HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

# DIPLOMARBEIT

Herr
**Ferenc Rozsa**

## Portierung und Erweiterung einer grafischen Benutzeroberfläche für das Mixed Reality Framework Avalon

2011

Fakultät Elektro- und Informationstechnik

# DIPLOMARBEIT

# Portierung und Erweiterung einer grafischen Benutzeroberfläche für das Mixed Reality Framework Avalon

Autor:
**Herr Ferenc Rozsa**

Studiengang:
**Multimediatechnik**

Seminargruppe:
**MK03w1**

Erstprüfer:
**Prof. Dr.-Ing. Klaus Müller**

Zweitprüfer:
**Dipl.-Ing. (FH) Philipp Klimant**

Einreichung:
**Mittweida, 31.01.2011**

Verteidigung/Bewertung:
**Mittweida, 2011**

Faculty Elektro- und Informationstechnik

# DIPLOM THESIS

# Portierung und Erweiterung einer grafischen Benutzeroberfläche für das Mixed Reality Framework Avalon

author:
**Mr. Ferenc Rozsa**

course of studies:
**Multimediatechnik**

seminar group:
**MK03w1**

first examiner:
**Prof. Dr.-Ing. Klaus Müller**

second examiner:
**Dipl.-Ing. (FH) Philipp Klimant**

submission:
**Mittweida, 31.01.2011**

defence/ evaluation:
**Mittweida, 2011**

**Bibliografische Angaben**

Rozsa, Ferenc: Portierung und Erweiterung einer grafischen Benutzeroberfläche für das Mixed Reality Framework Avalon. - 2011. – 99 S., Hochschule Mittweida, Fakultät Elektro- und Informationstechnik

Diplomarbeit, 2011

**Referat**

Ziel und Inhalt dieser Arbeit ist die bereits in C++ umgesetzte Steuerungsapplikation *avGo* für das Avalon-System nach Java zu portieren, um so zukünftig eine Plattformunabhängigkeit der Steuerungsapplikation zu gewährleisten. Dabei soll der bestehende Funktionsumfang von *avGo* abgedeckt und durch weitere Funktionen angereichert werden. Die Sicherstellung verschiedenster Funktionalitäten, setzt eine Manipulation bzw. Kommunikation der Steuerungsapplikation mit dem intern durch das Avalon-System abgebildeten Szenegrafen voraus. Das System bietet für diesen Service unter anderen das sogenannte External Authoring Interface (EAI) an. Um zusätzlich die Knotenstruktur der in VRML-Syntax notierten 3D-Objekte zu visualisieren, setzt die Steuerungsapplikation den VRML-Parser *pw* ein. Alle für die Arbeit der Steuerungsapplikation nötigen Einstellungen werden betriebssystemspezifisch in der Registry des Laufsystems hinterlegt. Die durch den Nutzer angelegten Sessions können in XML-konformen Dateien persistent abgespeichert werden.

# I. Inhaltsverzeichnis

# II. Abbildungsverzeichnis

# III. Tabellenverzeichnis

# IV. Abkürzungsverzeichnis

| | |
|---|---|
| AEI | Avalon External Interface |
| AEV | Additional Environment Variabeles |
| API | Application Programming Interface |
| AR | Augmented Reality |
| ASCII | American Standard Code for Information Interchange |
| AV | Augmented Virtuality |
| CAD | Computer-Aided Design |
| CAVE | Cave Automatic Virtual Enviroment |
| CLI | Common Language Infrastructure |
| EAI | External Authoring Interface |
| EDT | Event-Dispatch-Thread |
| GLUT | OpenGL Utility Toolkit |
| GUI | Graphical User Interface \| Grafische Benutzeroberfläche |
| HMD | Head Mounted Display |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| IEC | International Electronical Comission |
| IEEE | Institute of Electrical and Electronics Engineers |
| IGD | Institut für Graphische Datenverarbeitung |
| IP | Internet Protocol |
| ISO | International Standardisation Organization |
| IWP | Institut für Werkzeugmaschinen und Produktionsprozesse |

| | |
|---|---|
| JFC | Java Foundation Classes |
| JRE | Java Runtime Environment |
| JVM | Java Virtual Machine |
| LAN | Local Area Network |
| LED | Light Emitting Diode |
| MR | Mixed Reality |
| MVC | Model View Controller |
| NASA | National Aeronautics and Space Administration |
| OpenGL | Open Graphics Library (OpenGL) |
| PC | Personal Computer |
| SAV | Simple Avalon Viewer |
| SAX | Simple API for XML |
| SOAP | Simple Object Access Protocol |
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| UTF-8 | 8-Bit Unicode Transformation Format |
| VAG | VRML Architecture Group |
| VDP | Visual Decision Platform |
| VR | Virtual Reality \| Virtuelle Realität |
| VRCP | Virtual Reality Center Production Engineering |
| VRML | Virtual Reality Modelling Language |
| WWW | World Wide Web |
| X3D | Extensible 3D |
| XML | Extensible Markup Language |

ZGDV Zentrum für Graphische Datenverarbeitung

# 1. Einleitung

Das Virtual Reality Center Production Engineering (VRCP) am Institut für Werkzeugmaschinen und Produktionsprozesse (IWP) der TU-Chemnitz versteht sich als fachübergreifende Kommunikationsplattform für Forschung, Industrie und Lehre mit dem Schwerpunkt Maschinenbau und Produktionstechnik. Das VRCP bietet mit der Bereitstellung und Anwendung von Virtual Reality (VR)-Technologien seinen Partnern in Forschung, Entwicklung und Produktion verschiedenste Dienstleistungen an. Ausgestattet mit einer fünfseitigen Cave Automatic Virtual Environment (CAVE) und einer Powerwall als VR-Ausgabegeräte, lassen sich über die stereoskopische Visualisierung neben der realitätsgetreuen Simulation von Produkten, Prozessen und Anlagen, Fabrik- und Anlagenplanungen durchführen. Zum Steuern und Betreiben seines VR-Systems setzt das VRCP unter anderen das vom Fraunhofer Institut für Graphische Datenverarbeitung (IGD) übernommene und weiterentwickelte InstantReality-Framework als VR-Software ein. Im Zentrum des Frameworks steht das Avalon-System, mit seinen grafischen Frontends, dem InstantPlayer und dem Simple Avalon Viewer (SAV). Die für die virtuelle Darstellung bzw. stereoskopische Visualisierung vorgesehenen komplexen Objekte, wie auch die gesamte virtuelle Welt, sind durch eine Ansammlung von Komponenten beschrieben, die im Extensible 3D (X3D)-Standard spezifiziert sind bzw. mit ihm konform laufen. Der X3D-Standard bildet eine erweiterbare Auszeichnungssprache zur Definierung von virtuellen Welten auf Basis von XML. Das bedeutet, dass die virtuelle Welt, mit all in ihr befindlichen Objekten, durch sogenannte Knoten und deren Beziehungen untereinander, die als Kanten bezeichnet werden, über die X3D-Sytax in der entsprechenden Datei notiert ist. Das Avalon-System analysiert diese über seine grafischen Frontends geladenen Daten und bildet diese Ansammlung von Knoten und Kanten intern in eine hierarchische Datenstruktur ab, die als Szenegraph bezeichnet wird. Das Avalon-System verwendet dafür das ebenfalls vom IGD entwickelte Szenegraphensystem OpenSG. Nach dem Abbilden der Datenstruktur bzw. dem Interpretieren der in X3D-Sytax notierten Daten, erfolgt die Visualisierung der virtuellen Welt über das jeweils verwendete grafische Frontend des Avalon-Systems. Die interne Datenstruktur des Avalon-Systems ist so aufgebaut, das Daten, die der Visualisierungskonfiguration dienen und die, welche ausschließlich die darzustellende Szene betreffen, in eigene Graphen unter spezielle Namensräume abgebildet werden. Diese Namensräume besitzen wiederum eigene Knotenrepräsentationen, unter welchen die entsprechenden für den Namensraum zulässigen kontextbezogenen Knoten gruppiert werden. Über die Verwendung dieser Avalon-System spezifischen Knotenrepräsentation in der eigens erstellten X3D-Datei, lässt sich zum Beispiel die Konfiguration für die Visualisierung, neben der Szene an sich, in ein und der selben Datei vordefinieren. Zudem erlaubt das Avalon-System auch eine psychische Abgrenzung von Szene und Visualisierungskonfiguration, ohne das die für den Namensraum entsprechenden Knotenrepräsentationen verwendet werden müssen. In eine sogenannte .edv-Datei kann die Einstellung für das VR-Ausgabegerät hinterlegt werden, während in der X3D-Datei, mit dem entsprechenden Dateisuffix, die Szene notiert ist. Beide Dateien können beim Start des grafischen Frontends des Avalon-Systems über die Kommandozeile, als Parameter mit übergeben werden. Das System gruppiert nach der Analyse der Daten, die Knoten aus der .edv-Datei, wie auch aus der X3D-Datei automatisch in die

entsprechenden Namensräume. Das VRCP hat aufgrund dessen die Einstellungen für die Visualisierung in eigene .edv-Dateien hinterlegt und ausgelagert, die je nachdem, welches VR-Ausgabegerät für die Vorführung genutzt wird, entsprechend neben der Szene mit geladen werden. Durch die Implementierung von OpenSG als Szenegraphensystem eröffnet sich die Möglichkeit zur schnelleren Berechnung des Darstellungsprozesses, die Performance des hinter der CAVE stehenden Rechnerverbunds (engl.: Cluster) zu nutzen. Der am VRCP eingesetzte Cluster setzt sich aus zehn Grafik-PC's und einem Steuerungsrechner, auf dem die VR-Software läuft, zusammen, sodass die Berechnung der Bilder für die fünf Seiten der CAVE auf zehn Rechner verteilt werden kann. Der Arbeitsablauf am VRCP gestaltet sich so, dass, wie bereits erwähnt, je nachdem welches VR-Ausgabegerät für die VR-Vorführung genutzt wird, eine entsprechende Visualisierungskonfiguration in Form einer speziellen .edv-Datei vorhanden ist, welche neben der eigentlichen Szene dem InstantPlayer oder dem SAV beim Start über die Kommandozeile als Parameter übergeben wird. Die zu visualisierende Szene setzt sich aus einer Grundszenerie zusammen und den für die VR-Vorführung vorgesehenen 3D-Objekten. Die 3D-Objekte sind wiederum in eigene unabhängige X3D-Dateien ausgelagert. Der X3D-Standard spezifiziert zur Einbindung von externen Objekten den sogenannten Inline-Knoten, sodass vorab im Quellecode der Grundszenerie die entsprechenden 3D-Objekte über diese Knoten eingebunden sein müssen. Der Start der grafischen Frontends des Avalon-Systems über Kommandozeile mit der Pfadangabe der verwendeten Visualisierungskonfiguration und Grundszenerie als Parameter, sowie die Vorabeinbindung der 3D-Objekte über Inline-Knoten im Quellcode der Grundszenerie, hat sich für den Arbeitsablauf am VRCP als umständlich und unflexibel herausgestellt. Aus diesem Grund erfolgte in einer studentischen Projektarbeit die Entwicklung einer Steuerungs-Graphical User Interface (GUI) für den SAV des Avalon-Systems. Die in C++, unter Verwendung der Qt-Bibliothek, programmierte und für das Betriebssystem Linux kompilierte Steuerungs-GUI *avGo*, ist so aufgebaut, dass die verschiedenen Visualisierungskonfigurationen und Grundszenerien übersichtlich in den entsprechenden GUI-Elementen aufgelistet werden. Nach der Auswahl dieser, lässt sich der SAV direkt über die *avGo* Steuerungs-GUI starten. Die zur Visualisierung vorgesehen 3D-Objekte können bequem über einen Dateiauswahldialog ausgewählt werden. Über eine bestehenden Kommunikationsverbindung zwischen der *avGo*-GUI und dem SAV werden die 3D-Objekte direkt in die Grundszenerie eingebunden. Das Avalon-System implementiert zur Kommunikation und Manipulation der im System aufgebauten Datenstruktur bzw. Szenegraphen das Avalon External Interface (AEI). Über diese Interfaceverbindung wird das 3D-Objekt unter den Wurzelknoten des Szenegraphen über einen Inline-Knoten eingehängt. Alle geladenen 3D-Objekte werden wiederum durch entsprechende Einträge in einem Listen-Element innerhalb der *avGo*-GUI aufgeführt. Das Erzeugen und Löschen dieser Einträge im Listen-Element hat direkten Einfluss darauf, ob das durch den entsprechenden Eintrag repräsentierte 3D-Objekt Teil der Grundszenerie und damit visualisiert wird oder nicht. Der Abgleich mit der im Avalon-System abgebildeten Datenstruktur erfolgt, wie bereits erwähnt, über die Kommunikation mittels AEI-Interface. Wegen der programmiertechnischen Umsetzung der *avGo* Steuerungs-GUI mit Hilfe der Compilersprache. C++, manifestiert sich mit der Kompilierung der Applikation eine Plattformbindung. Um diesbezüglich die Applikation zukünftig flexibel einsetzen zu können, erwog das VRCP eine Portierung

der avGo Steuerungs-GUI nach Java. Unter der Verwendung der in den sogenannten Java Foundation Classes (JFC) enthaltenen SWING-Klassen, sollte die Funktionalität der bisherigen *avGo*-GUI abgedeckt und erweitert werden. Zu den zusätzlichen Funktionalitäten zählt unter anderen, die Visualisierung der Knotenhierachie, der in die Grundszenerie geladenen 3D-Objekte. Innerhalb dieser Visualisierung soll die Möglichkeit bestehen die einzelnen Knoten markieren zu können, um sich deren Eigenschaften anzeigen zu lassen. Die Portierung und Erweiterung der bestehenden *avGo*-Applikation stellt das Thema dieser Diplomarbeit dar. Das prinzipielle GUI-Layout, die interne Datenhaltung, die Kommunikation zwischen Avalon-System und externer Applikation, sowie die Funktionalitäten von *avGo*, bilden die Grundlage der neu zu programmierenden Steuerungs-GUI in Java. Die Visualisierung der hierarchischen Knotenstruktur der 3D-Objekte soll auf der Basis der Analysedaten eines VRML-Parsers erfolgen, der in das Projekt eingebunden wird. Für den Abgleich zwischen externer Applikation und Avalon-System wird auf das External Authoring Interface (EAI)-Interface zurückgegriffen, welches das Avalon-System als Softwareschicht über der AEI-Schnittstelle implementiert. Die über die Steuerungs-GUI gemachten Einstellungen, also gewählte Visualisierungskonfiguration, Grundszenerie, geladene 3D-Objekte und zusätzliche Parameter können persistent in XML-konformen Daten abgespeichert werden. Die so psychisch gesicherte Session wird mit dem Laden dieser sequentiell über den in der Java Standard Edition (JSE) bereits implementierten Simple API for XML (SAX)-Parser eingelesen, um so die für die Steuerungs-GUI entsprechenden Daten zur Initialisierung zu gewinnen. Die für das Avalon-System essentiellen Einstellungen, wie die Pfade zum InstantReality-Installationsordner, Ordner mit den Visualisierungskonfigurationen und Grundszenerien, sowie die Angaben zum Aufbau der Interfaceverbindung, werden betriebssystemspezifisch in der Registry des lokalen Computersystems hinterlegt.

# 2. Thematische Vorbetrachtungen

## 2.1 Virtual Reality - VR

Als virtuelle Realität wird eine Technologie beschrieben, die dem Nutzer computergenerierte Welten zur Verfügung stellt. Das Ziel hierbei ist, diese Welten so realitätsgetreu wie nur möglich erscheinen zu lassen. Um diesem Anspruch gerecht zu werden, müssen die Sinnesorgane des Nutzer mit den verschiedensten Technologien angesprochen werden. So wird über die stereoskopische Visualisierung und der Generierung von 3D Sound, der räumliche Eindruck beim Nutzer erzeugt. Haptisches Feedback macht das Fühlen von Konturen oder das Spüren von Vibration möglich.

In der Literatur finden sich verschiedene Definitionen über die Begrifflichkeit Virtuelle Realität (engl. Virual Reality). Nachfolgend sind zwei Definitionen aufgeführt

„Virtuelle Realität ist eine computererzeugte Simulation einer dreidimensionalen Umgebung, bei der der Anwender die Inhalte seiner Umgebung betrachten und manipulieren kann.“ (Matsuba & Roehl 1996, S. 83)

„Virtual reality is a high-end user-computer interface that involves realtime simulation and interactions through multiple sensorial channels. These sensorial modalities are visual, auditory, tactile, smell and taste.” (Burdea & Coiffet 2003, S. 3)

Aus den oben genannten Definitionen ergeben sich spezielle Merkmalsanforderungen, die erfüllt sein müssen, um von VR zu sprechen. Kennzeichnend dafür, sind die sogenannten drei I’s (Abbildung 2-1), gemeint ist damit Immersion, Interaction (deutsch: Interaktion, Wechselspiel) und Imagination (deutsch: Vorstellungskraft).

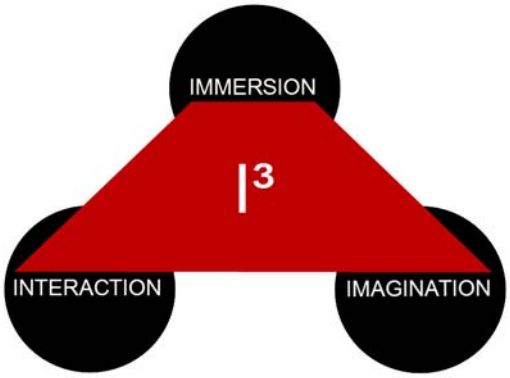


Abbildung 2-1: Die drei I's, Immersion, Interaction und Imagination[1]

Immersion beschreibt die Intensität, mit welcher der Anwender in die durch das VR-System visualisierte Szene eintauchen kann. Der Grad dieser Intensität ist davon abhängig, in wieweit die Sinne des Anwenders durch das VR-System berührt werden. Hier steht vorwiegend die visuelle Wahrnehmung des Anwenders im Vordergrund. Wie bereits erwähnt, wird mit Hilfe von stereoskopischen Verfahren die Szenerie dreidimensional visualisiert, um einen räumlichen Eindruck zu erzeugen. Mit Interaktion ist die Möglichkeit gemeint, mit VR-Interaktionsgeräten (Datenhandschuh, Zeigergerät) oder einfach mit der Fähigkeit der freien Bewegung innerhalb der virtuellen Umgebung in die Szenerie einzugreifen bzw. den Ablauf der Szenerie zu manipulieren. Die menschliche Vorstellungskraft wird durch den Begriff Imagination symbolisiert. Genauer ist damit der Sachverhalt gemeint, dass das VR-System nur als Unterstützung dienen kann, ein reales Problem zu visualisieren und somit bei der Lösung dieses Problems unterstützend mitzuwirken. Die Vorstellungskraft des Anwenders muss ebenfalls wesentlich ihren Teil dazu beitragen. Ob eine durch das VR-System visualisierte Simulation auch wirklich ihren Zweck erfüllt, hängt also maßgeblich mit von der Vorstellungskraft des Anwenders ab (vgl. Burdea & Coiffet 2003, S. 3).

In der Literatur werden neben den oben erwähnten Merkmalsanforderungen, noch weitere genannt, um der Begrifflichkeit Virtual Reality gerecht zu werden. So zum Beispiel, dass die visualisierte Simulation computergeneriert und in Echtzeit dargestellt sein muss (vgl. Matsuba & Roehl 1996, S. 83) (vgl. Hausstädler 2010, S.13). Des Weiteren sollte das visualisierte 3D Modell maßstabsgetreu dargestellt sein.

Die folgende Abbildung (Abbildung 2-2) veranschaulicht schemenhaft den Aufbau eines VR-Systems anhand eines fünf Komponenten-Modells. Zur Erklärung dieses Modells soll hierbei

[1] in Anlehnung an: Burdea & Coiffet 2003, S. 4

die am VRCP, angesiedelt an der Professur für Werkzeugmaschinen und Umformtechnik der TU-Chemnitz, eingesetzte fünfseitige CAVE dienen (Abbildung 2-3).

