

Institut für Medizinische Informatik
Universitätsklinikum der RWTH Aachen
Prof. Dr.rer.nat. Dipl.-Ing. Thomas M. Deserno

Visualisierung von HARAGs
Studienarbeit

Author:	Meysam Minoufekar
Matrikel nummer:	235535
Email Address:	meysam3@yahoo.com
Supervisor:	Prof. Dr.rer.nat. Dipl.-Ing. Thomas M. Deserno (geb. Lehmann)
Begin:	01. Oktober 2007
End:	01. Mai 2008

Abstract

This paper describes the design and the implementation of a visualization tool kit which is capable to load and to plot hierarchical attributed regional adjacent graphs – so called harags. The focus is on the methods used to develop the software which enclose object-oriented analysis and design techniques and general approaches of solving problems like transforming and reducing a problem to a known problem which can be solved.

Kurzfassung

Diese Studienarbeit beschreibt den Entwurf und die Implementierung einer Software-Anwendung zur Darstellung von hierarchischen attribuierten Regionen-Adjazenz-Graphen (HARAGs). Hierbei werden besonders Methoden aus der Objekt-Orientierten Softwarekonstruktion sowie Herangehensweisen aus der Informatik wie die Abstraktion, Transformation und Reduktion des Problems der HARAG-Visualisierung auf ein äquivalentes Problem, dessen Lösung schon bekannt ist, diskutiert.

Inhalt

1. Einleitung	1
2. Stand der Technik.....	3
2.1 HARAG.....	3
2.2 (Graph-)Visualisierung.....	4
3. Realisierung.....	11
3.1 Anforderungen an das System.....	11
3.2 OO-Analyse.....	12
3.3 Entwurf & Realisierung.....	13
4. Ausblick	17
4.1 Integration in das Gesamtsystem.....	17
4.2 HViz als Basis für 3D-Anwendungen	17
4.3 Erweiterung des Systems.....	18
Literatur	19

1. Einleitung

Zur graphischen Visualisierung von hierarchischen attribuierten Regionen-Adjazenz-Graphen (HARAG) bedarf es einer Software, die die Struktur eines gegebenen HARAG repräsentiert. Zurzeit existiert kein Werkzeug, das diese Aufgabe adäquat löst.

Im Rahmen des IRMA-Projektes sollte eine Software entwickelt werden, welche in der Lage ist HARAG-Definitionen aus einer Datei (z.B. einer XML-Datei) zu laden und darzustellen. Das hierdurch entwickelte Werkzeug ist als Hilfestellung zur Analyse von HARAGs gedacht und soll Beteiligten des IRMA-Projektes eine räumliche Darstellung der Elemente im HARAG und deren Beziehung zueinander bieten. Die Funktionalitäten der Software orientieren sich momentan eher an einen wissenschaftlich-technischen Anwender, können aber ohne weiteres weiterentwickelt werden, um sie den Anforderungen des medizinischen Personals anzupassen.

Bei dieser Arbeit wurden – neben den funktionalen Anforderungen – weitere Aspekte hinsichtlich Analyse und Realisierung der Software bereits während der Planung berücksichtigt und werden in dieser Ausarbeitung hervorgehoben. Diese Rahmenbedingungen beziehen sich sowohl auf die vorhandene interne Struktur des Betriebes (in dem Fall dem Institut für Medizinische Informatik des UK-Aachen) und weiteren Kriterien bedingt durch das IRMA-Projekt. Darunter fallen neben Performanz und kurzen Antwortzeiten etwa die Anforderung nach Interportabilität der geplanten Software und die Anforderung nach geeigneten und gut durchdachten Schnittstellen zur Kommunikation der Software mit anderen Komponenten des IRMA-Projektes.

Wie in den folgenden Abschnitten beschrieben, ist die präzise Schnittstellendefinition zu weiteren Komponenten einer der Hauptgründe, warum das Projekt überhaupt erfolgreich zu Ende gebracht werden konnte, trotz der Tatsache, dass leitende Personen – in dem Fall der Betreuer der Studienarbeit und Hauptverantwortlicher für die Realisierung einer allgemeinen HARAG-Definition und damit auch für das die Studienarbeit von essentieller Bedeutung – kurzfristig das Institut verließ.

Ein weiterer Aspekt, der im Folgenden diskutiert wird, ist der Aspekt der Wiederverwendung in der Softwaretechnik. Zwar ist dieses Prinzip nichts Neues und wird mit der objektorientierten Programmierung und Klassenvererbung in einem Atemzug genannt, jedoch geht dieses Prinzip über Vererbungsmechanismen der Implementierungssprache hinaus. Mit der Wiederverwendung bereits vorhandener Bibliotheken und Rahmenwerke ist ein sehr mächtiges Werkzeug der OO-Softwarekonstruktion, wodurch der Entwicklung beschleunigt wird. Außerdem ist in den meisten Fällen von Bibliotheken und Rahmenwerken zu erwarten, dass eventuelle Fehler bereits aufgefallen und behoben worden sind. Natürlich bringt die Verwendung externer Bibliotheken den Aufwand einer Einarbeitung mit sich, was vorher abgewogen werden muss.

Das in dieser Studienarbeit realisierte HARAG-Visualisierungstool (im folgenden HViz genannt) hat die Aufgabe, einen gegebenen HARAG hoher Komplexität visuell darzustellen. Das System muss die Struktur der Graphen berücksichtigen, in dem Sinne, dass unter den verschiedenen Kantentypen unterschieden wird und die Hierarchie der Knoten (Parent-Child-Beziehungen bzw. Nachbarschaftsbeziehungen) beachtet wird. Auf der z-Achse werden die verschiedenen Ebenen - stellvertretend für die verschiedenen Höhen des Graphen - aufgetragen. Knoten gleicher Höhe werden auf einer (x,y)-Ebene abgebildet. Diese können zu einander adjazent (Nachbarschaftskanten) sein. Außerdem ist jeder Knoten mit Ausnahme des Wurzelknotens adjazent zu einem Vaterknoten (Hierarchiekanten). Jeder Kantentyp wird entsprechend gekennzeichnet.

Das System ist plattformunabhängig, d.h. es ist sowohl unter Windows, als auch unter Linux und Mac OS lauffähig. Außerdem ist das System performant genug, um ein HARAG höherer Größenordnung zu verarbeiten und in vertretbarer Zeit auf den Benutzer zu reagieren. Das ist im weitesten Sinne durch die Implementierungssprache (C++) und der Design-Entscheidung für Trolltech's Qt (Version 4.2) und weiteren frei verfügbaren Frameworks (SoQt, Open Inventor und Open GL) gewährleistet. Für HViz

sind Schnittstellen zu anderen Produkten vorgesehen, die allerdings noch nicht fertiggestellt sind und nicht im Rahmen der Studienarbeit sind. Diese werden in Form eines XML-Datenformats realisiert.

Diese Studienarbeit ist folgendermaßen strukturiert: Im 2. Kapitel wird zunächst der Stand der Technik dargelegt. Es wird detailliert auf die Struktur von HARAGs eingegangen, die formal eingeführt werden. Darüberhinaus werden momentan auf dem Markt erhältliche Graph-Visualisierungstools gegenübergestellt. Es wird geklärt, warum diese Softwaretools für die Darstellung von HARAGs nicht geeignet sind, indem auf nicht-enthaltene Aspekte bei den bisherigen Visualisierungstools hingewiesen wird. Anschließend werden die in der Software HViz verwendeten Komponenten, Bibliotheken und Rahmenwerke vorgestellt und verglichen. Die erworbenen Erkenntnisse bilden dann eine Grundlage für die weiteren Überlegungen. Im 3. Kapitel wird auf die Realisierung des Software-Projekts eingegangen. Ausgangspunkt ist die Erfassung von Anforderungen (funktionale und nicht-funktionale), eine Objekt-orientierte Analyse und die Festlegung der Grenzen des Systems nach außen. Hierbei spielt die Definition von Software-Schnittstellen eine wichtige Rolle. Darauf aufbauend folgt im 4. Kapitel der Entwurf der Software-Architektur und die Beschreibung der Implementierung. Es wird auf die Algorithmen und den Datenstrukturen, die zur Lösung des Problems verwendet wurden, eingegangen. Wir werden sehen, wie sich die Darstellung von HARAGs auf ein anderes, äquivalentes Problem, nämlich die Transformation von einem HARAG in einen Szenegraphen übertragen lässt. Wir lösen dieses Problem über die Definition einer bijektiven Abbildung. Außerdem wird auf die Implementierung der bereits entworfenen Schnittstelle eingegangen. Ein weiterer Aspekt, der ebenfalls behandelt wird, ist die Benutzung existierender Bausteine. Abschließend werden im letzten und 4. Kapitel ein Ausblick und ein Resümee gegeben.

2. Stand der Technik

In diesem Kapitel wird zunächst der aktuelle Stand der Entwicklung dargelegt. Die Erkenntnisse, die hierdurch gewonnen werden, sind eine Hilfestellung, um den Sachverhalt und die Zielsetzung des Projektes zu verstehen und zu formulieren. Nach einer detaillierten Einführung von HARAGs und deren Struktur werden momentan auf dem Markt erhältliche Graph-Visualisierungstools gegenüber-gestellt. Vor Allem soll hier vermittelt werden, warum sich die kommerziellen Softwareprodukte für die konkrete Aufgabenstellung der HARAG-Visualisierung nicht eignen. Abschließend gehen wir auf verschiedene Komponenten ein, die zur Realisierung von Visualisierungsanwendungen verwendet werden.

