constant size at startup.

setRootSize If the user loads the parent of the base directory of the visualisation, this method is used to calculate its size.

moveRec Moves recursively all child documents of the given `Document` to a new position, so an entire pyramid can be moved.

Currently only two classes implement the abstract `Layout` class, `LayoutQD` and `LayoutQE`, whereby `LayoutQD` is used as the default layout engine.

LayoutQE

The *QE* in the name of this class stands for *quadratic* and *equal*. This simply means, that shapes for the directories have a square base shape and all subdirectories of a given directory have the same size. This layout is the most simple one with quadratic shapes. It is not very useful, but it shows the advantages of the other layout.

The central method is the `placeChildren` method. This method really calculates the size and positions of the subdocuments of the given document. Firstly, all the files are placed. They are arranged in a raster of rows and columns atop the base plateau, starting from the bottom left corner. To do this the width of the 3D file object is calculated by multiplying the height of the parent directory with 0.95. This size was determined experimentally and gives a good looking visualisation. If there are many files, it could happen, that there is not enough space on the directory object to place everything. If that happens, the size of the file object is scaled using following algorithm. This algorithm also scales the file objects if there are many subdirectories to place. So the subdirectories become larger and are visualised better.

```
if file objects can not be placed atop directory plateau then
  if no subdirectories to place then
    scale file object to use the entire space on the directory plateau
  else
  if there is at least one big subdirectory to place then
    scale file objects to use only 50% of the space on the directory plateau
  else
    scale file objects to use only 66% of the space on the directory plateau
else
  if there is at least one big subdirectory to place then
    scale file objects to use only 50% of the space on the directory plateau
  else
    scale file objects to use only 66% of the space on the directory plateau
```

After the size of the file object is determined, a simple loop calculates the actual position of the file objects in the raster. When setting the size of a file object, the `sizeCount` of this `Document` is also passed to this method. With this value it calculates the height of the file object (see Section 5.5.1). Since the `sizeCount` of a file `Document` corresponds to the file size, the height of the 3D object represents the file size.

Secondly, the subdirectories have to be laid out. This is done by calculating the free space on the plateau which was left by the file objects. After that the free space is divided into equal sized squares by the number of subdirectories. Then the size and position of the subplateaus are set to fit in these squares. This is all done in a simple loop.

The layout is done very fast, because every thing can be calculated straightforwardly. No time-consuming loops have to be used. However, the resulting layout is not very appealing.

LayoutQD

The `LayoutQD` class is very similar to the `LayoutQE` class. It also places the 3D objects on a directory plateau. The layout of file objects is done by the same algorithm as in the `LayoutQE` class. The major difference to the `LayoutQE` class is the layout algorithm for the subdirectories. The subdirectories are laid out as different sized plateaus. The subplateau size is proportional to the data contained in the corresponding subdirectory. The base shape of the plateau is also a square.

The `placeChildren` method implements the layout as defined in the abstract base class `Layout`. After the file objects are placed using the same algorithm as the `LayoutQE` class, the layout of the subdirectories starts. First of all the free space for the subplateaus is calculated. This results in a rectangular area as shown in the left picture of Figure 5.15. The resulting area is divided into pieces according to `sizeCount` of each directory. The `sizeCount` variable is a measure for the information content of a directory. It is determined prior to the layout by the *Data Server*. So we get the area for each directory object. With the area, the size of the base square of the shape can be easily calculated by using the square root of the area for this object. This size is set only tentatively for the shapes, because often the square shapes can not be placed to fit exactly on to the base plateau (see the right picture of Figure 5.15), although the sum of the subplateau areas equals the area of the free area on the base plateau. The problem of placing quadratic shapes exactly into a rectangular area is non-trivial. A simple but non-optimal iterative algorithm is currently used.

All directory objects are moved to the top left corner of the base plateau. Then every object is moved to a position on the base plateau where it fits on the base plateau and does not

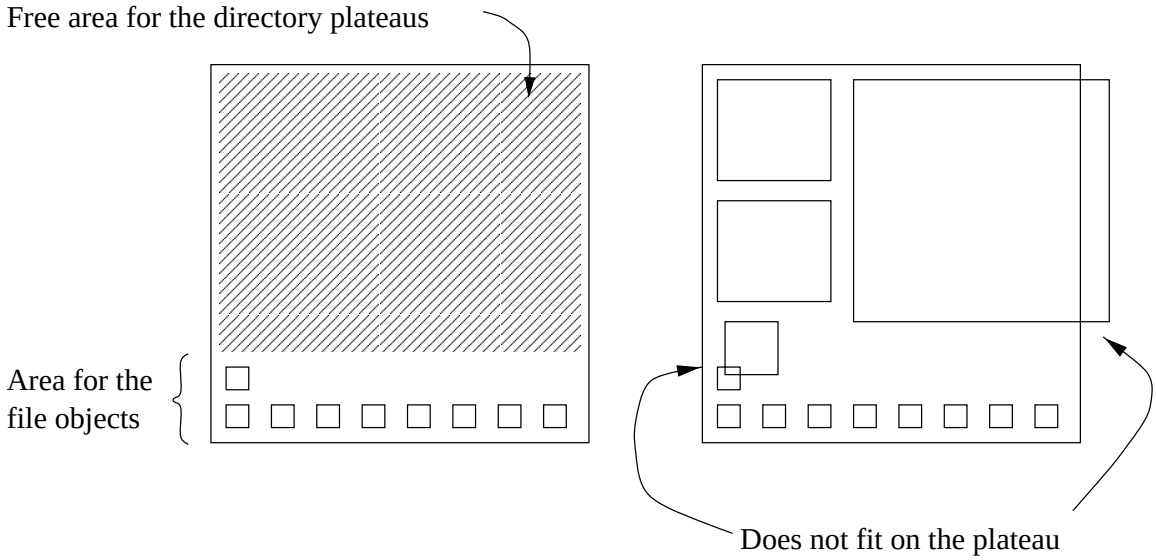


Figure 5.15: The problem of placing squares in a rectangular area.

intersect any of the objects positioned before. To find such a position a scan line algorithm is used. There is a horizontal and a vertical scan line, which represent the next free position for a shape. Starting this algorithm the scan lines cross each other at the top left corner of the base plateau. This is where the first shape is placed. After that the vertical scan line is moved the width of the shape to the right. And the horizontal scan line is not moved. So the next shape will be placed right to the first shape (see left picture of Figure 5.16). This is done until a shape can not be placed on the base plateau as shown in the middle picture of Figure 5.16. In this case, the horizontal scan line is moved down to the first bottom corner of a previous placed shape. An the vertical scan line is moved to to an position where the intersection with the horizontal scan line is not in a previous placed shape. This is shown in the right picture of the Figure 5.16.

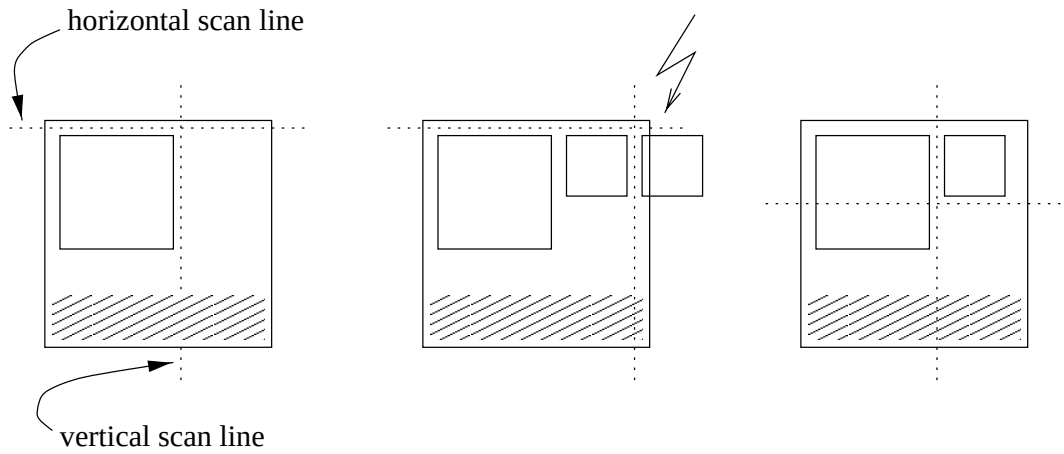


Figure 5.16: The scan lines to find a position for a directory object.

If one of the objects can not be placed on the base plateau all directory objects are shrunk

in size. The objects are then moved back to the top left corner of the base plateau and the positioning is tried again. The shrinking is not done by a constant factor. When an object can not find a correct position, only a position can be found where the object reaches over the plateau boundaries. This overlap is used to calculate a scale factor for all directory objects. Using this value less iterations have to be made.

Once all directory objects have a valid position, the tentative size and positions are set permanently. Usually five to six iterations have to be done to place the objects correctly. When directories with many subdirectories have to be laid out, this can take a while, but in normal use the speed of this algorithm is sufficient - far more time is used to read all the file and directory details from the hard disk.