Unter einer CAVE versteht man einen weitestgehend abgeschlossen würfelförmigen Raum, auf dessen Wände oder begrenzende Flächen von außen die Bilder auf die durchscheinenden Projektionsflächen abgebildet werden. Die oben erwähnte CAVE der TU-Chemnitz besitzt fünf Projektionsflächen der Abmaße 3x3m. Der Nutzer, ausgestattet mit VR-Interaktionsgeräten, befindet sich quasi innerhalb der CAVE.

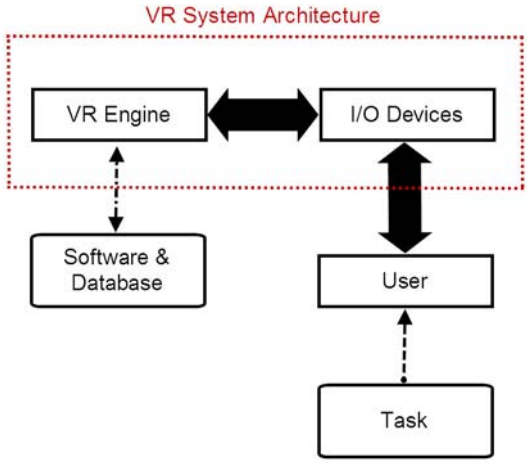


Abbildung 2-2: Die Fünf klassischen Komponenten eines VR-Systems[2]

Einen der wesentlichsten Bestandteile innerhalb des Modells machen die sogenannten I/O Devices aus. Als Input Device sei in Bezug auf die CAVE das dort eingesetzte passive optische Tracking erwähnt. Das für die fünf Seiten verwendete passive stereoskopische Projektionssystem fungiert als Output Device.

Tracking bedeutet übersetzt aus dem Englischen „verfolgen, aufzeichnen“. In der VR ist damit die Positions- und Lagebestimmung von Punkten in einem dreidimensionalen Raum gemeint. Um zum Beispiel aus der Kopfbewegung den projektionsgerechten Blickpunkt auf das 3D Objekt zu berechnen oder aus der Handposition den Ort des virtuellen Zeigers zur Bedienung zu bestimmen, müssen jeweils genaue Positionsdaten zur Verfügung stehen. Diese Daten werden mit Hilfe von einer Ansammlung von Trackern, dass auch als Trackingsystem bezeichnet wird, ermittelt.

- **Tracker: Unter einem Tracker versteht man eine spezielle Hardware, die in der VR genutzt wird, um Veränderungen in Position und Ausrichtung eines Objekts im Raum in Echtzeit zu messen (vgl. Burdea & Coiffet 2003, S. 16).**

[2] in Anlehnung an Burdea & Coiffet 2003, S. 16

Das passive optische Tracking arbeitet mit Reflektion. Das zu trackende Objekt wird mit Infrarotstrahlen beschossen, diese werden wiederum von sogenannten Markern reflektiert und entgegengesetzt der Ausstrahlungsrichtung zurückgeworfen. Aus den reflektierten Infrarotstrahlen erfolgt eine Auswertung, auf deren Grundlage dann die Ermittlung der Positionsdaten resultiert. Zur Anwendung kommt dabei ein spezieller Typ von Kamera, der in der Lage ist, Infrarotstrahlen auszuwerten, zusätzlich sind um das Objektiv mehrere Leuchtdioden – Light Emitting Diode (LED) angebracht, die für das Aussenden der Infrarotstrahlen verantwortlich sind. Derartige Kameras werden auch als Tracking-Kameras bezeichnet. Die folgende Abbildung (Abbildung 2-3) zeigt einen Blick in die CAVE. An der Decke befinden sich die fest installierten Tracking-Kameras (neongrün markiert). Diese sind so ausgerichtet, dass sie faktisch in den Bedienungsraum hineinschauen, eine derartige Konstellation wird auch als *outside-looking-in* (deutsch: von außen nach innen schauen) bezeichnet.

Abbildung 2-3: CAVE mit passivem optischen Trackingsystem[3]

Die um das Objektiv angebrachten LEDs blitzen mit der gleichen Frequenz auf, wie die Tracking-Kamera das Bild aufnimmt. Die Abgabe der Infrarotstrahlen und die Bildaufnahme sind somit deckungsgleich. Die am Objekt befestigten Marker reflektieren nun diese einfallenden Infrarotstrahlen unmittelbar. Um eine besonders gute Reflektionseigenschaft zu gewährleisten, wird für die Marker meist ein Werkstoff verwendet, der aus vielen kleinen Glaskügelchen besteht (vgl. Steger 2004, S. 39). Im Endeffekt spiegelt sich die Form des Markers aber als Kugel wieder, die aus der Nähe betrachtet eine Art glänzenden Überzug aufweist. Eine in die Tracking-Kamera integrierte Bildverarbeitungseinheit, sorgt dafür, dass die beleuchteten Marker

[3] eigene Darstellung

anschließend identifiziert und von Frame zu Frame verfolgt werden können. Mittels Triangulation erfolgt dann die eigentliche Positionsbestimmung des betreffenden Markers. In folgender Abbildung (Abbildung 2-4) ist eine einzelne Tracking-Kamera nochmals aus der Nähe dargestellt.


Abbildung 2-4: Tracking-Kamera[4]

Wie bereits erwähnt, verwendet die CAVE für die Projektion das Passiv-Stereo Verfahren. Darunter versteht man, dass der Tiefeneindruck des visualisierten Objekts dadurch erzeugt wird, dass die für das rechte Auge und linke Auge bestimmten perspektivisch leicht versetzten Bilder auf ein und die selbe Projektionswand mit unterschiedlicher Polarisation abgebildet werden. Über einen Polarisationsfilter, der vor jedem Beamerobjektiv angebracht ist, wird das Bild mit entsprechend polarisiertem Licht per Rückprojektion auf die Leinwand projiziert. Die eigentliche optische Trennung der beiden Bilder erfolgt beim Nutzer über eine sogenannte passive Polfilterbrille (Abbildung 2-5). Die verwendeten Brillengläser sind ebenfalls Polarisationsfilter, die nur das „passend“ polarisierte Licht der entsprechenden Ansicht durchlassen, sodass wiederum jedes Auge nur das für sich vorgesehene Bild erhält. Bei einer fünfseitigen CAVE sind insgesamt zehn Beamer notwendig, also pro Projektionswand zwei Beamer, die je einen Computer im Background besitzen, dessen Grafikpipeline das entsprechende Bild generiert. Rückprojektion besagt, dass sich der Standort der Beamer hinter der eigentlichen Projektionswand befindet und das Bild faktisch von hinten auf die durchscheinende Fläche projiziert wird. Die Projektionswände sind so beschaffen, dass sie polarisationserhaltend wirken, um keine Nachteile für das Passiv-Stereo Verfahren zu erleiden (vgl. Hausstädler 2010, S. 34).

[4] IWP, TU-Chemnitz

Abbildung 2-5: Passive Polfilter Brille als *Rigid-Body* mit Markern[5]

In obiger Abbildung (Abbildung 2-5) ist die schon erwähnte passive Polfilterbrille abgebildet, die zusätzlich als *Rigid-Body*, auch als Target bezeichnet, mit fünf Markern ausgestattet ist, um so über das passive optische Tracking der CAVE die Kopfbewegung des Nutzer nachzuvollziehen.

Die Computerarchitektur, die hinter der VR-Engine der CAVE steht, wird als PC-Cluster bezeichnet. Wie bereits erwähnt, wird jeder Beamer von der Grafikpipeline eines Rendering Servers angesteuert, pro Beamer ist also ein Rechner erforderlich. Bei einer fünfseitigen CAVE, mit je zwei Beamern pro Seite, ergeben sich dadurch zehn Rendering Server. Hinzu kommen noch sogenannte Control Server, die einer bestimmten Anzahl von Rendering Servern vorstehen. Bei der an der TU-Chemnitz eingesetzten CAVE, steuert genau ein Control Server alle zehn Rendering Server. Vernetzt ist die Gesamtheit der Rechner in einem Local Area Network (LAN) (Abbildung 2-6).

[5] IWP, TU-Chemnitz

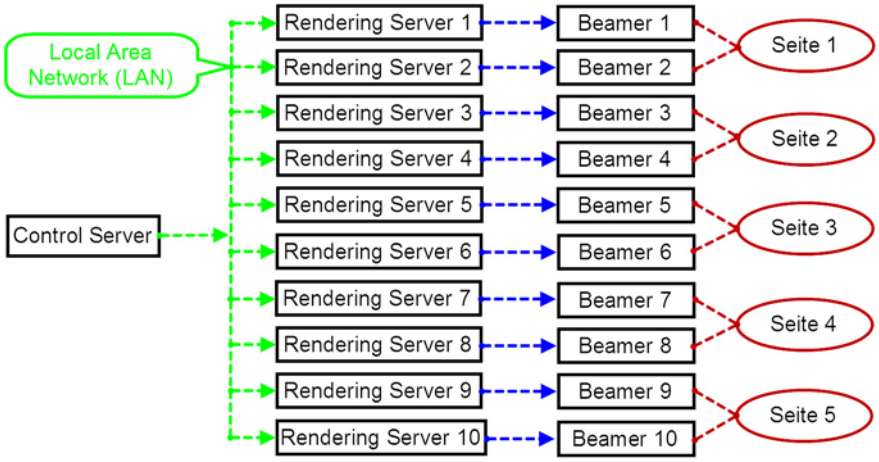


Abbildung 2-6: Vereinfacht dargestellter Aufbau eines PC-Clusters[6]

Nachfolgend Definitionen oben genutzter Begriffe:

- **VR-Engine: Die VR-Engine ist die Schlüsselkomponente in jedem VR-System. Sie liest ihre Eingabegeräte ein und führt die geforderte Echtzeitberechnung durch, um den Zustand der virtuellen Welt zu aktualisieren. Das Resultat dieser Berechnungen dient als Grundlage für die Ausgabe auf den Displays (vgl. Burdea & Coiffet 2003, S. 116).**
- **Rendering: Als Rendering wird der Konvertierungsprozess bezeichnet, 3D-Modelle, welche die virtuelle Welt ausmachen, in eine 2D Szene, die der Nutzer letztendlich über die Displays visuell wahrnimmt, umzuwandeln (vgl. Burdea & Coiffet 2003, S. 117).**
- **Grafikpipeline: Eine Technologie, bei der der Renderingprozess in funktionelle bzw. funktionsgemäße Stufen unterteilt wird. Dadurch ergibt sich die Möglichkeit, die einzelnen Stufen funktionsgemäß der entsprechenden Hardware zuzuordnen und in der Gesamtheit betrachtet, parallel zu verarbeiten.**

[6] eigene Darstellung

Das VRCP an der TU-Chemnitz nutzt für die CAVE folgende VR-Anzeige Software:

- **InstantReality**[7]
- **ICODO Visual Decision Platform (VDP)**[8]
- **Covise**[9]

Je nach Anwendungsschwerpunkt und Ausgangsmaterial entscheidet sich, welche Software zum Einsatz kommt. Sollen in VRML notierte 3D-Modelle bzw. komplette VRML-Animationen visualisiert werden, dann ist das InstantReality-Framework mit seinem InstantPlayer, aufgrund dessen kompletter Unterstützung von VRML 2.0 und X3D, die erste Wahl. Besteht das Ausgangsmaterial hingegen aus Computer-Aided Design (CAD)-Daten und soll im Verlauf der Visualisierung mit dem 3D-Modell interagiert werden, greift das VRCP zur VDP-Software. Die VDP-Software arbeitet sehr nah an den CAD-Daten und bildet diese intern auf eine eigene Datenstruktur ab. Dadurch ergeben sich weitreichende Interaktionsmöglichkeiten. So können zum Beispiel einzelne Teile des Gesamtmodells selektiert, bewegt oder ein- und ausgeblendet werden. Geht es darum Festkörpersimulationen zu visualisieren, kommt Corvise aus dem Hause VISENSO zu Anwendung.

Die Geschichte von VR ist untrennbar mit der Person Morton Heilig verbunden. Der Filmemacher, Kammermann und Erfinder entwickelte Ende der 50er Jahre das erste vollimmersive System. 1962 stellte Heilig seine multisensorische *Sensorama* vor (Abbildung 2-7). Dabei handelte es sich um eine Art Kinetoskop, welcher erstmals die normale Filmprojektion mit Ton, Vibration, Wind und sogar vorgefertigten Gerüchen kombinierte. Im gleichen Jahr präsentierte Heilig auch das erste am Kopf des Anwenders befestigte Display, damit war der Grundstein für die heutigen Head Mounted Displays (HMD) gelegt.

[7] www.instantreality.org

[8] www.icido.de/de/index.html

[9] www.visenso.de


Abbildung 2-7: *Sonsorama* - Werbeanzeige[10]

Einige Jahre später baute Ivan Sutherland, Professor für Computergrafik, dass erste computerbasierte VR-System, bekannt unter dem Namen: „Das Schwert des Damokles“. Mit dem aus einem Computermonitor, einem mechanischen Trackingsystem und einem Steuerungscomputer bestehenden System, war es dem Anwender möglich, einfache Gebilde, die vorher auf der Grundlage von Schwarz-Weiß Zeilen im Computer zusammengebaut wurden, aus verschieden Perspektive zu betrachten. In den 70er Jahren beschäftigte sich das National Aeronautics and Space Administration (NASA) Ames Research Center mit der virtuellen Realität. Im Laufe der Forschungsarbeit entstand das erste richtige VR-System. Bestehend aus einem einfachen HMD, welches über ein magnetisches Trackingsystem an einen Steuerungsrechner gebunden war, konnte nun mit Hilfe eines Datenhandschuhs (engl.: Data Glove) Objekte, die in der virtuellen Umgebung visualisiert waren, manipuliert werden. In der Folgezeit hat es etliche weitere Pioniere in der Entwicklung von VR gegeben, die den Fortschritt in diesem Bereich vorangetrieben haben. In Frankreich fand im März 1992 die erste internationale Konferenz zum Thema VR unter der Überschrift „Interfaces for the Real and Virtual Worlds“ in Montpellier statt. Im gleichen Jahr folgte eine weitere Konferenz in den USA, bei der diskutiert wurde, welches Potential VR als Unterstützendes Werkzeug für die Medizin darstellt. Im September 1993 hielt die weltweit größte Vereinigung von Ingenieuren aus dem Bereich Elektrotechnik und Informatik, das Institute of Electrical and Electronics Engineers (IEEE), seine erste Konferenz zum Thema VR ab. Mit diesem Ereignis etablierte sich VR in der Gemeinschaft der Wissenschaftler und Ingenieure (vgl. Burdea & Coiffet 2003, S. 8).

Die kommerzielle Fertigung von VR-Technologien erfolgte bereits in den späten 80er Jahren mit der Vorstellung des HMD’s *EyePhones* der Firma VPL. Im Jahr 1992 folgte von der gleichen Firma der Datenhandschuh *DataGlove* und von der Firma Nintendo der *PowerGlove*. Das erste

[10] http://www.humanproductivitylab.com/archive_blogs/2006/09/18/telepresence_defined_by_brent.php

kommerzielle VR-System führte VPL mit seinem *RB2 Model 2* ein. Mit dem Anspruch die einzelnen VR-Komponenten zu verkleinern und in eine einzige Workstation zu integrieren, veröffentliche die britische Firma Division 1991 ihre VR-Workstation *Vision*, der bald darauf die Weiterentwicklung *Provision 100* folgte. Einer der wichtigsten Nachteile der ersten kommerziellen VR-Entwicklungen war deren hoher Preis. Die von Silicon Graphics 1993 entwickelte Grafik Workstation *Reality Engine* kostete damals 100.000$. Dies führte dazu, dass der VR-Markt relativ klein war, da nur große Kooperationen, Regierungen oder gut ausgestatte Universitäten in der Lage waren, derartige Summen beim Kauf von VR-Geräten zu investieren. Mit dem enormen technischen Forstschritt im Bereich der Computerhardware erfuhr die VR Ende der 90er Jahre einen neuen Aufschwung. Durch die Entwicklung von neuen schnellen Computer- und Grafikprozessoren (Tabelle 2-1), steigerte sich die Leistungsfähigkeit der VR unterstützenden Hardware, gleichzeitig verringerte sich deren Preis. Dies alles Mündete darin, dass interaktive 3D-Grafik für jedermann erschwinglich wurde. Neue großflächige Displays, machten es möglich, dass mehrere Nutzer gleichzeitig an einer virtuellen Simulation teilhaben konnten. VR-Eingabegeräte, wie die HMDs, wandelten sich zu kleineren und dadurch handlicheren Geräten auf der einen Seite und auf der anderen Seite steigerte sich deren Auflösung. All diese Punkte führten bzw. führen dazu, dass der VR-Markt seit den 90er Jahren, enorme Wachstumsraten zu verzeichnen hat (Tabelle 2-2).

| **Rendering Geschwindigkeit (poly/sec)** | 486 PC/SPEA 7000 poly/sec | Pentium/Integraph 100.000 poly/sec | Pentium III/NVidia 31.000.000 poly/sec |
|---|---|---|---|
| **Jahr** | **1993** | **1997** | **2001** |

Tabelle 2-1: Grafikperformance unterschiedlicher Personal Computer (PC)[11]

| **Marktgröße von VR in US Dollar** | 50 Millionen | 560 Millionen | 1,4 Billionen | 3,4 Billionen |
|---|---|---|---|---|
| **Jahr** | **1993** | **1996** | **2000** | **2005** |

Tabelle 2-2: Wachstum der VR-Industrie seit 1993[12]

[11] Tabelle nach Burdea & Coiffet 2003, S. 12

[12] Tabelle nach Burdea & Coiffet 2003, S. 11

## 2.2 Mixed Reality - MR

Um die Thematik Mixed Reality (MR) erörtern zu können, müssen noch weitere Begriffe erläutert werden. Im Fordergrund stehen hier die Augmented Reality (AR) und die Augmented Virtuality (AV). Die Grundlegende Definition von Realität soll ebenfalls mit angeführt werden. Die virtuelle Realität war bereits Thema des Punktes 2.1.1, sodass hier nicht noch einmal darauf explizit eingegangen werden soll.

Realität zeichnet sich durch zwei Merkmale aus, sie ist zum einen objektiv und zum anderen messbar. Objektiv sagt in diesem Zusammenhang aus, dass die Realität völlig unabhängig von den subjektiven Eigenschaften des Menschen, wie dessen Wahrnehmung, Gefühlen und Wünschen, existent ist. Über philosophische und wissenschaftliche Betrachtungen und Erforschungen ist die Realität zugänglich und als solches sind deren Dinge messbar.

Als erweiterte Realität (engl.: Augmented Reality) wird die Anreicherung bzw. Erweiterung von realen Bildern mit digital generierten Zusatzinformationen bezeichnet. Das Ziel der AR-Technologie besteht also darin, computergenerierte Informationen in reale Videosequenzen so einzubinden, dass die realen und virtuellen Objekte in ein und derselben Welt existieren zu scheinen. (vgl. Stricker & Bockholt 2006, S. 8) Darüber hinaus stehen die erzeugten Objekte mit den realen Objekten in Beziehung, sodass eine situationsgerechte Interaktion durch den Nutzer erfolgen kann.

AR-Systeme werden durch drei Eigenschaften charakterisiert (vgl. Aschke 2007, S. 21):

- **Kombination aus realen Objekten und computergenerierten Objekten in einer realen Umgebung**
- **Echtzeitlauffähigkeit**
- **korrekte Anordnung von realen Objekten und virtuellen Objekten zueinander im dreidimensionalen Raum**

Typische Anwendungen von AR finden sich in der Medizin, zum Beispiel bei der Operationsplanung oder in der Automobilindustrie, beim Fahrzeugservice und Reparatur, wieder.

Unter erweiterter Virtualität (engl.: Augmented Virtuality) versteht man in Umkehrung zur AR, dass ein real existierendes Objekt in eine virtuelle Umgebung eingebunden wird. Ein interagieren des realen Objekts mit der computergenerierten Umgebung ist ebenfalls in Echtzeit möglich.