2.1 HARAG

Einige Anwendungen aus dem Bereich der medizinischen Bildanalyse wie etwa die automatische Objekterkennung oder Objekt-basierte Bildkompression benötigen eine Segmentierung medizinischer Bildaufnahmen. In der Regel werden hierbei mehrere Objekte betrachtet, was zu einer Partitionierung des Bildes führt. Diese Partitionen, so genannte *Regions* stellen visuell wichtige Objekte auf dem der medizinischen Aufnahme dar. Die Objekte im medizinischen Bereich sind Knochen, Gefäße, Organe und anderes hartes oder weiches Gewebe. Je nach medizinischer Anforderung und Aufgabenstellung ist es nötig ein Objekt in mehreren Auflösungen aufzunehmen. So entsteht eine Sequenz von Aufnahmen, die das Objekt in seiner Gesamtheit und Bereiche des Objektes in verschiedenen Auflösungen abbildet und hierarchisch ist.

Es gibt eine Anzahl von automatischen Bildsegmentierungsverfahren, die eine Aufnahme in mehrere Regionen unterteilen. Ein Verfahren ist das in [Error! Reference source not found.] vorgestellte Bottom-up Merging Algorithmus, das als Ausgabe ein HARAG liefert, in dem die Beziehungen (Nachbarschaften und Nachfolger) gespeichert werden (siehe Abbildung 1).

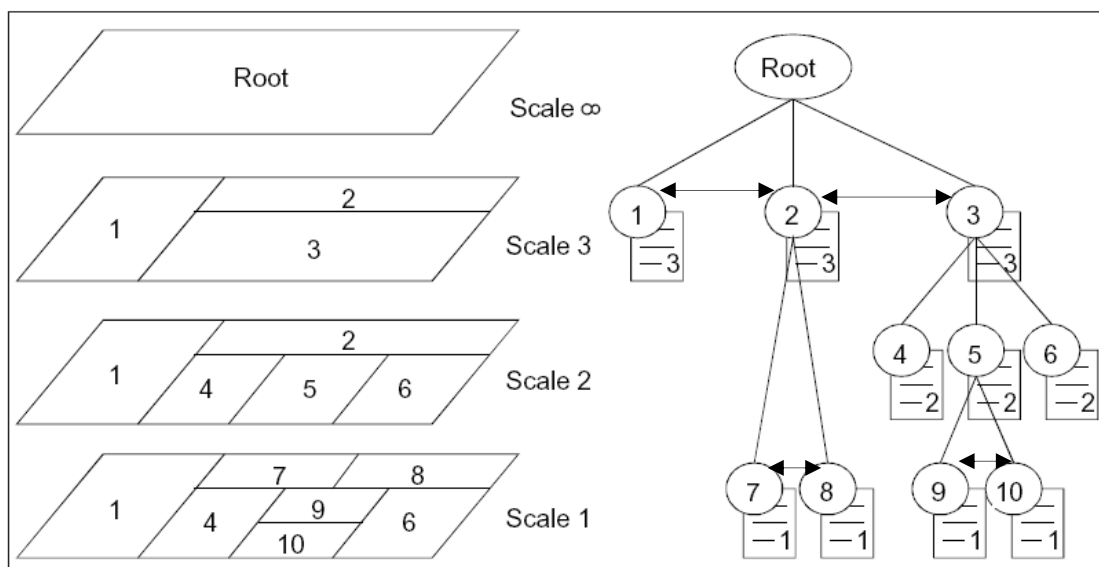


Abb. 1: Ein segmentiertes Bild und das entsprechende HARAG

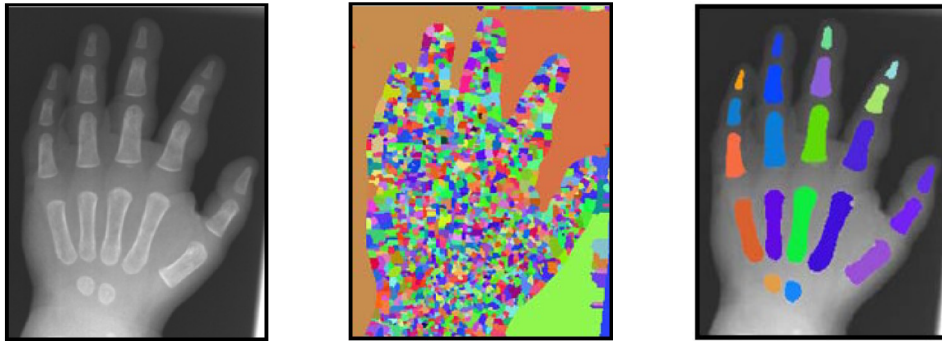


Abb. 2: Eine Aufnahme mit den entsprechenden Regionen nach der Segmentierung

Ein HARAG (hierarchical attributed region adjacency graph) ist ein Baum-ähnlicher zusammenhängender Graph, der sich für die Anwendung der Bildsegmentierung besonders gut eignet, da er sowohl die Eigenschaften eines Graphen als auch die eines Baumes vereint [2].

Definition (HARAG)

Ein HARAG $G = (V, E, T)$ ist ein Tripel, wobei V eine Menge von Knoten und $E = V \times V$ und $E' = V \times V$ Mengen von Kanten mit $|V| = n$, $|E| = m$ und $|E'| = m'$ sind. Für den durch G induzierten Baum $G' = (V, E')$ gilt zusätzlich

- (i) Zwischen je zwei Knoten von G' gibt es genau einen Weg.
- (ii) G' ist zusammenhängend und es gilt $m' = n - 1$. (Es gibt immer eine Kante weniger als Knoten)
- (iii) G' enthält keinen Zyklus und es gilt $m' = n - 1$.
- (iv) E und E' sind disjunkt: $E \cap E' = \emptyset$

2.2 (Graph-)Visualisierung

Wir haben bereits gesehen, dass HARAGs dazu benutzt werden, um eine segmentierte Struktur einer medizinischen Bildaufnahme zu reflektieren. Um diese Strukturen besser analysieren zu können, sollen Visualisierungsprogramme eingesetzt werden. Die meisten dieser Werkzeuge sind auf die Anwendungen in der Biochemie zugeschnitten und können Moleküle wie z.B. der DNS darstellen. An dieser Stelle seien zwei führende Produkte, nämlich VIDA von OpenEye und PyMOL von Delano Scientific erwähnt.

Jedoch reichten beim Testen und Ausprobieren von allen Produkten die Funktionalitäten nicht aus, um ein HARAG richtig darzustellen. Das Ergebnis der Anzeige war meist ein Gebilde, die die Baumstruktur des HARAGs ignorierte. Außerdem bieten die meisten Hersteller keine offenen APIs für das Exportieren und Importieren in ein Zielformat. Dies ist besonders wichtig, da sich die Software in das gesamte Projekt integrieren und mit anderen Programmen Daten austauschen können sollte.

Diese Überlegungen machen die Notwendigkeit nach einer auf das Problem zugeschnittenen Lösung deutlich. Um eine Struktur auf dem Rechner darzustellen, gibt es vielerlei Möglichkeiten (siehe Abbildung 2 für eine 2D-Darstellung). Und es gibt mindestens genauso viele Bibliotheken, die diese Aufgabenstellung unterstützen. Neben der zweidimensionalen Darstellung des segmentierten Bildes soll auch die Hierarchie des HARAGs im 3D-Raum dargestellt werden. Hierzu gibt es mehrere Grafik-APIs.

2.2.1 Grafik-APIs

Programmiersprachen wie C, C++ und auch Python bieten in standardmäßig keine Routinen zur Grafikverarbeitung. Möchte man eine Applikation wie z.B. ein Computerspiel oder eine Simulation programmieren, ist man auf die Nutzung spezieller Bibliotheken angewiesen. Diese APIs bilden eine Schnittstelle zur Applikation, die entwickelt werden soll und der Grafikkarte. Die Grafikkarte – vorausgesetzt, sie unterstützt die Schnittstelle einer bestimmten API – interpretiert die OpenGL-Befehle und stellt sie entsprechend auf dem Bildschirm dar. Es muss eine Entscheidung getroffen werden, welche Grafik-Schnittstelle für das jeweilige Projekt am besten geeignet ist. Es gibt eine Vielzahl solcher Bibliotheken, von denen wir vier betrachten: die Win32 GDI, SDL, OpenGL und DirectX.

	Performanz	Komfort	Flexibilität	Umfang
GDI	(-)(-)	(-)(-)	(-)(-)	(+)
SDL	(±)	(+)	(+)(+)	(+)(+)
DirectX	(+)(+)(+)	(+)	(-)(-)	(+)(+)
OpenGL	(+)(+)	(±)	(+)(+)	(-)

Abb. 3: Grafik-APIs im Vergleich

GDI

Im Microsoft-Windows-Umfeld können grafische Anwendungen mit der WIN32-API (Application Programming Interface) erstellt werden. Das ist der klassische Weg für die Entwicklung von Windows-Programmen. Innerhalb von Win32 steht mit der GDI (Graphics Device Interface) eine Bibliothek von Grafikfunktionen bereit, die über C-Befehle angesprochen werden können.