5.6 Data Server

The *Data server* is responsible for fetching the data from the file system. The main class for this task is the `Server` class. To easily adapt this application for other hierarchical visualisations, this class was designed to retrieve data from a variety of different data sources. So it is fairly easy to write a class to fetch the data from a Hyperwave server, for example.

The loading of the data is done in a separate thread, so it is possible to navigate in the landscape while loading is done in the background. The management of the different server threads is done in the `World` class. For this purpose the `World` class has two members:

```
private Server server_;
private Vector serverThreads = new Vector();
```

The `server_` variable is initialised in the constructor of the `World` class. It is the default server instance for the application. This default object is mainly used by the `LayoutParam` class to change the layout in a separate thread (see Section 5.4.1). The threaded design of `server_` is not used in this case, because the entire change of the layout is done in a separate thread.

The Vector `serverThreads` holds the active threads of the `Server` class. If the user initiates a loading operation, for instance by opening a directory, the method for this event calls the `getServer` method. This method of the `World` class returns an instance of the `Server` class, which is stored in the `serverThreads`. This method also deletes unused server threads from the `Vector`. With the returned `Server` object the calling method can now start a loading operation in a separate thread.

The `World` class has also two other methods to handle the server threads. Firstly, the `stopThread` method stops all `Server` threads in the `serverThreads` and deletes them from the `Vector`. This is done after the user pushed the *Stop* button in the status line. Of course, this button is only shown in during loading of data. Secondly, the `IsAServerActive` method. The method returns true if a thread in the `serverThreads` list is active. A thread to change the layout and one to load new data are not allowed to be active at the same time. So `IsAServerActive` is used before a new thread to change the layout is started. If `IsAServerActive` returns true the new thread will not be started and an error window is displayed.

5.6.1 A Threaded Server

The `Server` class encapsulates all the functions to load hierarchically structured data in a separate thread. The main members of this class are

```
private FileServer fserver_;
private TServer tserver_;
```

The `fserver_` is an instance of the `FileServer` class which encapsulates all the methods which really access the file system. To get the data from another source only this class has to be reimplemented for the type of data source. For instance, if a Hyperwave Server is to be visualised, only the methods of this class have to be reimplemented to access the Hyperwave Server instead of the file system. The `tserver_` extends the `Thread` class to generate a new thread for the `Server`.

When a `Server` object is instantiated, an `fserver_` object is constructed. After that the `tserver_` objects is instantiated. The constructors of both objects receive the `Server` as parameter. The constructor of `Server` then sets the priority of the thread. This is simply done to give this thread a lower priority than the main thread of the application. The server needs more time to load, but the user has reasonable navigation speed.

The thread is then started. The `run` method of the `TServer` class simply calls the `load` method of the `Server` class. At the beginning of the `load` method, the called thread is set to the wait state by calling `wait`. This state ends when the `load_now_` flags is true. This flag is set by the main thread of the application, when a user initiates some loading operation. If this is done the server thread wakes up and starts an operation depending on the `operation_` field. This variable was also set by the application thread. Currently three operations are implemented:

```
private final static int REC_LOAD = 1;
private final static int INIT_START = 2;
private final static int GET_PARENT = 3;
```

`INIT_START` is used to load the first plateau of the visualisation. This is a special case because the size of the shape is constant. If `operation_` is set to `GET_PARENT` the parent directory of the current directory is loaded. Of course, this is only done if the parent of the current directory was not loaded before. In such a case the parent is already in the internal data structure and has only to be made visible.

The third operation is the `REC_LOAD`. This is the most often used operation because it loads the hierarchy data to the internal data structure. This operation will therefore be discussed in detail here.

When the user double-clicks on a closed plateau, the loading process starts. Firstly, the event handler of the `LSCanvas` class gets an event and calls the `toggleColl` method of the `World` class. This method checks if the directory is open or closed. In our case it is closed and should be opened, so `loadChildren` is called with the `Document` object for this directory as parameter. In the other case the method `hideChildren` is called. `loadChildren` gets a `Server` object with its associated thread by using the `getServer` method. After that the method `recLoad` of the `Server` object is called. The `Document` whose children should be loaded is passed as an parameter. Also, the variable `loadLevel`

is passed on. This variable defines how many levels of the hierarchy should be opened by this operation. Usually this is one, but the user can set it to a higher value.

The `recLoad` method just stores the passed parameters in private member variables and sets the `operation_` flag to `REC_LOAD`. After that the status line is set to the loading status line as described in Section 5.3.2. Then the flag `load_now_` is set to `true`. The thread associated with this `Server` objects waits until this variable becomes true, as described above. To wake up the thread the method `notify` is called. After that, the event handling of the double click is done. The main thread goes back to its usual work, waiting for user input. Parallel to the main thread the `Server` thread starts to work, so the user can do other things while new data is loaded.

The `Server` thread checks the `operation_` value and starts to load the hierarchy data. This is done by calling the method `recLoader`. This method loads the children of the given `Document` recursively to the given depth level. It simply loads the children `Documents` with the `getChildren` method and then it calls itself with every newly loaded `Document` object representing a directory and the decreased level value. The algorithm is a *depth-first* algorithm.

`getChildren` uses the `Filserver` object to load the child object of the given `Document`. This is done by the `addChildren` method, which loads the file system data and then generates the `Document` objects representing these file system items. It also inserts these new objects into the *Document Tree*. After that, the size values for the added objects which represents a directory have to be calculated. As described in Section 5.5.2 these values are needed to lay out the objects in the 3D representation. Then the new objects are laid out by calling `placeChildren` and setting visible. Finally, the entire landscape is repainted to display the added object in the visualisation.

To calculate the size values for the added directories the method `calcCount` is used. Two different methods to calculate the size values were implemented:

- Add the sizes of the files in the directory up to a given depth. So the visualisation displays the disk usage.
- Add the number of files in the directory up to a given depth. The visualisation usually looks better, because directories with many files get a bigger plateau, no matter how big this files are. This strategy for a directory with many small files results in a small plateau, so the 3D objects for the files become very small.

Both calculation strategies are done by `calcCount`. This method is used recursively to calculate the value to a given depth. Of course the size value is only an estimation, because the hierarchy is scanned only to a given depth. To get the exact value, the whole subtree would have to be analysed. This is of course very time consuming, so only a given depth is analysed. By default the scan depth is set to one. The method `calcCount` is defined as:

```
private long calcCount(Document doc,int level)
```

`doc` represents the directory document, whose size value should be calculated. The parameter `level` is a counter which represents the current depth. On the first call the method receives the directory and the scan depth as parameter. First the it is checked if the `level`

is negative, if this is true the recursion ends here. After that the children of the subdirectories of the given directory are loaded. The children are only added to the *Document Tree*, no size calculation or layout is done. After the children are added to the tree, the method `calcCount` is now recursively called with every new subdirectory and a decremented level counter. This call returns the size value for this subtree and is added to size variable. Lastly the size values of the files are added to the size variable. The size value is either the file size or a constant (usually one), depending on the calculation strategy. At the end of the `calcCount` method, the calculated size value is returned.

This recursive algorithm is a so-called *depth first* algorithm. To improve this algorithm it is checked if the given subdirectory is nearly empty. If this is true, the level counter is increased by one to scan an additional level. File systems often have directories which contain no files but some subdirectories. In this case, the original algorithm adds only a constant for every directory. This results in a very small increase in size value, although the subdirectories may contain many files. The improved algorithm would also scan the next level for better subtree size estimation.

Chapter 6

Future Work

Since the 3D Explorer is the first application of the Information Pyramids visualisation technique, numerous improvements are possible. Some of them, such as texture support, were simply not included due to performance reasons. However, hardware in general and 3D graphics hardware in particular, is improving steadily. In future versions this will not be an issue. Some other features suggested here were simply not implemented for lack of time.

6.1 Edit Functions

Currently, the 3D Explorer is only a tool to visualise hierarchies. Typical authoring functionality such as:

- Cut
- Copy
- Paste
- Delete
- Undo

is not yet implemented. All these operations dynamically change the hierarchy. The visualisation has also to support the dynamic nature of these operations. The most important change will be in the layout classes. In particular, the *paste* operation needs an improved layout strategy, since objects have to be added to a directory and thus to the plateau in the visualisation. This addition of objects should be done in a way that it does not completely change the positions of objects. For instance, pasting a directory into the current one should not change the positions of the previous subdirectories. Of course they can not have exactly the same size and position, because some space has to be made for the new directory, but their rough position should be the same. If a directory “images” was located in the top left corner of the parent plateau, it should also be there after the insertion. Only after really large changes could it happen to find the “image” directory at the bottom right corner, for example. It is very important to retain the approximate position of objects in order to minimise the cognitive load of the user.

