Visualisierung von Verzeichnisbäumen

Eine Jugend-Forscht-Arbeit von Hannes Flicka

Deckblatt und Kurzfassung

Bereich:	Jugend forscht	**Bundesland:**	Niedersachsen
Fachgebiet:	Mathematik/Informatik	**Anzahl Gruppenteilnehmer:**	1
Titel der Arbeit:	Visualisierung von Verzeichnisbäumen		

	Einzelteilnehmer/in Gruppensprecher/in:	**2. Teilnehmer/in:**	**3. Teilnehmer/in:**
Geschlecht:	männlich		
Name:	Flicka		
Vorname:	Hannes		
Straße:	Sohldfeld 26		
PLZ:	31139		
Ort:	Hildesheim		
Telefon:	05121 267928		
E-Mail:	hflicka@gmx.net		
Geburtsdatum:	2.6.1985		
Klassenstufe:	13		
Frühere Teilnahme:	2002/2003		
Schule/Betrieb/ Dienststelle:	Scharnhorstgymnasium		
Schulform:	Gymnasium		
Straße:	Steingrube 19		
PLZ:	31141		
Ort:	Hildesheim		
Telefon:	05121/750947		
E-Mail:			
Webadresse:			
	Betreuer/in:		
Name:	May		
Vorname:	Otto		
Schule:	Scharnhorstgymnasium		
Telefon:			
E-Mail:			

Alle Angaben sind verbindlich und dürfen für die Presse- und Öffentlichkeitsarbeit der Stiftung Jugend forscht gespeichert und verwendet werden. Ich/wir erkenne/n die Teilnahmebedingungen des Wettbewerbs an.

3377	2003-11-28	....
id	**Datum**	**Unterschrift - bei Minderjährigen des Erziehungsberechtigten**

Kurzfassung:

Der Verzeichnisbaum ist das Grundgerüst der Daten, die auf einem PC gespeichert sind. Seine Struktur ist meist komplex, tief und unübersichtlich. Um einen Verzeichnisbaum zu durchsuchen und zweidimensional darzustellen, gibt es Programme wie den Windows-Explorer oder SequoiaView. In dieser Arbeit werden die Möglichkeiten einer dreidimensionalen Darstellung untersucht und im Programm getestet.

Die Shell bezeichnet im Allgmeinen die Benutzeroberfläche eines Betriebssystems. Sie soll dem Benutzer schnellen und einfachen Zugriff auf seine Dateien, Verzeichnisse und Anwendungen geben. Es gibt textbasierte und grafische Ansätze für Shells.

Um das Verzeichnis der lokalen Dateien anzuzeigen wird meist ein Baumdiagramm benutzt, wie etwa im Windows-Explorer. Große Verzeichnisse können dabei sehr unübersichtlich werden, da man viel scrollen und klicken muss.

In dieser Arbeit wird eine alternative Visualisierung, bei der die Navigation stets einfach und schnell bleibt, entwickelt, und ihre Einsatzmöglichkeit als Shell untersucht. Dabei wird die bislang in Shells ungenutzte Leistung der 3D-Beschleunigung eingesetzt.

1 Inhaltsverzeichnis

2 Shells & GUI's....3
3 Grundlagen der Baumdarstellung....4
4 Entwickeln einer anforderungsgerechten Visualisierung....5
4.1 Ziele und Voraussetzungen....5
4.2 Treemaps in der zweiten Dimension....6
4.3 Treemaps in der dritten Dimension....6
4.3.1 Ein weiterer Ansatz....8
5 Implementation....9
5.1 Programmaufbau....9
5.2 Datenstruktur des Baums....10
5.3 Darstellung der Treemaps....10
5.4 Dynamische Anzeige von Treemaps....12
5.4.1 Geometrische Vereinfachung je nach Kameraentfernung....12
5.4.2 Culling von Objekten....13
5.5 Bewertung....14
5.6 Hosting des Projekts....15
6 Aussichten....15
7 Literaturverzeichnis....17

2 Shells & GUI's

Vor allem unter Unix haben sich viele Text-basierte Shells entwickelt, denn die ersten Grafikkarten boten nur die Textwiedergabe und auch für Text-Terminals eignen sich diese Shells. Sie bieten grundsätzliche Funktionen und Befehle wie:

- Verzeichnisse auflisten
- Programme starten
- Dateien anzeigen
- Verwaltung der Dateien

Die erste grafische Shell wurde '73 am Xerox Palo Alto Research Center (PARC) entwickelt. Dort wurden die Grundbausteine einer grafischen Benutzerschnittstelle (auch GUI, Graphical User Interface) konzipiert. Dabei handelt es sich natürlich um Fenster und sogenannte Widgets (Buttons, Checkboxen, Radioboxen, Textfelder, Comboboxen, Scrollbalken)[1].

Grafische Shells wurden die Grundlage aller neueren Betriebssysteme, da sie einfach besser zu bedienen sind, und in mehreren Fenstern mehrere Anwendungen genutzt werden können (Multitasking).

Auf Unix-Systemen ist das X Window System weit verbreitet, das wegen der Teilung in Client und Server gleichzeitig grafisches Terminal ist. Zum X Window System sind unter Linux viele weitere Window-Manager entstanden, die einer grafischen Shell entsprechen.
Bei Microsofts Windows sind die Fenster schon im Namen enthalten und zentraler Bestandteil des Systems. Weiterhin gibt es Aqua, die Oberfläche zu Apples Macintosh-Computern, bei deren Entwicklung vor allem Wert auf Ästhetik gelegt wurde[2].

Heutige Weiterentwicklungen bestehender Shells verbessern die Benutzerfreundlichkeit, bringen neue, grafische Fähigkeiten wie Anti-Aliasing/Vektorgrafik, vereinfachen die Erweiterbarkeit,

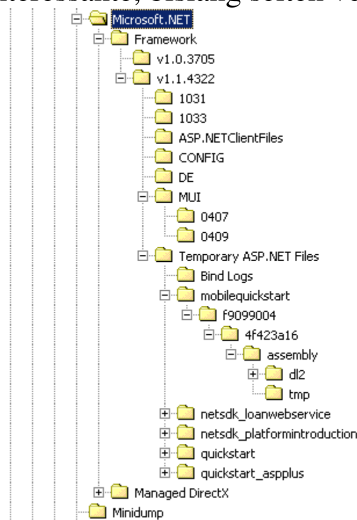
verankern das Internet in der Shell, etc... [3], [4]

Vereinzelt gibt es auch Ansätze und Konzepte, dreidimensionale Shells zu entwickeln, bei denen die Programmfenster als Textur auf Flächen dargestellt werden, oder die Programme gleich mit dreidimensionalen Objekten arbeiten [5].

An dieser Stelle knüpft dieses Projekt an. Hier soll eine übersichtliche dreidimensionale Darstellung für Verzeichnisbäume entwickelt werden. Eine solche Darstellung kann gleichzeitig als Shell dienen, wenn man die Interaktivität hinzufügt.

3 Grundlagen der Baumdarstellung

Um beliebige Bäume mit gewichteten Knoten (wie z.B. Dateien und Verzeichnissen) darzustellen, gibt es bereits bekannte aber auch interessante, bislang selten verwendete Methoden[6].



Im Windows-Explorer wird der Dateibaum diagrammartig angezeigt. Alle Knoten sind entlang einer Dimension (vertikal) aufgereiht, wobei einzelne Ordner auf- und zugeklappt werden können. Da ein Explorer-Fenster begrenzte Ausmaße hat, kann man natürlich nur eine begrenzte Anzahl an Knoten auf ein Mal anzeigen. Wenn man allerdings viele Knoten anzeigen und etwa vergleichen will, muss man entsprechend viel scrollen.

Um möglichst viel Informationen auf einer zweidimensionalen Fläche unterzubringen, hat Ben Shneiderman vom Human-Computer Interaction Laboratory [7], [8] sogenannte Treemaps erfunden.

In einer Treemap werden die Knoten des Baums als rechteckige Flächen dargestellt. Alle Knoten, die einem Knoten k untergeordnet sind, befinden sich dabei als kleine Rechtecke im Rechteck von k.
Die Flächeninhalte der Rechtecke zweier Knoten entsprechen im Verhältnis den Gewichten dieser Knoten.

So erhält man letztendlich eine Gesamtübersicht über alle Knoten des Baums und ihre Größenverhältnisse.

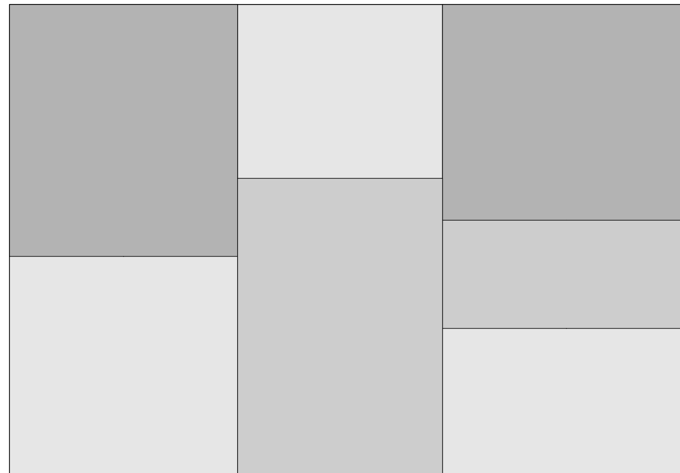


Abbildung 1: Beispiel einer Treemap

Setzt man die Gewichte der Knoten gleich der Größe der entsprechenden Dateien/Verzeichnisse, kann man schnell und einfach die Größe der Dateien anhand der Flächeninhalte der Rechtecke vergleichen. Dieses Anwendungsbeispiel ist im Programm “SequoiaView” zu sehen, das an der TU Eindhoven entwickelt wurde.

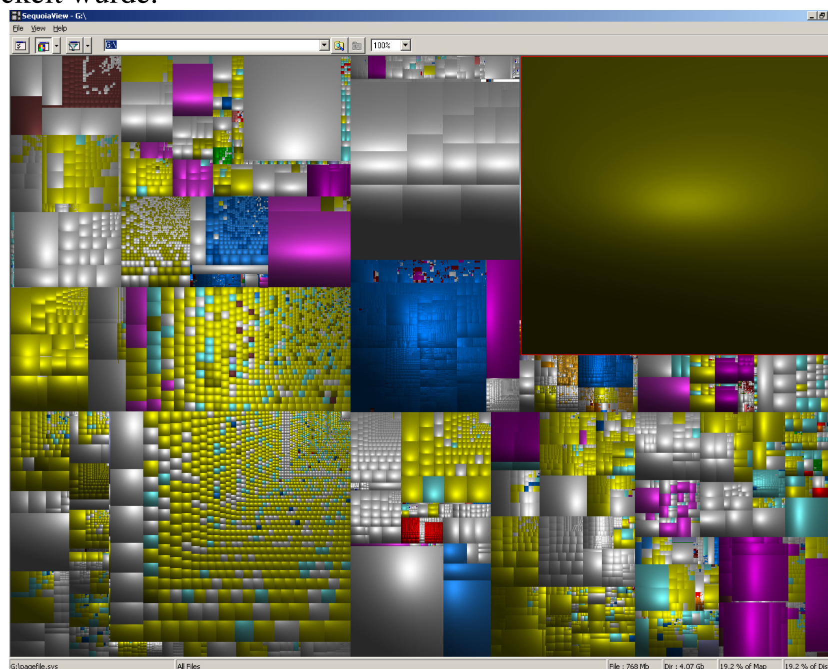


Abbildung 2: SequoiaView zeigt den Inhalt einer Festplatte an

Wie schon in der Abbildung zu erkennen, werden manche Knoten bzw. Dateien so klein, dass sie nicht mehr erkennbar sind oder verschwinden. Das legt nahe, eine Zoomfunktion für Treemaps zu verwenden.

4 Entwickeln einer anforderungsgerechten Visualisierung

4.1 Ziele und Voraussetzungen

Grundsätzlich sind Visualisiserung und Interaktion zu unterscheiden.

Bei Visualisierung/Darstellung sollten die Vorteile der Navigation in der dritten Dimension genutzt werden. Denn das Bewegen im virtuellen Raum mit Maus und Tastatur ist intuitiv und daher einfach.

In der dritten Dimension habe ich mir vorgenommen, ein Konzept, ähnlich dem der Treemaps zu verfolgen: Alle Dateien und Verzeichnisse sollten, ensprechend ihrer Größe, in einem Raum angeordnet sein, über den der Benutzer eine vollständige Übersicht hat.