Die GDI hat für größere Projekte schwerwiegende Nachteile: sie ist langsam und umständlich. Zudem sind fortgeschrittene Techniken der Grafikprogrammierung nicht vorgesehen, so zum Beispiel Texturen für dreidimensionale Modelle.

SDL

Für 2D-Anwendungen steht mit SDL (Simple Direct Media Layer) eine freie Grafik-Bibliothek zur Verfügung, die unabhängig vom Betriebssystem ist. SDL ist in C geschrieben und kann mit C++ genutzt werden. Über den Bereich Grafik hinaus unterstützt SDL auch die Wiedergabe von Sound, Maus- und Joysticksteuerung und die Tastaturabfrage. Im Vergleich zu den Möglichkeiten der Win32/GDI-Programmierung bietet SDL einen vereinfachten und Plattform-unabhängigen Weg. Zudem ist es möglich mit der SDL eine OpenGL-Anwendung zu entwickeln, wodurch beide Bibliotheken kombiniert werden können. Für die SDL gibt es Bindungen in einer Reihe weiterer Programmiersprachen (Ada, C#, Java, Objective C, u.v.m.).

DirectX

Im Bereich moderner 3D-Anwendungen haben sich DirectX und OpenGL durchgesetzt. DirectX ist das umfassende Multimedia-Entwicklungspaket von Microsoft. Die meisten 3D-Spiele basieren auf DirectX. Der Funktionsumfang von DirectX ist größer als OpenGL und in DirectX geschriebene Anwendungen sind schneller. Jedoch ist man bei einer Entscheidung für DirectX auch an das Microsoft-Framework gebunden, was ein Plattformunabhängigkeit unmöglich macht.

OpenGL

OpenGL ist die Alternative für Plattform-übergreifende Entwicklung von 2D und 3D Programmen, da sie für alle verbreiteten Systeme erhältlich ist. OpenGL ist im Bereich von Wissenschaft und Medizin weit verbreitet. Es ist eine Bibliothek, die über reine C-Befehle angesprochen wird und nach dem State-Machine-Muster arbeitet.

Applikationen mit aufwendiger Grafik werden mit C und C++ programmiert, da diese systemnahen Sprachen klare Geschwindigkeitsvorteile bieten. Compilersprachen wie C und C++ sind immer dann schneller als Interpretersprachen wie Python, wenn in kurzer Abfolge beispielsweise Benutzer-Interaktionen auf dem Bildschirm dargestellt werden sollen und die Simulation unmittelbar auf Reaktionen des Benutzer reagieren muss. (Selbstverständlich können auch mit Java Spiele programmiert werden. Allerdings lassen sich mit Java die klassischen Grafik-APIs Win32, OpenGL und DirectX nicht direkt ansprechen. Es werden Programmbibliotheken wie Java3D von Sun oder auch Jogl - der bekannteste Java-Wrapper für OpenGL - genutzt.)

Aufgrund der Vorteile, die OpenGL mitbringt, betrachten wir diese API etwas genauer. Genormt werden nur die Grafikkommandos, Fenster- und Eingabegeräteverwaltung wird dem jeweiligen Windowssystem überlassen. Portable Programmierung ist mit Toolkit Libraries möglich (Fenster öffnen, Behandlung der Resize Events, Tastatur- und Mouseevents). Im OpenGL state (Zustand des OpenGL servers) sind Informationen wie die momentane Zeichenfarbe, eingeschaltete Lichtquellen, aktuelle Projektions- und Objekttransformationsmatrizen usw. gekapselt und können durch OpenGL Kommandos geändert werden. Die Funktionsweise von OpenGL soll anhand eines kleinen Beispiel verdeutlicht werden:

Beispiel (OpenGL)

Angenommen wir möchten ein Quadrat auf dem Bildschirm zeichnen. Dann geschieht das unter OpenGL nach folgendem Schema:

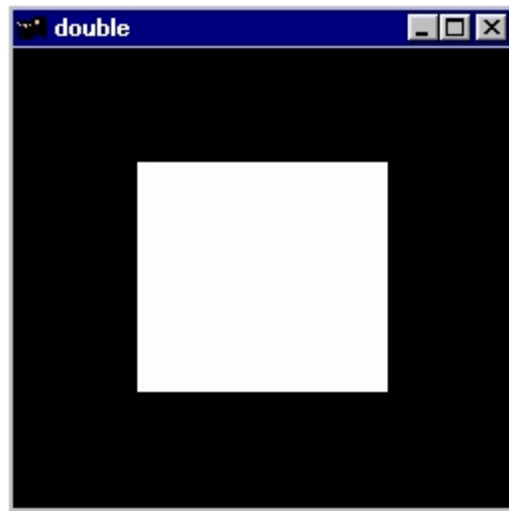


Abb. 4: Das ausführbare Programm zum Code-Fragment

```
#include <GL/glut.h>    //Glu-Toolkit anhänghen
...
int main(int argc, char **argv)
{
    /* GLUT initialisieren */
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);

    /* Fenster anlegen */
    glutInitWindowSize(400, 300);
    glutCreateWindow("Funktionsplotter");

    /* Projektionsmodus festlegen */
    glMatrixMode(GL_PROJECTION);

    /* Zeichenfunktion festlegen */
    glutDisplayFunc(plotfunc);

    /* GLUT-Hauptschleife aufrufen */
    glutMainLoop();
    return 0;
}
```

Die Zeichen-Funktion `plotfunc` kann eine beliebige Routine mit Befehlen, den sogenannten OpenGL-Primitiven sein:

```
void plotfunc()
{
    /* Fensterinhalt löschen */
    glClearColor (0.0, 0.0, 0.0, 0.0);

    /* Zeichenfarbe auf weiß setzen */
    glColor3f (1.0, 1.0, 1.0);

    /* Quadrat zeichnen */
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
    glBegin(GL_POLYGON);
        glVertex3f (0.25, 0.25, 0.0);
        glVertex3f (0.75, 0.25, 0.0);
        glVertex3f (0.75, 0.75, 0.0);
    glEnd();
}
```

```

    glVertex3f (0.25, 0.75, 0.0);
    glEnd();
    glFlush();

    /* Zeichenfunktionen ausführen */
    glFlush();
}

```

Die OpenGL-Bibliothek ist sehr umfangreich und effizient. Leider geht dabei die Benutzerfreundlichkeit auf Kosten der Effizienz verloren. Konzepte wie Objekt-Orientierung und Modularität, die aus Software-technischer Sicht wünschenswert wären, sind in OpenGL nicht realisiert. Es gibt allerdings Bibliotheken, die dort ansetzen, wo OpenGL aufhört. Eine Möglichkeit, eine Struktur und Abhängigkeiten zwischen graphischen Objekten zu erzeugen, ist über Szenegraphen realisierbar. Dieses Konzept werden wir nun kennenlernen.

2.2.2 Szenegraphen

Zur Programmierung von 3D-Grafikanwendungen gibt es, wie wir gesehen haben, komplexe Programmbibliotheken. Sie ermöglichen die Modellierung von 3D-Grafiken und das Zeichnen dieser Grafiken auf nahezu beliebigen Ausgabegeräten (Rendern). Dies zu bewerkstelligen stellt schon eine große Herausforderung dar. Aufgrund der Komplexität dieser Aufgabe werden Anwendungsprogramme praktisch

immer auf solchen speziellen 3D-Bibliotheken aufsetzen.

Weitergehende wichtige Anforderungen, die von fast jeder Anwendung an die Bibliotheken gestellt werden sind u.a.:

- Unterstützung von Interaktionen durch die Auswertung von Ereignissen des zugrundeliegende Fenstersystems (z.B. Mousebewegungen)
- intuitives grafisches Modell
- Möglichkeit für den Austausch von Modelldaten zwischen verschiedenen Programmen und Prozessen
- Unterstützung von Änderungen an den Modelldaten nach dem Rendern
- Unterstützung von Animationen

Jedoch werden diese Anforderungen von OpenGL nicht erfüllt, weshalb sich einige Hersteller sich dazu berufen fühlten, die vermissten Funktionalitäten zu implementieren und Benutzern bereitzustellen. Wir betrachten an dieser Stelle solche Programmier-Frameworks, die auf der Basis eines Szenegraphen arbeiten. Ein Szenegraph ist ein gerichteter azyklischer Graph mit nur einem Wurzelknoten, der alle Objekte, die die Szene in irgendeiner Weise beeinflussen, enthält. Für jede Szene existiert genau ein Szenegraph.