6.2 Attribute Mapping

Data attributes are currently mapped on to the 3D visualisation in a static and limited way. To make it more flexible a “mapping editor” should be implemented. With this tool, file extentions (types) could be mapped to specific colours or visual representations. Other attributes of a file should also interactively be mapped to some visual representation. For instance it could be possible to map:

- the age of a file to its colour. For example, new files would be red and the older they are, the more blue they become.
- the type to a specific 3D object. A text file could be a represented by a book or an image by a frame. Maybe different 3D icon sets could be created. On slow machines a simple, and on faster computers a very detailed 3D object could be used. Another feature would be to use the content of a file for texturing. The object for an image file could use a thumbnail of the image as a texture for the 3D representation. Or text files could have the first few lines represented by a texture.
- the type of the file to a texture. This textures should be carefully designed, because the user should be able to distinguish easily between different object. Too fancy textures would distract the user more than they would help.

Of course, not all of the mappings above could be used at the same time. A meaningful combination of some of these would be best. Maybe it could be possible to design different schema of possible mappings and methods. Users could then easily set a schema which suits them best. Due to the many different hardware configurations, it is use useful to let users select a scheme which is best for their particular hardware configuration.

6.3 Session Management

All options the user changes are now only active during the current session. Of course, it would be desirable for configuration settings to be saved between sessions. Also, the current base directory and the opened directories should be presented after a restart. If the user uses the command line arguments to open the application with a given directory, this new directory should have a higher priority than the last session.

The layout of the objects should also be included in the session management. The approximate positions and sizes of 3D objects should be the same in every session. This should work like the layout features mentioned in Section 6.1. At the end of a session the layout of the object should be stored in a file. When starting a new session this information should be used to layout the pyramids. If there were made only small changes between the sessions the overall structure should be the same as in the first session. That means that objects which were found in the top left corner of a plateau should also be there now. If, for instance, a whole directory was deleted, there could be a large empty space in the visualisation where the plateau of this directory was located. The user could then immediately see that some changes in the file structure were made. After some sessions this space could be used for

newly created directories. Only if really large changes were done between sessions should the layout reflect this. If there are no similarities in the hierarchical structure between two sessions, it is of course not a good idea to try to create a similar visualisation. In this case the layout can be done afresh without any constraints.

6.4 Navigation History

A navigation aid which is not implemented currently is a navigation history. Nevertheless, it is a very useful navigation tool which should increase usability. An "animation buffer" should store the completed animations and moves of the camera. To go back in the navigation history a new *Back* button in the tool bar could be included. Also a *Forward* button should be included to go forward in the history buffer. These buttons should work like the buttons in a conventional WWW Browser. Instead of displaying visited pages it would move the camera between previously visited positions.

6.5 Hyperwave Integration

Hyperwave [Mau96] is a second generation web server. It was initially developed under the name Hyper-G [KMS92] at the IICM [1] in Graz. The successful development team later founded the company Hyperwave and renamed the product. Hyperwave has many features which can not be found in ordinary, first generation web servers. In most cases Hyperwave will be accessed through a WWW browser, using the WWW gateway. For a browser a Hyperwave server acts like an ordinary WWW server. To offer the user the additional features of Hyperwave Javascript is used.

The Hyperwave server itself can be divided into four main parts:

1. **Document Database:** stores the documents and serves them to the clients. In most cases through the WWW gateway. This database also fetches and caches remote documents from other servers. Each object has a unique ID independent of the physical location. Since numerical IDs are not easy to handle, every object can have one or more symbolic names.

2. **Link Database:** stores the links and attributes of the documents. Storing links separately from the documents has many advantages. It is possible to follow links backwards, but the main advantage is that the server is responsible for link consistency. That means that there are no dangling links on a Hyperwave server.

3. **Structure Information:** there are some structural objects to store documents in a well-defined way. These objects are *collections*, which can contain documents and other collections, forming a hierarchical structure. Another object is a *cluster*, which can contain different multimedia documents. These multimedia documents are presented simultaneously when the cluster is viewed. A cluster could also hold documents in

[1] Institute for Information Processing and Computer Supported New Media at Graz University of Technology

multiple languages. A client can define its preferred language; when opening a multilingual cluster the document with the most suitable language is presented. Another structural element is a *sequence*, where documents are stored in a fixed sequence. On the client side, next and previous buttons are embedded into the presented documents, to navigate in the sequence. The main advantage of a sequence is that when adding or removing documents the next and previous links are updated automatically.

4. **Search Index:** provides built-in search facilities. Older WWW servers need an external search engine to provide this feature. The search engine supports full text search on the stored documents. Additionally users can do an attribute search on attributes of a document such as author, creation date, last modification date, or keywords.

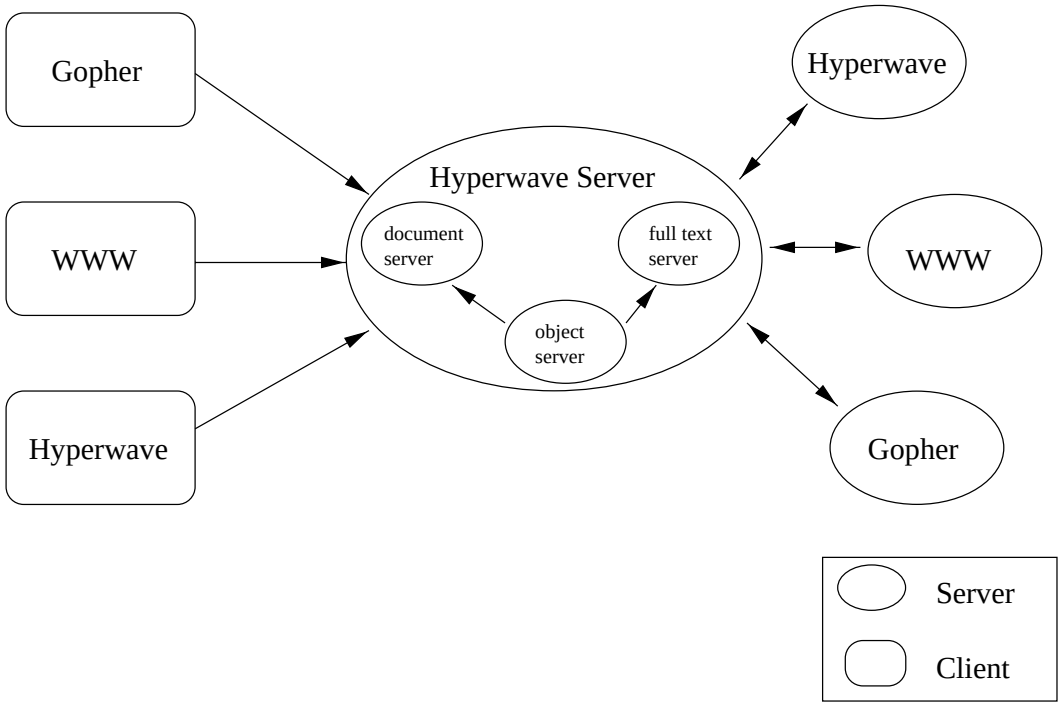


Figure 6.1: The structure of a Hyperwave server. [And94]

Another feature of Hyperwave is user accounting. Users and user groups with different access rights can be defined. There are numerous other features of Hyperwave which can not be mentioned here, but it should be clear that Hyperwave is a powerful solution for serving hyperlinked multimedia documents.

Currently, a Java API is under development, which can access a Hyperwave server. It is called ANT/Dino. All the features of a Hyperwave server can be accessed through this API. Of course all the attributes of the documents on the server can be accessed and modified. The access methods of this API are similar to the java.io API. In fact, it a superset of the java.io package with some new methods defined to access special features of Hyperwave. Since ANT/Dino gives access to Hyperwave through file system methods, it is easy to write a Hyperwave version of the 3D Explorer. Of course some new features have to be added to the 3D Explorer to take advantage of the full Hyperwave functionality.

6.6 Improved Layout

As described in Section 6.1 and Section 6.3, the layout algorithm has to be improved in many ways. The layout algorithm should not only have extended features, it should also be optimised. The layout of the 3D objects is a very time critical operation, because the user waits until the new objects are laid out and displayed. Too long calculation times will bore the users and they will not use this visualisation tool anymore. So layout time should as low as possible. The current algorithm performs well for average file systems, but for directories with many subdirectories it could take an unreasonable amount of time to complete the layout.

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 object 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.

To actually build clusters of objects with a given relation a *simulated annealing* algorithm could be used. These algorithms are usually very slow, because they are iterative algorithms where every iteration step needs $O(n^2)$ time. As the name of the algorithm suggests, it was developed in physics to model the annealing of metal. The algorithm needs as input some objects and relations between these objects. The relations are used as forces between the objects. Closely related objects have large forces of attraction. Objects with no relation have repulsion forces. The algorithm iteratively searches for a configuration of objects where the total energy is minimised. To speed up the algorithm Matthew Chalmers [Cha96b] published a modified version, which needs only linear time for every iteration. Normal algorithms calculate the forces to an object by adding the forces of the n-1 objects on this object. The linear algorithm uses only a subset of objects to calculate the force. These subsets are built according to some constraints, so the result is almost the same as that of the original algorithm. Since the layout has to be as fast as possible this algorithm would be a good choice, but nevertheless this algorithm needs quite a long time to finish. So the cluster layout should be not used by default. The user might perhaps issue a command to build clusters for the files of the selected directory.