Als ersten Ansatz der Visualisierung habe ich also versucht, den Algorithmus für das Treemap-Layout von der zweiten in die dritte Dimension zu übertragen.

4.2 Treemaps in der zweiten Dimension

Zum Erzeugen einer Treemap sind ein Baum mit gewichteten Knoten und eine rechteckige Fläche gegeben, die so aufgeteilt wird, dass jeder Blatt-Knoten eine Fläche zu seiner Repräsentation erhält.

Ein Knoten, der noch weitere Unterknoten hat, braucht nicht angezeigt werden, denn die Fläche dieses Knotens wird auf die untergeordneten Knoten aufgeteilt.

Das Problem der Treemaps lässt sich ideal rekursiv lösen: Ausgehend vom obersten Knoten müssen jeweils die unmittelbar untergeordneten Knoten auf die Fläche des übergeordneten Knoten aufgeteilt werden. Dann kann dieser Vorgang für alle die untergeordneten Knoten rekursiv angewendet werden.

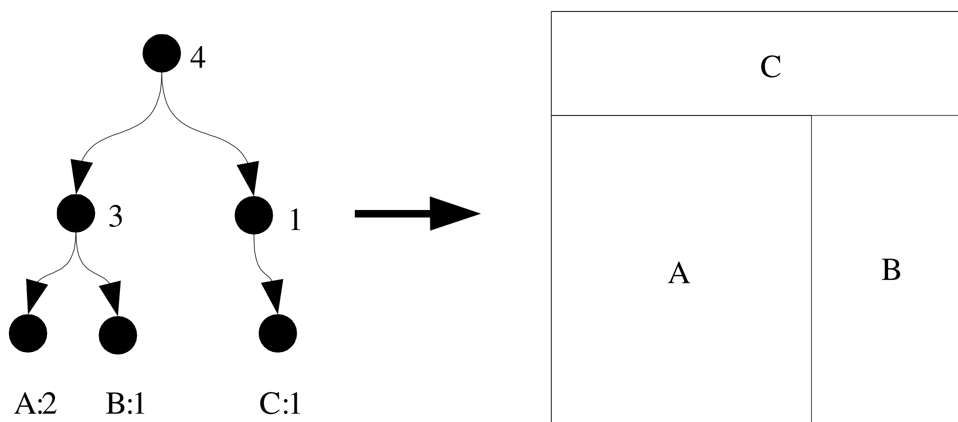


Abbildung 3: Funktionsweise des Treemap-Algorithmus: Aus einem Baum mit gewichteten Knoten wird eine Treemap erzeugt

Somit reduziert sich das Problem darauf, die einem Knoten untergeordneten Knoten dem Gewicht entsprechend auf dessen rechteckige Fläche aufzuteilen.
Hierzu gibt es verschiedene Lösungen bzw. Layout-Verfahren.
Das Einfachste ist es, die aufzuteilende Fläche entlang einer Achse in Streifen zu schneiden, deren Größe den Gewichten der Unterknoten entspricht. Sofort hat man das große Problem, daß die Streifen sehr schmal und schwer erkennbar werden können.

Eine Layout-Variante, die sogenannten *Squarified*-Treemaps [9] versuchen, möglichst quadratische Felder anzulegen, wodurch eine viel übersichtlichere Treemap erzeugt wird. Das ist eine

Notwendigkeit, wenn man etwa Schrift in die Felder schreiben will.

4.3 Treemaps in der dritten Dimension

In der dritten Dimension sind es eben Quader anstelle von Rechtecken, die die Knoten eines Baumes repräsentieren, und deren Volumina verhälnismäßig den Gewichten der Knoten angepasst werden müssen.

Um die zweidimensionalen Verfahren beibehalten zu können, kann man ein Quader in mehrere Ebenen einteilen, die dann jeweils nach zweidimensionalem Layout mit einem Teil der Unterknoten gefüllt werden. Auf einer Seitenfläche des Ebenenquaders wird ein zweidimensionales Layout aufgebaut, das dann über die Ebene extrudiert wird. Die Ebenen sind also eigentlich flache Quader.

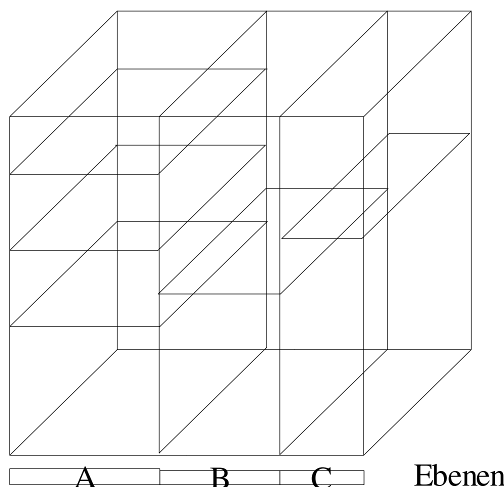


Abbildung 4: Mehrere Ebenen werden mit zweidimensionalem Layout gefüllt

Zunächst braucht man einen Algorithmus, der die Knoten auf verschiedene Ebenen einteilt und entscheidet, wieviele Ebenen es geben soll.

Wegen der Übersichtlichkeit sollten die Knoten möglichst kubisch sein.

Der naheliegende Ansatz lautet, solange Knoten zu einer Ebene hinzuzufügen, bis das beste Verhältnis zwischen Grundfläche eines Quaders, die sich mit jedem neuen Knoten verringert, und Höhe dieses Quaders, die sich mit jedem neuen Knoten vergrößert, erreicht ist. Ich habe hier die arithmetisch gemittelten Werte aller Knoten bzw. Quader verwendet.

Es ist

n die Anzahl der Knoten auf der Ebene,

V das Volumen der Ebene (V ist gleich der Summe aller Knotengewichte),

A die Grundfläche der Ebene,

h die Höhe der Ebene und damit die Höhe aller enthaltenen Quader (Es gilt $V = hA \Leftrightarrow h = \frac{V}{A}$),

a die durchschnittliche Grundfläche eines Quaders. Es ist $a = \frac{A}{n}$.

Damit die Quader möglichst kubisch sind, muss $\frac{h^2}{a}$ bzw. $\frac{a}{h^2}$ möglichst nah bei 1 liegen.

Während des Hinzufügens sind A und n ohnehin und V stets durch Addition der Größen aller bereits hinzugefügten Knoten bekannt.

Ziel des Algorithmus ist es, eine Ebene mit $r=\frac{a}{h^2}=\frac{A^3}{nV^2}$ möglichst nahe bei 1 zu erzeugen.

Wie oben beschrieben, verringert sich a mit jedem neuen Quader während sich h vergrößert. Also verringert sich r mit jedem neuen Knoten und nähert sich von oben der 1 an. Wenn $r \leq 1$ ist, muss die Ebene abgeschlossen und eine neue Ebene erstellt werden.

Es resultiert der Algorithmus:

```
1. Ebenenlayout(Liste von Knoten l , Quader q )
2.         Bestimme A als Produkt der kürzeren beiden Seiten des umfassenden
   Quaders q
3.         Solange r≤1
4.             Füge Knoten k aus l der Ebene hinzu
5.              V=V+Größe von k
6.              n=n+1
7.              r=A^3/(nV^2)
8.         Führe Layout für die Ebene durch
9.     Wiederhole 2-8, bis alle Knoten auf Ebenen verteilt sind
```

Das Layout-Problem in der dritten Dimension kann mit Hilfe dieses Algorithmus in Bezug auf die Anzahl der Knoten in linearer Zeit gelöst werden [10], [11].

Diese Methode baut jedoch auf unveränderten zweidimensionalen Layouts auf. Man kann die dritte Dimension auch effektiver nutzen.

4.3.1 Ein weiterer Ansatz

Der Algorithmus für Squarified Treemaps lässt sich auch in einem Detail verändern, damit er eine dreidimensionale Darstellung erzeugt.

Das Grundproblem ist es, ein Recheck, das einen Knoten repräsentiert mit anderen Rechtecken, die die Unterknoten darstellen, zu füllen. Die Unterknoten werden vom Squarified Layout absteigend sortiert verarbeitet.

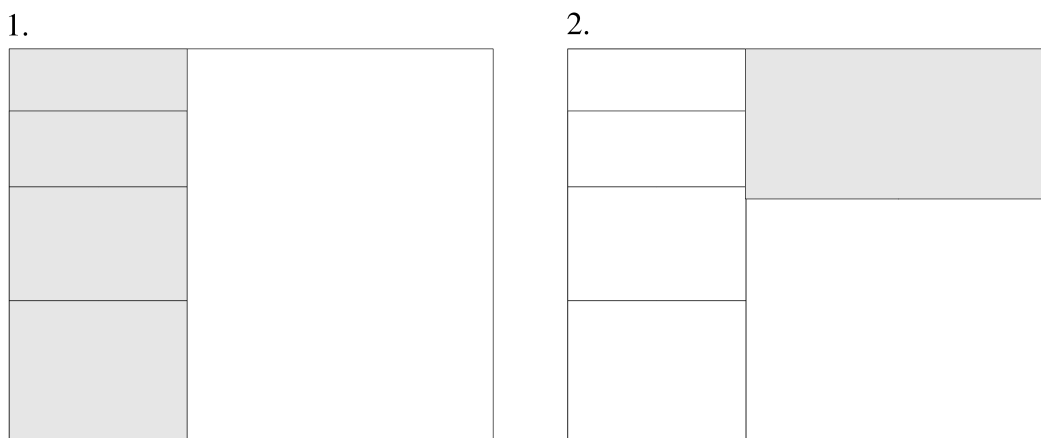


Abbildung 5: Haben die Seitenverhältnisse der Rechtecke einen Grenzwert erreicht, wird eine neue Reihe (markiert) angelegt

Bei Aufruf des Algorithmus wird an einer der kürzeren Seiten des umfassenden Rechtecks eine Reihe von Rechtecken angelegt. Zu dieser Reihe werden solange Knoten hinzugefügt und nebeneinander positioniert, bis ein bestimmtes Kriterium betreffend dem Seitenverhältnis der enthaltenen Rechtecke erreicht ist.
Der Algorithmus wird dann rekursiv für die restliche Fläche und die restlichen Knoten aufgerufen. Es wird eine neue Reihe angelegt, wieder entlang einer der kürzeren beiden Seiten des Rechtecks. Dieser Ablauf ist in der obigen Abbildung verdeutlicht.

Im dreidimensionalen Raum kann man anstelle der rechteckigen Reihe eine Quaderförmige annehmen. Diese muss auf einer der beiden kleinsten Flächen des zu füllenden Quaders ausgelegt werden. Das Auffüllen der Reihen und die Abbruchbedingung kann analog zur zweiten Dimension erfolgen. Schon sind die Squarified Treemaps in die dritte Dimension portiert.

```
1. Layout(Liste von Knoten l , Zielquader q )
2.     Baue eine neue Reihe auf
3.     Solange ein Knoten k in der Liste l enthalten ist und kein Grenzwert
   erreicht ist
4.         Füge k zur Reihe hinzu, entferne k aus l
5.     Bestimme die entgültige Größe der Reihe
6.     Setze die Positionen aller Knoten in der Reihe
7.     Wenn noch Knoten in l enthalten sind, rufe Layout rekursiv mit l und dem
   restlichen Raum auf
```

5 Implementation

Als experimentelle Platform und zur praktischen Umsetzung der Ideen, ist das Programmprojekt 'GLTreeView' entstanden.
Die verwendete Sprache ist C++, da der erzeugte native Code schnell ist, und C++ weitreichenden Zugriff auf 3D-Schnittstellen bietet. Als solche habe ich das bewährte OpenGL [12] verwendet, da es unter vielen Systemen unterstützt wird.

Das Ziel des Programmprojekts war es, dreidimensionale und zweidimensionale Visualisierungen auszuprobieren, Vor- und Nachteile zu bestimmen und zu vergleichen.

5.1 Programmaufbau

Die Struktur von GLTreeView ähnelt der einer 3D-Engine oder einem 3D-Spiel, denn neben der Grafik-Schnittstelle ist auch die Art der Navigation in GLTreeView ähnlich wie in 3D-Spielen gestaltet.