Die Szenenendatenbank kann einen oder mehrere Szenegraphen verwalten. Die Bezeichnung „Datenbank“ ist eigentlich nicht das richtige Wort, da die minimale Anforderung für Datenbanken, nämlich Daten persistent zu verwalten, bei der Szenendatenbank nicht zutrifft. Auch andere für Datenbanken übliche Konzepte gibt es im Allgemeinen nicht, wie z.B. Transaktionen. Vielmehr ist die Szenendatenbank eine komplexe Objektorientierte Datenstruktur im Hauptspeicher. Die Basis-Datenstruktur der Datenbank ist der Knoten (engl. *node*), von dem eine Vielzahl von Klassen abgeleitet ist. Ein Knoten wird als Teil des Szenegraphen von einem oder mehreren Knoten referenziert und er kann selbst einen oder mehrere Knoten referenzieren. Je nach Typ des Knotens enthält ein Knoten Informationen z.B. über die Form eines 3D-Körpers (Kugel, Quader, Kegel, etc.), das Oberflächenmaterial, Transformation, Licht oder Kamera. Auch interaktive Elemente (Manipulatoren) und sogar einfache Animationen werden mit Hilfe solcher Knoten abgebildet.

Mehrere Knoten können zu einer *Gruppe* zusammengefasst werden. Eine Gruppe ist ein spezieller Knoten, der Verweise auf einen oder mehrere andere Knoten hat. Abbildung 5 zeigt einen einfachen Szenegraphen mit sechs Knoten, zwei davon sind Gruppen.

An die Knoten sind die Typen (Klassen) der Knoten angeschrieben. Der Szenegraph wird beim Rendern wie folgt interpretiert: der Szenegraph wird von oben nach unten und von links nach rechts traversiert. Jeder Knoten wird typabhängig gerendert. Das Transformations-Objekt („Transform“) repräsentiert zum Beispiel eine geometrische Transformation und führt beim Rendern zu einer Veränderung der Transformationsmatrix des sogenannten Render-Status. Der Render-Status wird beim Rendern der in der Traversierungsreihenfolge folgenden Formen angewendet, sodass immer die zuletzt gesetzte

Eigenschaft das Rendern der Form beeinflusst. Beim Rendern werden bei einem Vorkommen eines Property-Objekts die entsprechenden Eigenschaften des Render-Status überschrieben. Das Shape-

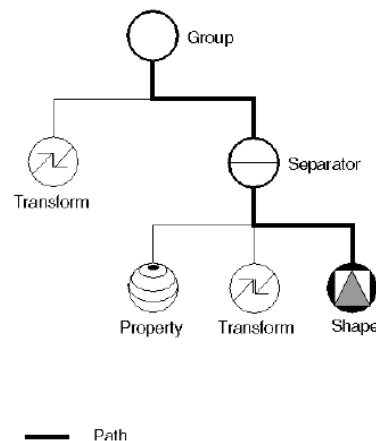


Abb. 5: Beispiel eines einfachen Szenegraphen

Objekt repräsentiert eine geometrische Form und wird beim Traversieren des Graphen unter Berücksichtigung des aktuellen Render-Status gerendert. Auch im Bereich von Szenegraphen gibt es einige Hersteller, von denen wir einen Auszug genauer betrachten.

Performer

Performer von SGI ist ein Pionierprojekt im Bereich Szenegraphen, welches in erster Linie ein Multiprozess-Transformationgraphen darstellte. Bereits in den frühen '90ern lieferte SGI Rechnersysteme mit mehreren Grafikprozessoren (bis zu 256Stück!). Performer unterstützte viele Techniken der Multiprozess-Programmierung (MP) und war sehr performant, allerdings auf Kosten der Wartbarkeit. Ein anderes Problem stellt sich beim Erlernen der API. Im Gegensatz zu Inventor, das ebenfalls von SGI entwickelt wurde, gibt es zu Performer bei Weitem nicht so eine gute Dokumentation und Einführung. Zur Zeit existiert eine freie Version unter OpenPerformer, die alle gängigen Plattformen unterstützt.

OpenSceneGraph (OpenSG)

OpenSG ist ein Open-Source Projekt mit der Zielsetzung, die Vorteile aus den bereits existierenden Szenegraph-Bibliotheken wie Inventor und Performer zu vereinen und die Nachteile zu umgehen. OpenSG ist ebenfalls multi-core-fähig und benutzt verschiedene Culling-Techniken zu Performanzsteigerung. Ausserdem ist OpenSG für alle Plattformen (MSWin, Linux, MacOS, HP UX, etc.) erhältlich. Für OpenSG spricht, dass einige renommierte Firmen (darunter Boeing) ihre Simulationen teilweise mit OpenSG realisieren.

Open Inventor (Coin)

Warum bei der Realisierung von HViz für Open Inventor entschieden wurde, lag vor Allem für die beispielsweise guten Dokumentation und Einführungsmaterialien zu Open Inventor. Die beste Bibliothek ist nutzlos, wenn es dazu keine Dokumentation gibt. Inventor wurde etwa zur selben Zeit wie Performer bei SGI entwickelt, jedoch mit einem ganz anderen Ziel. Im Vordergrund stand die Benutzerfreundlichkeit der Bibliothek vor der Performanz. Das Ergebnis war ein sehr durchdachter und hochgradig wiederverwendbarer Satz von Szenegraph-Knoten, allerdings auf Kosten der Performanz. Allerdings hat die Bibliothek eine ausreichende Leistungsfähigkeit, um die Struktur der HARAGs in Echtzeit zu verarbeiten, weshalb die Entscheidung dennoch zu Gunsten von Open Inventor getroffen wurde. Das Open Inventor Toolkit ist in C++ geschrieben und hat eine C- und eine C++-API. Open Inventor ist Open-Source für Linux und kann daher bei Bedarf an spezielle Anforderungen angepasst werden. Es existieren Versionen für IRIX, Unix, Linux und Windows. Open Inventor wurde speziell für interaktive Anwendungen entwickelt und enthält daher neben Klassen, die Grafik-Primitiven entsprechen, auch noch Tools, die vor allem die interaktive Arbeit mit 3D-Szenen erleichtern. Auf der Programmierenebene hat Inventor folgende Tools:

- Szenen-Datenbank zur Speicherung von Szenengraphen
- node kits zur Programmierung von vordefinierten Knoten-Gruppen
- Manipulatoren (handle box und trackball) für die Unterstützung von Interaktionen

- verschiedene Komponenten für das Fenstersystem (render area, material editor, directional light editor und examiner viewer).

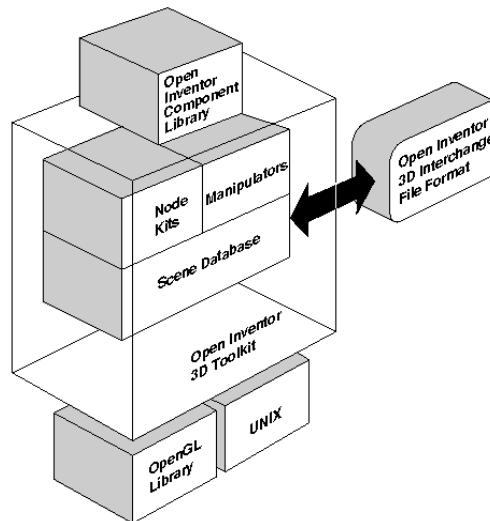


Abb. 6: Die Architektur von Open Inventor

Abbildung 6 zeigt die Architektur von Inventor. Inventor basiert auf OpenGL und Unix. Der für Anwendungsprogrammierer wohl interessanteste Teil der Architektur ist die *Szenendatenbank*, die den *Szenengraphen* verwaltet. *Node kits* sind ein fortgeschrittenes Konzept für die Strukturierung von Knoten innerhalb der Szenendatenbank, ein mit der objektorientierten Vererbung vergleichbares Konzept. *Manipulatoren* sind grafische interaktive Komponenten für die interaktive Veränderung der Szene. Sie werden mit der Szene gerendert und können mit einem Zeigegerät verändert werden, z.B. skaliert, gedreht oder verschoben werden. Inventor ist unabhängig von einem bestimmten Fenstersystem. Zum Austausch mit anderen Programmen und Prozessen existiert ein offenes Dateiformat. Inventor bildet praktisch fast alle Grafikprimitive von OpenGL irgendwie nach. Die Grafikprimitive ähneln sich sehr stark, sodaß die Performance und sonstige Eigenschaften von OpenGL so gut wie möglich ausgenutzt werden können.

Beispiel (Open Inventor)

Ein kurzes aber komplettes Programmbeispiel soll die Programmierung mit Inventor zeigen. Die einzelnen Schritte bei der Programmierung einer Szene sind normalerweise die folgenden:

- Erzeugen eines Fensters und eines Renderbereichs, in dem die Szene gerendert werden soll (hier ein Objekt vom Typ `SoQtRenderArea`)
- Konstruktion des Szenengraphen
- Rendern des Szenengraphen innerhalb des Render-Bereichs des Fensters

```
#include ... //Include-Section

main(int , char **argv)
{
    // Initialisiere Inventor. Gibt ein Fenster
    // zurück.
    Widget myWindow = SoQt::init(argv[0]);

    // Erzeuge eine Szene mit rotem Konus
    SoSeparator *root = new SoSeparator;

    //Für die Speicherverwaltung von Inventor nötig:
    //Referenzierung des Wurzelknotens
    root->ref();

    root->addChild(myMaterial);
    root->addChild(new SoCone);
}
```

```

SoQtRenderArea *myRenderArea
    = new SoQtRenderArea(myWindow);
...
myRenderArea->setSceneGraph(root);

myRenderArea->show();
}

```

Das `SoQtRenderArea` ist eine Klasse aus der SoQt-Bibliothek. SoQt bildet das Verbindungsglied zwischen Open Inventor und dem verwendeten Window-Manager. SoQt ist für Trolltechs Qt (Ver. 4.3) ausgelegt. Es gibt entsprechende Module für Windows (`SoWin`) und Unix (`SoXt`).