Um nun MR zu erläutern, bietet sich das von Prof. Paul Milgram (Universitiy of Toronto) entwickelte Reality-Virtuality-Kontinuum an (Abbildung 2-8). Es stellt eine Gerade dar, an deren Ende jeweils die reale Welt und die virtuelle Welt steht. Die Gerade selber veranschaulicht die Kombinationsmöglichkeiten zwischen einem realen Anteil und einem virtuellen Anteil. Je nachdem auf welches Ende man sich zu bewegt, entscheidet sich, in welchem Verhältnis realer Anteil zu

virtuellen Anteil steht. Diese Gewichtung von realem- und virtuellem Mengenanteil, ist auch ausschlaggebend dafür, ob man von AR oder AV spricht. Die Bandbreite an Kombinationsmöglichkeiten zwischen den beiden Anteilen wird als Mischzone bzw. MR bezeichnet.

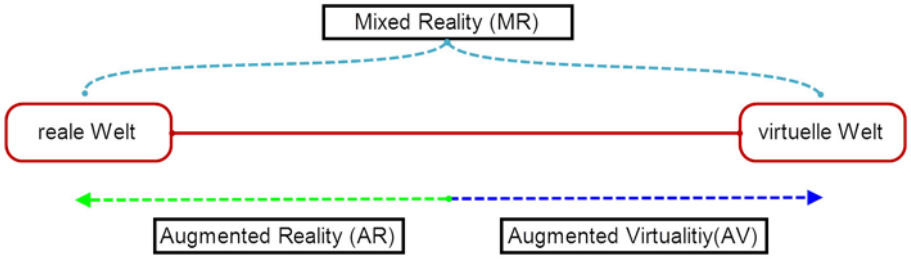


Abbildung 2-8: Reality-Virtualitiy-Kontinuum, entwickelt von Prof. Paul Milgram (Univerisity of Toronto)[13]

## 2.3 Virtual Reality Modeling Language - VRML

Die Virtual Reality Modeling Language (VRML) ist eine Auszeichnungssprache ähnlich der Hypertext Markup Language (HTML), nur das in dem Fall nicht 2D-Inhalte sondern 3D-Inhalte beschrieben werden. HTML-Dokumente bilden die Grundlage für das World Wide Web (WWW) und werden durch einen Webbrowser angezeigt. VRML wurde im Wesentlichen für das Internet konzipiert, die Interpretation, Ausführung und Präsentation des VRML-Dokuments übernimmt hier der VRML-Browser. Der VRML-Browser ist ein Plug-in, welches vom Internet-Browser aufgerufen wird, um auf Webseiten eingebettete 3D-Inhalte anzuzeigen (vgl. Matsuba & Roehl 1996, S. 78).

VRML bietet die Möglichkeit, über Knoten und Felder, die Geometrie und Positionierung von 3D-Objekten zu beschreiben. Ebenfalls lässt sich das Aussehen der Objekte oder andere Einstellungen, wie die Belichtung, festlegen. Die Interaktivität wird über sogenannte Sensoren eingebunden.

Nachteilig ist die Offenheit von VRML, einmal im Internet publizierte VRML-Dateien lassen sich unkompliziert nach vorherigem Download auf den eigenen Rechner einsehen und verändern.

VRML-Dateien besitzen den Dateisuffix .wrl („world“ oder „wörml“) bzw. .gz (wenn das VRML-Dokument in gepackter Form vorliegt) und sind im Grunde genommen einfacher Text, der in American Standard Code for Information Interchange (ASCII) oder 8-Bit Unicode Transformation Format (UTF-8) kodiert ist. Mit jedem Texteditor lassen sich somit VRML-Dateien erstellen, einsehen und editieren.

[13] eigene Darstellung

VRML, damals noch unter der Bezeichnung Virtual Reality Markup Language, wurde ab 1994 auf der Basis des von der Firma Silicon Graphics (SGI) entworfenen ASCII-Dateiformats Open Inventor entwickelt. Im Verlauf der Entwicklung wurde das Wort „Markup“ in „Modeling“ umbenannt und schließlich im Mai 1995 von der International Standardisation Organization (ISO) VRML 1.0 spezifiziert[14] und im Januar 1996 eine fehlerbereinigte Version, sogenannte „clerified Version“, der VRML 1.0 Spezifikation veröffentlicht.

Schnell wurden jedoch die Grenzen des doch sehr statischen VRML 1.0 offensichtlich und so begannen Ende 1995 unter anderem Firmen, wie SGI und Microsoft an einem Ersatz zu arbeiten, um das Defizit im Bereich Interaktion und Verhalten zu beseitigen. Um ein Zersplittern der VRML 1.0 Spezifikation zu verhindern, beschloss die VRML Architecture Group (VAG), die sich 1995 aus den Spitzen der VRML-Gemeinde gebildet hatte, die neu entwickelten Erweiterungen zur Abstimmung der VRML-Gemeinde zu unterbreiten. In dem sechswöchigen Abstimmungsprozess wurde schließlich die von SGI entwickelte Erweiterung *Moving Worlds* als Basis für VRML 2.0 ausgewählt. Mit einigen Ergänzungen und Berichtigungen konnte so VRML 2.0 auf der Siggraph96 Konferenz präsentiert werden.

Im Bestreben der VAG, VRML 2.0 in einen internationalen Standard umzuwandeln, wurde in Zusammenarbeit mit der ISO und der International Electronical Comission (IEC) im Dezember 1997 der ISO/IEC-14772-1:1997 Standard[15], auch bekannt als VRML97, genehmigt.

Auf Grund dieser Aktion kommt es immer wieder zu Missverständnissen in der Begrifflichkeit VRML97 und VRML 2.0. Beide Spezifikationen weisen aber keinen Unterschied in der Funktionalität auf. Die Unterschiede beschränken sich rein auf das Spezifikationsdokument des VRML97 Standards. So mussten Änderungen in der Formulierung und Layout vorgenommen werden, um den Anforderungen der ISO zu genügen, sodass VRML97 die Internationalisierung der VRML 2.0 Spezifikation darstellt.

Der komplette Titel des ISO/IEC-14772-1:1997 Standards lautet: „Information technology -- Computer graphics and image processing -- The Virtual Reality Modeling Language (VRML) -- Part 1:Fuctional specifications and UTF-8 encoding“. Im Jahre 2004 wurde der ISO Standard noch erweitert. Die Ergänzung wurde dem bestehendem ISO Standard ISO/IEC-14772-1:1997 noch mit hinzugefügt, unter der Bezeichnung: ISO/IEC-14772-2:2004[16]. Der komplette Titel lautet: „Information technology -- Computer graphics and image processing -- The Virtual Reality Language (VRML) -- Part2: External authoring interface (EAI)”. Der zweite Teil definiert

[14] VRML 1.0 Spezifikation: http://www.web3d.org/x3d/specifications/vrml/VRML1.0/index.html

[15] VRML 97: http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/

[16] VRML 97: http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/

das EAI-Interface, welches externe Programme benutzen können, um Objekte bzw. den Szenegrafen der Szenerie, die im VRML-Browser oder Player dargestellt wird, zu manipulieren.

Bei vielen Modellierungsprogrammen fungiert heute das VRML-Format neben den eigenen proprietären Formaten als In- und Exportformat für die erstellten 3D-Objekte. Ebenfalls ist jede VR-Anzeigesoftware in der Lage, VRML-Dateien zu laden. Bei einem kommerziellen und regelmäßigen Einsatz von VR, ist eine parallele Datenhaltung in verschieden Formaten nicht anzuraten. Dort, wo 3D-Daten zwischen Kunden bzw. verschieden Abteilungen innerhalb der Firma ausgetauscht werden, sollte man sich auf ein Format verständigen. VRML als quasi kleinster gemeinsamer Nenner in Sachen Austauschformat zwischen den verschieden Modellierungsprogrammen und VR-Anzeigeprogrammen sollte deshalb mit zur ersten Wahl gehören (vgl. Hausstädtler 2010, S. 118).

## 2.4 Avalon-System

Als Avalon wird das nunmehr vom Fraunhofer Institut für Graphische Datenverarbeitung (IGD) weiterentwickelte System bezeichnet, dass eine Entwicklungs-, Demonstrations- und Präsentationsumgebung für die verschiedensten in VRML und X3D notierten Anwendungen im VR/AR Bereich darstellt (vgl. Ziezold 2006, S. 8). Das Gesamtsystem ist programmiertechnisch in C++ umgesetzt und benutzt zur Datenhaltung der Szenenhierarchie, das quelloffene Szenegrafen-System OpenSG. Als Frontend implementiert das System einen Player, der zur Visualisierung der MR-Szenen dient. Darüber hinaus bietet das System externe Schnittstellen an, um zur Laufzeit Manipulationen an Teilen der abgebildeten Szenenhierarchie oder Komponentenparametern, vornehmen zu können.

Die Entwicklung von Avalon begann 1997 ursprünglich am Zentrum für Graphische Datenverarbeitung (ZGDV) in Darmstadt. In dieser Phase erfolgte die programmiertechnische Umsetzung der Visualisierung von 3D Modellen noch über die direkte Benutzung des Open Graphics Library (OpenGL) Application Programming Interface (API). In einem gemeinsamen Projekt des ZGDV mit dem IGD erfolgte später die Entwicklung von OpenSG. Auf Grund des immensen programmiertechnischen Aufwands, den die Umsetzung des Avalon Projekts unter der direkten Benutzung von OpenGL bereitete, ging man dazu über, eine Portierung nach OpenSG vorzunehmen. Im Verlauf der Weiterentwicklung von Avalon wurde das gesamte Projekt von der IGD übernommen.

Die Institute der Fraunhofer-Gesellschaft bekommen nur ein 1/3 ihres finanziellen Budgets vom Staat bezuschusst. Die restlichen 2/3 müssen durch die Vermarktung der eigenen Projekte bzw. deren Ergebnisse erwirtschaftet werden, sodass neben der Forschung auch immer mit die kommerzielle Komponente eine wichtige Rolle spielt.

Auf Grund dieser Tatsache hat das IGD das ganze System in InstantReality umbenannt und publiziert es seitdem unter einer eigens erstellten Homepage[17]. Für den nichtkommerziellen Gebrauch bzw. zu Testzwecken, kann das nun auf ein Framework angewachsene System frei heruntergeladen werden. Der kommerzielle Einsatz setzt das Vorhandensein einer individuell erstellten Lizenz durch die entsprechende Kontaktperson auf Seiten der IGD voraus.

Mit zunehmenden Erfolg des Avalon unter dem Vermarktungslabel InstantReality, begann das IGD auch weitere eigene Softwareentwicklungen unter der schon vorhandenen Marke zu vertreiben. Sodass mit der Zeit ein ganzes Framework unter der Begrifflichkeit InstantReality entstand, dass neben dem schon erwähnten Avalon, noch das Gerätemanagement InstantIO, den OpenSG basierten Renderer Server InstantCluster und das Kamera Tracking und Echtzeitbildverarbeitungsprogramm InstantVision beinhaltet.

### 2.4.1 InstantPlayer

Der im InstantReality-Framework eingebundene Player, liegt den verschiedenen plattformabhängigen Paketen in unterschiedlichen Varianten bei. Dabei macht aber nur das verwendete GUI-API die Differenz zwischen den Playern aus. Der InstantPlayer (Abbildung 2-9) verwendet jeweils die native Betriebssystembibliothek, während die ebenfalls dem Paket beigefügte Kommandozeilenversion des Players, der SAV (Abbildung 2-9) auf dem OpenGL Utility Toolkit (GLUT) basiert. Der InstantPlayer und auch der SAV bilden die programmiertechnische Umsetzung, Mixed Reality Szenen, die auf der Syntax von VRML oder X3D basieren, darzustellen. Konkret handelt es sich bei beiden, um eine Implementierung, des im Punkt 2.1.4 erwähnten Avalon-Systems.

[17] http://www.instantreality.org/


Abbildung 2-9: Simple Avalon Viewer (SAV) (links im Bild) | InstantPlayer (rechts im Bild)[18]

Zur Laufzeit wird die zu visualisierende Szene auf eine intern generierte Szenenhierarchie, einem Szenegrafen, abgebildet. Das Avalon-System ist dabei in der Lage, verschiedene Komponenten über die Einführung und Zuordnung zu speziellen Namensräumen, gegenseitig abzugrenzen. Beginnend mit der Generierung eines sogenannten Laufzeit-Kontextes erfolgt weiter die Erstellung eines *scene*, *engine* und *contextsetup* Namensraums. Im *scene* Namensraum wird der Wurzelknoten der zu visualisierenden Welt, also der Wurzelknoten des Szenegrafen, definiert. Die Komponenten zur applikationsunabhängigen Beschreibung und Konfiguration der Laufzeitumgebung, werden im *engine* Namensraum zusammengefasst. Es besteht so die Möglichkeit, Einstellungen bezüglich der Visualisierung vorzunehmen, ohne dabei die Szene an sich anzutasten. Der Namensraum *contextsetup* speichert die ganzen Einstellungen bzw. Voreinstellungen, die den Laufzeit-Kontext betreffen. Der Benutzer hat nun die Gelegenheit in seiner X3D- oder VRML-Datei über die Verwendung des entsprechenden Knotentyps (*Scene*, *Engine*, *ContextSetup*) eine eigene Vorinitialisierung des Laufzeit-Kontextes schon mit dem Laden seiner Datei vorzunehmen. Dabei ist aber darauf zu achten, dass pro Datei immer nur je ein Knoten des Typs *Scene*, *Engine* und *ContextSetup* verwendet werden darf, da der Laufzeit-Kontext immer nur auf je einen aktiven *scene*, *engine* und *contextsetup* Namensraum verweisen kann. Beim Laden einer X3D- oder VRML-Datei ohne die Verwendung der oben genannten Knotentypen, nimmt das System faktisch die Vorinitialisierung selber vor. Das bedeutet, dass der komplette in der Datei notierte Knotenbaum in den vom System erstellten Wurzelknoten *DEF scene Scene* integriert wird. Ähnlich verhält es sich, wenn per Kommandozeile dem SAV oder dem InstantPlayer neben der VRML-Datei noch eine InstantPlayer Engine (EDV) Datei mit übergeben wird. Der in der EDV-Datei notierte Knotenbaum wird dann in den vom System generierten Wurzelknoten für den *engine* Namensraum *DEF engine Engine* untergeordnet. So

[18] eigene Darstellung

lässt sich zum Beispiel die ganze Visualisierungskonfiguration in eine eigene Datei auslagern, die beim Laden neben der eigentlichen Szene einfach mit getrennt übergeben wird.

Über die Eigengenerierung des gesamten Laufzeit-Kontextes mit den dazugehörigen Namensräumen und der Initialisierung der entsprechenden Komponenten baut das Avalon-System zusätzliche Funktionalitäten ein, die über den Player dem Nutzer zugänglich gemacht werden. Etwaige Funktionalitäten wären zum Beispiel:

- **das Auswählen verschiedener Navigationsmodi durch die Szenerie**
- **das Durchschalten der in der Szenerie verwendeten Kameras**
- **das Verwenden verschiedener Rendermodi**
- **etc.**

Beim InstantPlayer können diese Funktionalitäten bequem über Menüs angewählt werden, während beim SAV der Benutzer mit Tastenkombinationen (engl.: Shortcuts) arbeiten muss.

### 2.4.2 Externe Schnittstellen des Avalon-System

Wie im Punkt 2.2.1 bereits erwähnt, implementiert das Avalon-System externe Schnittstellen zur Manipulation des Szenegrafen. Diese Veränderungen werden wiederum in Echtzeit durch den Player visualisiert. Als sogenannte netzwerktransparente Application Services bietet das Avalon-System neben der Hypertext Transfer Protocol (HTTP)- und Simple Object Access Protocol (SOAP)-Anbindung, noch zusätzlich unter dem Begriff Avalon External Interface, das External Authoring Interfaces an (Abbildung 2-10). So können Manipulationen über applikationsunabhängige Schnittstellen am applikationsabhängigen Szenegrafen des Systems vorgenommen werden.

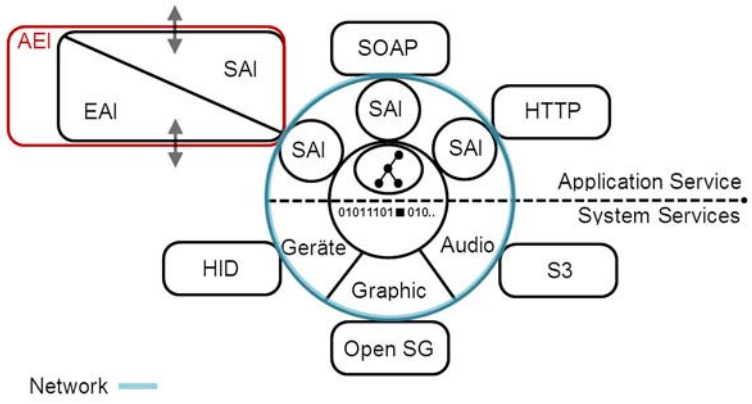


Abbildung 2-10: Application und System-Services[19]

Da die Anbindung sowohl über HTTP und SOAP nur eine unidirektionale Kommunikation zwischen externer Applikation und dem Player / dem Avalon-System zulässt, steht in dieser Arbeit nur das EAI-Interface im Fokus. Das System gruppiert zwar noch das durch den X3D-Standard spezifizierte Scene Access Interface (SAI) mit unter AEI, aber dessen Integrierung beschränkt sich nur auf eine rudimentäre Implementierung, sodass per SAI nur reine Statusinformationen des Avalon-System abgerufen werden können.

Über das EAI Interface lassen sich folgende Manipulationen vornehmen:

- **das Hinzufügen und Löschen von Knoten**
- **das Empfangen und Aussenden von Ereignissen (engl.: events)**
- **das Setzen/Verändern von Feldwerten**

Die Datenübertragung erfolgt dabei mittels systemspezifisch kodierter Pakete, die über das Transmission Control Protocol / Internet Protocol (TCP/IP) zwischen externer Applikation und dem Avalon-System verschickt werden. Aufgrund dieser Tatsache müssen sowohl auf Client Seite, wie auch auf Server Seite, spezielle Java / C++ Bibliotheken zum Einsatz kommen. Das durch das Avalon-System bereitgestellte EAI Interface bietet zur Nutzung seiner Services eine Anbindung nach Java und nach Programmiersprachen, die auf der Microsoft .NET Common Language Infrastructure (CLI) basieren (C#, Visual Basic, C++/CLI, J# und JScript). Die im Zuge dieser Arbeit umgesetzte Applikation, wurde auf der Grundlage von Java geschrieben und als solches wurde die Java-Anbindung des EAI-Interface benutzt. Das InstantReality-Paket stellt dafür eine entsprechende Java-Bibliothek *instantreality.jar* bereit. Der Benutzer kann dann über das direkte Einbinden der Java-Bibliothek in den Applikations-Projektordner seiner Java-Entwicklungsumgebung oder über das Hinzufügen zur globalen Umgebungsvariable

[19] in Anlehnung an Behr, Johannes 2005, S. 61

*CLASSPATH*, auf die speziellen Funktionen zugreifen und letztendlich seine Applikation kompilieren. Standardmäßig erfolgt die EAI-Verbindung zur Laufzeit des Avalon-Systems bzw. dessen grafischen Frontend über den Port 4848.