GLTreeView hat eine Hauptschleife, in der die Aufgaben

- Abfrage der Benutzereingaben und Systemnachrichten
- Animation und Bewegung
- Zeichnung in OpenGL

wiederholt werden.

Die Animation umfasst die Bewegung der Kamera und das dynamische Anpassen des Baums.

Gezeichnet wird im Wesentlichen die Baumdarstellung und einige Anzeigen des Benutzerinterfaces. Der Benutzer kann z.B. eine Konsole anzeigen, an der Programmeinstellungen und Befehle eingegeben werden können.

Die Maus und einige Keyboard-Tasten dienen zur Navigation in der Visualisierung.

Mit diesem Framework als Grundlage kann man sich auf die Implementierung der Visualisierung konzentrieren.

5.2 Datenstruktur des Baums

Der Name sagt eben doch noch nicht alles, denn unter C++ hat man die Wahl, den Baum als Struktur oder als Klasse zu schreiben, wobei ich mich für ein Klassenmodell entschieden habe.

In diesem Modell hat jeder Knoten des Baums einen Zeiger auf den übergeordneten oder Elternknoten, einen Zeiger auf den untergeordneten Knoten und Zeiger auf 'Geschwisterknoten', um bei den Stammbaum-Beschreibungen zu bleiben. Alle Knoten, die demselben Elternknoten direkt untergeordnet sind, sind durch eine bidirektional verkettete Liste verbunden. Jeder Knoten hat also noch einen Zeiger auf den Vorgänger und Nachfolger in dieser Liste. Die einfache Navigation im Baum ist durch diese Klassen-Member gewährleistet.
Die Member-Funktion `Depth()` gibt die Tiefe eines Nodes an, wobei die Tiefe bei der Wurzel mit 1 anfängt.

Die nächste von der grundlegenden Knoten-Klasse abgeleitete Klasse heisst `CLayoutNode` und implementiert in ihren Methoden die zwei- und dreidimensionalen Treemap-Verfahren. Sie beinhaltet auch die Attribute, die die Größe und Position der Quader-Repräsentation angeben.

Darauf baut wiederum die Klasse `CDirNode` auf, die den Knoten den Datei/Verzeichnisnamen als Attribut hinzufügt, wodurch ein wirklicher Dateibaum aufgebaut werden kann. Sie besitzt auch die Fähigkeit, unter Verwendung der Betriebssystem-API den Baum aufzubauen.
Die Funktion `Size()` von `CDirNode` gibt die Größe der Datei zurück.

5.3 Darstellung der Treemaps

Wie oben erwähnt, gibt es in GLTreeView eine dreidimensionale und eine zweidimensionale Form der Treemaps. Beide werden in OpenGL im dreidimensionalen Raum angezeigt. Bei der zweidimensionalen Anzeige sind die Treemaps auf der x-y-Ebene angezeigt, die z-Achse ist Zoomachse.

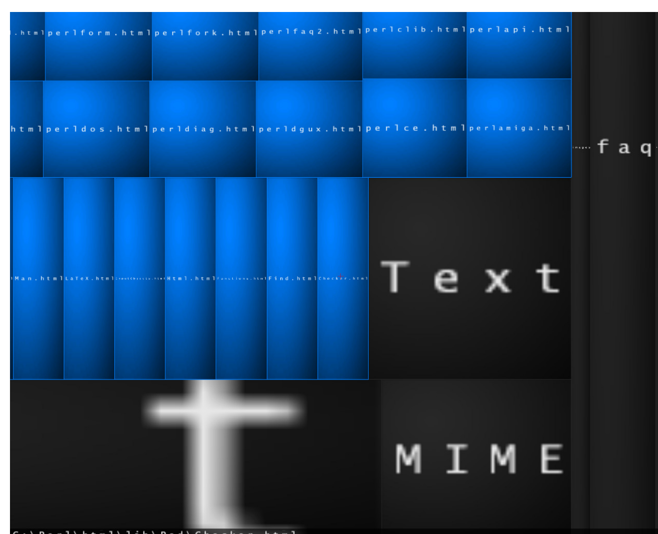


Abbildung 6: Zweidimensionale Ansicht in GLTreeView

Sehr Praktisch sind die Farbcodierungen der Dateien: Verzeichnisse sind grau, grün sind alle nicht näher bestimmten Dateien, Quellcode ist in Orangetönen gehalten, Dokumentation, wie etwa

HTML in blau, ausführbare Dateien sind hellblau gefärbt.

Beim dreidimensionalen, quaderförmigen Layout liegen die Quader nahtlos nebeneinander. Man kann die Quaderflächen nun nicht deckend zeichnen, da dahinterliegende Quader in der Projektion verdeckt würden.
Testweise habe ich in GLTreeView nur die Umrisse der Quader gezeichnet. Das Ergebnis wurde leider bei vielen Dateien und Verzeichnissen zu unübersichtlich. Daraufhin habe ich versucht, die Repräsentation der Dateien zu vereinfachen.

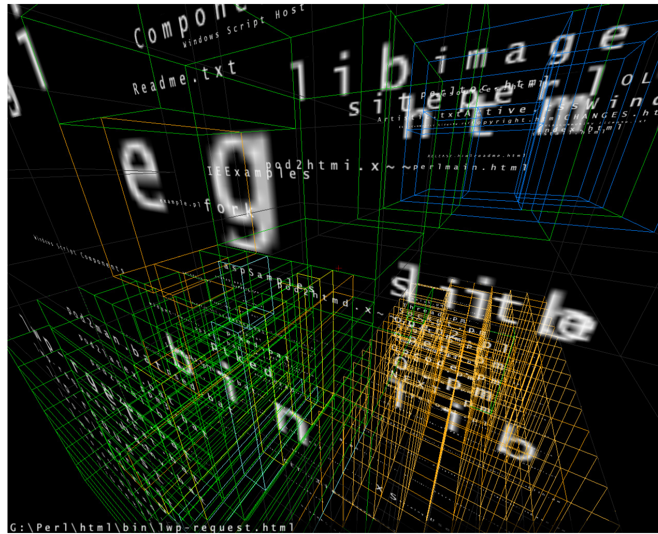


Abbildung 7: Umrisszeichnung der Quader ist unübersichtlich

Anstatt der Quader habe ich Billboards eingesetzt. Billboards sind Objekte, die sich immer dem Betrachter bzw. der Kamera zuwenden [13]. Ich habe als Billboard eine einfache Textur von einem gefüllten Kreis benutzt. Im Zentrum je eines Quaders habe ich dann ein Billboard gesetzt, dessen Größe dem Volumen des Quaders entspricht. Durch die Transparenz werden keine Objekte vollständig verdeckt.
Viele Billboards auf einer kleinen Fläche sehen wegen der weichen Kanten eher wie eine Wolke aus. Das bietet zusammen mit der Farbcodierung eine vage Übersicht über die Dateitypen, in der Abbildung unten ist z.B. eine Wolke HTML-Dateien (blau), Perl-Skripte(gelb) und weitere Dateien in grün zu sehen.
Wenn man allerdings eine bestimmte Datei sucht, sollte man die Anzahl der angezeigten Dateien reduzieren.

Abbildung 8: Billboards mit 90% Transparenz. Da hier viele Billboards gezeichnet sind, kann man nur Wolken erkennen.

Für die Beschriftung der Dateien und Verzeichnisse habe ich keine Billboards sondern einfach auf der x-y-Ebene liegende Schrift eingesetzt.

Welche Art der Darstellung nun die beste ist, zeigte sich durch praktische Tests.

5.4 Dynamische Anzeige von Treemaps

Bei der Anzeige des kompletten Dateibaums einer Festplatte muss man bis zu 100000 Knoten zeichnen. Selbst auf modernen PC's dauert das zu lange, um noch eine nützliche Interaktion, wie z.B. Ein- und Auszoomen ermöglichen zu können, denn beim Zoomen muss die Map ja neu gezeichnet werden.

Man muss also auf jeden Fall auf das Zeichnen einiger Rechtecke, Quader oder Billboards verzichten.

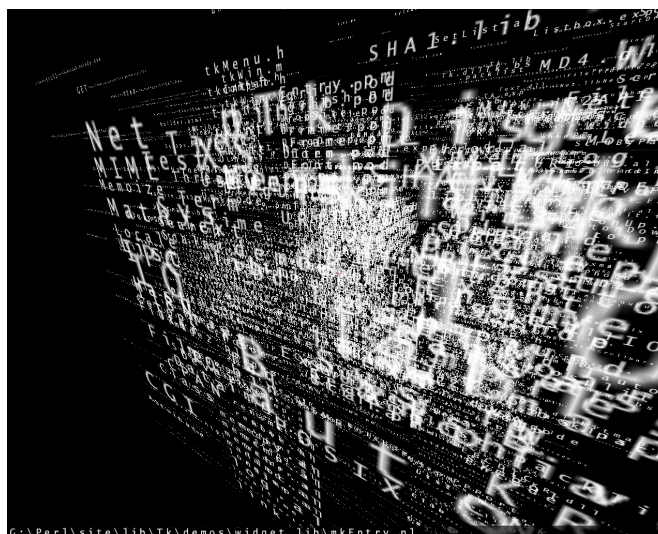


Abbildung 9: Beschriftung aller Dateien des Perl-Verzeichnisses

5.4.1 Geometrische Vereinfachung je nach Kameraentfernung

In unserem konkreten Fall der Verzeichnisstruktur, können Dateien, die so klein sind, dass man ihren Namen nicht erkennen kann, übersprungen werden. Anstattdessen zeichnet man das noch erkennbare übergeordnete Verzeichnis.

Beim rekursiven Zeichnen des Baums kommt also die Abbruchbedingung hinzu, nur Knoten, deren Gewicht einen Mindestwert überschreitet, rekursiv weiterzuzeichnen.

Der Mindestwert ist allerdings auch abhängig von der Entfernung. Das Produkt aus Größe eines Objekts in der Projektion und Entfernung dieses Objekts zur Kamera bleibt wegen dem der Projektion zugrundeliegenden Strahlensatz konstant, wenn man die Kamera verschiebt.

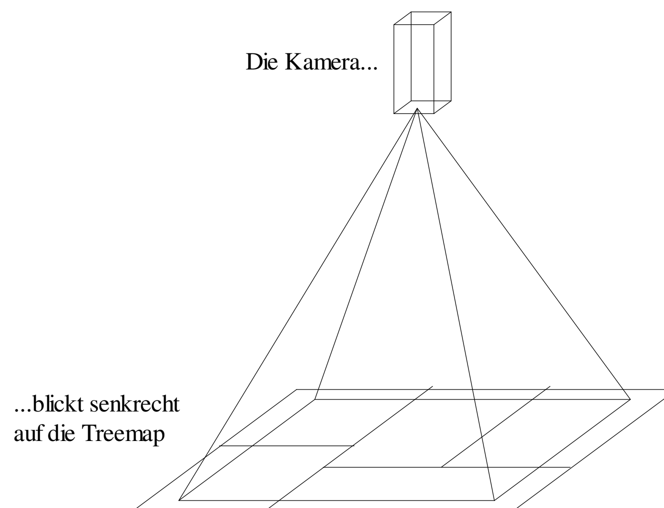


Abbildung 10: Zweidimensionaler Ansichtsmodus in GLTreeView

In der zweidimensionalen Darstellung blickt die Kamera senkrecht auf die x-y-Ebene. Die z-Distanz der Kamera zu einem Objekt kann vereinfacht als Distanz angenommen werden. Schließlich muss die Entfernung beim Zeichnen für jedes Objekt berechnet werden, und das Wurzelziehen bei der Euklidischen Distanz kostet viel Zeit.

Letztendlich habe ich folgendes Größenkriterium benutzt:

w : Breite eines Objekts

h : Höhe eines Objekts

d : Entfernung der Kamera(d ist gleich der z-Koordinate der Kamera, wenn das betrachtete Objekt auf der x-y-Ebene liegt)

g : Grenzwert

Pseudocode:

1. Wenn $\frac{w\,d}{z^2} > g$
2.   zeichne Objekt

Im praktischen Einsatz zeigte sich das Verfahren sehr brauchbar. Zoomt man mit dem Mausrad an ein Verzeichnis heran, wird es immer größer, bis es dann rekursiv weitergezeichnet wird. Es wird also ohne Mausklick geöffnet.