6.7 Populated Information Terrain

Populated Information Terrains (PIT) are virtual data spaces that may be inhabited by multiple users, so people can work co-operatively within data as opposed to with data. Users of a PIT are aware of the presence and the actions of other users within the same PIT. They are also able to communicate. A user interested in a particular subject may question other users browsing the same information or even ask directions to relevant data.

The Information Pyramids could be extended to a PIT. In the case of visualising a file

system it would be difficult to implement and not even very useful. When visualising data of a web server, for instance a Hyperwave server, this would be a great feature. The information stored on large web servers is enormous, and co-operative work would potentionally be very helpful. Inexperienced users can easily get information by asking other users.

Users could be represented by user configurable 3D objects, so-called *avatars*. A simple chat-application will be necessary to enable communication between users. Two or more users should be able to talk to one another. To start the communication the user could double click on the avatar of another user. There should also be a way to visualise if a user is busy and therefore is not able to talk. Maybe the colour of the avatar or a special texture could represent this. Another feature of co-operative work would be a *guide function*. When a user has been asked the way to some specific data, they could enable the guide function. This will "tie" the asking user to them, so that they could move to the location with the other user. During this move the tied users are not able to navigate by themselves. Of course, the guide function should be disabled when the data is reached. Maybe some robots could be implemented to serve as guides, so this would be a guided tour through the information on the server.

An Information Pyramid application with PIT functionality would be a useful feature. The features of the implementation suggested here are just the beginning, surely many other useful features could be developed to support co-operative browsing of information.

Chapter 7

Concluding Remarks

Numerous techniques to visualise hierarchies are available today. Several techniques go beyond the traditional approach of 2D scrolling browsers with horizontal tree layout. Information Pyramids, presented in this theses, utilise 3D graphics to compactly visualise large hierarchies.

A plateau represents a tree node. Smaller plateaus arranged atop such a plateau represents its child nodes. Separate 3D icons are used to represent leaf nodes of the hierarchy. The size of a plateau is proportional to the size of the subtree it contains. These simple layout rules lead to the overall impression of pyramids growing from the ground plane. This technique also scales well to both wide and deep hierarchies.

3D objects have many attributes such as size, shape, colour or textures which could be used to represent attributes of the tree nodes. For instance, colours could be used to show different types of node. This attribute mapping is one of the strength of the Information Pyramids. Users can perceive much information on tree nodes simply by looking at them. It also reduces the cognitive load, because every subtree has its unique appearance, so it is fairly easy to recognise.

Information Pyramids use a landscape metaphor. It typically uses less 3D space than other 3D visualisations such as the Cone Trees (see Section 2.6), but is much easier to handle. Even novice users understand the landscape metaphor and can navigate through the world.

The first implementation of the Information Pyramid technique, the *3D Explorer* file browser, provides many different navigational facilities. For example, users can fly freely through the landscape, or can use viewpoints to quickly move from one place to another. Early testers expressed a desire for a navigation mode to easily move from one hierarchy level to the next or previous one. Such a navigation mode is equivalent to the one in conventional tree views, which users are used to. So the *3D Explorer* provides such an mode called *browse mode* by default.

The Information Pyramids technique is patent pending by Hyperwave Inc.

Appendix A

User Guide

The 3D Explorer is a three-dimensional visualisation of a file system. It is the first application of the *Information Pyramids* technique for visualising and exploring large hierarchies. This application visualises directories as plateaus. Subdirectory plateaus are placed atop the parent plateau. Files are displayed as 3D objects atop their parent plateau.

Many attributes of files and directories are mapped to properties of the 3D objects. For instance, the directory plateaus' size is proportional to the cumulative size of their contained data. The height of the file objects is proportional to their file size, and the colour of the file objects reflects their file type (extension).

Since the 3D Explorer is the first application of *Information Pyramids* there are many options and configurations which are not particularly useful, but show the evolution of the visualisation. A brief description of the user interface is given here. More information can be obtained in the online help of the 3D Explorer and in the readme files provided with the application.

A.1 Installing the 3D Explorer

To install the 3D Explorer the `zip` archive has to be uncompresseded with an appropriate program. Windows users may use *WinZip*, and Unix users *gunzip*. When uncompressing the archive, an `3dexpl` directory will be created. This directory contains the starting scripts and the native library. There are also some subdirectories for the help files, the fonts, and images. Of course, there is also a `class` directory for the Java code. Before the application can be started the Java environment must be set correctly.

- JDK 1.1.x must be installed.
- To use the online help the `Swing` classes must be installed.
- The `PATH` variable must include the JDKs `bin` directory.
- The `CLASSPATH` variable should include:
  - path/to/JDK/lib/classes.zip

- path/to/Swing/swingall.jar
- path/to/3dexplorer/classes

A.2 Starting the 3D Explorer

The 3D Explorer is started under Unix by executing the simple `run` script:

```
#! /bin/sh
#
#
LD_LIBRARY_PATH=../native/$CPU:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
set -x

java iicm.lscape.Explorer3D $* &
```

First the path to the native 3D library is set. It has to be added to the `LD_LIBRARY_PATH` to ensure that the Java runtime can find it at startup. Finaly, the Java interpreter is executed with the `Explorer3D` class and any additional command line parameters the user entered. If Windows95/98/NT is used the `explorer.bat` file has to be started:

```
@echo off
rem start 3D Explorer

java iicm.lscape.Explorer3D %1
```

It is not necessary to set the library path here, because the native library and the `.bat` file are in the same directory. The scripts take parameters according the follwoing schema:

```
explorer [Options] [Directory]
```

The *Directory* parameter defines the base directory of the visualisation. If no directory is passed to the script, the apllication uses the current directory as the base of the visualisation.

The *Options* are described in Table A.1.

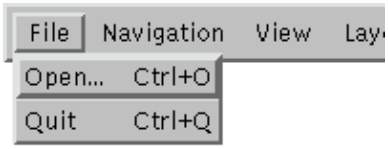
Parameter	**Description**
-h, -help	Print the usage help
-verbose	Verbose output (window creation etc.)
-home path	Explorer ”home” directory (to locate images, fonts and help files). Should be used if the script is not executed in the 3D Explorer home directory.

Table A.1: Command line parameters.

A.3 Menu Bar

The menu bar provides all the basic commands for the 3D Explorer, except navigation options, which can only be issued by the buttons in the navigation bar.

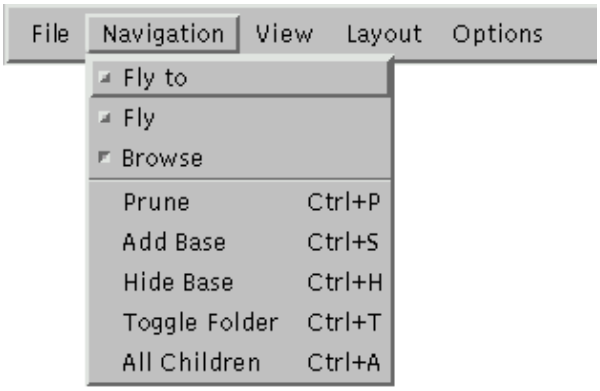
A.3.1 File Menu



Open Shows a file selection dialog, to select a new base directory for the visualisation. On some platforms (e.g. Windows95/98/NT) the file dialog does **not** allow to select a directory only. In such a case a file in the directory has to be selected. This file is not used further, it only makes it possible to close the file dialog and to open the selected directory.

Quit Closes the application.

A.3.2 Navigation Menu



Fly To Switch to the *Fly To* mode, where the user can select a target point in the landscape. By using some buttons in the navigation bar the camera can be moved toward this point, or away from this point. For a detailed description see Section A.6.

Fly Switch to *Fly* mode, where the user can freely move through the visualisation. For details see Section A.6.

Browse Switch to *Browse* mode, where the visualisation can be browsed almost like in a conventional *tree view*. In this mode the user can move easily from one directory to another. Also an easy way to examine the files in a directory is provided. For details see Section A.6.

Prune The currently selected directory will now be the base of visualisation. It appears as though the directory tree is pruned at the specified directory.

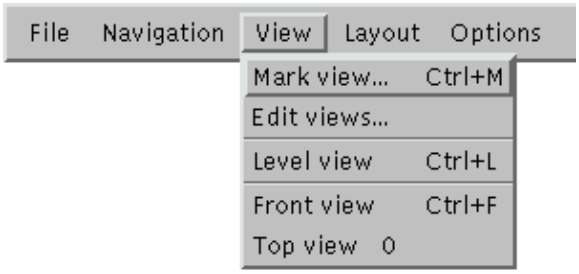
Add Base Adds the parent of the current base directory of the visualisation.

Hide Base Hides or prunes the current base directory. To do this, a directory has to be selected, to to determine which subdirectory is the new base of the visualisation.

Toggle Folder This operation opens the selected directory if it was closed, or closes it if it was open. Open and close in this context means to show the children of the directory or not.

All Children All children of the selected directory are opened and displayed recursively.

A.3.3 View Menu



Mark Views... Save the current camera view (i.e. camera position and orientation). For details see Section A.4.