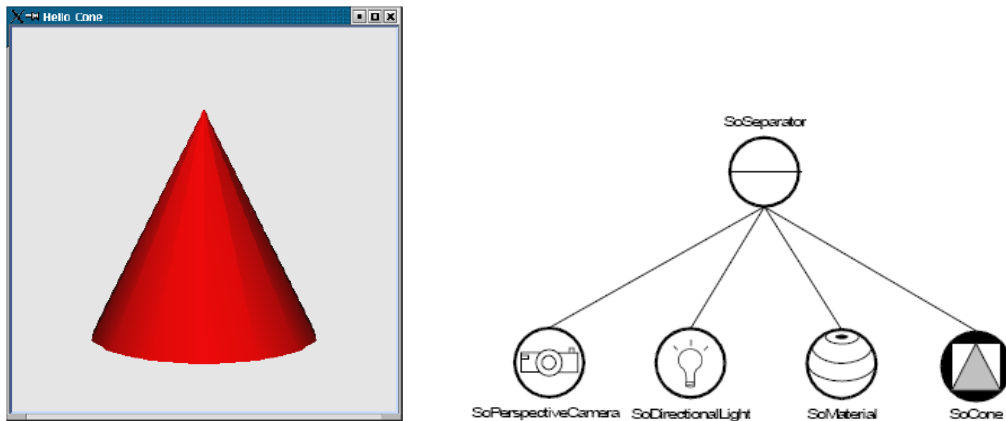


Abb.7: Das Programmbeispiel und sein Szenegraph

Nun haben wir alle Werkzeuge, um ein Applikation zur Darstellung von Strukturen wie etwa HARAGs zu entwickeln. Im nächsten Abschnitt wird ein solches Tool nach und nach entwickelt.

3. Realisierung

Nachdem wir im vorherigen Abschnitt einen Überblick über die notwendigen Werkzeuge, erhalten haben, um ein Programm zur Visualisierung einer Graphstruktur zu entwickeln, beschäftigt sich dieses Kapitel mit der konkreten Realisierung eines HARAG-Visualisierers. Allerdings wurde hierbei auf Mitteln der Objekt-orientierten Softwarekonstruktion zurückgegriffen. Anstatt direkt „loszuprogrammieren“, wurden zunächst die Anforderungen (Requirements) erfasst. Aufbauend auf eine OO-Analyse (Usecases, etc.) entstand dann ein Klassenmodell, was die Grundlage der Implementierung war. Bei dem Entwurf wurde die wichtige Rolle einer sauberen Schnittstellendefinition von Anfang an berücksichtigt.

3.1 Anforderungen an das System

Zu Beginn eines Software-Projektes – gleichgültig, ob es sich um eine Webapplikation für ein Bankkonzern oder um einen Schnittstellentreiber für eine Hardwarekomponente handelt – werden im Rahmen der Software-Qualitätssicherung die Rahmenbedingungen und die Anforderungen, die ein System erfüllen muss in einer Anforderungsspezifikation mit Anwendungsfällen (AFs) formuliert und festgehalten. Die Spezifikation dient als Grundlage für die Kommunikation mit dem Kunden, für Reviews und Blackbox-Tests. Hierdurch sollen bereits vor der Realisierung des Projektes Missverständnisse und Fehler verhindert werden.

Im Falle einer relativ kleinen Anwendung, wie es sich hierum handelt, wäre die Ausarbeitung eines kompletten Anforderungsdokuments sicherlich zu aufwendig. Daher konzentrieren wir uns auf die zentralen Elemente, den funktionalen und nicht-funktionalen Anforderungen in komprimierter Form. Zunächst gehen wir auf die nicht-funktionalen ein. Diese enthalten Rahmenbedingungen, die die Systemumgebung im Allgemeinen betreffen. Sie lassen sich wie folgt zusammenfassen:

- NF1 Minimale Antwortzeiten auf die Benutzer-Interaktion
- NF2 Plattformunabhängigkeit: Die Anwendung soll auf Windows-, Linux- und Unix Systeme lauffähig sein
- NF3 Kosten: Es sollen keine weiteren Lizenzkosten entstehen

Die funktionellen Anforderungen ergeben sich wie folgt:

- AF1 Das Laden einer HARAG-Definition
- AF2 Die Darstellung aller Knoten/ Kanten des HARAGs (im 3D-Raum)
- AF3 Die Darstellung von mit einem Knoten assoziierten Region auf Anfrage (durch Mouse-Click auf einen Knoten)
- AF4 Das Anzeigen/ Ausblenden der Kanten mit Beachtung der Kantentypen
 - a. Ein/ Ausblenden von Nachbarschaftskanten
 - b. Ein/ Ausblenden von Hierarchiekanten
- AF5 Festlegen einer Filterauswahl zum Ausblenden von Knoten und Kanten
- AF6 Benutzer-Interaktion: Rotieren, Bewegen und Zoomen des Graphen
- AF7 Exportieren der Ansicht als Bildformat

NF1 lässt sich dadurch realisieren, indem auf eine performante Compilersprache zurückgegriffen wird. Im Bereich der Visualisierung hat sich hier die Objekt-orientierte Programmiersprachen C++ etabliert. Zwar existieren für Java diverse Grafikbibliothek wie Java3D, allerdings sind implementieren sie eine API-Schicht, die die Java-Befehle an die OpenGL-Bibliothek, die in der Sprache C geschrieben ist, weiterreicht. Durch dieses Schichtenmodell geht natürlich sehr viel Performanz verloren, so dass sich

Java und Java3D aufgrund der hohen Anzahl der Elemente, die gleichzeitig gerendert werden sollen, für diese Anwendung nicht eignen. C++ hat dieses Problem nicht, da der C++-Compiler die C-Befehle (fast) genauso wie ein nativer C-Compiler umsetzt.

Allerdings hat die Verwendung einer System-nahen Sprache i.d.R. den Nachteil, dass sie nicht Plattform-unabhängig ist. Dennoch lässt sich eine Lösung im Sinne von NF2 finden, indem Komponenten verwendet werden, die auf allen Betriebssystemen erhältlich sind. Es entsteht dadurch ein geringer Aufwand, dass der Quelltext der Anwendung für eine beliebige Zielplattform kompiliert werden muss. Als Window-Toolkit, dass für die Anzeige von grafischen Bedienungselementen verantwortlich ist, wurde Trolltechs Qt in der Version 4.3 verwendet. Um 3D-Objekte zu rendern, wurde auf die Grafik-API OpenGL zurückgegriffen, der Szenegraph und die Benutzer-Interaktion mithilfe von OpenInventor (Coin3D) realisiert. Eine weitere Bibliothek – SoQt – kam ebenfalls zum Einsatz, um OpenInventor und das verwendete Window-Toolkit miteinander zu „verheiraten“. Der Einsatz dieser Bausteine hat den großen Vorteil, dass keine weiteren Lizenzkosten entstehen, da man sich OpenSource-Produkten im Rahmen der GPL (Gnu Public Licence) bedient. Demnach sind NF2 und NF3 gleichermaßen erfüllt. Die funktionalen Anforderungen werden gesondert als Anwendungsfälle betrachtet.

3.2 OO-Analyse

Die funktionalen Anforderungen beschreiben das Verhalten eines Systems. Hier werden im Kontext der Objektorientierung oft Methoden der OO-Analyse benutzt wie z.B. Use-Cases(UCs) und UC-Diagramme. Diese helfen die gesamte Anwendung zu verstehen und dienen als Kommunikationsmedium zwischen Entwickler und Kunden. Das System lässt sich mit einzelnen Anwendungsfällen als Use-Case Diagramm folgendermaßen beschreiben:

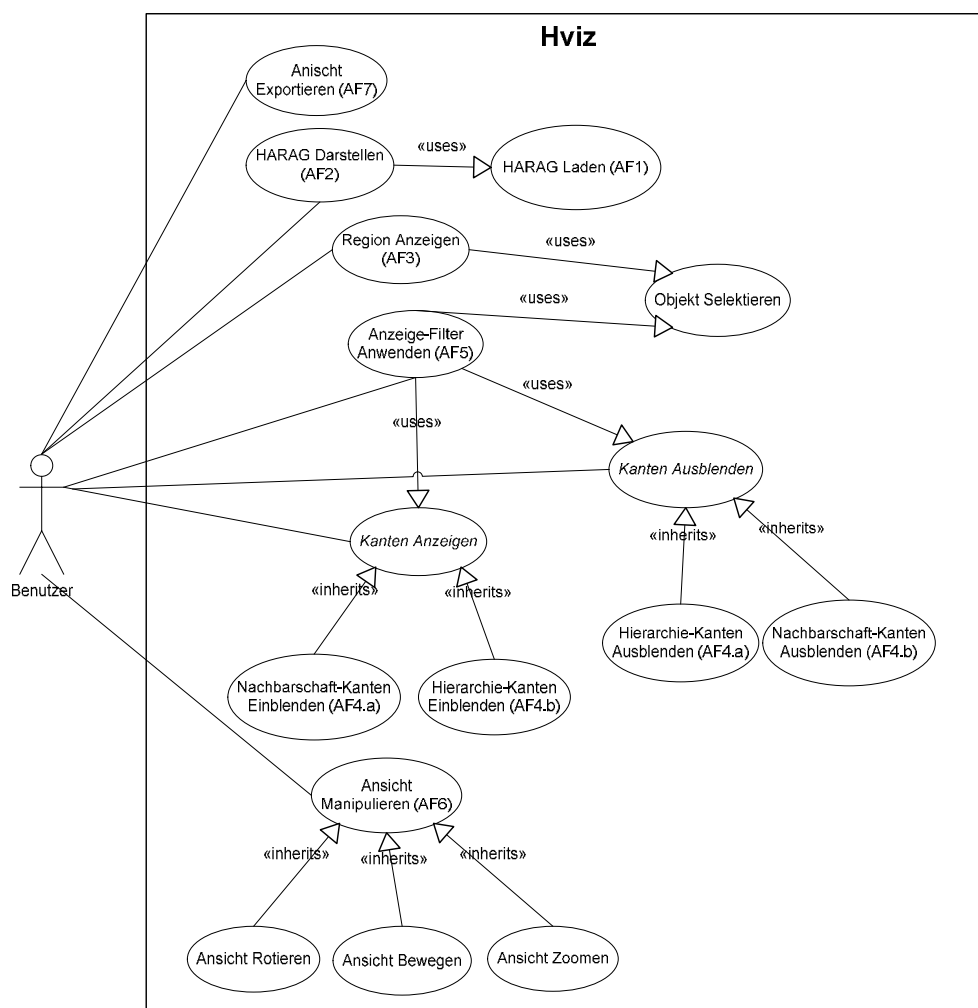


Abb. 8: Das HViz-System als Use-Case Diagramm

Das Use-Case Diagramm hilft bei der Strukturierung der Anwendungsfälle. Das Darstellen von HARAGs (AF2) setzt das Laden einer Definition (AF1) voraus. Das Anzeigen einer Region (AF3) erfolgt erst dann, wenn ein Objekt – im Speziellen ein Knoten – selektiert wurde. Analog kann ein Selektionsfilter auf Kanten und/ oder Knoten (AF5) angewandt werden, wenn sie selektiert wurden. Das Ein-/Ausblenden von Kanten (AF4) ist eine Verallgemeinerung von Ein-/Ausblenden von Nachbarschaft-Kanten (AF4.a) und Hierarchie-Kanten (AF4.b). Ebenso sind die Rotieren, Bewegen und Zoomen der Ansicht Spezialisierungen der Model View Matrix (AF6).

3.3 Entwurf & Realisierung

Ein UC-Diagramm zeigt nicht den inneren Ablauf eines Anwendungsfalls oder des Systems in ihrer Gesamtheit. In der Objektorientierung modelliert man ein System, das aus Elementen (Objekten) besteht, die durch Komponenten realisiert werden. Die Elemente haben aus einer statischen Sicht betrachtet Beziehungen zu einander, die beim Entwurf identifiziert werden. Der Prozess des Entwurfs umfasst die Architektur, die Gliederung des Systems in Komponenten und die Definition von Schnittstellen. In dem Sinne wurde die Klassenarchitektur wie in Abbildung 8 zu sehen entworfen. Die Attribute und die Operationen der jeweiligen Klasse wurden hier ausgeblendet, damit das Diagramm nicht an Übersichtlichkeit verliert.

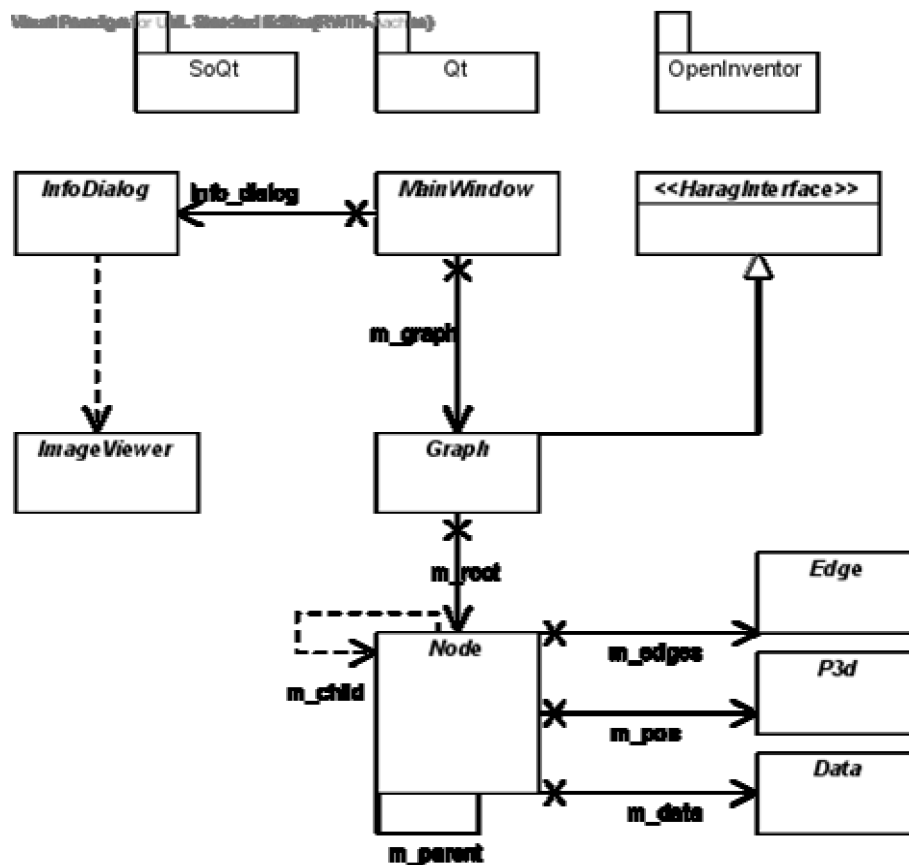


Abb.8: Das UML-Klassendiagramm von HViz

Im Diagramm bildet das zentrale Element die Klasse MainWindow. Diese Klasse verbindet die Datenansicht (View) mit dem Datenmodell (Model) und steuert die Interaktion zwischen Benutzer und Anwendung. In dem Sinne erfüllt die Applikation das MVC-Paradigma (siehe Abbildung 9) mit MainWindow als Controller. Die Benutzerinteraktionen werden an die View propagiert, die sich entsprechend aktualisiert. Die View wird durch die Komponenten von SoQt realisiert. Hierzu werden vordefinierte Operationen aus der SoQt-Bibliothek verwendet, sodass der Aufwand der Implementierung einer Benutzer-Interaktion sich als äußerst gering darstellt.

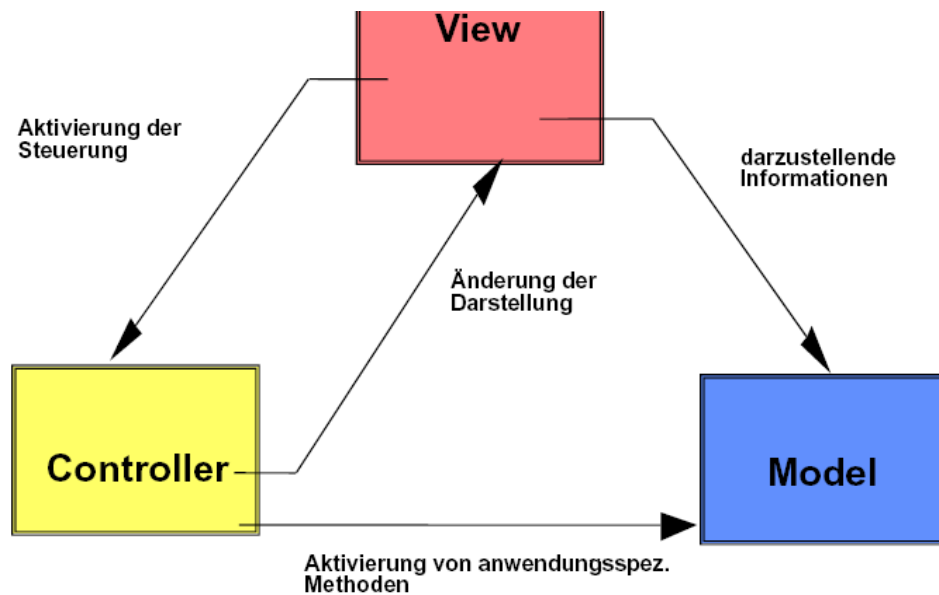


Abb.9: Der MVC-Architekturstil

Der Szenegraph aus OpenInventor und die Graph-Klasse, die einer Verallgemeinerung von HARAGs entspricht, stellen das Model dar. Das Model wird während dem gesamten Ablauf nicht verändert, sondern einmalig initialisiert. Allerdings ist die Architektur des Systems sehr flexibel gegenüber Veränderungen und Erweiterung. So ist beispielsweise eine Benutzeroperation zum Verändern der Konnektivität zwischen zwei Kanten auf der Grundlage von MVC leicht realisierbar. Der Benutzer wendet eine Aktion im Ansichtsfenster an und MainWindow benachrichtigt das Model über die Veränderung des Datenstatus, worauf Graph seinen Datenbestand aktualisiert.