### 2.4.3 Fehlende Funktionalitäten des Avalon-Systems

Die am VRCP der TU-Chemnitz eingesetzten VR-Anwendungen lassen sich in der Regel in folgende Bestandteile unterteilen:

- **Grundszenerie, welche die eigentliche virtuelle Welt beschreibt**
- **Visualisierungskonfiguration, ausgelagert in eine EDV-Datei**
- **die einzelnen 3D-Objekte, wiederum jeweils als eigenständige VRML-Dateien**

Der Anspruch des VRCP, die einzelnen 3D-Objekte flexibel in die schon im InstantPlayer dargestellte Grundszenerie hineinzuladen bzw. bei Bedarf wieder herauslöschen zu können, wird so vom Avalon-System bzw. dem InstantPlayer nicht angeboten. Etwaige Notlösungen sind eher unkomfortabel und nicht einsatztauglich. Zudem besteht, wenn dennoch der Weg über einen Behelf genommen wurde, keine Möglichkeit einer Abspeicherung. Das VRCP setzt neben der CAVE als VR-Umgebung noch die Powerwall ein. Sodass je nachdem, welches VR-Ausgabegerät zur Visualisierung genutzt wird, verschiedene Visualisierungskonfigurationen für das Avalon-System zum Einsatz kommen. Eine Möglichkeit, im Arbeitsprozess schnell und flexibel die entsprechenden Visualisierungskonfigurationen und Grundszenerien auszuwählen, um das Avalon-System damit neu zu initialisieren, wird ebenfalls vom InstantPlayer nicht angeboten.

# 3. Bestehende GUI als Ausgangspunkt

Am IWP/VRCP der TU-Chemnitz wurde 2007 als studentisches Projekt eine Steuerungs-GUI für das Avalon-System, mit dem Namen *avGo* entwickelt. Auf Grund der fehlenden Plattformunabhängigkeit soll *avGo* unter Beibehaltung seines kompletten Funktionsumfanges nach Java portiert werden. Zusätzlich soll die Portierung dazu genutzt werden, um weitere Funktionalitäten einzubinden.

## 3.1 Programmiersprache und Plattform

Die bisherige Benutzeroberfläche *avGo* wurde in C++ unter Verwendung der Qt[20] (engl. cute) - Bibliothek programmiert und für das Betriebssystem Linux kompiliert. Die Qt-Bibliothek befähigt zum Erstellen von plattformübergreifenden Benutzeroberflächen ohne den Quellcode der Applikation umschreiben zu müssen. Da die Qt-Bibliothek, genau wie die Applikation *avGo* selber, in C++ implementiert ist, bleibt der Nachteil einer Kompiliersprache. Bei jedem Kompilierungsvorgang wird der Applikationsquellcode direkt in Maschinencode übersetzt und anschließend in ein betriebssystemspezifisches Format für ausführbare Dateien gepackt (Abbildung 3-11). Daraus ergibt sich, dass bei jedem Plattformwechsel die Applikation neu kompiliert werden muss. Um in der Weiterführung des Projekts eine Plattformunabhängigkeit zu gewährleisten, hat man die Portierung nach Java forciert. Java arbeitet nach dem Konzept „Write Once, Run Anywhere“ (deutsch: „Einmal schreiben, überall ausführen“). Eine Java Applikation braucht also nur einmal geschrieben und kompiliert werden. Der Java-Compiler übersetzt den Quellcode in plattformunabhängigen Javabytecode. Dieser Zwischenschritt wird durch den Interpreter, der Bestandteil der Java Virtual Machine (JVM) ist, zur Laufzeit in Maschinencode für den jeweiligen Prozessor übersetzt. Die JVM ist wiederum Teil der Java Laufzeitumgebung Java Runtime Environment (JRE), sodass als Kriterium für die Lauffähigkeit einer Java-Applikation nur das Vorhandensein einer entsprechenden JRE auf dem System entscheidet (Abbildung 3-11). Das Format einer ausführbaren Java-Datei (*.jar Archiv) ist allein javaspezifisch, sodass keine Abhängigkeit zum Betriebssystem entsteht.

[20] http://qt.nokia.com/

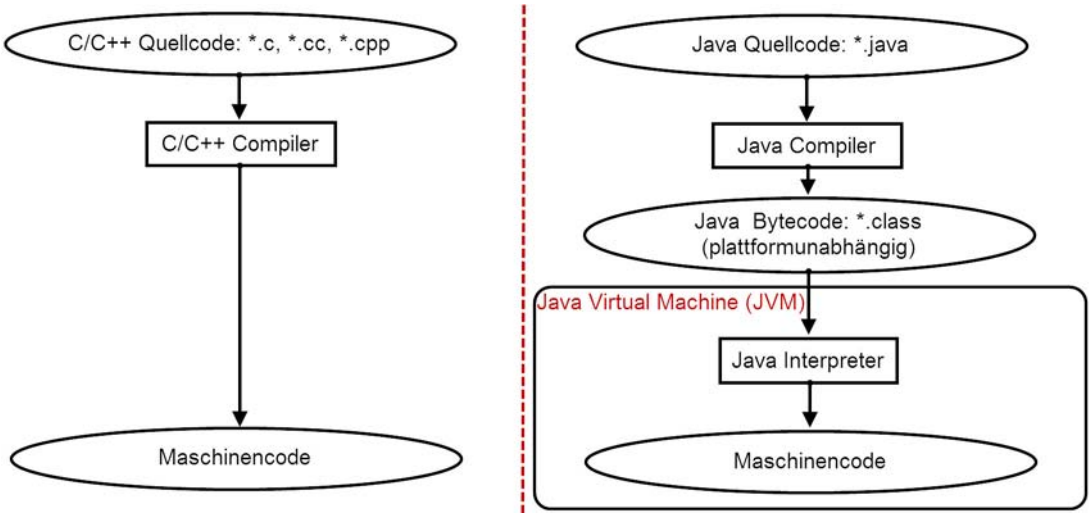


Abbildung 3-11: Umwandlungsprozess vom Quellcode zum aufführbaren Programm – Gegenüberstellung C/C++ (links) und Java (rechts)[21]

## 3.2 Funktionsumfang

Die Hauptfunktionalität der *avGo* Steuerungs-GUI besteht darin, die bereits im Punkt 2.1.7 genannten Anforderungen für den Arbeitsablauf des VRCP, die vom Avalon-System über dessen Visualisierungs- Frontend nicht angeboten werden, nachzuliefern. Folgend sollen diese nochmals aufgezählt werden.

- **Flexibles Umschalten von Visualisierungskonfiguration (EDV-Datei) und Grundszenerie (WRL-Datei)**
- **Flexibles Einbinden und Löschen von 3D Objekten, die in eigenen WRL-Dateien hinterlegt sind, in die Grundszenerie**
- **Möglichkeit, die so zusammengestellte Szenerie in einer Datei abzuspeichern**

Zusätzlich implementiert die *avGo* Steuerungs-GUI noch weitere Funktionalitäten. So lässt sich sowohl global, als auch für jedes einzelne in die Grundszenerie geladene 3D Objekt, steuern, ob es vom Avalon-System gerendert (dargestellt) werden soll oder nicht. Über Eingabemasken hat der Benutzer zusätzlich die Möglichkeit, eine globale Transformation, wie auch eine spezifische, auf die einzeln geladenen 3D-Objekte gerichtete, Transformation vorzunehmen.

Über einen eigenen Settings-Dialog (Abbildung 3-12) wird dem Benutzer die Möglichkeit gegeben, verschiedenste Einstellungen vorzunehmen, die dann per Bestätigung persistent in der Registry des Betriebssystems abgespeichert werden. Zu den Einstellungen zählen:

[21] eigene Darstellung

- **Auswahl des Avalon-Ordner**

Hierbei handelt es sich um den Installations-Ordner des InstantReality-Frameworks. Die *avGo* Steuerungs-GUI benutzt zum Start des Avalon-System dessen Kommandozeilen-Visualisierungsfrontend, den SAV-Player. Die Richtigkeit dieser Einstellung entscheiden also darüber, ob das Avalon-System über die *avGo* Steuerunsg-GUI gestartet werden kann oder nicht.

- **Auswahl des Konfigurations-Ordner**

Der Benutzer hat die Möglichkeit, den Ordner, in dem alle Visualisierungskonfigurationen (EDV-Dateien) und Grundszenerien (WRL-Dateien) zentral abgelegt sind, als Konfigurations-Order festzulegen. Nach Übernahme der Settings-Einstellungen analysiert *avGo* den festgelegten Konfigurationsordner, um die enthaltenen EDV-und WRL Dateien für die spätere Auswahl in entsprechenden Tabellen und Comboboxen in der GUI aufzulisten.

- **Auswahl der Lizenz-Datei**

Das Avalon-System braucht zur Lauffähigkeit das Vorhandensein der Datei *license.xml*. In dieser Datei befinden sich unter anderem Registrierungsschlüssel, die die einzelnen Komponenten des InstantReality-Frameworks für deren Lauffähigkeit freischalten und etwaige Reglementierungen aufheben bzw. beibehalten lassen. Die Lizenz-Datei liegt nach der Installation des InstantReality-Frameworks standardmäßig im *bin*-Verzeichnis des Installationsordners. Beim Start des Avalon-System wird das Vorhandensein der Datei *license.xml* im *bin*-Verzeichnis überprüft, bleibt dies ohne Erfolg, wird die Existenz der globalen Umgebungsvariable *TALIC_LICENSE_FILE* abgefragt. Mit der Deklarierung dieser Umgebungsvariable hat der Nutzer die Möglichkeit, bei ausgelagerter Lizenz-Datei, deren Standort dennoch für den speziellen Betriebssystem-Prozess, dem gestarteten Avalon-System, bekannt zu geben. Nichts anderes wird über die Auswahl der Lizenz-Datei bewerkstelligt. Die dort gemachte Auswahl führt zu Deklaration der *TALIC_LICENSE*_FILE-Umgebungsvariable mit der entsprechenden Pfadangabe der Lizenz-Datei.

- **Abbilden der Kommandozeile**

Per Kommandozeile können dem SAV nicht nur die Visualisierungskonfiguration und die Grundszenerie übergeben werden, sondern zusätzlich noch weitere Parameter, die die Initialisierung des Laufzeit-Kontext steuern. Der Settings-Dialog integriert dafür ein eigenes Textfeld, über welches vom Benutzer die Parameter eingegeben werden können.

All die genannten Einstellungsmöglichkeiten sind im Settings-Dialog unter der Rubrik *directories and executable* gruppiert (Abbildung 3-12 (A)).

- **Anlegen \ Editieren \ Löschen von Umgebungsvariabeln für den externen Prozess**

Im Bereich *additional environment variables* des Settings-Dialog (Abbildung 3-12 (B)), ist eine Tabelle implementiert, über welche vom Benutzer Umgebungsvariablen für den externen Prozess, in dem Fall dem Avalon-System, angelegt, editiert und gelöscht werden können.

- **Einstellen der Parameter für das AEI Interface**

Um eine Verbindung der *avGo* Steuerungs-GUI über das AEI-Interface zum Avalon-System herstellen zu können, muss sowohl die IP-Adresse, als auch das Port des Rechners angegeben werden, auf dem das Avalon-System läuft. Standardmäßig öffnet das Avalon-System für die TCP/IP Verbindung zur Laufzeit das Port 4848. Der Settings-Dialog gruppiert diese Einstellungen unter *Avalon External Interface* (Abbildung 3-12 (C)).

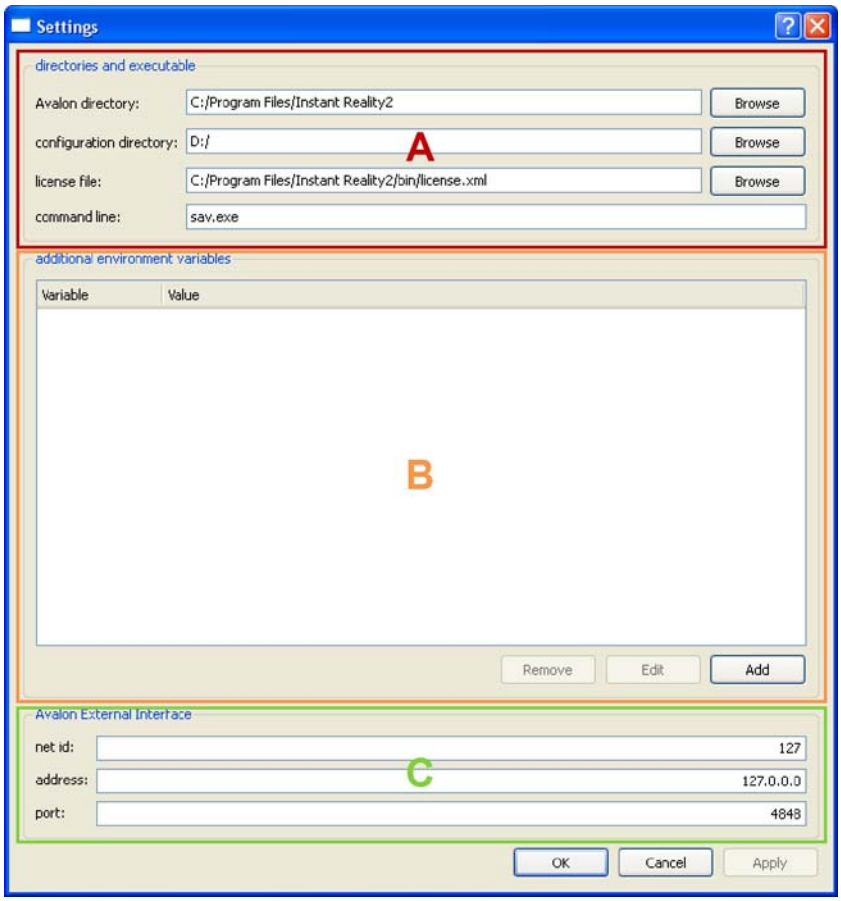


Abbildung 3-12: Settings-Dialog der *avGo* Steuerungs-GUI[22]

Die eigentliche *avGo* Steuerungs-GUI setzt sich aus einem Menu, einer ToolBar und einem Hauptfenster zusammen. Die wesentlichen Funktionen die das Menüs und die ToolBar zugänglich machen, sind folgend aufgezählt.

[22] Screenshot – eigene Darstellung

- **erzeugen, laden und speichern einer Session**
- **laden und entfernen von 3D Objekten (WRL-Dateien)**
- **starten und stoppen des Avalon-Systems**
- **öffnen des Settings-Dialogs**

In die ToolBar sind zusätzlich noch ComboBox-Elemente integriert, um sowohl die geladenen 3D-Objekte in Form von WRL-Dateien, als auch die im Konfigurations-Ordner enthalten Visualisierungskonfigurationen und Grundszenerien aufzulisten bzw. auswählen zu können (Abbildung 3-13).


Abbildung 3-13: ToolBar der Steuerungs-GUI *avGo*[23]

Die Session bildet eine Art Container, der beim Start von *avGo* virtuell angelegt wird. Innerhalb dieses Containers bestehen Vorrichtungen, welche die vom Nutzer individuell vorgenommenen Einstellungen hinsichtlich globaler Sichtbarkeit, globaler Transformation, geladenen 3D Objekten, deren Sichtbarkeit und Transformation, erstmalig zwischenspeichert (Abbildung 3-14). Beim Schließen bzw. auch schon während des Arbeitens mit *avGo*, hat der Nutzer die Möglichkeit, diesen rein virtuellen Container in Form einer auf Extensible Markup Language (XML) basierenden Datei (XSN-Datei) abzuspeichern.

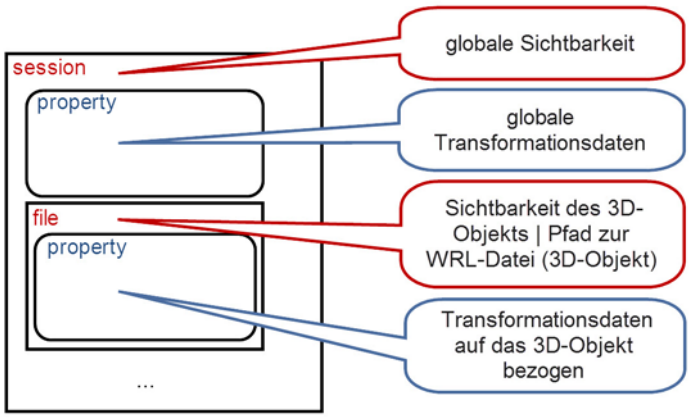


Abbildung 3-14: Aufbau einer Session bei *avGo*[24]

Das Avalon-System führt für bestimmte Komponententypen das sogenannte *render* Feld ein, mit dem Knoten bzw. auch ganze Teilbäume aus dem Darstellungsprozess (engl.: rendering) des System ausgeschlossen werden können. Jede Gruppierungskomponente besitzt so ein

[23] Screenshot – eigene Darstellung

[24] eigene Darstellung

*render* Feld. Hinter der Steuerung der globalen Sichtbarkeit verbirgt sich nichts weiter, als ein direkter Verweis auf das *render* Feld des Wurzelknotens der Szene. Je nachdem ob das Feld auf *True* oder *False* gesetzt wird, erfolgt das Darstellen der im Wurzelnknoten gruppierten Knoten. Jedes in die Session geladene 3D Objekt wird über einen Inline-Knoten in die Grundszenerie eingebunden. Zur Steuerung der spezifischen, direkt auf das 3D Objekt bezogenen Sichtbarkeit, besteht hier der Verweis auf das durch den entsprechenden Inline-Knoten integrierte *render* Feld.

Das Hauptfenster (Abbildung 3-15) von *avGo* wird durch ein Tabbed-Element (deutsch: Registernavigation) bestimmt, in dem die Tabs: Avalon, Session und File enthalten sind (Abbildung 3-15 (C)). Auf der linken Seite des Hauptfensters befinden sich Listen-Elemente, die im Endeffekt die gleichen Funktionen übernehmen, wie die ComboBox-Elemente in der ToolBar. Gruppiert unter dem Punkt *Configuration* findet eine getrennte Auflistung der im Konfigurations-Ordner enthaltenen Visualisierungskonfigurationen und Grundszenerien statt (Abbildung 3-15 (A)). In der Liste unter dem Punkt „File“ (Abbildung 3-15 (B)) werden sämtliche in die Session geladenen 3D-Objekte aufgeführt. Auch hier hat der Nutzer die Möglichkeit, 3D-Objekte in die Session neu zu laden oder zu entfernen. Vor jedem Eintrag befindet sich ein CheckBox-Element, über welches die Sichtbarkeit des betreffenden 3D Objekts in der Grundszenerie gesteuert werden kann.

Die im Tabbed-Element des Hauptfensters von *avGo* enthaltenen Tabs bieten folgende Funktionalitäten.

- **Avalon-Tab**

Auf das im Avalon-Tab integrierte Dokument, gibt das Avalon-System nach dem Start seine Statusmeldungen aus. Der Benutzer hat die Möglichkeit, die Ausgabe im Dokument in eine LOG-Datei abzuspeichern.

- **Session-Tab**

Das Session-Tab ist in die Bereiche *Attributes* und „Transformation“ unterteilt. Die Einstellungen bezüglich der Steuerung der globalen Sichtbarkeit, die Festlegung eines Arbeitsverzeichnisses für den externen Prozess des Avalon-Systems und die Angabe, ob die WRL-Dateien der 3D-Objekte sich relativ zum Pfad der abgespeicherten Session-Datei befinde, sind unter *Attributes* gruppiert (Abbildung 3-16). Im *Transformation* Bereich (Abbildung 3-17) finden sich die Eingabemasken zur Eingabe der Transformationsdaten für die globale Transformation wieder. Der Nutzer hat die Möglichkeit die Daten über das Axis-Angle (AA) Format oder über das Yaw-Pich-Roll (YPR) Format einzugeben. Per Submit-Button lassen sich die Daten in den virtuellen Container „Session“ übernehmen bzw., wenn schon eine Verbindung über das AEI Interface besteht, direkt an das Avalon-System übermitteln. Über den Default-Button können die Transformationsdaten wieder auf den Initialisierungsstand zurückversetzt werden.

- **File-Tab**

Im File-Tab können weitestgehend die gleichen Einstellungen vorgenommen werden, wie im Session-Tab. Nur das dieses Einstellungen sich auf das jeweilige 3D Objekt beschränken, welches über die entsprechende ComboBox in der ToolBar oder aus der *Files* Liste im Hauptfenster ausgewählt wurde. Die ComboBox-Elemente stehen in Beziehung zu den Listen-Elementen im Hauptfenster von *avGo*, um eine synchrone Selektion zu gewährleisten. Der *Attributes* Bereich gruppiert die Steuerung der Sichtbarkeit des 3D Objekts in der Grundszenerie, sowie die Pfadanzeige zur WRL-Datei, die das 3D Objekt repräsentiert (Abbildung 3-16).