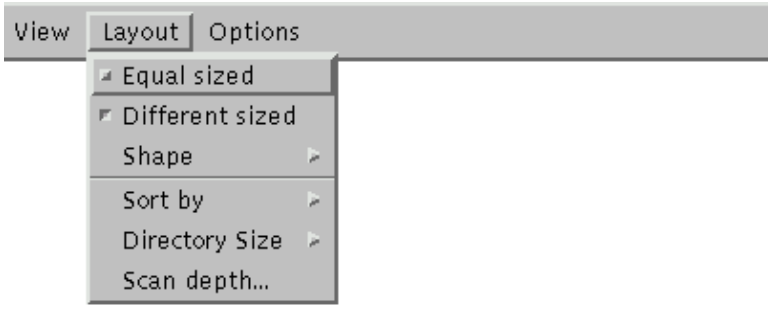
Edit Views... Opens a dialog to edit the marked views. Views can be renamed, deleted and the user can also go to a selected view. For details see Section A.4.

Level View Rotates the camera to an orientation where the viewing ray is parallel to the ground plane. After many rotations and moves through the landscape the user might loose orientation. By using this operation the camera is given a well defined orientation, so it is easier for users to regain their orientation.

Front View Moves the camera automatically in front of the landscape.

Top View Moves the camera automatically over the base directory. The camera orientation is set to look straight down to the ground.

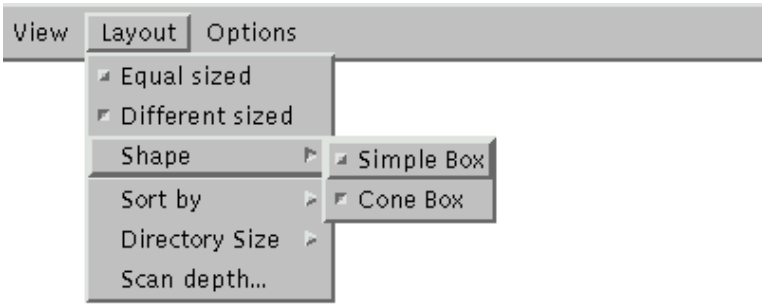
A.3.4 Layout Menu



Equal sized The subplateaus of an directory have all the same size. The size is not proportional to the data content of the directory.

Different sized The size of the plateaus is proportional to the content of the directory. The strategy to calculate the content of a directory is determined by the **Directory size** menu.

Shape This menu determines the shapes of the directory plateaus.

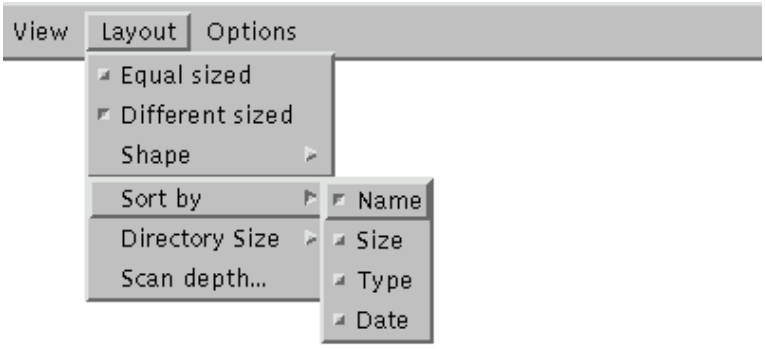


Simple Box The shape for a directory will be a simple recangulal box.

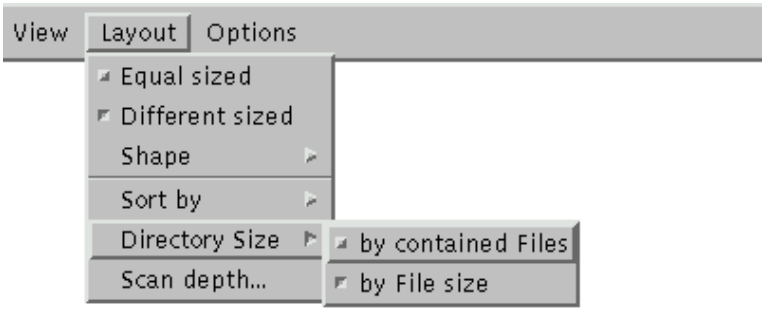
Cone Box The plateau has the same base size as the simple box, but the top area is slightly smaller. The overall appearance of this shape is that of a cone shaped rectangular box. The major advantage of this shape is that the plateaus can be better recognised when looking from above down to the shapes (i.e. the top view).

The above layout options were related to the shape of the directory objects, the following are related to other layout parameters.

Sort by This menu defines the sort order of the 3D objects on a plateau. The default order is to sort by name. So the directories are placed starting from the top left corner and the file objects starting from the bottom left corner according the sort order. The sort order can be changed at any time to size, name, type, and date.



Directory size or better the directory plateau size, can be determined by two strategies.



by contained files Simply counts the files in the given directory and its subdirectories to a given depth.

by file size The size of the contained files are added. This is also done recursively to a given depth.

Both strategies return only an approximation for the amount of contained data, sice we do not want to analyse the whole subtree because this would be too time-consuming. The first strategy results in better looking visualisations, because the directory plateau sizes correspond to the number of contained files. The second strategy can produce very large subdirectory plateaus containing only one large file. But this strategy gives an overview of the disk usage. The user can easily see which directories are using the most disk space.

Scan depth... Selecting this item brings up a dialog as shown in Figure A.1. With the slider the user can select how many levels should be used for the size calculation of a directory object. As mentioned above, the size of a directory is calculated by recursively scanning the subdirectories up to a given level. The higher the scan depth, the better the estimation for the directory size, but the longer it takes to calculate the size value and to place and display the directory plateau.

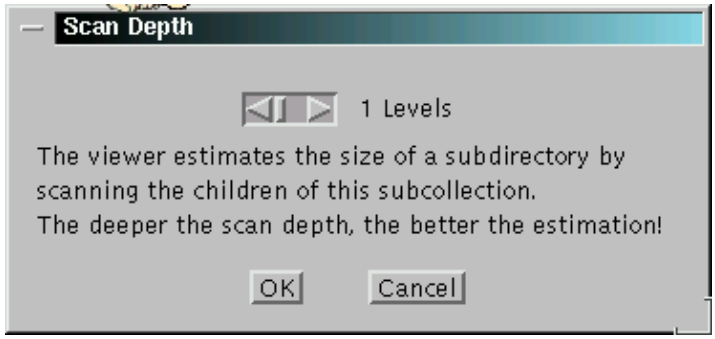
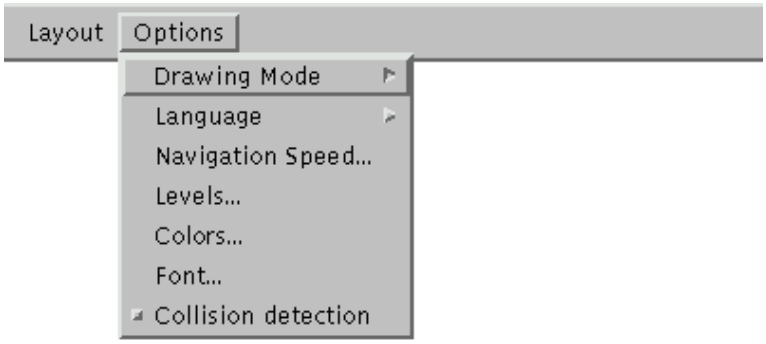


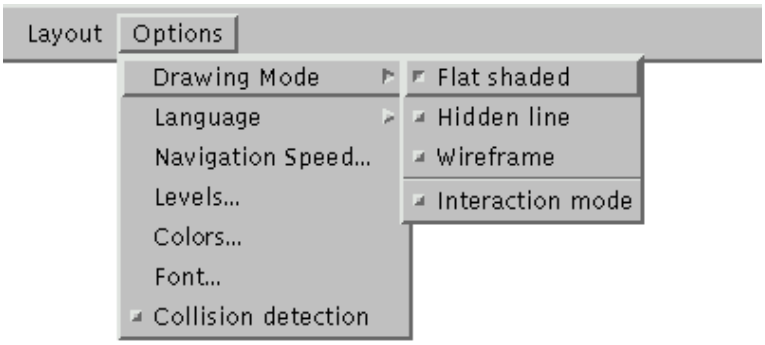
Figure A.1: The “Scan depth” dialog.