Die Graph-Klasse implementiert die Schnittstelle nach außen. Diese Schnittstelle wurde sehr sorgfältig entworfen, da sie ein Bindeglied zwischen der Anwendung und externen Systemen bildet. Wir betrachten daher diese Klasse etwas genauer. Sie stellt eine rekursive Datenstruktur, ähnlich wie ein Baum dar und besitzt einen Root-Knoten aus der Klasse Node. Der Root-Knoten hat wiederum beliebig viele Nachfolger, die ebenfalls Instanzen von Node sind und in einer Listen-Datenstruktur gehalten werden. Soweit wäre die Baumstruktur eines HARAGs modelliert, jedoch werden die Eigenschaften eines mit dem HARAG induzierten Graphen ignoriert. Hierzu wird in jedem Node-Objekt eine Liste mit allen Kanten zwischen den Nachfolge-Knoten gespeichert, wobei ausgenutzt wird, dass die Nachfolgeknoten zu jeweils zwei Knoten disjunkt sind.

Wie bereits erwähnt, wird jeder Knoten mit einer Region assoziiert. Anstatt die Region im Node-Element zu halten, wurde ein Data-Objekt entworfen, das alle Informationen über die Region enthält. Das Node-Objekt enthält lediglich einen Zeiger auf das Data-Objekt. Der Grund für diese Designentscheidung ist, dass die Knoten des Graphen in den Szenegraphen geladen und auf dem Bildschirm geplottet werden müssen. Wenn man sich vorstellt, dass mehrere hundert oder sogar tausende Node-Objekte gleichzeitig angezeigt werden und die Interaktion mit dem Benutzer ruckel-frei sein soll, so muss die Größe einer Node-Instanz im Hauptspeicher möglichst gering gehalten werden. Ein Zeiger belegt weniger Bytes an Speicher-Ressourcen.

Um eine räumliche Ansicht zu erhalten muss jedem Knoten eine Raumposition zugeordnet werden. Dabei wird eine Interpretation der vorhandenen Informationen, die von einem Knoten ableitbar sind, verwendet. Ein HARAG-Knoten ist mit einem Bildsegment verbunden, das eindeutig über die Grenzen in Pixelwerte identifiziert wird (siehe Abbildung 10). Es liegt nahe als xy-Koordinaten eines Knotens den Schwerpunkt des Rechtecks zu wählen, das das Bildsegment eingrenzt. Als z-Koordinate wird die Tiefe des Knotens im HARAG gewählt. Somit sind alle Raumkoordinaten eines Knotens festgelegt. Diese Überlegungen sind die Grundlage für einen Schnittstellenentwurf zwischen HViz und ein externes System, was HARAGs erzeugt. Das Konzept, das hinter einer Schnittstellenspezifikation steht, wird weiter unten erläutert.

Zunächst soll das Darstellen eines HARAGraphen als dreidimensionale Struktur mithilfe von Szenegraphen diskutiert werden. Wie bereits erwähnt, wird als eines der wichtigsten Bausteine der Szenegraph aus OpenInventor verwendet. Ein Szenegraph ist ein azyklischer Graph und stellt eine allgemeinere Struktur als ein HARAG dar. Somit kann also das Problem der Darstellungen eines HARAGs als

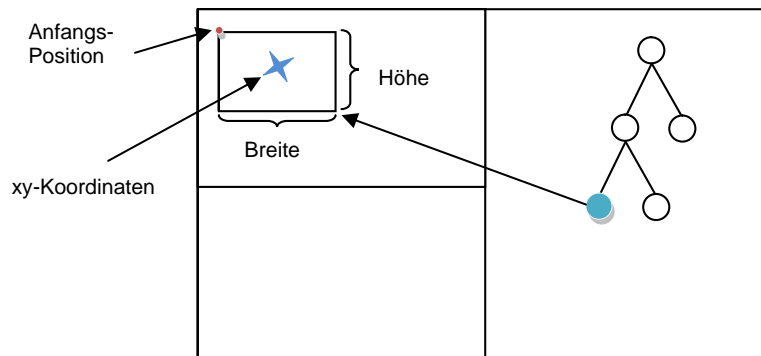


Abb. 10: Ein HARAG-Knoten und die dazugehörige Region.

ein äquivalentes Problem, nämlich der Transformation eines HARAGs in ein Szenegraph formuliert werden. Diese Reduktion geschieht mithilfe der Funktion `plot_graph()` in `HViz`, die jedes Graph-Objekt, das eine passende Schnittstellendefinition liefert, in einen Szenegraphen aus `OpenInventor` umrechnet. Dabei wird angefangen bei einem ausgezeichneten Root-Knoten rekursiv über alle Knoten und Kanten des Graphen traversiert. Knoten werden dabei in `Node`-Objekte transformiert und in die richtige Raumposition verschoben, Nachbarschaftskanten und Hierarchiekanten entsprechend zwischen zwei benachbarten `Node`-Objekten bzw. zwischen einem Vater- und seinen Kinderobjekten erstellt, verschoben und rotiert. In einer separaten Liste wird vermerkt, ob es sich um eine Nachbarkante oder einer Hierarchiekante handelt. Es musste auf diese rudimentäre Methode zurückgegriffen werden, da sie sich als einzige in diesem Setting als performant genug herausstellte. Im folgenden Code-Fragment ist die Transformation eines (HARA)Graphen in ein Szenegraph realisiert.

```
void MainWindow::plot_graph(Graph* gr)
{
    //plot the nodes of the graph gr starting with the root node
    plot_node( gr->getRoot() );
}
void MainWindow::plot_node( Node* node )
{
    //Construct a Scene Graph Node
    SoGroup *group = new SoGroup;
    SoSphere *sphere = (SoSphere*) node;
    SoTransform *xForm = new SoTransform (node->getPos());
    //Add the sphere and the transformation to the group Node
    group->addChild( xForm );
    group->addChild( sphere );
    //Add the the group Node to the top-level scene graph
    _scene->addChild( group );
    if ( node->getParent() != 0 )
        //if the node has a parent, construct a parent edge
        plot_edge(node, node->getParent());
    //Visit all the child nodes of the current node
    if(node->children_size() > 0){
        //recursive call for the child nodes
        for ( int i = 0; i < node->children_size(); i++)
            plot_node( node->getChild(i) );
        //Add all edges between the neighbors in the sub graph
        for(int i = 0; i < node->edges_size(); i++){
            Edge e = node->getEdge(i);
            plot_edge(e.getFirst(), e.getSecond());
        }
    }
}
```

Zu beachten ist, dass ein Objekt der Klasse `Graph` in einen Szenegraph transformiert wird, was auf dem ersten Blick nach einem Fehldesign aussieht. Tatsächlich wurde dieser Entwurf mit Absicht gemacht. Hierdurch werden HARAGs auf allgemeine Graphen abstrahiert. Dieser Abstraktionsschritt hat den großen Vorteil, dass beliebige Graphen – selbst solche, die keine HARAGs sind – mit HViz visualisiert werden können.

Es gibt mehrere Aspekte anhand derer ein System in Teilsysteme zerlegt werden kann. Ein sehr wichtiger Aspekt ist die Aufteilung des Gesamtsystems in Teile anhand von Zuständigkeiten. In Falle von HViz als Gesamtsystem und im Rahmen des IRMA-Projektes als Subsystem wurde darauf geachtet, dass Komponenten, die an der gleichen Aufgabe beteiligt sind, gruppiert und von denen abgegrenzt wurden, die für andere Aufgaben zuständig sind. Ein Subsystem sollte aus einer Schnittstelle, die die Funktionalitäten und die Zugänglichkeit der externen Verwendung definiert, und aus der Implementierung, die den tatsächlichen Code enthält, bestehen. Die Festlegung einer Schnittstellenspezifikation ganz zu Beginn eines Entwurfs bringt aus Software-technischer Sicht diverse Vorteile mit sich. So können Teilsysteme parallel entwickelt werden, da im Vorfeld die Zuständigkeiten und die Grenzen festgelegt wurden. Zum Einen wirken sich Änderungen nur lokal an einer Stelle – nämlich nur innerhalb des jeweiligen Systems – aus, zum Anderen bleiben die Teilelemente austauschbar. Man erhält also eine gewisse Unabhängigkeit gegenüber Veränderungen. Dieses Prinzip ist als das Open-Closed-Prinzip bekannt. Das Modul kann gefahrlos verwendet werden, da sich seine Schnittstelle nicht mehr ändert (Geschlossenheit) und weil das Modul problemlos erweitert werden kann (Offenheit). Im konkreten Fall von HViz wurde die Schnittstelle durch die abstrakte Klasse `HaragInterface`, realisiert. Diese Klasse definiert alle Funktionalitäten, die von einer konkreten Klasse implementiert werden müssen. D.h. eine konkrete HARAG-Definition muss die Schnittstellenfunktionalität erfüllen.