5.4.2 Culling von Objekten

Weiterhin brauchen Rechtecke, die nicht mehr im Sichtfeld liegen, ebenfalls nicht gezeichnet zu werden. Da das Sichtfeld bei 3D-Anwendungen auch Frustum genannt wird, nennt man diese Technik Frustum-Culling [14].
OpenGL bietet die Möglichkeit zu beliebigen Objekten, die Anzahl der eigentlich gezeichneten Pixel auszugeben, ohne das Objekt wirklich zu zeichnen. Man kann ein Objekt durch einfache Geometrie annähern, diese zeichnen und dann entscheiden, ob man das eigentliche Objekt ebenfalls zeichnet oder nicht. Weil diese Erweiterung auch zum Test, ob Objekte einander verdecken, verwendet werden kann, nennt man sie Occlusion Test (Verdeckungs-Test).

```
1. Setze OpenGL in Occlusion Test-Modus
2. Zeichne angenäherte Geometrie (Bounding-Box) eines Objekts
3. Werte Test aus
4. Wenn die Pixelanzahl größer als ein Grenzwert ist
5.     Setze OpenGL in Zeichen-Modus
6.     Zeichne Objekt
```

Obwohl diese Methode mit Hardware-Beschleunigung arbeitet, war die Geschwindigkeit des Occlusion Tests im praktischen Einsatz nicht ausreichend.

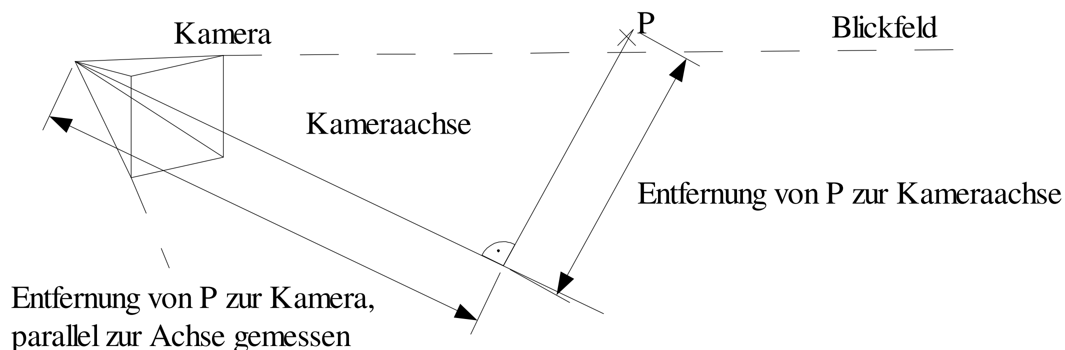


Abbildung 11: Vereinfachtes Frustum Culling

Daraufhin habe ich eine einfachere, heuristische Methode des Occlusion Tests verwendet:
Wenn die Entfernung eines Punkts zur Mittelachse der Kamera hinreichend größer als die Entfernung dieses Punktes zur Kamera ist, dann ist dieser höchstwahrscheinlich nicht im Blickfeld.

Der Grenzwert für die Entfernung des Punktes zur Kameraachse muss nach dem Strahlensatz proportional mit der Entfernung von P zur Kamera steigen. Man könnte hier Trigonometrische Funktionen anwenden, um einen genauen Proportionalitätsfaktor zu bestimmen, jedoch ist ein exaktes Kriterium hier nicht erforderlich.

Mit den Bezeichnern
z : Entfernung von P zur Kamera, parallel zur Achse gemessen
d : Entfernung von P zur Kameraachse
f : Proportionalitätsfaktor zwischen z und `dist`
muss folgende Prüfung durchgeführt werden:

1. wenn $z f < d$
2. Zeichne P

Allerdings handelt es sich bei den gezeichneten Objekten ja nicht nur um einfache Punkte, sondern um Quader oder Rechtecke. Nun kann man entweder alle Ecken eines Quaders oder Rechtecks prüfen, oder, wie ich es letztendlich in GLTreeView implementiert habe, das Zentrum des Quaders

oder Rechtecks als Punkt annehmen.

Bei der Verwendung der Methode gab es einige vereinfachende Umstände: In der zweidimensionalen Baumdarstellung liegen ja alle Rechtecke in der x-y-Ebene. Die Distanz zur Kamera ist gleich deren z-Position, da sie senkrecht auf die Rechtecke blickt. Die Distanz zur Kameraachse muss nur an den x- und y-Koordinaten abgelesen werden, hier kann die einfachere Manhattan-Distanz anstelle der Euklidischen Distanz berechnet werden.

5.5 Bewertung

Erst im praktischen Einsatz stellten sich die Vor- und Nachteile der Visualisierungsmethoden heraus.
In der dreidimensionalen Darstellung fehlt die Gesamtübersicht über ein Verzeichnis. Man sieht immer nur die Teile des Verzeichnisses, die nahe bei der Kamera liegen, die Anderen verschwinden in der Ferne. Sehr unpraktisch ist es, wenn Objekte von Anderen verdeckt werden, sich überlagernde Aufschriften sind extrem schlecht zu lesen.
Man kann zwar prinzipiell schnell navigieren doch sollte man vorher wissen, wo die gesuchte Datei liegt. Zum Erforschen der Verzeichnisstruktur ist diese Ansicht nicht geeignet.

Die dreidimensionale Darstellung hat viele der erhofften Vorteile nicht erbracht.
Die zweidimensionale Ansicht bietet hingegen Übersicht und Navigation ohne Mausklicks. Die in GLTreeView implementierte, stufenlose Zoomfunktion kann allerdings auch als dritte Dimension interpretiert werden, nur blickt man lediglich auf eine zweidimensionale Fläche.

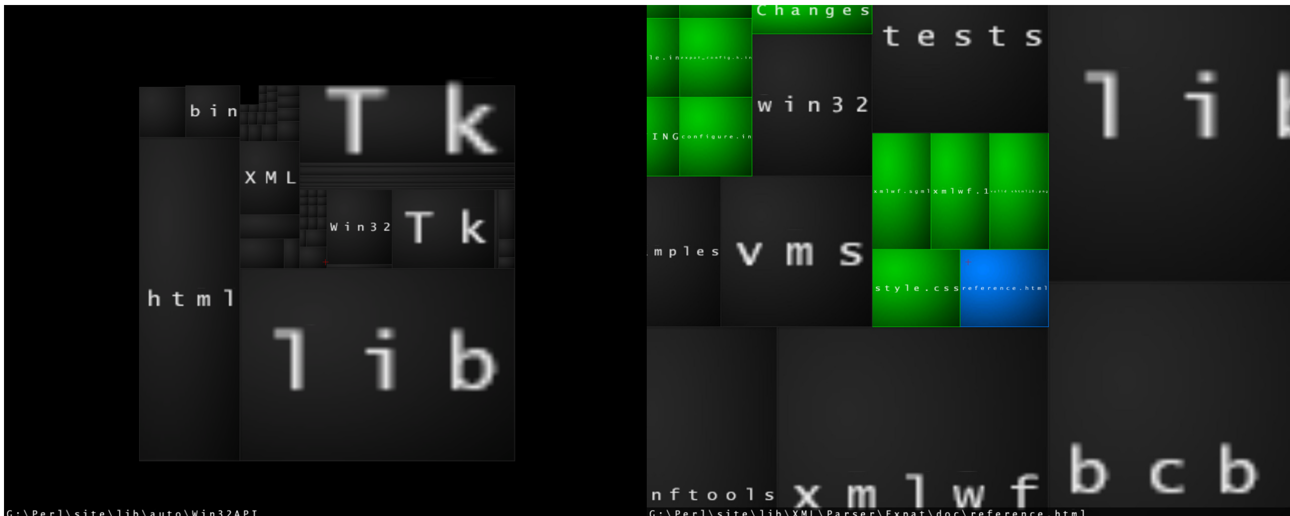


Abbildung 12: Das Perl-Verzeichnis von Weitem ...

Abbildung 13: ... und aus nächster Nähe

5.6 Hosting des Projekts

Schon zu Beginn des Projekts stand fest, den Quellcode von GLTreeView zu veröffentlichen, denn zur kommerziellen Nutzung ist das experimentelle Programm noch nicht geeignet.
Bei der Suche nach einer Hosting-Möglichkeit bin ich schnell bei Sourceforge.net gelandet. Dort können ausschließlich Open-Source-Projekte veröffentlicht werden, jedes Projekt erhält die Möglichkeit der Quellcode-Verwaltung, eine Dokumentationswebsite und man kann Downloads anbieten.
Vom Austausch mit der Open-Source-Gemeinde, sobald sie auf das Projekt aufmerksam wird, erhoffe ich mir konstruktive Kritik und Anregungen.

Das Projekt ist unter http://gltreeview.sourceforge.net/ abrufbar.

6 Aussichten

Bislang befindet sich GLTreeView noch in der Entwicklungsphase. Das Programm kann zu einem nützlichen Datei-Browser weiterentwickelt werden. Dann müssten allerdings noch einige Dinge hinzugefügt werden, wie z.B. Icons, Kopier- und Verschiebemöglichkeit, Anzeige von Bildern und Dateien, etc...
Ob und wann ich weitere Funktionen hinzufüge, ist noch ungewiss.

Mit den Treemaps ist erst ein kleiner Teil der möglichen Visualisierungen abgedeckt. Effektivere Baumdarstellungen können gefunden werden.
Es gibt zudem andere Möglichkeiten Dateien einzuordnen außer anhand ihrer Pfade. Dateien können auch anhand von extrahierten Schlüsselwörtern, anhand der Dateitypen oder anhand der Namen miteinander verknüpft werden...

7 Danksagung

In erster Linie danke ich Herrn Dr. Otto May, der diese Arbeit durch seine Motivierung nicht unerheblich vorangetrieben hat.

Weiterhin danke ich den Entwicklern von hilfreicher, kostenloser Software, als da wären:
Ulli Meybohm, der einen praktischen Allround-Editor, mit dem die meisten Sourcen zu GLTreeView entstanden sind, als Freeware anbietet.
Die Projekt-Mitglieder von OpenOffice.org, dem OpenSource-Office-Paket, dass dem Marktführer durchaus gewachsen ist.
Danke auch an das Team von sourceforge.net, das der halben Open-Source-Welt eine Heimat bietet.

8 Literaturverzeichnis

[1] "GUI - Grafische Benutzer-Schnittstelle", http://de.wikipedia.org/wiki/GUI
[2] "The GUI Gallery", http://www.toastytech.com/guis/
[3] "Longhorn Developer Center", http://msdn.microsoft.com/Longhorn/
[4] "Doom as an Interface for Process Management",
http://www.cs.unm.edu/~dlchao/flake/doom/chi/chi.html
[5] "3D Window Manager", http://www.3dwm.org/
[6] "An Atlas of Cyberspaces - Information Space Maps",
http://www.cybergeography.org/atlas/info_maps.html
[7] B. Johnson and B. Shneiderman, "Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures", 1991
[8] "Treemaps for space-constrained visualization of hierarchies",
http://www.cs.umd.edu/hcil/treemap-history/
[9] D. M. Bruls, C. Huizing, J. J. van Wijk, "Squarified Treemaps", 2000
[10] Sedgewick, Robert, "Algorithmen in C", 1992, Addison Wesley
[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, "Introduction to Algorithms", 1990, The MIT Press
[12] Ute Claussen, "Programmieren mit OpenGL", 1997, Springer-Verlag
[13] "Billboarding", http://www.flipcode.com/articles/article_rtr2billboards.shtml
[14] "Frustum Culling", http://www.flipcode.com/articles/article_frustumculling.shtml

9 Anhang

Im Anhang befindet sich der Quelltext zu den für die Darstellung der Treemaps wichtigsten Klassen:

`CTreemapNode` - Baumstruktur

`CLayoutNode` - Treemap-Layout

`CDirNode` - Datei- / Verzeichnisbaum

9.1 CTreemapNode.h

```
1.  // GLTreeView - Visualizing directory trees in OpenGL
2.  // Copyright (C) 2003-2004 by Hannes Flicka
3.  //
4.  // This program is free software; you can redistribute it and/or
5.  // modify it under the terms of the GNU General Public License
6.  // as published by the Free Software Foundation; either version 2
7.  // of the License, or (at your option) any later version.
8.  //
9.  // This program is distributed in the hope that it will be useful,
10. // but WITHOUT ANY WARRANTY; without even the implied warranty of
11. // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12. // GNU General Public License for more details.
13. //
14. // You should have received a copy of the GNU General Public License
15. // along with this program; if not, write to the Free Software
16. // Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307,
    USA.
17.
18.
19. #ifndef _CTREEMAPNODE_H_
20. #define _CTREEMAPNODE_H_
21.
22. #include "common.h"
23.
24. class CTreemapNode {
25. public:
26.     CTreemapNode() :
27.             mparent(NULL),
28.             mchild(NULL),
29.             mnext(NULL),
30.             mprev(NULL),
31.             mdepthiscached(false),
32.     mmxdepthiscached(false) {}
33.
34.     ~CTreemapNode() {}
35.
36.     virtual REAL Size() {
37.         return 1.0;
38.     }
39.
40.     virtual int Depth() {
41.         if (mdepthiscached)
42.             return mcacheddepth;
43.         if (mparent) {
44.             mdepthiscached = true;
45.             return mcacheddepth = mparent->Depth() + 1;
46.         } else {
47.             mdepthiscached = true;
48.             return mcacheddepth = 1;
49.         }
50.     }
51.
52.     virtual int MaxDepth() {
53.         if (mmxdepthiscached)
54.             return mcmxdepth;
55.         if (!mchild) {
56.             mmxdepthiscached = true;
57.             return mcmxdepth = Depth();
58.         } else {
```

```
59.             mcmxdepth = 0;
60.             for (CTreemapNode * tn = mchild; tn; tn = tn->mnext)
61.                 if (tn->MaxDepth() > mcmxdepth)
62.                     mcmxdepth = tn->MaxDepth();
63.             mmxdepthiscached = true;
64.             return mcmxdepth;
65.         }
66.     }
67.
68.     virtual CTreemapNode * Root() {
69.         if (mparent)
70.             return mparent->Root();
71.         else
72.             return this;
73.     }
74.
75.     virtual CTreemapNode * LevelLast() {
76.         if (mnext)
77.             return mnext->LevelLast();
78.         else
79.             return this;
80.     }
81.
82.     virtual CTreemapNode * LevelFirst() {
83.         if (mprev)
84.             return mprev->LevelFirst();
85.         else
86.             return this;
87.     }
88.
89.     bool SortChildren();
90.     bool RecursiveSort();
91.
92.     CTreemapNode * mparent;
93.     CTreemapNode * mchild;
94.     CTreemapNode * mnext;
95.     CTreemapNode * mprev;
96.
97.     int mcacheddepth;   // depht cache
98.     bool mdepthiscached;
99.     int mcmxdepth;   // max depht cache
100.     bool mmxdepthiscached;
101.private:
102.};
103.
104.#endif
```

9.2 CTreemapNode.cpp

```
1. // GLTreeView - Visualizing directory trees in OpenGL
2. // Copyright (C) 2003-2004 by Hannes Flicka
3. //
4. // This program is free software; you can redistribute it and/or
5. // modify it under the terms of the GNU General Public License
6. // as published by the Free Software Foundation; either version 2
7. // of the License, or (at your option) any later version.
8. //
9. // This program is distributed in the hope that it will be useful,
10. // but WITHOUT ANY WARRANTY; without even the implied warranty of
11. // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12. // GNU General Public License for more details.
```

```
13. //
14. // You should have received a copy of the GNU General Public License
15. // along with this program; if not, write to the Free Software
16. // Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307,
    USA.
17.
18.
19. #include "common.h"
20. #include "CTreemapNode.h"
21.
22. /** SortChildren() sortiert alle untergeordneten Knoten der Gre nach.
23.     Es wird Bubble Sort verwendet und absteigend sortiert */
24. bool CTreemapNode::SortChildren() {
25.     CTreemapNode *pln,    // node
26.     *pnn,       // Next node
27.     *pnnn,        // Next next node
28.     *ppn = NULL;    // previous node
29.
30.     if (!mchild)
31.         return true; // kein element
32.     if (!mchild->mnext)
33.         return true;     // ein element
34.
35.
36.     bool switched;
37.     do {
38.         switched = false;
39.         for (pln = mchild; pln != NULL; pln = pln->mnext) {
40.             pnn = pln->mnext;
41.             if (pnn == NULL)
42.                 break;
43.             if (pnn->Size() > pln->Size()) { // Die Knoten sind nicht
    absteigend sortiert
44.                 ppn = pln->mprev;
45.                 pnnn = pnn->mnext;
46.                 // Knoten austauschen
47.                 // previous node->next updaten
48.                 if (ppn != NULL)
49.                     ppn->mnext = pnn;
50.                 // this node updaten
51.                 pln->mprev = pnn;
52.                 pln->mnext = pnnn;
53.                 // next node updaten
54.                 pnn->mnext = pln;
55.                 pnn->mprev = ppn;
56.                 // next next node updaten
57.                 if (pnnn != NULL)
58.                     pnnn->mprev = pln;
59.
60.                 switched = true;
61.             }
62.         }
63.     } while (switched);
64.
65.     mchild = pln->LevelFirst();
66.
67.     return true;
68. }
69.
70. bool CTreemapNode::RecursiveSort() {
71.     if (!SortChildren())
```

```
72.         return false;
73.     for (CTreemapNode * ptn = mchild; ptn; ptn = ptn->mnext)
74.         if (!ptn->RecursiveSort())
75.             return false;
76.     return true;
77. }
```

9.3 CLayoutNode.h

```
1. // GLTreeView - Visualizing directory trees in OpenGL
2. // Copyright (C) 2003-2004 by Hannes Flicka
3. //
4. // This program is free software; you can redistribute it and/or
5. // modify it under the terms of the GNU General Public License
6. // as published by the Free Software Foundation; either version 2
7. // of the License, or (at your option) any later version.
8. //
9. // This program is distributed in the hope that it will be useful,
10. // but WITHOUT ANY WARRANTY; without even the implied warranty of
11. // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12. // GNU General Public License for more details.
13. //
14. // You should have received a copy of the GNU General Public License
15. // along with this program; if not, write to the Free Software
16. // Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307,
    USA.
17.
18.
19. #ifndef _CLAYOUTNODE_H_
20. #define _CLAYOUTNODE_H_
21.
22. #include "common.h"
23. #include "CTreemapNode.h"
24. #include "CGlbox.h"
25.
26. class CLayoutNode : public CTreemapNode, public CGlbox {
27. public:
28.     CLayoutNode() {}
29.     CLayoutNode(char * dir) {}
30.
31.     // Simples, lineares Layout:
32.     bool LayoutChildren();
33.
34.     // Kondition zum SquareLayout:
35.     bool Condition(REAL big, REAL Small, REAL val0, REAL valc1, REAL sum,
    REAL total);
36.
37.     // Mglichst kubische Nodes:
38.     bool SquareLayout(CLayoutNode * pfirst, CLayoutNode * pstop, VECTOR pos,
    VECTOR size);
39.
40.     // Die kleinste Dimension des Vektors size wird automatisch gesucht
41.     bool SquareLayoutMultiDim(CLayoutNode * pfirst, CLayoutNode * pstop,
    VECTOR pos, VECTOR size);
42.
43.     // Das obige auf zwei Dimensionen reduziert
44.     bool SquareLayoutTwoDim(CLayoutNode * pfirst, CLayoutNode * pstop, VECTOR
    pos, VECTOR size);
45.
46.     // Lineares Layout,
47.     bool LayerFill(CLayoutNode * pfirst, CLayoutNode * pstop, VECTOR pos,
```

```
   VECTOR size);
48.
49.     // Container in Layer einteilen, in denen 2-Dimensionales Layout
   angewandt wird
50.     bool LayerLayout(CLayoutNode * pfirst, CLayoutNode * pstop, VECTOR pos,
   VECTOR size);
51.
52.     bool RecursiveSquaredTwoDim();
53.
54.     // Layer-Layout auf alle Nodes anwenden
55.     bool RecursiveLayer();
56.
57.     // Alle LayoutNodes von der Wurzel an Rekursiv auslegen
58.     bool RecursiveSquarify();
59.
60.     // Lineares Layout rekursiv anwenden
61.     bool RecursiveLayout();
62.
63.     // Zeichnen ...
64.     bool DrawChildren(CCamera * pcam);
65. private:
66. };
67.
68. #endif
```