A.3.5 Options Menu

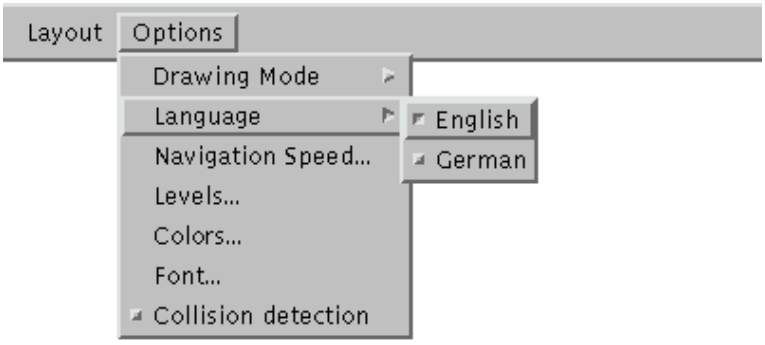


In the *Options* menu the user can change all properties which are not related to the layout of the Information Pyramids. All options of the layout, are handled in the *Layout* menu (see Section A.3.4)

Drawing Mode Changes the drawing mode of the 3D engine. Currently wireframe, hidden line and flad shaded modes are supported. If the fourth option called **Interaction mode** is enabled, the draw mode for an interaction will be set to wireframe regardless of the selected drawing mode. An interaction in this context is the motion of the camera after some user interaction. For instance pushing one of the standard views buttons of the navigation bar (see Section A.6).



Language The application determines the default language of the operating system and uses it also as the default language. But the user can also switch to other languages by using this menu. Currently only german and english are supported. When the language is changed, the whole interface changes the language immediately to the new setting.



Naviagtion Speed... It brings up a dialog to change the speed of the different camera movements. This is necessary to get useful animation speeds on different machines. On very slow machines the animation speed will be set faster, and on powerful 3D accelerated computers the animation should be done slower. In this dialog the desired frame rate can be set. The application then tries to display only as many details as possible to achieve the desired frame rate. Changing the frame rate is also very useful on very slow or fast machines to get a reasonable performance or better display quality. This dialog has an `Apply` button, so the user can change the values and by pushing this button the effects in the visualisation can be tested. If the changes are useful the user can set the values by pushing `OK` or discard the changes by pushing `Cancel`.

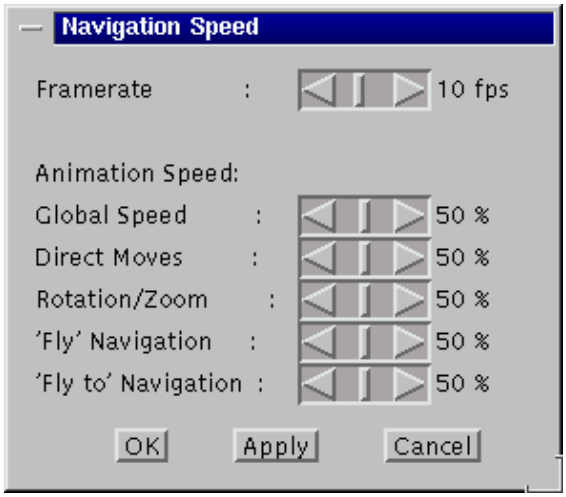


Figure A.2: The Naviagtion Speed dialog.

Levels... Brings up a dialog (see Figure A.3) where the numbers of subdirectory levels which should be opened when the user opens or unfolds a directory can be selected. In

conventional *Tree Views* this would be one, since only the next level of directories pops up. Since the calculation and placement of the object can take a while in the 3D Explorer it is recommended to open more than one subdirectory level at a time. Because this is much faster than opening one level after another. Of course, this value should not be too high, because this could lead to unreasonably long opening times.

Figure A.3: The Levels dialog.

Colors... Menu item to change the colours of the landscape objects. It brings up a dialog as seen in Figure A.4. At the top right corner of the dialog a visual attribute of the landscape can be selected. Its colour can be changed by using the sliders for the RGB or HSB values or by selecting one of the predefined colours.

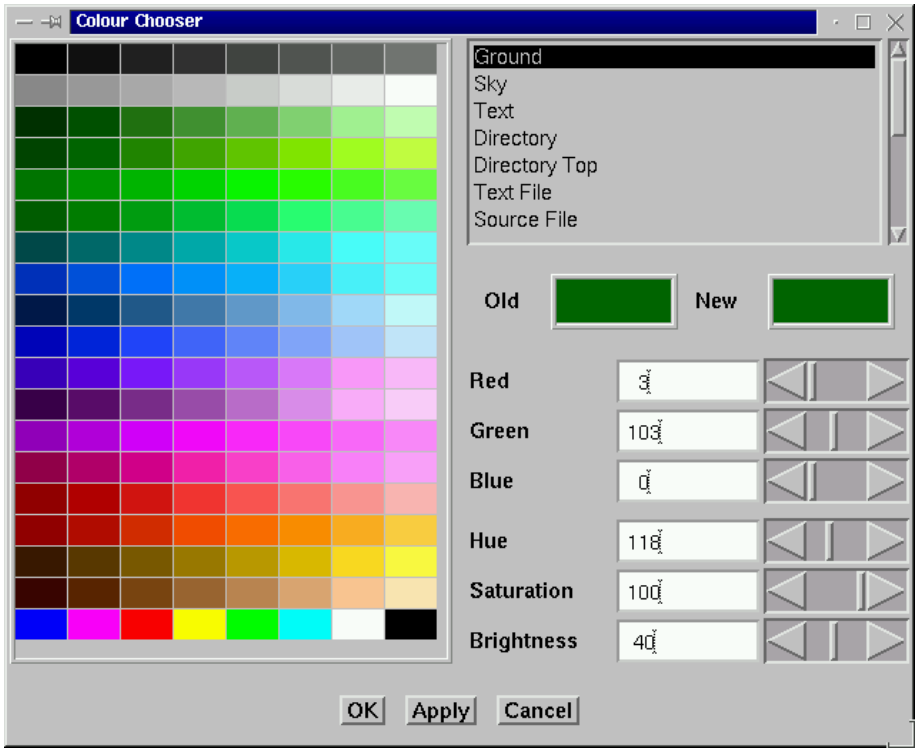


Figure A.4: The Colour Chooser dialog.

Font... Starts a dialog where the user can change the font properties. The first checkbox enables or disables the text in the 3D landscape. Next the font can be selected. Currently only *Helvetica* and *Times* are available. The text properties can be changed by selecting the *Bold* and *Italic* checkboxes. All changes in the dialog are set immediately in the 3D window.

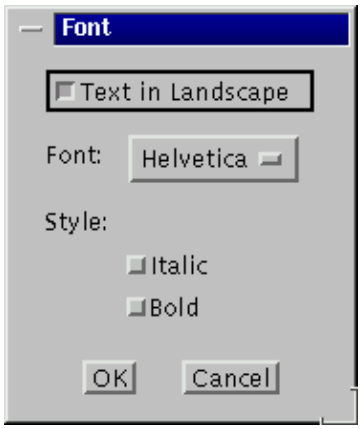


Figure A.5: The Font dialog.

Collision detection... If this option is enabled the user can not move the camera through objects in the landscape. Since the collision detection algorithm requires time-consuming calculations, this option is not very useful when visualising large pyramids.

A.3.6 Help Menu

Content... Brings up the online help dialog. This dialog uses Swing components, so it will only be displayed when the Swing classes are in the CLASSPATH. The help is based on HTML and therefore it uses hyperlinked documents which can be easily browsed. The online help is basically a short form of this user guide.

About... Displays a dialog with the 3D Explorer logo, the version information, and copyright information.

A.4 Using Views

To speed up navigation it is possible to store a particular camera position, a view. Users can move to stored views at any time. Often used directories can be easily accessed by marking

the view to this directory. It is for instance very helpful to use views when copying files from one directory to another. The two directories can be marked and so the camera can be moved from one directory to the other by a simple mouse click.

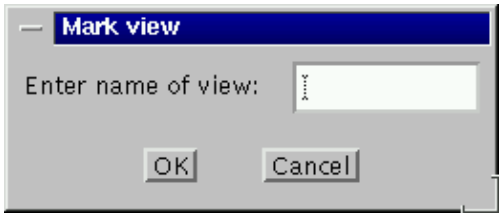


Figure A.6: The dialog to mark a view.

Marking a view can be done by different methods. Firstly, the menu **View - Mark view...** can be used. Another way would be to use the keyboard shortcuts **Ctrl-M** or simply **M**. The last method to mark a view is to push the appropriate icon in the tool bar (see Section A.5). All these actions opens a simple dialog to enter a name for this view (see Figure A.6).

When a view is marked, it is added to the popup list of the navigation bar. This list at the left side of the navigation bar, by clicking on it a list with all available marked views pops up. Selecting an item from the list takes the camera to this view immediately via smooth animation.

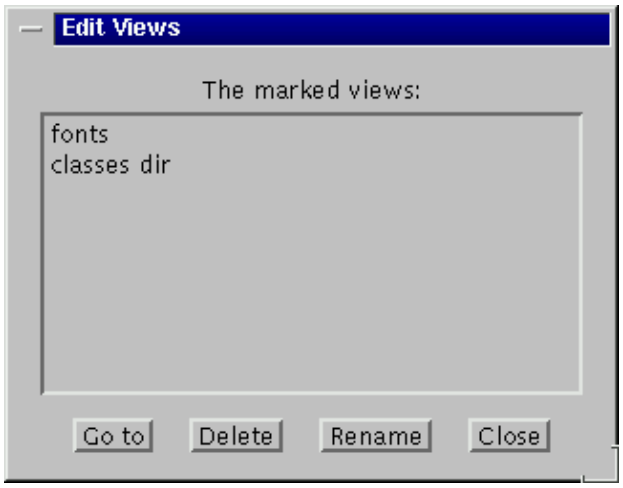


Figure A.7: The dialog to edit marked views.