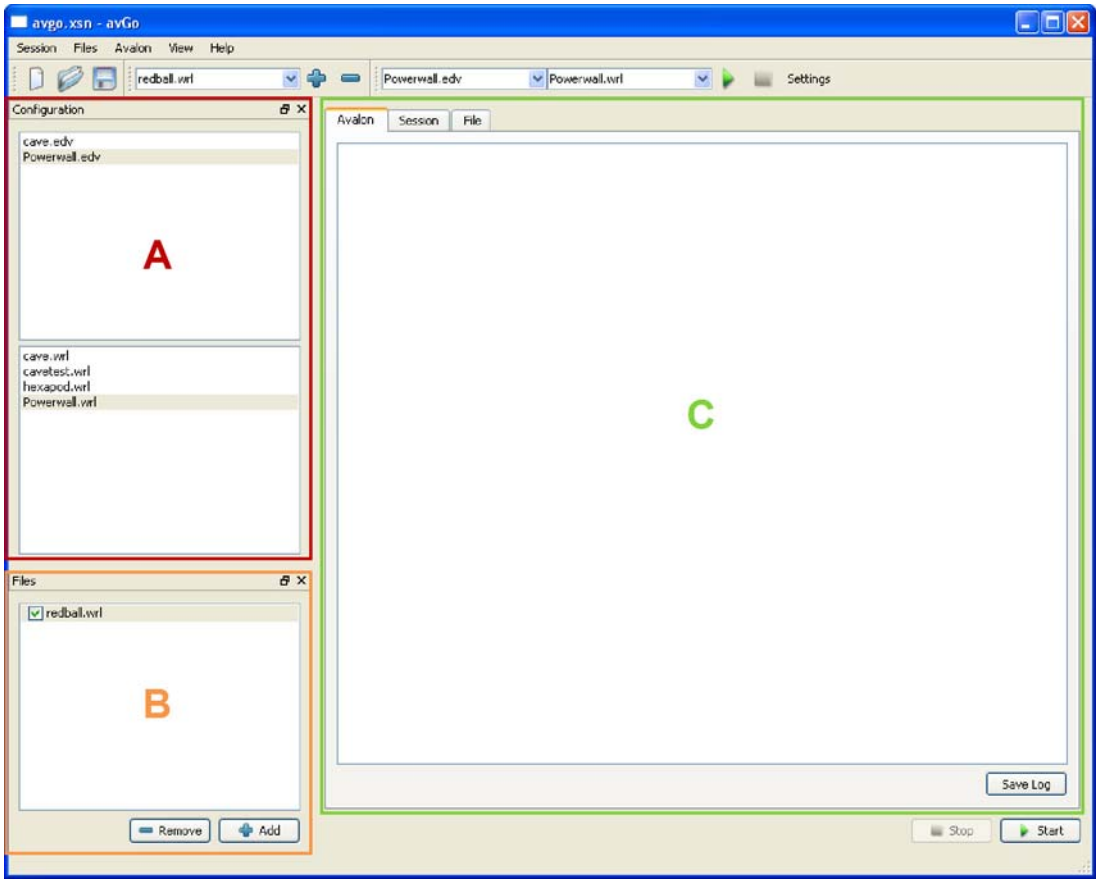


Abbildung 3-15: Die Steuerungs-GUI *avGo* für das Avalon-System[25]

[25] Screenshot - eigene Darstellung

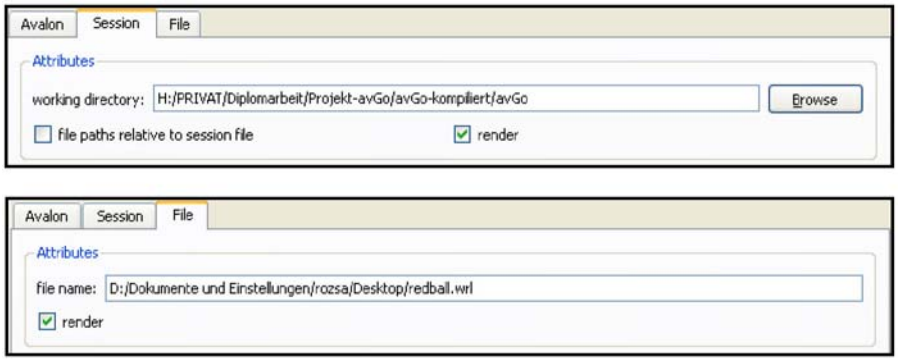


Abbildung 3-16: Attributes-Bereiche in den Tabs, Session (oben) und File (unten)[26]

Die folgende Abbildung (Abbildung 3-17) zeigt die Eingabemasken für die Transformationsdaten, so wie sie im Session-und File-Tab und dort im Bereich *Transformation* integriert sind. Mit dem Start von *avGo* und dem Erzeugen des virtuellen Containers *Session*, erfolgt auch eine Vorinitialisierung der Transformationsdaten, sowohl global, als auch bei dem einzeln geladenen 3D Objekten. Wie bereits erwähnt, integriert *avGo* die geladenen 3D-Objekte über Inline-Knoten in die Grundszenerie. Jeder dieser Inline-Knoten wird wiederum in einen eigenen Transformations-Knoten gruppiert, um so die spezifische, nur auf das jeweilige 3D Objekt bezogene, Transformation vornehmen zu können.

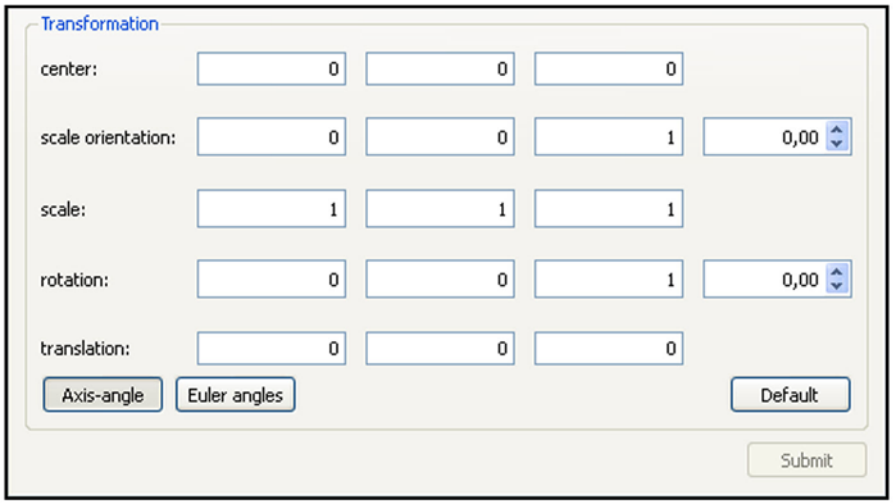


Abbildung 3-17: Transformation-Bereich in den Tabs, Session und File[27]

Um die in obiger Abbildung (Abbildung 3-17) ersichtlichen Begriffe (*center*, *scale orientation*, *scale*, *rotation* und *translation*) besser verstehen zu können, sollen nachfolgend grundlegende Dinge in Bezug auf die Transformation von Körpern wiedergegeben werden.

[26] Screenshot - eigene Darstellung

[27] Screenshot-eigene Darstellung

Die Transformation von Körpern im Raum zeichnet sich durch drei Grundtypen aus: dem Skalieren, dem Rotieren und der Translation.

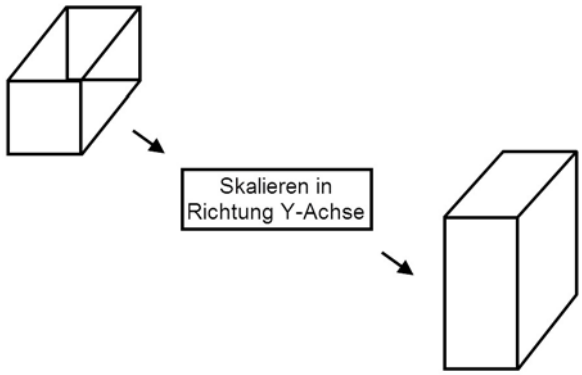


Abbildung 3-18: Skalieren eines Körpers[28]

Das Skalieren (engl.: scale) steht für das Strecken bzw. Stauchen eines Körpers im dreidimensionale Raum (Abbildung 3-18). Je nachdem in welche Richtung der Körper skaliert werden soll, wird über einen sogenannten Skalierungsfaktor die Einheit der entsprechenden Achse vergrößert oder verkleinert. Die Skalierung erfolgt über die Angabe von drei Zahlen, welche die jeweiligen Faktoren für die entsprechenden Achsen(X, Y und Z) bilden.

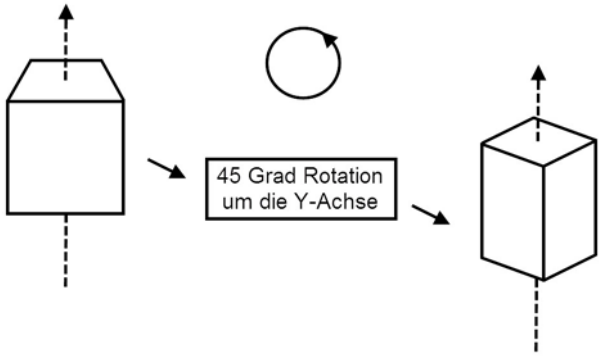


Abbildung 3-19: Rotieren eines Körpers[29]

Die Rotation (engl.: rotation) eines Körpers im dreidimensionalen Raum (Abbildung 3-19) kann auf zwei verscheiden Arten angegeben werden. Einmal über die sogenannte Achsen-Winkel Form (engl.: Axis-Angle (AA) oder über die Eulersche-Winkel Form (engl.: Yaw-Pith-Roll (YAW)). Bei der Achsen-Winkel Form erfolgt die Rotation des Körpers über die Angabe eines Vektors und eines Winkels, der angibt, um wie viel Grad der Körper um den Vektor rotieren soll.

[28] in Anlehnung an Matsuba und Roehl 1996, S. 167

[29] in Anlehnung an Matsuba und Roehl 1996, S. 168

Mit der Festlegung von drei Winkeln, die angeben, um wie viel Grad der Körper um die entsprechende Achse rotiert werden soll, erfolgt die rotation nach der Eulersche-Winkel Form. Nach dem Prinzip *Yaw-Pit-Roll* werden die einzelnen Rotationsphasen um die Achsen abgearbeitet. Zuerst erfolgt die horizontale Rotation des Körpers (*Yaw*), danach das Heben oder Senken der Blickrichtung auf den Körper (*Pitch*), mit der Rotation der Blickrichtung selber (*Roll*), wird der gesamte Rotationsvorgang abgeschlossen. Die Winkelangaben bei der Achsen-Winkel Form und Eulersche-Winkel Form werden in Bogenmaß vorgenommen.

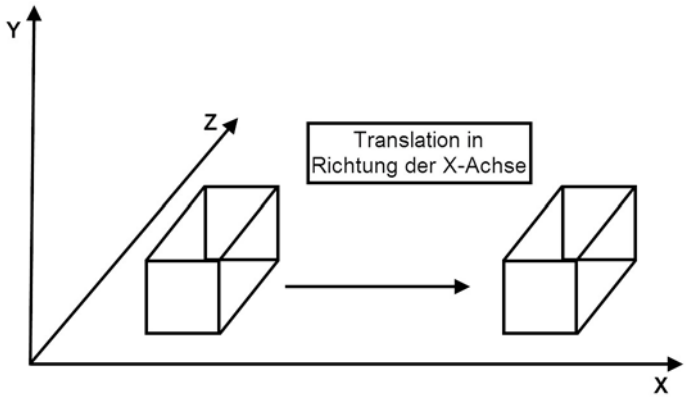


Abbildung 3-20: Translation eines Körpers[30]

Die Translation (engl.: translation) beschreibt die einfache Bewegung eines Körpers im Raum (Abbildung 3-20). Mit Hilfe eines Vektors wird die Bewegungsrichtung des Körpers im dreidimensionalen Raum vorgegeben.

Die Eingabemaske für die Transformationsdaten (Abbildung 3-17), führt neben den bereits erwähnten Parametern (*scale*, *rotation*, *translation*) noch weitere an, die nachfolgend erklärt werden.

Über *scale orientation* besteht die Möglichkeit, die Skalierung des Körpers an einem vorher in seiner Ausrichtung veränderten Koordinatensystem zu vollziehen. Da die Änderung der Ausrichtung durch Rotation erreicht wird, gelten für *scale orientation* die gleichen Notierungsvorgaben, wie für die *rotation* selber. Nach erfolgter Skalierung des Körpers, wird das Koordinatensystem wieder in seine in seine ursprüngliche Ausrichtung zurückversetzt.

Hinter *center* verbirgt sich die Veränderung der Lage des Koordinatenursprungs und damit des gesamten Koordinatensystems durch einfaches Bewegen, also Translation.

[30] in Anlehnung an Matsuba und Roehl 1996, S. 169

## 3.3 Fehlende Funktionalitäten

Wie im Punkt 3.1 bereits erwähnt, ist die Steuerungs-GUI *avGo* nicht ohne weiteres plattformübergreifend einsetzbar, jeder Plattformwechsel setzt für die Benutzung der *avGo*-Applikation eine Neukompilierung voraus. Die während des Einsatzes von *avGo* ausgewählte Visualisierungskonfiguration und Grundszenerie wird nicht mit in der abgespeicherten Session hinterlegt, sodass der Nutzer beim erneuten Laden der Session die Auswahl wiederholt ausführen muss. Die *avGo*-Applikation ist nicht in der Lage, den Szenegrafen, der in die Grundszenerie geladenen 3D-Objekte, zu visualisieren, sodass eine etwaige Kontrolle der Datenstruktur durch den Nutzer nicht möglich ist.

# 4. Konzept der neuen GUI

Sowohl das Layout-Konzept als auch der Funktionsumfang der Steuerungs-GUI *avGo* bilden die konzeptionelle Grundlage zur Erstellung der neuen GUI. Neben dem Menü und dem Hauptfenster, soll ebenfalls wieder eine ToolBar, sowie eine Statusleiste in die GUI integriert werden (Abbildung 4-1).

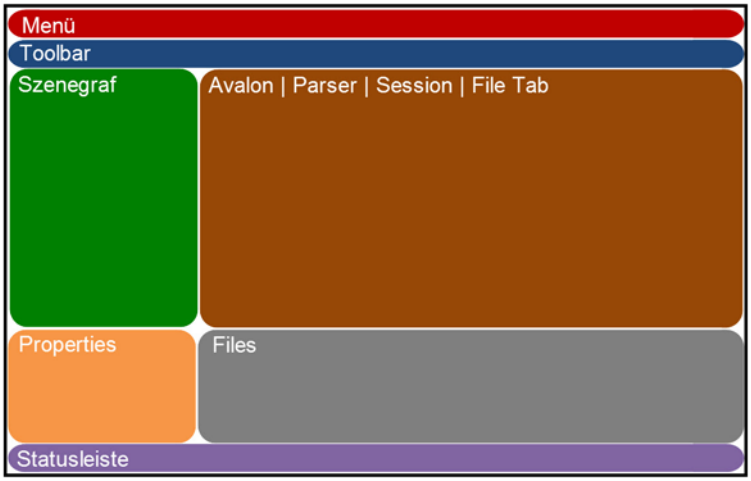


Abbildung 4-1: Konzept für das Hauptfenster[31]

Im Hauptfenster behält das Tab-Element seine zentrale Lage bei. Die linke Seite des Hauptfensters wird dazu benutzt, Elemente zur Visualisierung des Szenegrafen und der Knoten-Eigenschaften, zu platzieren. Die hierarchische Struktur des Szenegrafen soll in Form eines Baums dargestellt werden, dessen einzelne Bestandteile durch den Nutzer anwählbar sind. Eine Mehrfachselektierung soll allerdings nicht möglich sein. Die in den Feldern des Felddatentyps *SFNode* und *MFNode* enthaltenen VRML-Knoten bilden die Blätter bzw. Zweige des Baumes. Das Selektieren durch den Nutzer hat die Auflistung der im VRML-Knoten enthaltenden Felder mit dazugehörigen Feldwerten im Bereich *Proberties* zur Folge. Zur Anzeige kommen keine Felder des Felddatentyps *SFNode* und *MFNode*. Die im Bereich Files integrierten Elemente listen zum einen die in die Grundszenerie geladenen 3D-Objekte auf, zum anderen bieten sie die Möglichkeit 3D-Objekte neu hinzu zu laden bzw. bestehende zu entfernen. In die Auflistung soll ebenfalls, wie in der *avGo*-Applikation, die direkte Steuerung der Sichtbarkeit des entsprechenden 3D Objekts möglich sein. Die Selektierung eines 3D Objekts in der Auflistung hat automatisch die Darstellung dessen interner Knotenstruktur im Bereich Szenegraf zur Folge.

[31] eigene Darstellung

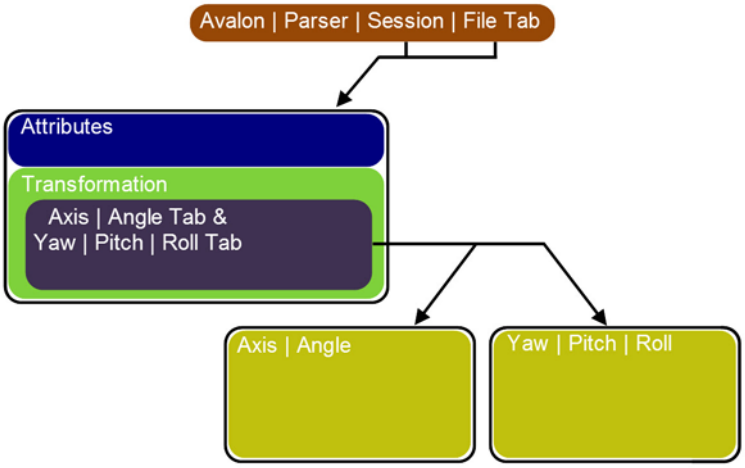


Abbildung 4-2: Avalon | Parser | Session | File Tab[32]

Der im Tab-Element integrierte *Session*-, *File*- und *Avalon*-Tab übernimmt das Layout und weitestgehend die Funktionalität, wie sie bereits der entsprechende Pendant in der *avGo*-Applikation vorgibt. Sowohl der *Session*-Tab als auch der *File*-Tab integriert wiederum ein Tab-Element, um die Eingabe der Transformationsdaten und dort vor allem der verschiedenen Notierungsformen für die *rotation* und *scale orientation* gerecht zu werden bzw. zu ermöglichen (Abbildung 4-2). Die veränderten Daten können direkt über das Betätigen einer *submit*-Schaltfläche in den virtuellen Container Session übernommen werden. Ist das Avalon-System bereits gestartet, findet darüber hinaus ein Datenupdate über das Interface statt. Sämtliche Transformationsparameter lassen sich über eine *default*-Schaltfläche wieder auf ihren Initialisierungstand zurückversetzen. Auf die Möglichkeit im Attributsbereich des *Session*-Tab das Arbeitsverzeichnis für den externe Prozess des Avalon-Systems festzulegen, wurde verzichtet, statt dessen soll nur der Pfad und Name der aktuell geladenen Session-Datei angezeigt werden. Die Attributbereiche beider Tabs (*Session* und *File*) bieten gleichermaßen die Gelegenheit, die Sichtbarkeit zu steuern. Das Konzept, die Sichtbarkeit über ein CheckBox-Element zu regeln, wird aus der *avGo*-Applikation übernommen. Eine Änderung der Sichtbarkeit benötigt keine Betätigung der *submit*-Schalfläche zur Übernahme, diese wird automatisch bei Aktivierung oder Deaktivierung des CheckBox-Elements vorgenommen. Die *submit*-Schaltfläche bezieht sich rein auf die Speicherung bzw. Übernahme der Transformationsdaten. Die im *File*-Tab dargestellten Einstellungsmöglichkeiten beziehen sich immer auf das im *Files*-Bereich selektierte 3D Objekt, während die sich auf gesamte virtuelle Welt beziehenden Einstellungen im *Session*-Tab visualisiert werden. Das Starten des Avalon-System über das SAV Frontend führt dazu, dass die Statusmeldungen des Avalon-Systems auf der Konsole ausgegeben werden. Diese Ausgabe wird auf ein Document-Element im *Avalon*-Tab umgelenkt. Das gleiche Prinzip steht hinter dem zusätzlich eingefügten Parser-Tab. Die vom VRML-Parser erzeugte Ausgabe von Statusmeldungen wird auf ein im Parser-Tab platziertes Document-Element

[32] eigene Darstellung

umgelenkt und visualisiert. Neben der reinen Visualisierung der Statusmeldungen des Avalon-Systems und des VRML-Parsers im entsprechenden Tab, besitzt der Nutzer die Möglichkeit, die jeweiligen Document-Inhalte in eine externe LOG-Datei abzuspeichern.