9.4 CLayoutNode.cpp

```
1. // GLTreeView - Visualizing directory trees in OpenGL
2. // Copyright (C) 2003-2004 by Hannes Flicka
3. //
4. // This program is free software; you can redistribute it and/or
5. // modify it under the terms of the GNU General Public License
6. // as published by the Free Software Foundation; either version 2
7. // of the License, or (at your option) any later version.
8. //
9. // This program is distributed in the hope that it will be useful,
10. // but WITHOUT ANY WARRANTY; without even the implied warranty of
11. // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12. // GNU General Public License for more details.
13. //
14. // You should have received a copy of the GNU General Public License
15. // along with this program; if not, write to the Free Software
16. // Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307,
   USA.
17.
18. #include "common.h"
19. #include "CLayoutNode.h"
20.
21. bool CLayoutNode::LayoutChildren() {
22.     REAL totalsize = 0, cursize = 0, r;
23.     CLayoutNode * pln;
24.
25.     // Grssen der Nodes zusammenzhlen
26.     for (pln = (CLayoutNode*)mchild; pln; pln = (CLayoutNode*)pln->mnext)
27.         totalsize += pln->Size();
28.
29.     if (mscale.x >= mscale.y) { // breiter als hoch
30.         // Rechtecke einfach linear anordnen
31.         for (pln = (CLayoutNode*)mchild; pln; pln = (CLayoutNode*)pln->mnext)
   {
32.             r = pln->Size();
```

```
33.             pln->mscale.z = mscale.z;
34.             pln->mscale.y = mscale.y;
35.             pln->mscale.x = mscale.x * r / totalsize;
36.             pln->mpos.z = mpos.z + mscale.z;
37.             pln->mpos.y = mpos.y;
38.             pln->mpos.x = mpos.x + mscale.x * cursize / totalsize;
39.             cursize += r;
40.         }
41.     } else {    // hher als breit
42.         // Rechtecke linear anordnen
43.         for (pln = (CLayoutNode*)mchild; pln; pln = (CLayoutNode*)pln->mnext)
    {
44.             r = pln->Size();
45.             pln->mscale.z = mscale.z;
46.             pln->mscale.x = mscale.x;
47.             pln->mscale.y = mscale.y * r / totalsize;
48.             pln->mpos.z = mpos.z + mscale.z;
49.             pln->mpos.x = mpos.x;
50.             pln->mpos.y = mpos.y + mscale.y * cursize / totalsize;
51.             cursize += r;
52.         }
53.     }
54.
55.     return true;
56. }
57.
58. // kondition, um zu testen ob ein neues rechteck eingereiht wird oder nicht
59. // Big ist die lngere Seite des Container-Rechtecks
60. // Small ist dessen krzere Seite
61. // val0 ist die Grsse des ersten Rechtecks
62. // valc1 ist die Gre des nchsten Rechtecks
63. bool CLayoutNode::Condition(REAL big, REAL Small, REAL val0, REAL valc1, REAL
    sum, REAL total) {
64.     REAL r1 = big * sum * sum / total / Small / val0;
65.     REAL r2 = big * (sum + valc1) * (sum + valc1) / total / Small / val0;
66.     if (r1 < 1.0)
67.         r1 = 1.0 / r1;
68.     if (r2 < 1.0)
69.         r2 = 1.0 / r2;
70.     return r1 > r2;
71. }
72.
73. // Nodes, angefangen bei pfirst an die Position 'pos' mit Gre size
74. // packen
75. bool CLayoutNode::SquareLayout(CLayoutNode * pfirst, CLayoutNode * pstop,
    VECTOR pos, VECTOR size) {
76.     CLayoutNode *pln, *plast;
77.
78.     // Sonderflle: wenn 1 oder kein Node, zurckkehren
79.     if (!pfirst)
80.         return true;
81.     if (!pfirst->mnext) {
82.         pfirst->mpos = pos;
83.         pfirst->mscale = size;
84.         return true;
85.     }
86.     if (pfirst == pstop)
87.         return true;
88.
89.     REAL totalsize = 0;
90.     for (pln = pfirst; pln != pstop; pln = (CLayoutNode*)pln->mnext)
```

```
91.         totalsize += pln->Size();    // alle gren ab pfirst zusammenrechnen
92.
93.     REAL sum;
94.     if (size.x > size.y) {    // breiter als hoch => reihe wird an die seite
  gepackt
95.         plast = pfirst;
96.         sum = pfirst->Size();
97.         while (plast->mnext != pstop) {    // Solange Rechtecke zur aktuellen
  Reihe hinzufgen, bis ...
98.             if (!Condition(size.x, size.y, pfirst->Size(), pfirst->mnext-
  >Size(), sum, totalsize))
99.                 break;    // Eine Kondition nicht mehr zutrifft
100.            plast = (CLayoutNode*)plast->mnext; // entscheidungen treffen...
101.            sum += plast->Size();
102.        }
103.
104.        // layoutrow, Reihe anlegen
105.        REAL w;
106.        REAL h;
107.        REAL rpos;
108.        rpos = 0;
109.        w = size.x * sum / totalsize;
110.        for (pln = pfirst; pln != plast->mnext; pln = (CLayoutNode*)pln-
  >mnext) {
111.            h = size.y * pln->Size() / sum;
112.            pln->mscale = VECTOR(w, h, size.z);
113.            pln->mpos = pos + VECTOR(0, rpos, 0);
114.            rpos += h;
115.        }
116.
117.        // rekursion, nchste Reihe aufbauen
118.        VECTOR nsize = size * VECTOR(1.0 - sum / totalsize, 1.0, 1.0);
119.        VECTOR npos = pos + VECTOR(size.x * sum / totalsize, 0, 0);
120.        if (!SquareLayout((CLayoutNode*)plast->mnext, pstop, npos, nsize))
121.            return true;
122.    } else {
123.        plast = pfirst;
124.        sum = pfirst->Size();
125.        while (plast->mnext != pstop) {
126.            if (!Condition(size.y, size.x, pfirst->Size(), pfirst->mnext-
  >Size(), sum, totalsize))
127.                break;
128.            plast = (CLayoutNode*)plast->mnext; // entscheidungen treffen...
129.            sum += plast->Size();
130.        }
131.
132.        // layoutrow
133.        REAL w;
134.        REAL h;
135.        REAL rpos;
136.        rpos = 0;
137.        w = size.y * sum / totalsize;
138.        for (pln = pfirst; pln != plast->mnext; pln = (CLayoutNode*)pln-
  >mnext) {
139.            h = size.x * pln->Size() / sum;
140.            pln->mscale = VECTOR(h, w, size.z);
141.            pln->mpos = pos + VECTOR(rpos, 0, 0);
142.            rpos += h;
143.        }
144.
145.        // rekursion
146.        VECTOR nsize = size * VECTOR(1.0, 1.0 - sum / totalsize, 1.0);
```

```
147.         VECTOR npos = pos + VECTOR(0, size.y * sum / totalsize, 0);
148.         if (!SquareLayout((CLayoutNode*)plast->mnext, pstop, npos, nsize))
149.             return true;
150.     }
151.     return true;
152.}
153.
154.
155.bool CLayoutNode::SquareLayoutMultiDim(CLayoutNode * pfirst, CLayoutNode *
  pstop, VECTOR pos, VECTOR size) {
156.     CLayoutNode *pln, *plast;
157.
158.     // Sonderflle: wenn 1 oder kein Node, zurckkehren
159.     if (!pfirst)
160.         return true;
161.     if (!pfirst->mnext) {
162.         pfirst->mpos = pos;
163.         pfirst->mscale = size;
164.         return true;
165.     }
166.     if (pfirst == pstop)
167.         return true;
168.
169.     REAL totalsize = 0;
170.     for (pln = pfirst; pln != pstop; pln = (CLayoutNode*)pln->mnext)
171.         totalsize += pln->Size();    // alle gren ab pfirst zusammenrechnen
172.
173.     // Lngste Dimension finden ...
174.     int mindim;
175.     int Da, Db;    // Nebendimensionen Da > Db
176.     if (size.x < size.y) {
177.         if (size.x < size.z) { // max: x
178.             mindim = 0;
179.             if (size.z < size.y) {
180.                 Db = 2;
181.                 Da = 1;
182.             } else {
183.                 Da = 2;
184.                 Db = 1;
185.             }
186.         } else { // z
187.             mindim = 2;
188.             Da = 1;
189.             Db = 0;
190.         }
191.     } else {
192.         if (size.y < size.z) { // y
193.             mindim = 1;
194.             if (size.x < size.z) {
195.                 Db = 0;
196.                 Da = 2;
197.             } else {
198.                 Da = 0;
199.                 Db = 2;
200.             }
201.         } else { // z
202.             mindim = 2;
203.             Db = 1;
204.             Da = 0;
205.         }
206.     }
```

```
207.    if (size[Da] < size[Db] || Da == Db || Db == mindim || Da == mindim) {
208.        GENERR("strange");
209.    }
210.
211.    REAL sum;
212.
213.    plast = pfirst;
214.    sum = pfirst->Size();
215.    while (plast->mnext != pstop) { // Solange Rechtecke zur aktuellen Reihe
  hinzufgen, bis ...
216.        if (!Condition(size[Da], size[Db], pfirst->Size(), pfirst->mnext-
  >Size(), sum, totalsize))
217.            break;  // Eine Kondition nicht mehr zutrifft
218.        plast = (CLayoutNode*)plast->mnext; // entscheidungen treffen...
219.        sum += plast->Size();
220.    }
221.
222.    // layoutrow, Reihe anlegen
223.    REAL w;
224.    REAL h;
225.    REAL rpos;
226.    rpos = 0;
227.    w = size[Da] * sum / totalsize;
228.
229.    for (pln = pfirst; pln != plast->mnext; pln = (CLayoutNode*)pln->mnext) {
230.        h = size[Db] * pln->Size() / sum;
231.
232.        pln->mscale[Da] = w;
233.        pln->mscale[Db] = h;
234.        pln->mscale[mindim] = size[mindim];
235.
236.        pln->mpos = pos;
237.        pln->mpos[Db] += rpos;
238.
239.        rpos += h;
240.    }
241.
242.    // rekursion, nchste Reihe aufbauen
243.    VECTOR nsize = size;
244.    nsize[Da] *= 1.0 - sum / totalsize;
245.    VECTOR npos = pos;
246.    npos[Da] += size[Da] * sum / totalsize;
247.
248.    if (!SquareLayoutMultiDim((CLayoutNode*)plast->mnext, pstop, npos,
  nsize))
249.        return false;
250.    return true;
251.}
252.
253.
254.
255.bool CLayoutNode::SquareLayoutTwoDim(CLayoutNode * pfirst, CLayoutNode *
  pstop, VECTOR pos, VECTOR size) {
256.    CLayoutNode *pln, *plast;
257.
258.    // Sonderflle: wenn 1 oder kein Node, zurckkehren
259.    if (!pfirst)
260.        return true;
261.    if (!pfirst->mnext) {
262.        pfirst->mpos = pos;
263.        pfirst->mscale = size;
```

```
264.        return true;
265.    }
266.    if (pfirst == pstop)
267.        return true;
268.
269.    REAL totalsize = 0;
270.    for (pln = pfirst; pln != pstop; pln = (CLayoutNode*)pln->mnext)
271.        totalsize += pln->Size();    // alle gren ab pfirst zusammenrechnen
272.
273.    // Lngste Dimension unter x und y finden ...
274.    int mindim = 2;
275.    int Da, Db;    // Nebendimensionen Da > Db
276.    if (size.x < size.y) {
277.        Da = 1; //y
278.        Db = 0; //x
279.    } else {
280.        Da = 0;
281.        Db = 1;
282.    }
283.
284.    REAL sum;
285.
286.    plast = pfirst;
287.    sum = pfirst->Size();
288.    while (plast->mnext != pstop) { // Solange Rechtecke zur aktuellen Reihe
  hinzufgen, bis ...
289.        if (!Condition(size[Da], size[Db], pfirst->Size(), pfirst->mnext-
  >Size(), sum, totalsize))
290.            break;  // Eine Kondition nicht mehr zutrifft
291.        plast = (CLayoutNode*)plast->mnext; // entscheidungen treffen...
292.        sum += plast->Size();
293.    }
294.
295.    // layoutrow, Reihe anlegen
296.    REAL w;
297.    REAL h;
298.    REAL rpos;
299.    rpos = 0;
300.    w = size[Da] * sum / totalsize;
301.
302.    for (pln = pfirst; pln != plast->mnext; pln = (CLayoutNode*)pln->mnext) {
303.        h = size[Db] * pln->Size() / sum;
304.
305.        pln->mscale[Da] = w;
306.        pln->mscale[Db] = h;
307.        pln->mscale[mindim] = size[mindim];
308.
309.        pln->mpos = pos;
310.        pln->mpos[Db] += rpos;
311.
312.        rpos += h;
313.    }
314.
315.    // rekursion, nchste Reihe aufbauen
316.    VECTOR nsize = size;
317.    nsize[Da] *= 1.0 - sum / totalsize;
318.    VECTOR npos = pos;
319.    npos[Da] += size[Da] * sum / totalsize;
320.
321.    if (!SquareLayoutTwoDim((CLayoutNode*)plast->mnext, pstop, npos, nsize))
322.        return false;
```

```
323.     return true;
324.}
325.
326.
327./*
328.     LayerFill wird von LayerLayout aufgerufen
329.     Die verketteten Nodes von pfirst bis pstop mssen durch 2-dimensionales
330.     Layout in den durch pos und size gegebenen Quader gepackt werden.
331.*/
332.bool CLayoutNode::LayerFill(CLayoutNode * pfirst, CLayoutNode * pstop, VECTOR
   pos, VECTOR size) {
333.
334.     CLayoutNode * pln;
335.     REAL totalsize = 0;
336.     for (pln = pfirst; pln != pstop; pln = (CLayoutNode*)pln->mnext)
337.         totalsize += pln->Size();    // alle gren ab pfirst zusammenrechnen
338.
339.     // Lngste Dimension finden ...
340.     int maxdim;
341.     if (size.x > size.y) {
342.         if (size.x > size.z) { // max: x
343.             maxdim = 0;
344.         } else { // z
345.             maxdim = 2;
346.         }
347.     } else {
348.         if (size.y > size.z) { // y
349.             maxdim = 1;
350.         } else { // z
351.             maxdim = 2;
352.         }
353.     }
354.
355.     REAL curpos = pos[maxdim];
356.     for (pln = pfirst; pln != pstop; pln = (CLayoutNode*)pln->mnext) {
357.         pln->mpos = pos;
358.         pln->mscale = size;
359.         pln->mscale[maxdim] = size[maxdim] * pln->Size() / totalsize;
360.         pln->mpos[maxdim] = curpos;
361.         curpos += pln->mscale[maxdim];
362.     }
363.
364.     return true;
365.}
366.
367./*
368.     Layerlayout: Das durch pos und size gegebene Container-Rechteck
369.     wird in Ebenen eingeteilt, die dann durch 2-dimensionales
370.     Layout geflt werden.
371.     Die sortierten Nodes mssen also auf die Layer verteilt werden.
372.     Dabei ist erst einmal die Optimale Anzahl der Layer zu bestimmen
373.*/
374.bool CLayoutNode::LayerLayout(CLayoutNode * pfirst, CLayoutNode * pstop,
   VECTOR pos, VECTOR size) {
375.     CLayoutNode *pln;
376.
377.     // Sonderflle: wenn 1 oder kein Node, zurckkehren
378.     if (!pfirst)
379.         return true;
380.     if (!pfirst->mnext) {
381.         pfirst->mpos = pos;
```

```
382.         pfirst->mscale = size;
383.         return true;
384.     }
385.     if (pfirst == pstop)
386.         return true;
387.
388.     int numnodes = 0;
389.     REAL totalsize = 0;
390.     for (pln = pfirst; pln != pstop; pln = (CLayoutNode*)pln->mnext) {
391.         totalsize += pln->Size();    // alle gren ab pfirst zusammenrechnen
392.         numnodes++;
393.     }
394.
395.     // Layer entlang der lngsten Dimension schichten ...
396.     // Lngste Dimension finden ...
397.     int maxdim;
398.     if (size.x > size.y) {
399.         if (size.x > size.z) { // max: x
400.             maxdim = 0;
401.         } else { // z
402.             maxdim = 2;
403.         }
404.     } else {
405.         if (size.y > size.z) { // y
406.             maxdim = 1;
407.         } else { // z
408.             maxdim = 2;
409.         }
410.     }
411.     REAL maxdimlen = size[maxdim];
412.
413.     //REAL optnumlayers = (REAL)pow((double)numnodes, 1.0 / 3.0);
414.     // Bei kubischem container muss jeder Layer etwa numnodes^(1/3) Nodes
  aufnehmen.
415.     // Allgemein sind es numnodes *  maxdim / volume Nodes:
416.     REAL optnumlayers = (REAL)numnodes * maxdimlen / (size.x * size.y *
  size.z);
417.
418.     // Layer auslegen ...
419.     REAL sum = 0;
420.     pln = pfirst;
421.     VECTOR vpos = pos, vsize = size;
422.
423.     for (int layer = 0; layer < optnumlayers; layer++) {
424.         // Wert, den sum nach fllen dieses Layers haben sollte:
425.         REAL sumperlayer = (REAL)(layer + 1) * totalsize / optnumlayers;
426.         REAL layersum = 0;            // Inhalt dieses Layers
427.         CLayoutNode * playerfirst = pln;  // Erster Node des Layers
428.         // Layer solange fllen, bis sumperlayer erreicht ist
429.         while (pln != pstop) {
430.             sum += pln->Size();
431.             layersum += pln->Size();
432.             pln = (CLayoutNode*)pln->mnext;
433.             if (sum > sumperlayer) {
434.                 break;
435.             }
436.         }
437.         vsize[maxdim] = size[maxdim] * layersum / totalsize; // layerhhe nach
  dreisatz
438.         SquareLayoutMultiDim(playerfirst, pln, vpos, vsize);
439.         vpos[maxdim] += vsize[maxdim];    // eine ebene weiter gehen
```

```
440.     }
441.
442.     return true;
443.}
444.
445.bool CLayoutNode::RecursiveSquaredTwoDim() {
446.     if (!SquareLayoutTwoDim((CLayoutNode*)mchild, NULL, mpos, mscale))
447.         return false;
448.     for (CLayoutNode * pln = (CLayoutNode*)mchild; pln; pln = (CLayoutNode*)
   pln->mnext)
449.         if (!pln->RecursiveSquaredTwoDim())
450.             return false;
451.     return true;
452.}
453.
454.bool CLayoutNode::RecursiveLayer() {
455.     if (! LayerLayout((CLayoutNode*)mchild, NULL, mpos, mscale) )
456.         return false;
457.     for (CLayoutNode * pln = (CLayoutNode*)mchild; pln; pln = (CLayoutNode*)
   pln->mnext)
458.         if (!pln->RecursiveLayer())
459.             return false;
460.     return true;
461.}
462.
463.// Alle LayoutNodes von der Wurzel an Rekursiv auslegen
464.bool CLayoutNode::RecursiveSquarify() {
465.     if (!SquareLayoutMultiDim((CLayoutNode*)mchild, NULL, mpos, mscale))
466.         return false;
467.     for (CLayoutNode * pln = (CLayoutNode*)mchild; pln; pln = (CLayoutNode*)
   pln->mnext)
468.         if (!pln->RecursiveSquarify())
469.             return false;
470.     return true;
471.}
472.
473.// Lineares Layout rekursiv anwenden
474.bool CLayoutNode::RecursiveLayout() {
475.     if (!LayoutChildren())
476.         return false;
477.     for (CLayoutNode * pln = (CLayoutNode*)mchild; pln; pln = (CLayoutNode*)
   pln->mnext)
478.         if (!pln->RecursiveLayout())
479.             return false;
480.     return true;
481.}
482.
483.bool CLayoutNode::DrawChildren(CCamera * pcam) {
484.     for (CLayoutNode * pln = (CLayoutNode*)mchild; pln; pln = (CLayoutNode*)
   pln->mnext)
485.         if (!pln->Draw(pcam))
486.             return false;
487.     return true;
488.}
```