To manage or edit the marked views the menu item **View - Edit views...** can be selected or alternatively the icon in the tool bar can be pushed. This brings up the dialog shown in Figure A.7. At the top of this dialog there is a list of the currently marked views. Selecting one item from the list and pushing one of the four buttons of the dialog performes the corresponding operation:

- **Go to:** Moves the camera to the selected view of the list.

- **Delete:** Deletes the selected view. This of course is only done if the `Yes` button of the confirmation dialog is pushed.
- **Rename:** Displays a dialog (see Figure A.8) with a text field to enter a new name for the selected list item. If the new name is allready used another name has to be entered. Of course the whole operation can also be discarded by pushing the `Cancel` button.

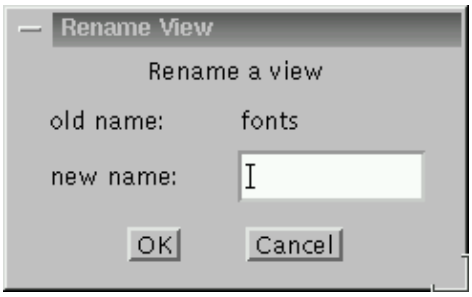


Figure A.8: The dialog to rename a marked view.

- **Close:** Closes the dialog.

A.5 Tool Bar

Figure A.9: The tool bar of the 3D Explorer.

The tool bar provides easy access to some functions of the 3D Explorer. None of these functions deal with navigation. The tool bar contains currently seven buttons:

- **Cut:** To cut the selected 3D objects. Currently disabled all the time.
- **Copy:** To copy the selected 3D objects. Currently disabled all the time.
- **Paste:** To paste previously copied or cut 3D objects. Currently disabled all the time.
- **Delete:** To delete the selected 3D objects. Currently disabled all the time.
- **Mark View:** Stores the current view, i.e. camera position and orientation. Details in Section A.4.
- **Edit View:** Edit and manage the marked views. Details in Section A.4.
- **Properties:** Displays a dialog with the properties of the selected object. If multiple objects were selected, only the properties of the last selected objects are displayed.

A.6 Navigation Bar

The navigation bar provides access to all navigation tools in an elegant and simple way. This part of the 3D Explorer was redesigned after usability tests with the former user interface. This graphically enhanced interface is easier to use, gives more feedback to the user and provides facilities to help the user with the interface.

Buttons with a light cyan symbol are activated. A dark cyan symbol is shown on deactivated buttons. If the button is disabled an almost invisible gray symbol will be displayed. To give the user feedback, buttons become light gray when the mouse cursor is moved over them.

A.6.1 Basic Structure

Figure A.10: The navigation bar with help text for the *Fly to* button.

On the left side of Figure A.10 the navigation mode switch is located. In this case the *fly mode* (see Section A.6.3) is enabled, because the airplane symbol of the switch is highlighted. By pushing the steering wheel symbol, *browse mode* (see Section A.6.2) can be activated. These two modes are the basic navigation methods of the 3D Explorer.

To the right of the navigation mode switch there is a popup list of the marked views. By clicking on this element a list of all previously marked views is displayed. Clicking on one of these views starts an animated translation of the camera to this view. For details on views see Section A.4.

In the middle of the navigation bar is the main button panel of the navigation mode. This panel is different for every navigation mode. The different styles of this panel will be discussed in the following subsections. The only two buttons which are always here are the left and right buttons of the central dashboard. The left button moves the camera to the top view, which is a camera position right above the base plateau, where the camera is looking down to the ground. This position gives a good overview of the whole structure. The right button moves the camera right in front of the information pyramids. This button is useful to get a good starting point when flying over the pyramids.

On the right side of the navigation bar there are another two buttons. These buttons are very similar to the two buttons mentioned above. The left one takes the camera to the top view of the currently selected 3D object. And the right one moves the camera to a position infront of the currently selected 3D object. These translations of the camera are only done while these buttons are pushed. This give the user the control to move to a position above or in front of any object in the landscape.

The navigation bar also provides a help text for every button. This text appears in the bottom right corner of the navigation bar when the mouse cursor is moved over a button in the

navigation bar. In Figure A.10 the mouse cursor is moved over a button, which is therefore highlighted and a short help text appears in the bottom right corner of the navigation bar. This help text briefly describes the meaning of a button.

A.6.2 Browse Mode

Figure A.11: The navigation bar in *Browse* mode.

This mode should provide a navigation method similar to the one used in ordinary tree views. It provides a simple and fast way to move up and down in the directory hierarchy. To activate this mode the switch on the left side of the navigation bar has to be pushed, so that the browse icon (a steering wheel) is highlighted (see Figure A.11). The button panel in the middle of the navigation bar now shows two round buttons.

Navigation with the help of the buttons is reasonably simple in this mode. The camera simply moves to a directory by clicking on it with the left mouse button. The camera moves to a position slightly above the directory plateau where the user can view the plateau.

To examine the plateau, the round button with the rotation symbol has to be activated. Dragging the mouse in the 3D window rotates the camera around the current directory plateau. If the other round button is activated, than the dragging of the mouse zooms the camera in or out.

Between the round buttons there is a small square-shaped button. Pushing this button moves the camera to the parent directory of the current directory.

A.6.3 Fly Mode

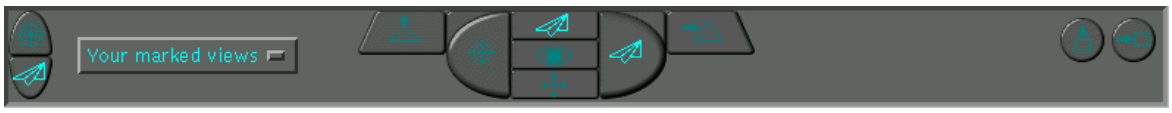


Figure A.12: The navigation bar in *Fly* mode.

In this mode the user can freely move through the three-dimensional landscape. To activate this mode the navigation mode switch to the left of the navigation bar has to be pushed. The airplane symbol of this switch must be highlighted. Now the middle panel of the navigation bar shows an elliptic dashboard as shown in Figure A.12. The fly mode consists actually of two different modes. The real *fly* mode and the *fly to* mode. The *fly to* mode is discussed in the next section. To activate the real fly mode the right elliptic button with the plane symbol has to be activated.

Between the two elliptic buttons there are now three small rectagular buttons. These buttons build a button group, so only one of these buttons can be activated at a time. The buttons control the meaning of the dragging of the mouse in the 3D window. From top to down these buttons acivate:

- **Fly:** Move the camera parallel to the ground.
- **Look:** Rotate the camera around the current position. Like moving your head to look around.
- **Pan:** Moves the camera parallel to the view plane.

A.6.4 Fly To Mode

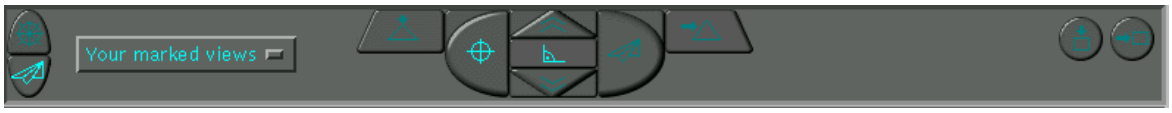


Figure A.13: The navigation bar in *Fly to* mode.

This mode provides an easy method to fly to a selected spot on the Information Pyramids. It is activated by switching the navigation switch at the left of the navigation bar to highlight the fly symbol. To distinguish the real fly mode described in the last section and the *fly to* mode, the left elliptic button has to be activated. This button actually switches to the *fly to* mode.

Now two triangular and one rectangular button are displayed between the two elliptic buttons (see Figure A.13). To select a target to fly to or away, this point has to be marked by a single mouse click with the left mouse button. To fly to or away of this point the triangular shaped buttons have to be used. While pushing these buttons the camera moves straight to or from this point. If the rectangular button in the middle is activated, then the camera approaches the selected surface along its normal vector.

A.7 Mouse Functions

To move the camera the mouse is used. The particular meaning of the mouse operation depends on the currently selected navigation mode. Although there are some mouse actions which can be applied independently of the navigation mode:

- **Left mouse button - single click:** Selects the 3D object which the mouse is pointing at. A coloured frame is painted around the object to show its selected state. If the mouse is clicked over the sky or the ground plane any selection will be deselected.
- **Left mouse button - double click:** If this is applied to an directory plateau the children will be displayed if they are not currently displayed or vice versa. A double click on any other 3D object will be handled as two single mouse clicks.

- **Right mouse button - single click:** The focused 3D object will be selected if it was not before. Additionally the *Properties* dialog will be opened to show the object properties. Of course no proprties will be shown if the mouse points to the sky or the ground plane. If the *Properties* dialog is allready open only the new properties will be displayed. In case of the right clicking on the sky or ground plane the properties fields will be left blank.