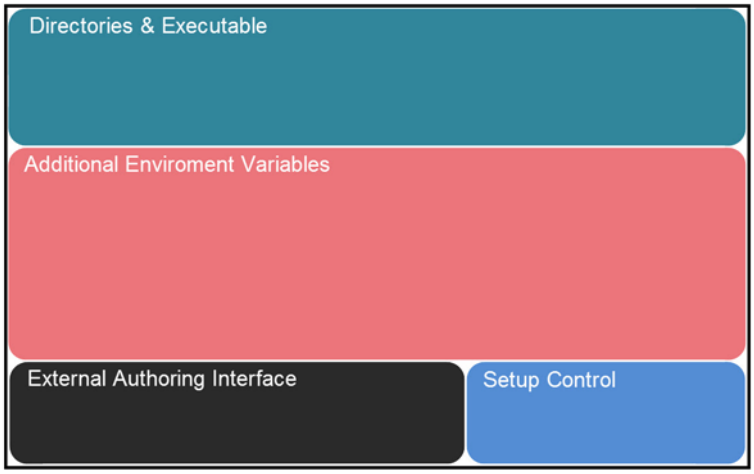


Abbildung 4-3:Konzept für den Settings-Dialog[33]

Der Settings-Dialog (Abbildung 4-3) nimmt den Funktionsumfang seiner avGo Entsprechung auf und bietet ihn in einem leicht veränderten Layout an. Alle Einstellungen die durch den Settinsg-Dialog dargestellt bzw. vorgenommen werden können, sind persistent in der Registry des jeweiligen Betriebssystems hinterlegt.

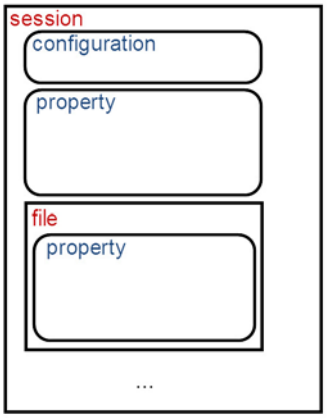


Abbildung 4-4: Konzeptioneller Aufbau des Session-Containers[34]

Die Struktur des Session Containers, sowohl als Objektstruktur für die virtuelle Präsentation der Session bei der Ausführung des Programms, als auch für die physische Darstellung als Sessi-

[33] eigene Darstellung

[34] eigene Darstellung

on-Datei, wird von der *avGo*-Applikation übernommen. Um die Funktionalität zu gewährleisten, dass die ausgewählte Visualisierungskonfiguration und Grundszenerie mit in der Session-Datei hinterlegt wird, soll zusätzlich in der auf XML basierenden Datei der Berech *configuration* eingefügt werden (Abbildung 4-4).

Zur Kommunikation zwischen dem externen Avalon-Prozess bzw. dem Avalon-System, wird auf das EAI Interface zurückgegriffen. Die Verbindung wird sofort nach dem Start des Avalon-System, welcher über die GUI angestoßen wurde, aufgebaut. Die in die GUI integrierte Statusleiste soll dem Benutzer signalisieren, wenn das Avalon-System vollständig initialisiert ist und eine Verbindung über das Interface besteht. Das Starten des Avalon-Systems ist über das Menü und über die ToolBar möglich.

Das Menü und die ToolBar decken den durch die *avGo*-Applikation vorgegebene Funktionsumfang für die entsprechenden Elemente ab.

# 5. Umsetzung

## 5.1 Aufbau der GUI über SWING

Java bietet zur Umsetzung von Grafischen Benutzeroberflächen ein Sammelsurium an verschiedensten GUI-Elementen an, die in den sogenannten Java Foundation Classes (JFC) zusammengefasst werden. Ab der Java Version 1.2 und der Java Development Kit (JDK) Version 1.1, sind die JFC Bestandteil der Java Plattform. Davor nahm das Abstract Window Toolkit (AWT) die Rolle des JFC ein. Aufgrund des Arbeitsprinzips, dass sich hinter dem AWT verbirgt und der daraus resultierenden recht überschaubaren Anzahl an GUI-Elementen, kam es zu einer Weiterentwicklung, die in den JFC mündeten. Das bestehende AWT-Arbeitsprinzip wurde ersetzt, wobei die alten Komponenten beibehalten und mit neuen angereichert wurden. Die Menge der neu hinzugekommenen Elemente läuft unter der Projektbezeichnung SWING.

Die folgende Abbildung (Abbildung 5-1) stellt die Arbeitsweise des AWT und SWING gegenüber.

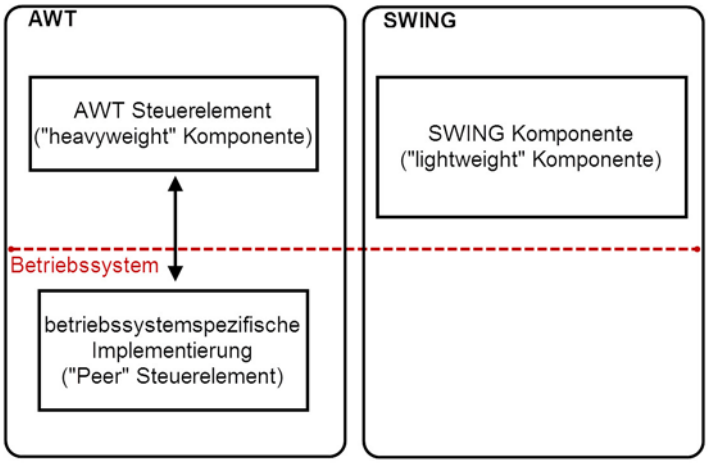


Abbildung 5-1: Abstract Window Toolkit (AWT) und SWING[35]

Grafische Benutzeroberflächen, die unter Verwendung der AWT Klassenbibliothek umgesetzt wurden, zeigen das betriebssystemspezifische Aussehen des jeweiligen Zielsystems, auf dem die GUI läuft. Dieses Verhalten stellt sogleich das Prinzip des AWT dar. Die einzelnen im AWT enthaltenen Grafikelemente brauchen zur Funktion die betriebssystemspezifische Implementierung als Pendant. Die jeweilige AWT Klasse dient faktisch nur als Vermittler zwischen dem AWT Steuerelement und der dazugehörigen betriebssystemeigenen Implementierung. Die AWT

[35] eigene Darstellung

Steuerelemente werden deshalb auch als *heavyweight* Komponenten bezeichnet und ihre Entsprechung auf Seiten des Betriebsystems als *Peer* Steuerelement. Im Gegensatz dazu sind die Grafikkomponenten in SWING nicht mehr auf ihre betriebssystemeigenen *Peer* Steuerelemente angewiesen. SWING erzeugt die Grafikkomponenten weitestgehend selbst (abgesehen von den Top-Level Containern *JFrame*, *JDialog*, *JWindow* und *JApplet*), was wiederum bedeutet, dass der Code, der das Aussehen und die Funktionsweise des jeweiligen Steuerelements bestimmt, nicht mehr im nativen Code des Betriebssystems umgesetzt ist sondern in echten Java Code verfasst ist. SWING Komponenten bezeichnet man deshalb auch als *leightweight* Komponenten.

Durch die oben beschriebenen unterschiedlichen Arbeitsprinzipien von SWING und dem AWT, ergeben sich etliche Vorteile, die für eine Verwendung der SWING Klassenbibliothek sprechen.

- **Da die SWING Komponenten völlig losgelöst von der betriebssystemspezifischen Implementierung der jeweiligen Grafikelemente sind, beschränkt sich deren Funktion eben nicht mehr auf den kleinsten gemeinsamen Nenner der unterschiedlichen Peer Elemente, der jeweiligen Betriebsysteme. Die SWING Grafikelemente können plattformübergreifend realisiert werden und stehen nach der Implementierung ohne Portieraufwand auf allen Systemen zur Verfügung. Dadurch kann die SWING Klassenbibliothek mit einer umfassenderen Menge an anspruchsvollen Grafikelementen aufwarten als die AWT Klassenbibliothek.**
- **Betriebssystemspezifische Besonderheiten entfallen für die SWING Grafikkomponenten. Auf allen Plattformen finden sich dasselbe Aussehen und die Bedienung der jeweiligen Komponenten wieder.**
- **Der Code für die Implementierung des jeweiligen SWING Grafikelements ist deutlich vereinfacht und logischer.**

In diesem Zusammenhang ist zu erwähnen, dass sich mit der Entwicklung der JFC eine Neustrukturierung der Architektur für die einzelnen Grafikelemente vollzogen hat. Der SWING Architekturaufbau, der auf dem Model-View-Controller (MVC) Prinzip basiert, zieht sich wie ein roter Faden durch sämtliche leichtgewichtige Komponenten, vom einfachen Button bis zur komplexen Tabelle (Abbildung 5-2).

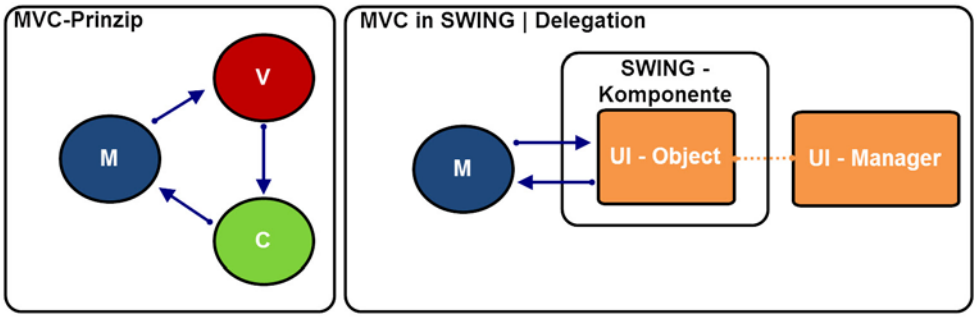


Abbildung 5-2: Klassisches Model-View-Controller-Prinzip (MVC) (links) und SWING Architekturaufbau (rechts)[36]

Durch die ebenfalls mit den JFC eingeführte Möglichkeit das Aussehen (engl.: look) und Verhalten (engl.: feel) der gesamten GUI-Elemente flexibel zu steuern, setzt der SWING Architekturaufbau nicht eins zu eins den klassischen MVC Ansatz um. Sowohl *Controller* als auch *View* werden durch das sogenannte User Interface (UI) Object zusammengefasst, das durch den UI Manager bereitgestellt wird. Sämtliche der so bezeichneten Look & Feel Aspekte werden durch die GUI Komponenten and das ihr entsprechenden UI Object weitergeleitet. Zwischen diesem und dem Model erfolgt die Kommunikation, um etwaige Änderungen an den Daten bekanntzugeben bzw. die Neustrukturierung der Daten zu visualisieren.

### 5.1.1 Layout

Um auf die verschiedenen Layout-Prinzipien, die in der JavGo-GUI zur Anwendung kommen, eingehen zu können, sollten nochmals grundlegende Dinge über den Aufbau der SWING Top-Level-Container Erwähnung finden.

Wie im Punkt 5.1 bereits genannt, ist SWING nicht gänzlich von schwergewichtigen Komponenten befreit. Die sogenannten Top-Level-Container bauen alle auf den entsprechenden AWT-Klassen auf, bzw. erweitern diese. Das *JFrame* SWING-Element symbolisiert einen solchen Top-Level-Container, dessen ganz spezieller Aufbau ist allen Top-Level-Containern zueigen. Ein *JFrame* beruht prinzipiell auf einer so genannten Root-Pane. Bei jeder Instanzierung eines Top-Level-Containers wird automatisch eine Root-Pane erstellt, sodass eine explizite Erstellung entfällt. Die Root-Pane fungiert als leichtgewichtiger Container für den schwergewichtigen Top-Level-Container. Dieser leichtgewichtige Container setzt sich wiederum aus vier Bestandteilen zusammen (Abbildung 5-3), die jede für sich ihre ganz bestimmten Aufgaben besitzen.

[36] in Anlehnung an http://java.sun.com/products/jfc/tsc/articles/architecture/index.html

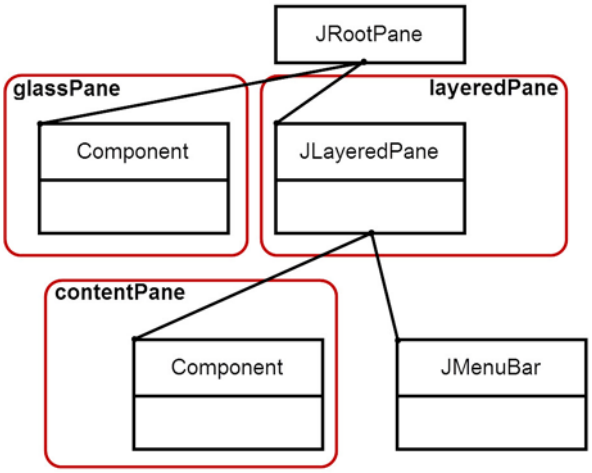


Abbildung 5-3: Bestandteile der RootPane[37]

Die Layered-Pane verwaltet die Content-Pane und die optionale *MenuBar*. Die Content-Pane beinhaltet sämtliche sichtbaren *leightweight* SWING Komponenten. Die *MenuBar* fungiert als eigenständige Komponente, die außerhalb der Content-Pane direkt in den Top-Level-Container eingefügt wird. Die Glass-Pane liegt wie in folgender Abbildung (Abbildung 5-4) zu sehen, über allen anderen Pane's und dient dazu, etwaige Nutzerinteraktionen, die für die Root-Pane bestimmt sind, abzufangen oder übergreifend über Content-Pane und *MenuBar* zu zeichnen. Standardmäßig ist die Glass-Pane auf Inaktivität geschalten.

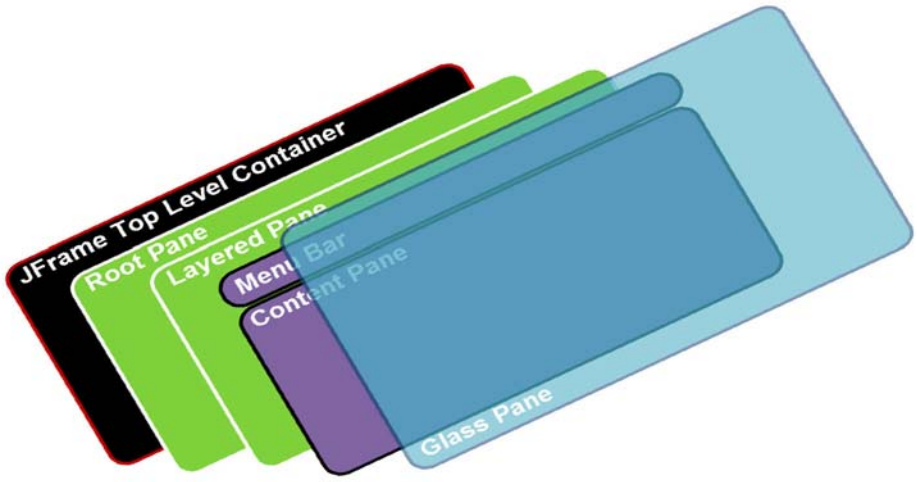


Abbildung 5-4: Aufbau eines Top-Level-Containers - JFrame[38]

[37] eigene Darstellung

[38] eigene Darstellung

Die *Layered-Pane* besitzt die Besonderheit, dass in ihr befindliche Komponenten wie auf einer Art Z-Achse über Konstanten angeordnet werden können. Eine Komponente lässt sich auf diese Weise über eine andere tiefer gelegene platzieren.

Für die Annordnung der in die Content-Pane eingefügten Komponenten, bietet die JFC sogenannte *LayoutManager* an. Mit Hilfe dieser Klassen lässt sich über qualitative Angaben jedes einzelne Element innerhalb eines Containers anordnen, sodass eine recht aufwendige Platzierung über absolute Werte nicht mehr erforderlich ist. Der Benutzer hat die Möglichkeit eigene *LayoutManager* zu erstellen bzw. auf vordefinierte der JFC zurückzugreifen. Die Content-Pane, wie auch alle anderen Container, sind bereits mit entsprechenden *LayoutManagern* vorinitialisiert. Diese können aber über entsprechende Methoden bzw. auch schon mit der Instanzierung des Containers neu gesetzt werden.

Die Content-Pane benutzt als Standard-*LayoutManager* das *BorderLayout* (Abbildung 5-5).

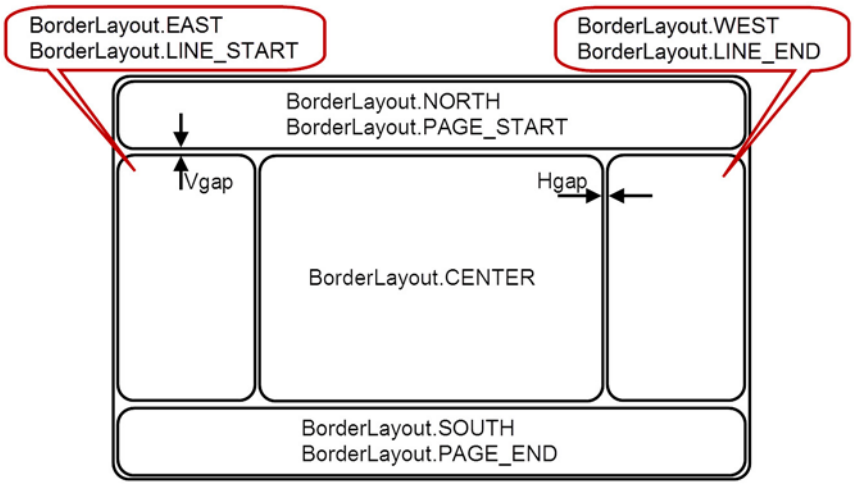


Abbildung 5-5: Prinzip des BorderLayout[39]

Das *BorderLayout* kennzeichnet sich dadurch aus, dass es den Container in fünf Bereiche aufteilt. Unter Angabe der entsprechenden *BorderLayout*-Konstante kann das GUI-Element in dem zur Konstante korrespondierenden Bereich im Container gesetzt werden. Die Abstände zwischen den einzelnen Bereichen kann ebenfalls über die Integer-Variablen *Hgab* und *Vgab* gesteuert werden. Ab der Java Version 1.4 hat man den bisherigen Konstanten eine verständlichere Entsprechung dazugestellt.

- **BorderLayout.NORTH** ⇨ **BorderLayout.PAGE_START**
- **BorderLayout.EAST** ⇨ **BorderLayout.LINE_START**
- **BoderLayout.WEST** ⇨ **BorderLayout.LINE_END**

[39] eigene Darstellung

- **BorderLayout.SOTH ⇨ BorderLayout.PAGE_END**

Die folgende Abbildung (Abbildung 5-6) zeigt das Hauptfenster der JavGo-GUI. Für die Content-Pane des Hauptfensters wurde der Standard-*LayoutManager* beibehalten.