9.5 CDirNode.h

```
1. // GLTreeView - Visualizing directory trees in OpenGL
2. // Copyright (C) 2003-2004 by Hannes Flicka
3. //
4. // This program is free software; you can redistribute it and/or
```

```
5. // modify it under the terms of the GNU General Public License
6. // as published by the Free Software Foundation; either version 2
7. // of the License, or (at your option) any later version.
8. //
9. // This program is distributed in the hope that it will be useful,
10. // but WITHOUT ANY WARRANTY; without even the implied warranty of
11. // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12. // GNU General Public License for more details.
13. //
14. // You should have received a copy of the GNU General Public License
15. // along with this program; if not, write to the Free Software
16. // Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307,
    USA.
17.
18.
19. #ifndef _CDIRNODE_H_
20. #define _CDIRNODE_H_
21.
22. #include "common.h"
23. #include "globals.h"
24. #include "CLayoutNode.h"
25. #include "CGlSphere.h"
26.
27. typedef enum {
28.     ftunknown, ftdir, ftfile
29. } FTYPE;
30.
31. class CDirNode : public CLayoutNode {
32. public:
33.     CDirNode () :
34.             mnname(NULL),
35.             mscached(false),
36.     mftype(ftunknown) {}
37.     CDirNode(char * dname) :
38.             mnname(NULL),
39.             mscached(false),
40.     mftype(ftdir) {
41.
42.         int len = strlen(dname);
43.         mparent = NULL;
44.
45.         mnname = new char[len + 1];
46.         if (!mnname)
47.             GENERR("kein speicher");
48.         strncpy(mnname, dname, len + 1);
49.     }
50.     CDirNode(CDirNode * parent, char * sdname) :
51.             mnname(NULL),
52.             mscached(false),
53.     mftype(ftunknown) {
54.         int len = strlen(sdname);
55.
56.         mparent = parent;
57.
58.         mnname = new char[len + 1];
59.         if (!mnname)
60.             GENERR("kein speicher");
61.         strncpy(mnname, sdname, len + 1);
62.     }
63.     ~CDirNode () {
64.         if (mnname)
```

```
65.             delete [] mnname;
66.     }
67.
68.     bool RecursiveBuild() {
69.         if (mftype == ftdir)
70.             if (!Build())
71.                 return false;
72.
73.         for (CDirNode * pdn = (CDirNode*)mchild; pdn; pdn = (CDirNode*)(pdn-
  >mnext))
74.             if (!pdn->RecursiveBuild())
75.                 return false;
76.
77.         return true;
78.     }
79.
80.     bool RecursiveDestroy() {
81.         if (mchild) {
82.             CDirNode * pnext;
83.             for (CDirNode * pdn = (CDirNode*)mchild; pdn; pdn = pnext) {
84.                 if (!pdn->RecursiveDestroy())
85.                     return false;
86.                 pnext = (CDirNode*)pdn->mnext;
87.                 delete pdn;
88.             }
89.         }
90.         return true;
91.     }
92.
93.     bool Build();
94.
95.     bool GetPathStr(char * buf) {
96.         if (!mparent) {
97.             strcpy(buf, mnname);
98.         } else {
99.             ((CDirNode*)mparent)->GetPathStr(buf);
100.            strcat(buf, "\");
101.            strcat(buf, mnname);
102.        }
103.        return true;
104.    }
105.
106.    bool Draw(CCamera* pcam);
107.    bool DrawBounds(CCamera* pcam);
108.
109.    REAL Size();
110.    REAL Resize();
111.
112.    bool RecursiveDraw(CCamera * pcam);
113.    bool RecursiveDrawTwoDim(CCamera * pcam);
114.
115.    char * mnname;
116.    FTYPE mftype;
117.    INT64 mfsize;
118.    REAL msizecache;  // Grencache
119.    bool mscached;    // Gre gecached
120.    REAL mdispsize;   // Ungefhre Gre bei Anzeige
121.private:
122.};
123.
124.
```

```
125.#endif
```