All other mouse actions depend on the selected navigation mode. To switch to the different modes and their usage is described in Section A.6.

The basic method to move the camera in a way depending on the selected mode is to push the left mouse button and move the mouse while holding down the mouse button. This drag operation is visualised through a white cross on the position where the mouse button was pushed and a red line to the current mouse position during dragging.

Browse Mode: This mode uses the drag operation in two different operations:

- **Examine:** Mouse movement while pushing the left mouse button rotates the camera around the currently selected directory. If nothing is selected, the rotation centre is set to the center of the base plateau.
- **Zoom:** Moving the mouse back and forth zooms the camera out and in to the center of the selected directory plateau.

Additionally, in this navigation mode, a single left button mouse click is handled slightly different to the other modes. When clicking on a directory the camera is automatically moved to a position where the directory plateau can be best viewed. Clicking on a file rotates the camera around the centre of the directory plateau where the file is located, to a position where the selected file is closest to the camera. Note that in case of clicking on a file no zooming is done.

Fly Mode: The drag operation is used here for three different movements:

- **Fly:** Moves the camera parallel to the ground. While dragging the direction can be changed by moving the mouse left or right. The distance of the mouse pointer to the cross in the vertical direction determines the speed of the camera. If the mouse pointer is beneath the white cross the camera moves backwards.
- **Look:** Moving the mouse rotates the camera around the current camera position. Thus the position stays the same, and only viewing direction changes.
- **Pan:** In this mode the camera moves up/down or left/right on the virtual vertical viewing plane.

Fly To Mode: Using this mode a target can be selected by a single left mouse button click. This target is then used to fly to or away from. Details on this navigation mode can be found in Section A.6.

A.8 Keyboard Shortcuts

Shortcut	**Function**
[F1] or Ctrl-1	Opens online help.
[F2] or Ctrl-2	Reload selected directory/plateau.
Ctrl-Q	Exit the application.
Ctrl-O or [F3] or Ctrl-3	Opens the file selection dialog to select a new base directory.
[F5] or Ctrl-5	Switch to “Fly to” navigation mode.
[F6] or Ctrl-6	Switch to “Fly” navigation mode.
[F7] or Ctrl-7	Switch to “Browse” navigation mode.
Ctrl-P or P	Prune the Information Pyramids at the selected directory.
Ctrl-S or +	Adds a new base directory to the visualisation.
Ctrl-H or -	Hides the base directory plateau.
Ctrl-T or T or [ENTER]	Open or closes a directory/plateau.
Ctrl-A or A	Loads recursivly all children of the selected directory.
Ctrl-M or M	Store the current camera view (position and orientation).
Ctrl-L or L	Levels the camera.
Ctrl-F or F	Moves the camera to the front view.
0	Moves the camera to the top view.
D	Enables or disables collision detection.

Table A.2: Keyboard shortcuts.

Bibliography

[And94] Keith Andrews. Multimedia information systems (lecture notes), 1994.

[ANS88] ANSI. *Computer Graphics — Programmer's Hierarchical Interactive Graphics System (PHIGS); Functional Description, Archive File Format, Clear-Text Encoding of Archive File; ANS X3.144*. ANSI, September 1988.

[AWP97] Keith Andrews, Josef Wolte, and Michael Pichler. Information pyramids: A new approach to visualising large hierarchies. In *Late Breaking Hot Topic - Proc. IEEE Visualization '97*, October 1997.

[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`.

[Bra96] Tim Bray. Measuring the web. *Computer Networks and ISDN Systems*, 28(7–11):993–1005, May 1996. Proc. Fifth International World-Wide Web Conference, WWW5, Paris, France. `http://www5conf.inria.fr/fich_html/papers/P9/Overview.html`.

[Cha] Leo Chan. An unofficial port of opengl to java. Computer Graphics Lab, University of Waterloo. `http://www.meta.cgl.uwaterloo.ca/SourceCodeAndDemos/OpenGL4java.html`.

[Cha93] Matthew Chalmers. Using a landscape metaphor to represent a corpus of documents. In *Spatial Information Theory, Proc. COSIT'93*, pages 377–390, Boston, Massachusetts, September 1993. Springer LNCS 716.

[Cha96a] Matthew Chalmers. Adding imageability features to information displays. In *Proc. UIST'96*, Seattle, Washington, November 1996. ACM. `http://www.ubs.com/webclub/ubilab/publications/e_cha96d.htm`.

[Cha96b] Matthew Chalmers. A linear iteration time layout algorithm for visualising high-dimensional data. In *Proc. Visualization'96*, pages 127–132, San Francisco, California, October 1996. IEEE Computer Society. `http://www.ubs.com/webclub/ubilab/publications/e_cha96a.htm`.

[Der95] Christian Derler. The world-wide web gateway to hyper-g: Using a connectionless protocol to access session-oriented services. Master's thesis, Graz University of Technology, Austria, March 1995. `ftp://ftp.iicm.edu/pub/theses/cderler.pdf`.

[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`.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, Massachusetts, second edition, 1990.

[HKW94] Matthias Hemmje, Clemens Kunkel, and Alexander Willet. Lyberworld - a visualization user interface supporting fulltext retrieval. In *Proc. SIGIR'94*, Dublin, Ireland, July 1994. ACM. `ftp://ftp.darmstadt.gmd.de/pub/VISIT/papers/hemmje/SIGIR94.ps.gz`.

[ISO87] ISO. *Information Processing Systems — Computer Graphics — Programmer's Hierarchical Interactive Graphics System (PHIGS), Parts 1-3, ISO DIS 9592*. ISO, October 1987.

[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`.

[KAF+94] Frank Kappe, Keith Andrews, Jörg Faschingbauer, Mansuet Gaisbauer, Michael Pichler, and Jürgen Schipflinger. Hyper-G: A new tool for distributed hypermedia. IIG Report 388, IIG, Graz University of Technology, Graz, Austria, May 1994. `ftp://ftp.iicm.edu/pub/papers/`.

[KM93] Frank Kappe and Hermann Maurer. Hyper-G: A large universal hypermedia system and some spin-offs. IIG Report 364, IIG, Graz University of Technology, Austria, May 1993. `ftp://ftp.iicm.edu/pub/papers/`.

[KMS92] Frank Kappe, Hermann Maurer, and Nick Scherbakov. Hyper-G – a universal hypermedia system. IIG Report 333, IIG, Graz University of Technology, Austria, March 1992. `ftp://ftp.iicm.edu/pub/papers/`.

[KMT91] Frank Kappe, Hermann Maurer, and Ivan Tomek. Hyper-G – specification of requirements. IIG Report 284, IIG, Graz University of Technology, Austria, April 1991. Also appeared in Proc. CIS '91. `ftp://ftp.iicm.edu/pub/papers/`.

[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 5,619,632, Xerox Corporation, April 1997. Filed 14th Sept. 1994, granted 8th April 1997. Available from USPTO `http://www.uspto.gov/`.

[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, editor. *HyperWave: The Next Generation Web Solution*. Addison-Wesley, May 1996. `http://www.iicm.edu/hgbook`.

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

[Ope92] OpenGL Architecture Review Board. *OpenGL Reference Manual*. Addison-Wesley, 1992.

[Pau] Brian Paul. The Mesa 3-D graphics library. `http://www.ssec.wisc.edu/\~brianp/Mesa.html`, Space Science and Engineering Center, University of Wisconsin.

[Pic93] Michael Pichler. Interactive browsing of 3D scenes in hypermedia: The Hyper-G 3D viewer. Master's thesis, Graz University of Technology, Austria, October 1993. `ftp://ftp.iicm.edu/pub/theses/mpichler1.pdf`.

[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.

[RMC94] George G. Robertson, Jock Mackinlay, and Stuart K. Card. Display of hierarchical three-dimensional structures with rotating substructures. US Patent 5,295,243, Xerox Corporation, March 1994. Filed 29th Dec. 1989, granted 15th March 1994. Available from USPTO `http://www.uspto.gov/`.

[SG86] R. W. Scheifler and J. Gettys. The x window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[Shn92] Ben Shneiderman. Tree visualization with tree-maps: A 2-d space-filling approach. *ACM Transactions on Graphics*, 11(1):92–99, January 1992. `ftp://ftp.cs.umd.edu/pub/papers/papers/2645/2645.ps.Z`.

[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`.

[Sil90] Silicon Graphics, Inc. *Graphics Library Programming Guide, Document Version 2.0*. Silicon Graphics, Inc., Mountain View, California, May 1990.

[ST96a] Steven L. Strasnick and Joel D. Tesler. Method and apparatus for displaying data within a three-dimensional information landscape. US Patent 5,528,735, Silicon Graphics, Inc., June 1996. Filed 23rd March 1993, granted 18th June 1996. Available from USPTO `http://www.uspto.gov/`.

[ST96b] Steven L. Strasnick and Joel D. Tesler. Method and apparatus for navigation within three-dimensional information landscape. US Patent 5,555,354, Silicon Graphics, Inc., September 1996. Filed 23rd March 1993, granted 10th Sept. 1996. Available from USPTO `http://www.uspto.gov/`.

[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`.