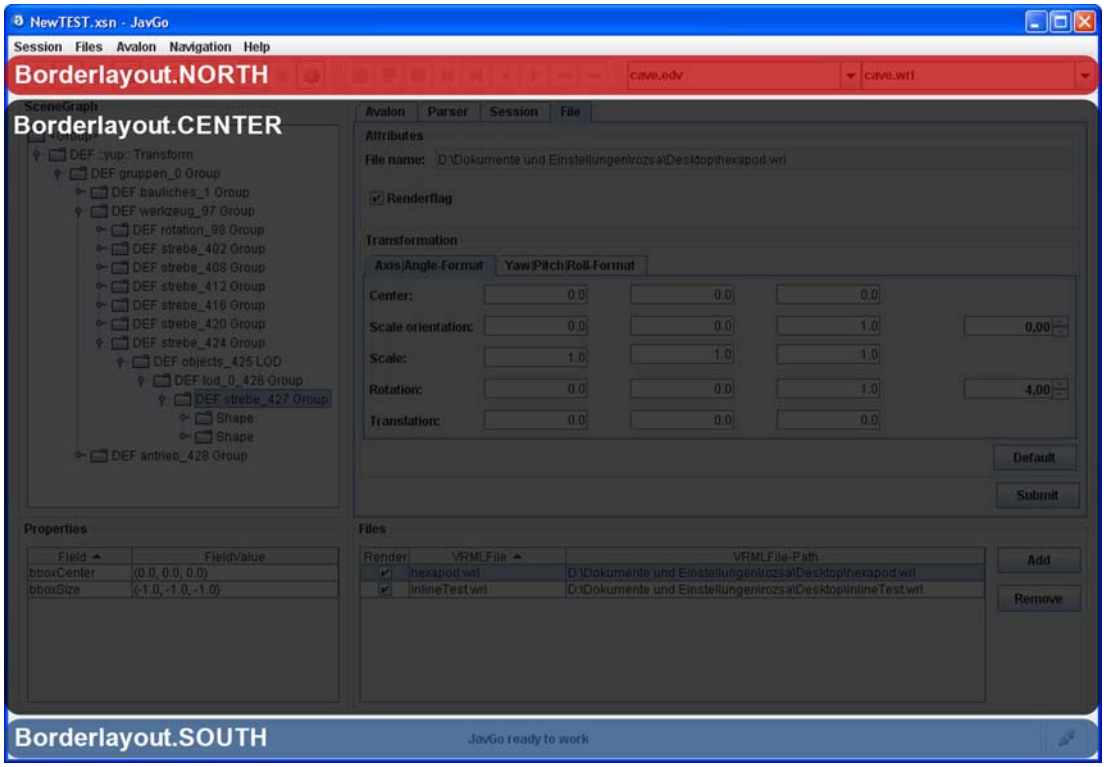


Abbildung 5-6: Standard-LayoutManager(*BorderLayout*) der Content-Pane des Hauptfensters JavGo-GUI[40]

Die einzelnen Bereiche mit der das *BorderLayout* die Content-Pane aufteilt sind nochmals farbig hinterlegt und mit der entsprechenden *BorderLayout*-Konstante gekennzeichnet. In nördlicher Ausrichtung befindet sich die *ToolBar.* Der zentrale Bereich bleibt dem Container *mainPnl* vorbehalten. Dieser gruppiert alle wichtigen Komponenten, die in ihrer Gesamtheit die JavGo-GUI ausmachen. Der Center-Bereich nimmt sich immer den kompletten verfügbaren Platz. Nicht benutzte *BorderLayout*-Bereiche werden deshalb durch den Center-Bereich verdrängt, sodass der Benutzer nicht darauf angewiesen ist, sämtliche Bereiche des *BorderLayouts* zu benutzen. Die Statusleiste befindet sich in südlicher Ausrichtung.

Hinter dem im Center-Bereich gelegenen *mainPnl* verbirgt sich die SWING Komponente *JPanel*. Es handelt sich dabei um einen leichtgewichtigen Container, der seinerseits wieder leichtgewichtige Komponenten aufnehmen kann. Die Anordnung der Komponenten erfolgt über den zugewiesenen *LayoutManager*. Standardmäßig benutzt das *JPanel* das *FlowLayout* als Lay-

[40] eigene Darstellung

outManager. Da aber für das *mainPnl* das *GridBagLayout* zur Anwendung kam (Abbildung 5-7), soll auf das *FlowLayout* nicht weiter eingegangen werden.

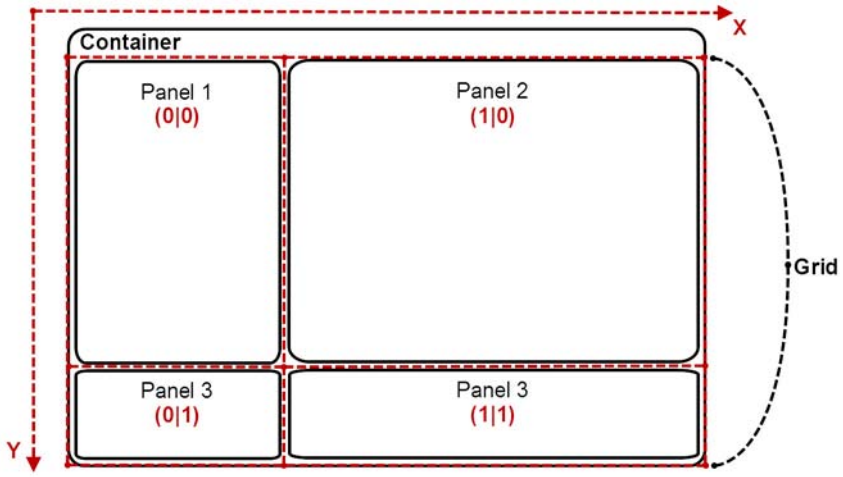


Abbildung 5-7: Prinzip des *GridBagLayout* mit *mainPnl* Container als Grundlage[41]

Das *GridBagLayout* gehört zu den flexibelsten und zugleich komplexesten *LayoutManagern* die SWING zur Anordnung von Komponenten anbietet. Das Prinzip des *GridBagLayout* ist Aufteilung des Containers in einer Art Netz, in dem die Komponenten in die entsprechenden Zellen platziert werden können. Über sogenannte *GridbagConstraints* besteht darüber hinaus die Möglichkeit, Darstellungseigenschaften für die Komponente in der Zelle festzulegen. Etwaige Eigenschaften wären:

- **Komponentenanzeige über mehrere Gitterzellen hinweg**
- **Zellen mit unterschiedlicher Breite und Höhe**
- **Anordnung der Komponente in der Zelle**
- **Komponente soll Zelle voll ausfüllen etc.**

Die gesetzten Eigenschaften werden beim Hinzufügen der Komponente in den Container einfach mit in Form einer *GridbagConstraints* Instanz übergeben.

Folgende Komponenten werden im *mainPnl* über das *GridBagLayout* angeordnet (Abbildung.5-8).

[41] eigene Darstellung

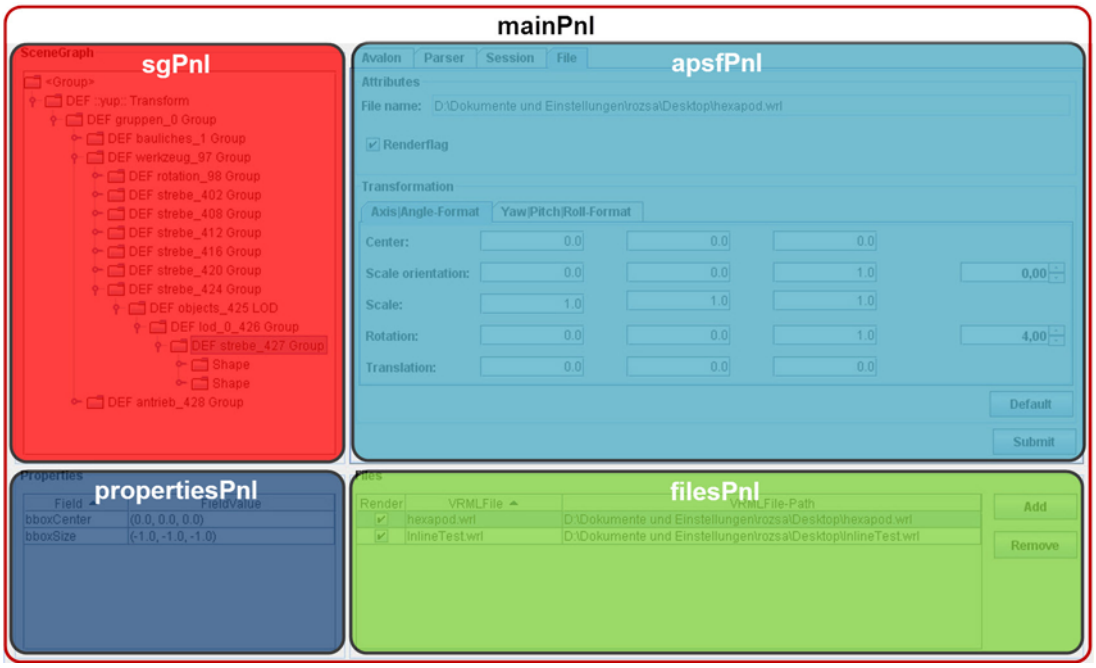


Abbildung 5-8: Anordnung der Komponenten über das *GridBagLayout* innerhalb des *mainPnl*[42]

- ***sgPnl* – Zelle(0,0)**

Das *sgPnl* symbolisiert den Container der das Tree-Element zur Visualisierung der Knotenstruktur des 3D Objekts beinhaltet.

- ***propertiesPnl* – Zelle(1,0)**

Im Container *propertiesPnl* befindet sich das Tabellen-Element zur Aufführung der Knoteneigenschaften. Auf der Grundlage der im *sgPnl* visualisierten Knotenstruktur erfolgt die Anzeige der Feldnamen und dazugehörigen Feldwerte des entsprechend ausgewählten Elements.

- ***apsfPnl* – Zelle(0,1)**

Das Tab-Element mit den Tabs für die Statusmeldungen des Avalon-Systems und VRML-Parsers, sowie für die globalen und 3D Objekt spezifischen Einstellungen bezüglich der Sichtbarkeit und Transformation, befindet sich im *apsfPnl* Container.

- ***filesPnl* – Zelle(1,1)**

Der Container *filesPnl* nimmt das Tabellen-Element auf, welches die in die Grundszenerie geladenen 3D-Objekte aufführt.

[42] Eigene Darstellung

## 5.1.2 Tabellen

In diesem Punkt soll auf die Tabelle als GUI-Komponente, repräsentiert durch das SWING-Element *JTable*, näher eingegangen werden. Die *JavGo*-Applikation verwendet das *JTable*-Element genau dreimal.

- **Files-Tabelle ⇨ Führt die in die Session geladenen VRML-Modelle, vorhanden als .wrl Dateien, in einer Tabelle auf**
- **AEV-Tabelle ⇨ Listet die Additional Environment Variabeles (AEV), also die zusätzlichen Umgebungsvariablen für den externen Avalon-Prozess in einer Tabelle auf**
- **Properties-Tabelle ⇨ Führt die Felder mit den entsprechenden Feldwerten in einer Tabelle auf, in Abhängigkeit vom markierten VRML-Knoten in der visualisierten Knotenstruktur**

Wie im Punkt 5.1 bereits erwähnt, wird mit SWING die MVC-Architektur für die einzelnen GUI-Elemente eingeführt. In Bezug auf die Tabelle bedeutet dies, dass das *JTable*-Objekt an sich keine Daten enthält, sondern nur für die Visualisierung der Daten verantwortlich ist. Die *JTable* fungiert also hier nur als View-Bestandteil innerhalb der MVC-Architektur. Die eigentlichen Daten liegen in einem entsprechenden Model-Objekt im Backendbereich. Die *JTable* benutzt standardmäßig ein sogenanntes *DefaultTableModel* als Datenhalter. Zusätzlich hat der Programmierer aber die Möglichkeit über das Erben von der Klasse *AbstractTableModel* oder über das Implementieren des Interface *TableModel* ein eigenes, individuell angepasstes Model zu erzeugen.

Die folgende Abbildung (Abbildung 5-9) zeigt den Aufbau einer GUI-Komponente des Typs *JTabel*.

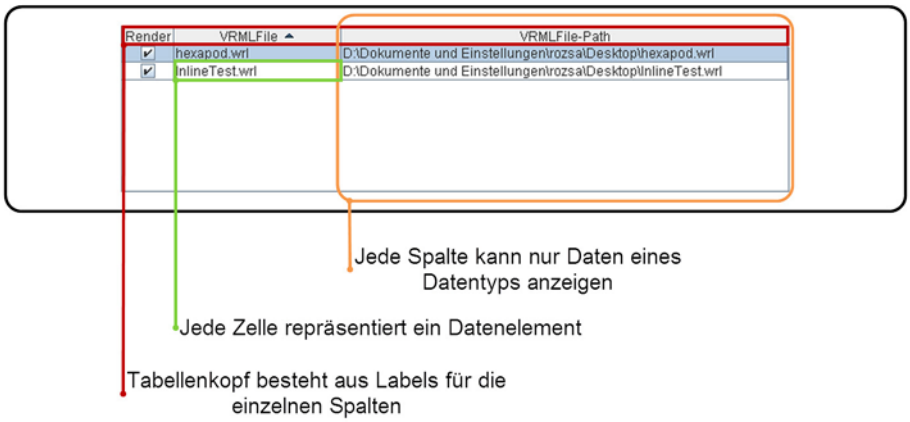


Abbildung 5-9: Tabelle – *JTable* SWING Element[43]

[43] eigene Darstellung

Die einzelnen Zellen innerhalb der Tabelle symbolisieren jeweils Datenelemente. Die eigentlichen Daten werden, wie schon erwähnt in einem so genannten Model-Objekt gehalten, dahinter verbirgt sich meist ein dynamisches Feld, jeder Eintrag in diesem Feld bildet genau eine Zeile in der Tabelle. Dabei kann der Eintrag wieder aus einem Feld oder Objekt bestehen. Bei der Files-Tabelle zum Bespiel besteht der Eintrag aus einem Objekt. Das Objekt besitzt drei Variablen, die je genau ein Datenelement für die Zelle innerhalb einer Zeile ausmachen. Der Tabellenkopf besteht aus lauter Label's, die die Spaltenüberschriften repräsentieren. Jede Spalte kann nur Daten eines Datentyps anzeigen bzw. visualisieren. Der Grund dafür ist, dass die Visualisierung über sogenannte Cell-Renderer verarbeitet wird. Für jeden Datentyp ist bei der View, also dem *JTable* Objekt, ein spezieller Cell-Renderer registriert. Er gibt der *JTable* letztendlich eine Anweisungsvorschrift, wie sie die entsprechenden Daten zu visualisieren hat.

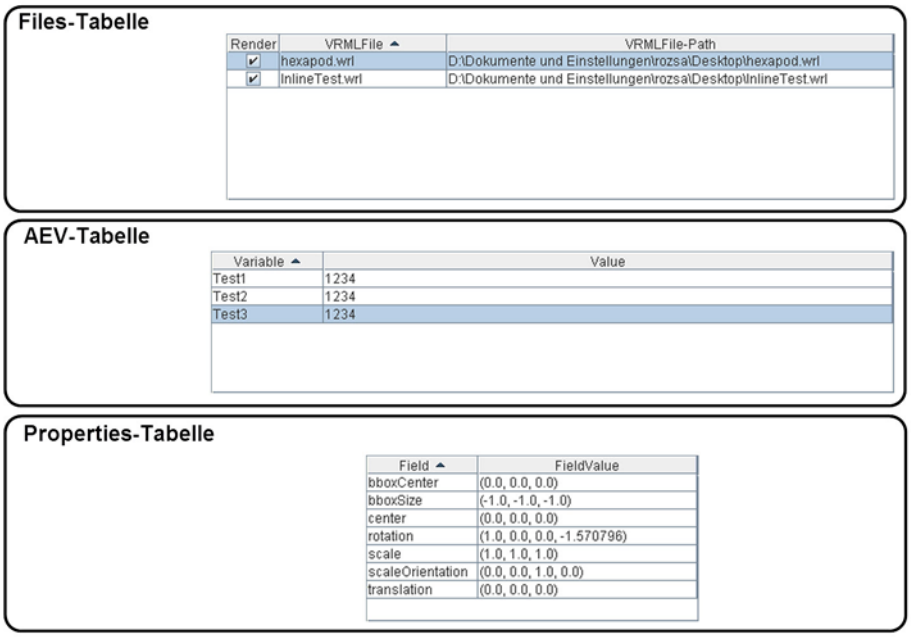


Abbildung 5-10: Sämtliche in *JavGo* verwendete Tabelle[44]

Die in *JavGo* verwendeten Tabellen (Abbildung 5-11) benutzen bis auf die Properties-Tabelle eigene, individuell angelegt Model's zur Datenhaltung. Auf die Model's für die jeweiligen Tabellen soll im Punkt 5.6.noch genauer eingegangen werden. Zu erwähnen wäre noch die Problematik der Sortierung der Daten innerhalb der Tabelle. Wie in der Abbildung 5-11 ersichtlich, sortieren die Tabellen die Daten aufsteigend.

- **Files-Tabelle ⇨ nach den Dateinnamen der .wrl Dateien (1. Spalte)**
- **AEV-Tabelle ⇨ nach den Namen der Variablen (0. Spalte)**
- **Properties-Tabelle ⇨ nach den Namen der Felder (0. Spalte)**

[44] eigene Darstellung

Auch hier kommt wieder das MVC Prinzip zum tragen, denn nur in der View, in dem Fall dem *JTable*, werden die Daten sortiert, dass Model bleibt davon unberührt. Die *JTable* benutzt standardmäßig einen *DefaultRowSorter*, der die zu visualisierenden Daten sortiert. Per default sind über diesen Standard-Sortierer alle Spalten der Tabelle auf sortable gestellt, das bedeutet, dass mit einem Mouse-Klick in den jeweiligen Spaltenkopf, die Daten in der View entweder aufsteigend oder absteigend sortiert werden. Die Datenelemente in den Zellen der Spalten müssen aber als solches vergleichbar sein, um eine Sortierung vornehmen zu können. Die Datentypen String, Long, Float, Double und Integer zum Beispiel implementieren das Interface Comparable, dadurch erhalten sie die Methode *compareTo()*. Über diese Methode kann der Sortierer die Datenelemente miteinander vergleichen und eine Sortierung vornehmen.

Bei den Tabellen in der *JavGo*-GUI wurden eigene Sortierer des Typs *TableRowSorter* erstellt. Über diese *TableRowSorter* Objekte wurde die Möglichkeit, der Sortierung per Klick in den Spaltenkopf abgestellt und stattdessen eine feste Sortierung vorgegeben. Über eine Liste mit sogenannten SortKeys wurde dem *TableRowSorter* Objekt die jeweilige Sortiervorschrift zugewiesen.

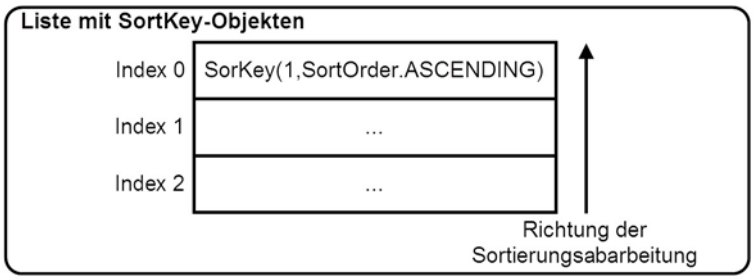


Abbildung 5-11: Liste mit SortKey-Objekten[45]

Um ein *SortKey* Objekt zu erzeugen, muss dem *SortKey* Konstruktor der Spaltenindex und die Sortierungsordnung übergeben werden. Der Spaltenindex gibt an, nach welcher Spalte die Zeilen in der Tabelle geordnet werden sollen. Hinter der Sortierungsordnung verbirgt sich eine Konstante, die angibt, ob die Daten in der betreffenden Spalte und damit die Zeilen aufsteigend, absteigend oder unsortiert beleiben sollen. Der *TabelRowSorter* arbeitet die Liste mit den *SortKey* Objekten von unten nach oben ab (Abbildung 5-12). Also der erste Sortierungsvorgang geschieht mit dem SortKey-Objekt welches den größten Listenindex aufweist, der letzte Sortierungsvorgang wird durch das *SortKey* Objekt mit dem Listenindex 0 abgeschlossen.