Ein weiterer Aspekt beim Entwurf von HViz ist die Wiederverwendung. Dies bezieht sich sowohl auf den Einsatz erprobter OO-Designprinzipien und den verwendeten Entwurfsmustern, die bereits erwähnt worden sind, als auch auf die Wiederverwendung von existierenden Open-Source Komponenten. Anstatt jede Primitive zur Anzeige von 3D-Objekten von Hand zu schreiben, wurde auf bereits existierende Rahmenwerke und Bibliotheken wie Trolltechs Qt und OpenInventor zurückgegriffen, was allerdings eine gewisse Einarbeitungszeit in die Bibliotheken erforderte. Jedoch stellte sich raus, dass die Dokumentationen zu den beiden Produkten hervorragend sind (siehe [3] und [4]), wodurch sich der Aufwand lohnte (Als Daumenregel wird gesagt, dass die Einarbeitungszeit etwa 10% der Zeit betragen sollte, die man zum Programmieren des Produkts benötigt). Abgesehen von dem erhöhten Entwurfs- und Entwicklungsaufwand, der bei einer Neuimplementierung anfallen würde, kann man bei der Verwendung von Software-Bibliotheken (meistens) davon ausgehen, dass sie (fast) keine Fehler enthalten, wodurch sich das umständliche Generieren von Testfällen und das Debuggen von Fehlern erübrigt.

4. Ausblick

Im Folgenden werden die Eingliederung der Studienarbeit in das IRMA-Projekt sowie ein Ausblick die Erweiterungsmöglichkeiten von HViz gegeben.

4.1 Integration in das Gesamtsystem

HViz ist im Rahmen der Studienarbeit soweit fertiggestellt worden, dass die Ziele und Anwendungsfälle in einer Anforderungsspezifikation die ganz zu Beginn der Studienarbeit festgelegt wurden, erreicht und realisiert wurden. Allerdings konnte die HViz als Teilsystem betrachtet nicht in der Gesamtumgebung integriert werden. Der Grund hierfür liegt darin, dass die Schnittstelle, auf die im vorherigen Abschnitt bereits eingegangen wurde, nicht fertiggestellt werden konnte. Das HARAG-Datenformat liegt derzeit in verschiedenen Versionen vor, die zu einem allgemeinen XML-Datenformat zusammengefasst werden sollen. Derzeit existiert ein Entwurf der DTD des XML-Formats, die eine Struktur des HARAGs in allgemeiner Form beschreibt. Um eine Integration zu bewerkstelligen, bedarf es eines Parser-Moduls, das in der Lage ist XML-Dateien zu laden und auf Klasseninstanzen abzubilden. Dadurch können andere Systeme wie etwa HViz das Parser-Modul benutzen und HARAG-Definitionen als XML laden. Bei der Programmierung von HViz musste darauf geachtet werden, dass die Anfertigung eines allgemeinen HARAG-Datenformats parallel zur Erstellung von HViz geschieht. Deshalb wurde HViz so konstruiert, dass eine nachträgliche Anpassung möglichst einfach durchgeführt werden kann, indem nur die Schnittstellen nach außen, d.h. die `Graph`-Klasse und `HaragInterface` angepasst werden müssen. Diese Vorgehensweise der Trennung interner Implementierung von den Elementen, die nach außen kommunizieren, ist ein wichtiger Aspekt, der bei jedem Projekt berücksichtigt werden sollte.

4.2 HViz als Basis für 3D-Anwendungen

Im gewissen Sinne stellt diese Studienarbeit eine Basis für weitere 3D-Anwendungen, auf OpenGL basieren sollen. Die Anwendung, die entwickelt wurde, kann als Vorlage für künftige Projekte verwendet werden, weil sie die wichtigsten und im Kontext der Interaktion mit einem Benutzer die am häufigsten vorkommenden Aktionen abdeckt. Diese sind Mouse-Interaktionen wie das Verändern (Bewegen, Rotieren und Zoomen) der Ansicht und das Auswählen von Objekten auf dem Bildschirm. HViz integriert diese Funktionalitäten und baut auf die Routinen in OpenInventor auf. Eine weitere Klasse von Operationen, die in HViz keine Anwendung fanden, jedoch von besonderer Wichtigkeit sind, sind sogenannte Manipulatoren, mit denen die Attribute von Objekten in einer 3D-Szene, beispielsweise die Position manipuliert werden können. Bei der Entwicklung von HViz konnte die Implementierungszeit weiterhin beschleunigt werden, weil man sich bei der Organisation der Objekte in einer 3D-Szene keine Gedanken machen musste. Die gesamte Organisation des Objekt-Datenbestands wie z.B. die Einordnung der Objekte, die Generierung von Displaylisten und Konsistenzüberprüfungen wurde an die Szenendatenbank von OpenInventor abgegeben.

Außerdem wurde anhand von HViz die Verwendung eines Szenegraphen und eines 3D-Frameworks für die Visualisierungen von Daten exemplarisch implementiert. Der Aufwand der Benutzung eines Szenegraphen macht sich mindestens dann bezahlt, wenn mehrere Objekte visualisiert werden müssen, die zu einander in Relation stehen und/ oder miteinander interagieren. In dem Kontext der HARAG-Visualisierung ist ein Szenegraph für diese Anwendung wie geschaffen. Ein beliebiger Graph kann fast eins-zu-eins in ein Szenegraph aus OpenInventor umgewandelt werden. Aber auch im Falle von einzel-

nen 3D-Objekten sind die Funktionen eines Szenegraphen sehr wichtig, wenn es etwa darum geht, einzelne Flächensegmente einer Netzstruktur auszuwählen und zu manipulieren.

4.3 Erweiterung des Systems

In dem aktuellen Status ist HViz als ein Werkzeug für Anwender und Programmierer im Rahmen des IRMA-Projektes gedacht, die über ein ausreichendes Hintergrundwissen verfügen, um die Anwendung zu bedienen und die dreidimensionale Darstellung eines HARAGs richtig zu interpretieren.

Die Anwendung steht ganz am Ende der Prozesskette der automatischen Segmentierung medizinischer Bilddaten. Demnach kommt HViz zum Einsatz, wenn die Segmentierung schon stattgefunden hat. Somit kommt der Anwendung eine Analyserolle zu, die um Analysefunktionen weiter ausgebaut werden kann. Vorstellbar wären Funktionalitäten im Kern zur interaktiven Manipulation der Graphstruktur in der 3D-Ansicht, die die Änderungen an dem HARAG abspeichern. Das HARAG hätte dann im Zusammenhang einer MVC-Architektur, wie in Abschnitt 3.3 beschrieben die Rolle des (Daten-)Modells. Dies setzt in jedem Fall eine Erweiterung der Schnittstellendefinition um Schreibroutinen voraus. Die Änderungen könnten sich auf die Struktur des Graphen wie z.B. das Ändern der Konnektivitäten zwischen benachbarten Knoten, das Ändern der Konnektivität zwischen Vater- und Kinderknoten und das Löschen von Kanten und/ oder Knoten im HARAG. Weitere Manipulationen könnten sich auf das Bearbeiten der assoziierten Regionen beziehen wie etwa das Umtauschen der Regionen zweier Knoten und das Verändern/ Bearbeiten des Inhalts der Bildinformation in der jeweiligen Region (Kontrast, Helligkeit, etc.) in einem separaten Bearbeitungsfenster.

Eine weitere Erweiterung bezieht sich auf die Ausdehnung der Zielgruppe auf das medizinische Fachpersonal. Diese Ergänzungen betreffen weniger die Anwendungsfälle im Kern, sondern sind vielmehr aus den Ableitungen der bereits existierenden Funktionalitäten. Möchte man das medizinische Personal ansprechen, so muss die Benutzerführung entsprechend erweitert und angepasst werden. Dies ist mit einem Aufwand für eine gute Benutzerergonomie, der Einführung von Sinnbildern anstatt technischer Begriffe für die spezielle Zielgruppe und das Abfangen von Benutzerfehlern verbunden.

Literatur

For literature references, a bibliography file has to be created. A reference is then entered into the Lyx document by choosing *Insert -> Citation Reference* from the menu bar. In the appearing box the bibliography label can be entered. The bibliography file has then to be inserted in the Addendum document (Addendum.lyx). Examples for a literature references are [EV99] or [Fär94].

1. Segmentation of medical images combining local, regional, global, and hierarchical distances into a bottom-up region merging scheme, Thomas M. Lehmann¹, Daniel Beier, Christian Thies, and Thomas Seidl
2. Boyle R: Image Processing, Analysis and Machine Vision. Sonka M, Hlavac V, Chapman & Hall, London, 1993.
3. C++ GUI Programmierung mit Qt 4, von von Jasmin Blanchette, Mark Summerfield, Programmer's Choice
4. The Inventor Mentor: Programming Object-Oriented 3D Graphics with OpenInventor, Release 2 (OTL), von Josie Wernecke, Inventor Architecture Group Open

This document was created with Win2PDF available at <http://www.win2pdf.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.
This page will not be added after purchasing Win2PDF.