9.6 CDirNode.cpp

```
1. // GLTreeView - Visualizing directory trees in OpenGL
2. // Copyright (C) 2003-2004 by Hannes Flicka
3. //
4. // This program is free software; you can redistribute it and/or
5. // modify it under the terms of the GNU General Public License
6. // as published by the Free Software Foundation; either version 2
7. // of the License, or (at your option) any later version.
8. //
9. // This program is distributed in the hope that it will be useful,
10. // but WITHOUT ANY WARRANTY; without even the implied warranty of
11. // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12. // GNU General Public License for more details.
13. //
14. // You should have received a copy of the GNU General Public License
15. // along with this program; if not, write to the Free Software
16. // Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307,
    USA.
17.
18.
19. #include "common.h"
20. #include "globals.h"
21.
22. #include "GLTools.h"
23.
24. #include "CDirNode.h"
25. #include "CDirTree.h"
26. #include "CColorTable.h"
27.
28.
29. /** Zeichnen der Umrandung, der Schattierung und des Textes zu den Dateien /
    Verzeichnissen */
30. bool CDirNode::Draw(CCamera* pcam) {
31.
32.     VECTOR center = mpos + 0.5 * mscale;
33.     REAL camdist = ~(pcam->mpos - center);
34.
35.     // Billboards zeichnen
36.     if (gsettings.Sint("drawbill", 0) && camdist < gsettings.SREAL
    ("billdist", 100.0)) {
37.         VECTOR tocam = !(pcam->mpos - center);
38.         VECTOR sky(0, 1, 0);
39.         VECTOR board_u = !(sky % tocam);
40.         VECTOR board_v = tocam % board_u;
41.
42.
43.         REAL billsize = 0.5 * ~VECTOR(mscale.x, mscale.y, 0); // * edist; //
    Vergrernde Billboards
44.         board_u *= billsize;
45.         board_v *= billsize;
46.         VECTOR board_ur = board_u + board_v;
47.         VECTOR board_ul = board_v - board_u;
48.
49.         glColor4f(mcolor.x, mcolor.y, mcolor.z, 0.1);
50.         glEnable(GL_BLEND);
51.         glEnable(GL_TEXTURE_2D);
52.         glBindTexture(GL_TEXTURE_2D, gdirtree.mtex.texID);
53.         glBegin(GL_QUADS ); /*GL_LINELOOP*/
54.
```

```
55.         glTexCoord2f(0.0, 0.0);
56.         glVertex3f(center.x - board_ur.x, center.y - board_ur.y, center.z -
  board_ur.z);
57.
58.         glTexCoord2f(0.0, 1.0);
59.         glVertex3f(center.x + board_ul.x, center.y + board_ul.y, center.z +
  board_ul.z);
60.
61.         glTexCoord2f(1.0, 1.0);
62.         glVertex3f(center.x + board_ur.x, center.y + board_ur.y, center.z +
  board_ur.z);
63.
64.         glTexCoord2f(1.0, 0.0);
65.         glVertex3f(center.x - board_ul.x, center.y - board_ul.y, center.z -
  board_ul.z);
66.
67.         glEnd();
68.         glDisable(GL_TEXTURE_2D);
69.         glDisable(GL_BLEND);
70.     }
71.
72.     // Fllung im zweidimensionalen Modus zeichnen
73.     if (gsettings.Sint("drawshdw", 0)) {
74.         glColor4f(mcolor.x, mcolor.y, mcolor.z, malpha);
75.
76.         glEnable(GL_TEXTURE_2D);
77.         glBindTexture(GL_TEXTURE_2D, gdirtree.mshadowtex.texID);
78.         glBegin(GL_QUADS); /*GL_LINELOOP*/
79.
80.         glTexCoord2f(0.0, 1.0);
81.         glVertex3f(mpos.x + mscale.x, mpos.y + mscale.y, mpos.z);
82.
83.         glTexCoord2f(1.0, 1.0);
84.         glVertex3f(mpos.x, mpos.y + mscale.y, mpos.z);
85.
86.         glTexCoord2f(1.0, 0.0);
87.         glVertex3f(mpos.x, mpos.y, mpos.z);
88.
89.         glTexCoord2f(0.0, 0.0);
90.         glVertex3f(mpos.x + mscale.x, mpos.y, mpos.z);
91.
92.         glEnd();
93.         glDisable(GL_TEXTURE_2D);
94.     }
95.
96.     // Vorderste Flche der Quader als Umriss zeichnen
97.     if (gsettings.Sint("drawfront", 0) && mdispsize > gsettings.SREAL
  ("frontsize", 0.0)) {
98.         glColor4f(mcolor.x, mcolor.y, mcolor.z, malpha);
99.         glBegin(GL_LINE_LOOP);
100.         glVertex3f(mpos.x, mpos.y, mpos.z);
101.         glVertex3f(mpos.x, mpos.y + mscale.y, mpos.z);
102.         glVertex3f(mpos.x + mscale.x, mpos.y + mscale.y, mpos.z);
103.         glVertex3f(mpos.x + mscale.x, mpos.y, mpos.z);
104.         glEnd();
105.     }
106.
107.     // Umriss der Quader zeichnen
108.     if (gsettings.Sint("drawbox", 0) && camdist < gsettings.SREAL("boxdist",
  0.0)) {
109.         glColor4f(mcolor.x, mcolor.y, mcolor.z, malpha);
110.
```

```
111.        glBegin(GL_LINE_LOOP);
112.        glVertex3f(mpos.x, mpos.y, mpos.z);
113.        glVertex3f(mpos.x, mpos.y + mscale.y, mpos.z);
114.        glVertex3f(mpos.x + mscale.x, mpos.y + mscale.y, mpos.z);
115.        glVertex3f(mpos.x + mscale.x, mpos.y, mpos.z);
116.        glEnd();
117.
118.        glBegin(GL_LINE_LOOP);
119.        glVertex3f(mpos.x, mpos.y, mpos.z + mscale.z);
120.        glVertex3f(mpos.x, mpos.y + mscale.y, mpos.z + mscale.z);
121.        glVertex3f(mpos.x + mscale.x, mpos.y + mscale.y, mpos.z + mscale.z);
122.        glVertex3f(mpos.x + mscale.x, mpos.y, mpos.z + mscale.z);
123.        glEnd();
124.
125.        glBegin(GL_LINES);
126.        glVertex3f(mpos.x, mpos.y, mpos.z);
127.        glVertex3f(mpos.x, mpos.y, mpos.z + mscale.z);
128.        glVertex3f(mpos.x, mpos.y + mscale.y, mpos.z);
129.        glVertex3f(mpos.x, mpos.y + mscale.y, mpos.z + mscale.z);
130.        glVertex3f(mpos.x + mscale.x, mpos.y + mscale.y, mpos.z);
131.        glVertex3f(mpos.x + mscale.x, mpos.y + mscale.y, mpos.z + mscale.z);
132.        glVertex3f(mpos.x + mscale.x, mpos.y, mpos.z);
133.        glVertex3f(mpos.x + mscale.x, mpos.y, mpos.z + mscale.z);
134.        glEnd();
135.
136.        glEnd();
137.    }
138.
139.    // Namen zeichnen, wenn die ausreichende Gre erreicht ist
140.    if (gsettings.Sint("drawaltname", 0) && mdispsize > gsettings.SREAL
  ("namesize", 20.0)) {
141.        glColor4f(1.0, 1.0, 1.0, 0.9);
142.
143.        int i = strlen(mnname);
144.        REAL height = mscale.x / (REAL)i * 14.0 / 16.0;
145.
146.        glEnable(GL_BLEND);
147.        // Text auf der Front des Quaders zeichnen
148.        gltPrint(
149.            mpos.x + mscale.x,       // xyz-Position des Rechtecks
150.            mpos.y + 0.5 * mscale.y + 0.5 * height,
151.            mpos.z,
152.            -mscale.x / (REAL)i,
153.            -height,
154.            mnname);
155.        glDisable(GL_BLEND);
156.    }
157.
158.    // Namen zeichnen, wenn sie nah genug bei der Kamera liegen
159.    if (gsettings.Sint("drawmidname", 0) && camdist < gsettings.SREAL
  ("namedist", 20.0)) {
160.        glColor4f(1.0, 1.0, 1.0, 0.9);
161.
162.        int i = strlen(mnname);
163.        REAL height = mscale.x / (REAL)i * 14.0 / 16.0;
164.
165.        glEnable(GL_BLEND);
166.        // Text auf der Front des Quaders zeichnen
167.        gltPrint(
168.            mpos.x + mscale.x,       // xyz-Position des Rechtecks
169.            mpos.y + 0.5 * mscale.y + 0.5 * height,
170.            mpos.z + 0.5 * mscale.z,
```

```
171.            -mscale.x / (REAL)i,
172.            -height,
173.            mnname);
174.        glDisable(GL_BLEND);
175.    }
176.
177.    return true;
178.}
179.
180./** Bounding Geometrie zeichnen */
181.bool CDirNode::DrawBounds(CCamera* pcam) {
182.    glBegin(GL_QUADS);
183.    glVertex3f(mpos.x, mpos.y, mpos.z);
184.    glVertex3f(mpos.x, mpos.y + mscale.y, mpos.z);
185.    glVertex3f(mpos.x + mscale.x, mpos.y + mscale.y, mpos.z);
186.    glVertex3f(mpos.x + mscale.x, mpos.y, mpos.z);
187.    glEnd();
188.    return true;
189.}
190.
191./** Rekursives Zeichnen fr das zweidimensionale Layout */
192.bool CDirNode::RecursiveDrawTwoDim(CCamera * pcam) {
193.    if (Size() == 0)
194.        return true;
195.
196.    int numdrew = 0;
197.    // Alle Nodes Zeichnen
198.    for (CDirNode * pdn = (CDirNode*)mchild; pdn; pdn = (CDirNode*)pdn-
  >mnext) {
199.        // Distanz berechnen
200.        VECTOR vxydist = pdn->mpos + 0.5 * pdn->mscale - pcam->mpos;
201.        vxydist.z = 0;
202.        REAL xydist = ~vxydist;
203.        // Achtung: Die z-Position der Kamera ist negativ
204.        // Soll das Objekt gezeichnet werden, oder ist es zu weit weg ?
205.        bool drawNode = (xydist < -1.0 * gsettings.SREAL("drawdist", 2.0) *
  pcam->mpos.z);
206.
207.        // ist die kamera innerhalb des Rechtecks von 'pdn'
208.        bool camInBox =
209.            (pdn->mpos.x <= pcam->mpos.x) &&
210.            (pdn->mpos.y <= pcam->mpos.y) &&
211.            (pdn->mpos.x + pdn->mscale.x >= pcam->mpos.x) &&
212.            (pdn->mpos.y + pdn->mscale.y >= pcam->mpos.y);
213.
214.        drawNode = (drawNode || camInBox);
215.
216.        // Die sehr grobe Anzahl der gezeichneten Pixel
217.        //    = Gre der Flche / Ungefhre Gre des Ansichtsausschnitts
218.        REAL approxpixels = pdn->mscale.x * pdn->mscale.y * 1024.0 / pcam-
  >mpos.z / pcam->mpos.z;
219.        //drawNode = (drawNode && (approxpixels > gsettings.SREAL("minpx",
  100.0)));
220.        pdn->mdispsize = approxpixels;
221.
222.        // Soll rekursiv weiter gezeichnet werden ?
223.        bool recurse =
224.            camInBox || (approxpixels > gsettings.SREAL("manypx", 1000.0));
225.
226.        if (drawNode && (approxpixels > gsettings.SREAL("minpx", 0.1))) {
227.            if (camInBox) {  // Dateipfad fr die Anzeige aufheben
```

```
228.                pdn->GetPathStr(gdirtree.mcurpath);
229.                gdirtree.mcurtype = pdn->mftype;
230.            }
231.
232.            if (recurse && pdn->mftype == ftdir) {
233.                pdn->RecursiveDrawTwoDim(pcam);
234.            } else {
235.                pdn->Draw(pcam);
236.            }
237.
238.            numdrew++; // Gezeichnete Nodes abzhlen
239.            if (numdrew >= gsettings.Sint("drawmaxnodes", 100))
240.                break;
241.        }
242.    }
243.
244.    return true;
245.}
246.
247./**
248. *  Rekursives Zeichnen bei dreidimensionalem Layout.
249. *  Es wird immer bis zur Tiefe 'cutoff' gezeichnet
250. */
251.bool CDirNode::RecursiveDraw(CCamera * pcam) {
252.    int cutoffdepth = gsettings.Sint("cutoff", 1);
253.    if (Size() == 0)
254.        return true;
255.    for (CDirNode * pdn = (CDirNode*)mchild; pdn; pdn = (CDirNode*)pdn-
   >mnext) {
256.        if (Depth() < cutoffdepth) {
257.            pdn->Draw(pcam);
258.            pdn->RecursiveDraw(pcam);
259.        } else if (Depth() >= cutoffdepth) {
260.            if ( ((pdn->mpos.x <= pcam->mpos.x) &&
261.                    (pdn->mpos.y <= pcam->mpos.y) &&
262.                    (pdn->mpos.z <= pcam->mpos.z)) &&
263.                    ((pdn->mpos.x + pdn->mscale.x >= pcam->mpos.x) &&
264.                     (pdn->mpos.y + pdn->mscale.y >= pcam->mpos.y) &&
265.                     (pdn->mpos.z + pdn->mscale.z >= pcam->mpos.z)) ) {
266.                pdn->GetPathStr(gdirtree.mcurpath);
267.                gdirtree.mcurtype = pdn->mftype;
268.                pdn->Draw(pcam);
269.                pdn->RecursiveDraw(pcam);
270.            }
271.        }
272.    }
273.    return true;
274.}
275.
276./** Groesse der Dateien */
277.REAL CDirNode::Size() {
278.    REAL s;
279.    if ((s = gsettings.SREAL("constantsize", 0.0)) != 0.0)
280.        return s;
281.
282.    // Ist die Gre im Cache ?
283.    if (mscached)
284.        return msizecache;
285.
286.    // Gre bestimmen
287.    if (mftype == ftfile) {
```

```
288.        if (gsettings.Sint("constantfilesize", 1))
289.            return 1.0;
290.        else
291.            return (REAL)mfsize;
292.    }
293.
294.    msizecache = gsettings.SREAL("dirsize", 0.0);
295.    for (CDirNode* pdn = (CDirNode*)mchild; pdn; pdn = (CDirNode*)pdn->mnext)
    {
296.        msizecache += pdn->Size();
297.    }
298.    mscached = true;
299.    return msizecache;
300.}
301.
302./** Loescht den Cache. Gibt neue Groesse zurueck */
303.REAL CDirNode::Resize() {
304.    REAL s;
305.    if ((s = gsettings.SREAL("constantsize", 0.0)) != 0.0)
306.        return s;
307.
308.    mscached = false;
309.
310.    if (mftype == ftfile)
311.        return 1.0;
312.
313.    msizecache = 1.0;
314.    for (CDirNode* pdn = (CDirNode*)mchild; pdn; pdn = (CDirNode*)pdn->mnext)
    {
315.        msizecache += pdn->Size();
316.    }
317.
318.    mscached = true;
319.    return msizecache;
320.}
321.
322.#include <windows.h>
323.
324./** Aufbau des Verzeichnisbaums */
325.bool CDirNode::Build() {
326.    WIN32_FIND_DATA wfd;
327.    char dir[1024];
328.    HANDLE sh;
329.
330.    if (mftype != ftdir)
331.        return false;
332.
333.    GetPathStr(dir);
334.    strcat(dir, "\*");
335.
336.    sh = FindFirstFile(dir , &wfd);
337.    if (!sh)
338.        GENERR("fehler bei suchen");
339.    do {
340.        if (wfd.cFileName[0] && strcmp(wfd.cFileName, ".") && strcmp
    (wfd.cFileName, "..")) {
341.            if (!mchild) {
342.                mchild = new CDirNode(this, wfd.cFileName);
343.                if (!mchild)
344.                    GENERR("kein speicher");
345.                if (wfd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
```

```
346.                    ((CDirNode*)mchild)->mftype = ftdir;
347.                else {
348.                    ((CDirNode*)mchild)->mftype = ftfile;
349.                    ((CDirNode*)mchild)->mfsize = (wfd.nFileSizeHigh *
  MAXDWORD) + wfd.nFileSizeLow;
350.                }
351.                ((CDirNode*)mchild)->mcolor = gcolortable.ColorByDirNode
  (((CDirNode*)mchild));
352.            } else {
353.                CDirNode * last = (CDirNode*)mchild->LevelLast();
354.                last->mnext = new CDirNode(this, wfd.cFileName);
355.                if (!last->mnext)
356.                    GENERR("kein speicher");
357.                last->mnext->mprev = last;
358.                last = (CDirNode*)last->mnext;
359.                if (wfd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
360.                    last->mftype = ftdir;
361.                else {
362.                    last->mftype = ftfile;
363.                    last->mfsize = (wfd.nFileSizeHigh * MAXDWORD) +
  wfd.nFileSizeLow;
364.                }
365.                last->mcolor = gcolortable.ColorByDirNode(last);
366.            }
367.        }
368.    } while (FindNextFile(sh, &wfd));
369.
370.    FindClose(sh);
371.
372.    return true;
373.}
```