Der Sortierungsvorgang an sich soll kurz anhand der Files-Tabelle gezeigt werden. Dem *TableRowSorter* Objekt wurde:

[45] eigene Darstellung

- **das Model zugewiesen, dessen Daten für die View sortiert werden sollen (Model der Files-Tabelle)**
- **die Liste mit den SortKeys zugewiesen, in dem Fall befindet sich in der Liste nur ein SortKey Objekt ⇨ Zeilen anhand der 1. Spalte aufsteigend sortieren**

Nachdem die Daten in das Model der Files-Tabelle eingetragen wurden, holt sich der *TableRowSorter* über die im Model implementierte *getValueAt()* Methode die entsprechenden Daten aus dem Model. Er übergibt der Methode jeweils den Spaltenindex und den Zeilenindex. Der Spaltenindex ist durch das *SortKey* Objekt vorgegeben. Der Zeilenindex richtet sich hingegen nach den im Model eingetragenen Datensätzen. Im Fall der Files-Tabelle traversiert der *TableRowSorter* nun die einzelnen Zeilen durch und erfragt dort jeweils die Daten aus der 1. Spalte bzw der Spalte mit dem Index 1. Intern werden nun diese Daten über die *compareTo()* Methode verglichen. Anschließend erfolgt die Eintragung in die View. Zu erwähnen ist noch, dass das *TableRowSorter* Objekt, nachdem ihm das Model der Tabelle zugewiesen wurde, über die im Model implementierte Methode *getColumnClass()* den Datentyp der entsprechenden Spalte abruft. Somit weiß das TableRowSorter Objekt in Bezug auf die Files-Tabelle, dass es in der 1. Spalte nur Daten des Typs String zu erwarten hat. Wie bereits erwähnt implementiert String von Hause aus schon das Interface Comparable und somit die *compareTo()* Methode.

Mit der Sortierung ergibt sich aber wieder ein neues Problem, denn die Daten im Model und die visualisierten Daten in der View sind somit nicht mehr deckungsgleich. Ein etwaiges Ansprechen von Daten im Model über den selektierten Zeilenindex in der View würde dann zwangsläufig zu einem falschen Ergebnis führen. Die Java-Entwickler haben für dieses Problem aber sogenannte Konvertierungs-Methoden implementiert, die über das *JTable* Objekt benutzt werden können. Diese Methoden geben dem Benutzer die Möglichkeit, die Indizes von Spalte oder Zeile zwischen View und Model hin und her zu konvertieren.

- ***convertColumnIndexToModel*(viewColumnIndex) ⇨ SpaltenIndex(View) → SpaltenIndex(Model)**
- ***convertColumnIndexToView*(modelColumnIndex) ⇨ SpaltenIndex(Model) → SpaltenIndex(View)**
- ***convertRowIndexToModel*(viewRowIndex) ⇨ ZeilenIndex(View) → ZeilenIndex(Model)**
- ***convertRowIndexToView*(modelRowIndex) ⇨ ZeilenIndex(Model) → ZeilenIndex(View)**

### 5.1.3 Tree

Zur Darstellung von Daten in einer Baumstruktur, mit der anschließenden Möglichkeit durch diese zu Navigieren bzw. Bearbeitungen, Editierungen vornehmen zu können, bietet Java das SWING Element *JTree* an. Die *JavGo*-Applikation benutzt die *JTree* Komponente, um auf Grundlage der vom VRML-Parser erzeugten Daten, den Szenegrafen in einer Baumstruktur abzubilden (Abbildung 5-12).

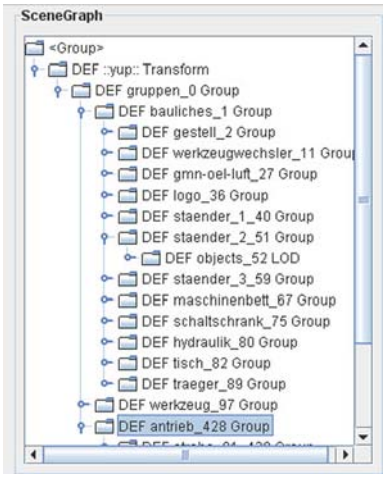


Abbildung 5-12: Tree zur Visualisierung des Szenegrafen[46]

Wie bei der *JTable*, setzte sich auch bei der *JTree* Komponente der Aufbau nach dem MVC Prinzip fort. Dabei symbolisiert das *JTree* Objekt innerhalb des MVC die View und ein entsprechendes TreeModel Objekt, dass zugrundeliegende Model. Damit eine hierarchische Visualisierung in einer Baumstruktur erfolgen kann, muss das TreeModel die Daten so aufbereiten, dass die View, in dem Fall das *JTree* Objekt, diese in einem Baum abbilden kann. Zur näheren Erläuterung, sollen in diesem Zusammenhang grundsätzliche Begriffe angeführt werden.

- **Ein Element eines Baums wird als Knoten bezeichnet.**
- **Die einzelnen Knoten können wiederum Unterknoten (Kindknoten) beinhalten.**
- **Das dem Kindknoten übergeordnete Element wird als Vaterknoten oder Elternknoten bezeichnet**
- **Das einzige Element, was Vaterknoten sämtlicher im Baum enthaltenen Elemente ist und selber keinem anderen Knoten untergeordnet ist, wird als Wurzel bzw. Wurzelknoten bezeichnet.**
- **Ein Baum kann nur einen Wurzelknoten besitzen.**
- **Knoten, die keine Kindknoten unter sich gruppieren, werden als Blätter bezeichnet**

[46] eigene Darstellung

Ein jegliches Model, was als TreeModel fungiert, implementiert das Interface TreeModel. Durch das Interface werden der TreeModel Klasse Methodenrümpfe bereitgestellt, über deren konkrete Implementierung das *JTree* Objekt die Daten so interpretieren kann, dass diese letztendlich auch in einer Baumstruktur abgebildet werden können. Auch hier besteht wieder die Möglichkeit, ein Standardmodel (*DefaultTreeModel*) zu benutzen. In diesem Fall müssen die Daten mit Hilfe sogenannter *DefaultMutableTreeNode* Objekte abgebildet werden, um darüber den kompletten Wurzelknoten aufzubauen. Der *JTree* der *JavGo*-Applikation verwendet zur Interpretierung der vom VRML-Parser gelieferten Daten, ein eigens erstelltes TreeModel. Dessen genaue Erläuterung ist Aufgabe des Punkt 5.5.1.2, sodass an dieser Stelle nur grob die Thematik des TreeModel abgehandelt wird.

Anhand folgender Funktionalitäten, die durch das TreeModel bereitgestellt werden, kann das *JTree* Objekt die Daten in der Baumstruktur abbilden.

- **Abfrage des Wurzelknoten**
- **Abfrage, ob es sich bei den jeweiligen Knoten um ein Blatt handelt.**
- **Abfrage, wie viel Kindknoten durch den jeweiligen Elternknoten gruppiert werden.**
- **Abfrage des entsprechenden Kindknotens**
- **Abfrage, an welcher Stelle sich der Kindknoten in Bezug zu seinem direkten Elternknoten befindet.**

Die für die Visualisierung großer Strukturen vorgesehenen SWING Elemente (*JList*, *JComboBox*, *JTable* und *JTree*) benutzen alle für die grafische Darstellung der Komponenten innerhalb der Struktur, sogenannte Cell-Renderer. Als Default-Renderer ist immer eine von *JLabel* abgeleitete Komponente voreingestellt. Was Bedeutet, dass eine jegliche Komponente innerhalb der Struktur, für die Visualisierung als *JLabel* aufgefasst wird. Wird vom Programmierer kein eigener Renderer für den *JTree* festgelegt, erfolgt die Benutzung des *DefaultTreeCellRenderer*. Die Bezeichnung der Knoten innerhalb des Baums, erstellt der *DefaultTreeCellRenderer* über den Aufruf der toString() Methode. Wie oben schon erwähnt, erhält das *JTree* Objekt vom TreeModel Objekt unter anderen die Funktionalität, Wurzel und Kindknoten abzufragen. Die dafür zuständigen Methoden liefern als Rückgabe einen Wert des Typs Object. Jede von Object abgeleitete Klasse ererbt die *toString()* Methode. Der *DefaultTreeCellRenderer* macht diesbezüglich also nichts anderes, als den über die *toString()* Methode erzeugten Text des zurückgelieferten Objekts intern per *setText()* Methode als Bezeichnungstext des *JLabels* zu setzen. Um an dieser Stelle eine sinnvolle Ausgabe zu ermöglichen, ergibt sich für den Programmierer die Notwendigkeit, die *toString()* Methode jeder Baumkomponente zu überschreiben, da sonst nur die Stringrepräsentation des jeweiligen Objekts angezeigt wird.

Für die Anzeige des Szenegrafen im *JTree* ergab sich an diesem Punkt ein Problem, da die vom VRML-Parser gelieferten Objekte die *toString()* Methode ursprünglich nicht überschreiben. Um hier dennoch zu vernünftigen Ausgaben zu kommen, muss die *convertValueToText()* Methode des *JTree* Objekts überschrieben werden. Der *DefaultTreeCellRenderer* ruft diese Methode des *JTree* Objekts auf, um auf Grundlage des Rückgabewertes den Text des *JLabels*

zu setzen. Normalerweise ist die *convertValueToText()* Methode so implementiert, dass es im Methodenrumpf zur Ausführung der *toString()* Methode des übergebenen Objekts kommt und deren Ergebnis als Rückgabewert dient. Für die Anwendung des *JTree* in der *JavGo*-Applikation, wurde die *convertValueToText()* so überschrieben, dass im Methodenrumpf geprüft wird, ob das übergebene Objekt (geparster Knoten) eine DEF Bezeichnung besitzt oder nicht. Auf Grundlage dieser getroffenen Entscheidung erfolgt entweder nur die Rückgabe des Knotentyps oder der gesamten *DEF*-Bezeichnung mit Knotentyp als String. Die vom VRML-Parser gelieferten Objekte sind mit Methoden ausgestattet, die eine Abfrage der *DEF*-Bezeichnung und des Knotentyps gewährleisten.

- ***objname()* ⇨ liefert die DEF Bezeichnung als String**
- ***nodeName()* ⇨ leifert den Knotentyp als String**

## 5.2 Session- und VRMLFile-Objekt

Sowohl die Session, als auch die in die virtuelle Welt geladenen 3D-Objekte brauchen eine javatechnische Repräsentation, um sie mit Hilfe von Java bzw. der in Java umgesetzten *JavGo*-Applikation verarbeiten zu können. Der Ansatz für den Aufbau der Klassenrepräsentation folgt dabei dem Prinzip, nach welchem der Szenegraf aufgebaut sein muss, um folgende Funktionalitäten zu gewährleisten.

- **Einbinden und Steuern einer globalen Transformation und Sichtbarkeit**
- **Einbindend von 3D-Objekten in die virtuelle Welt**
- **Einbinden und Steuern einer objektbezogenen Transformation und Sichtbarkeit**

Um eine globale Transformationssteuerung zu ermöglichen, wird in den Wurzelknoten der virtuellen Welt ein Transform-Knoten als Kind-Knoten eingebunden. Die jeweils in die Grundszenerie geladenen 3D-Modelle werden explizit per Inline-Knoten eingehängt. Jeder Inline-Knoten ist wiederum von einem Transform-Knoten ummantelt, der eine objektbezogene Transformationsstueerung gestattet. Die einzelnen 3D-Objekte tauchen also nicht direkt als Kinder unter dem Wurzelknoten der virtuellen Welt auf, sondern sind mit ihren entsprechenden Transform-Knoten über den Knoten zur allgemeinen Transformationssteuerung im Szenegraf angelegt. Wie bereits erwähnt, implementiert das Avalon-System unter anderen für die VRML-Gruppierungsknoten ein sogenanntes *render* Feld, über welches Teilbäume des Szenegrafen aus dem Renderingprozess ausgeschlossen werden können. Die Transform-Knoten lassen also nicht nur eine Transformationssteuerung zu, sondern geben über das Setzen des *render* Feldes auch dem Benutzer die Möglichkeit, die jeweils im *chlidren* Feld des Transform-Knoten gruppierten Knoten aus dem Visualisierungsprozess auszuklammern.

Die folgende Abbildung (Abbildung 5-13) veranschaulicht das bereits Erklärte nochmals zum besseren Verständnis.

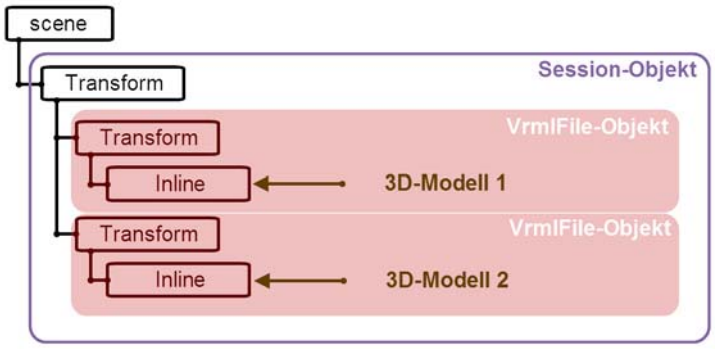


Abbildung 5-13: Aufbau des Szenegraf und entsprechende Java-Objekt- Repräsentation

Das Session-Objekt stellt das in Java implementierte Abbild zum globalen Transform-Knoten dar. Die von der *VrmlFile*-Klasse erzeugten Objekte bilden die entsprechenden Transform-Knoten ab, in deren *children* Feld die Inline-Knoten zur Einbindung der jeweiligen 3D-Modelle in die Grundszenerie sitzen.

Um später über das EAI-Interface eine Steuerung der Transformation und Sichtbarkeit vorzunehmen, muss ein expliziter Zugriff auf die entsprechenden Transform-Knoten erfolgen. Da ein derartiger Zugriff nur über die jeweils festgelegte *DEF*-Bezeichnung erlangt werden kann, wird jeder Transform-Knoten mit einer eindeutig definierten *DEF*-Bezeichnung in den Szenegrafen integriert.

- **Transform-Knoten zur globalen Transformations- und Sichtbarkeitsstuerung ⇨ DEF GlobalTransform{}**
- **Transform-Knoten für die objektbezogene Transformations- und Sichtbarkeitsstuerung ⇨ *DEF File ID Transform Transform{children[DEF File ID Inline Inline{exportNameSpace TRUE url [“ absoluter Pfad des 3D-Objekts ”]}]}***

Damit das Session-Objekt die unter dem Transform-Knoten zur globalen Transformationsstuerung gruppierten Kind-Knoten halten kann, ist das Objekt mit einem dynamischen Feld ausgestattet, welches Elemente des Typs *VrmlFile* aufnehmen kann. Die in der *DEF*-Bezeichnung für die objektspezifischen Transform-Knoten und der gleichfalls unter diesen eingehängten Inline-Knoten verwendete ID, ist deckungsgleich mit dem Element-Index, unter welchem das korrespondierende *VrmlFile*-Objekt im Vector (dynamisches Feld) des Session-Objekts eingetragen ist.

Sowohl die *Session* als auch die *VrmlFile*-Klasse bilden Subklassen von *SceneNode*, dadurch werden folgende Funktionalitäten ererbt.

- **Objektvariable für die Speicherung der Referenz auf den entsprechenden Transform-Knoten**
- **Objektvariablen für die Speicherung der einzelnen Transfromationsparameter**
- **Getter- und Setter-Methoden, um die in den entsprechenden Objektvariablen gehaltenen Transformationsparamter zu setzen bzw. abzufragen**
- **Methode, um die kompletten Transformationsparameter auf null zu setzen**
- **Methoden, um in den entsprechenden Eltern-Knoten eingetragen bzw. entfernt zu werden**
- **Abstrakte Methoden über deren konkrete Implementierung die Referenz auf den entsprechenden Eltern-Knoten und Transform-Knoten geliefert werden kann.**

Um das Eintragen in das *chlildren* Feld des Eltern-Knoten zu gewährleisten, implementiert die SceneNode-Klasse die abstrakte Methoden *parentNode()*. Die konkrete Implementierung dieser Methode in der Session-Klasse, wie auch in der VrmlFile-Klasse ist so umgesetzt, dass immer eine Referenz auf den entsprechenden Eltern-Knoten als Rückgabe geliefert wird.

- **Session-Klasse ⇨ Wurzelknoten des Szenegrafen der visualisierten Szene ⇨ *DEF scene Scene***
- **VrmlFile - Klasse ⇨ Transform-Knoten zur globalen Transformationssteuerung ⇨ *DEF GlobalTransform***

Mit Hilfe der Referenz auf den Eltern-Knoten kann die direkte Eintragung in dessen *children* Feld erfolgen. Nach dem Eintragevorgang wird im Objekt selber einer Referenz auf den entsprechenden Transform-Knoten in einer Objektvariable abgelegt. Diese Referenz dient dazu, die Initialisierung der Objektvariablen für die Transformationsparameter durchzuführen bzw. um die Steuerung der Transformation und Sichtbarkeit zu gewährleisten. Sowohl die Erzeugung der Referenz auf den Eltern-Knoten, der Eintragevorgang selber, die Erzeugung der Referenz auf den Transform-Knoten und die Initialisierung der Variablen zur Haltung der Transformationsparameter, erfolgt über die Services des EAI-Interface bzw. über die EAI-Interface Verbindung selber.

## 5.3 Ausgelagerte Prozesse

### 5.3.1 Externer Avalon Prozess

Um aus einer Java Anwendung heraus Betriebssystem-Prozesse bzw. einen externen Prozess zu erzeugen, bietet Java die Klasse *ProcessBuilder* an. Sämtliche für den externen Prozess nötige Eigenschaften werden durch die *ProcessBuilder* Instanz verwaltet. Zu den Eigenschaften zählen:

- ***command***

Der *command* symbolisiert eine Liste in welche Elemente des Typs String abgelegt werden können. Die Liste dient dazu, den Programm-Namen mit den etwaigen Programm-Argumenten aufzunehmen. Das Element mit dem Index 0 repräsentiert dabei immer den Programm-Namen. Sowohl die Angabe des Programm-Namens als auch die Programm-Argumente sind system-spezifisch. Je nachdem, ob bereits eine entsprechende Umgebungsvariable gesetzt wurde, entscheidet sich, ob die Angabe des Programm-Namens an sich ausreicht, da ansonsten der komplette absolute Pfad hinterlegt werden muss. Die Argumente nehmen normalerweise je einen Eintrag in der *command* Liste ein, dennoch kann es auch hier sein, dass systembedingt sämtliche Argumente in einem einzigen Eintrag abgelegt werden müssen. Die *command* Liste wird beim Instanzieren des *ProcessBuilder* direkt an den Konstruktor übergeben.

- ***environment***

Die Methode *environment()* liefert ein Abbild sämtlicher im System konfigurierter Umgebungsvariablen in Form einer *Map*. Der Benutzer hat nun die Möglichkeit, die zurückgelieferte *Map* zu bearbeiten, indem neue Umgebungsvariablen hinzugefügt bzw. editiert werden können. Der *ProcessBuilder* konstruiert genau aus dieser *Map* die Umgebungsvariablen für den externen Prozess.

- ***working directory***

Der *ProcessBuilder* implementiert Methoden, um das Arbeitsverzeichnis mit welchem der externe Prozess arbeitet, zu erfragt bzw. zu setzen. Standardmäßig wird das Arbeitsverzeichnis des aktuellen Prozesses übernommen.

- ***redirectErrorStream***

Das mit dem Start der *ProcessBuilder* Instanz erzeugte *Process* Objekt, stellt drei Streams zu Verfügung, mit welchem der Benutzer zum einen die Ausgaben des externen Prozess auslesen und zum anderen Befehle zum externen Prozess schicken kann. Die *ProcessBuilder* Klasse

stellt nun die Möglichkeit zu Verfügung, den Ausgabestrom für die Normalmeldungen und Fehlermeldungen über den Schalter *redirectErrorStream* zusammenzulegen.

Um den über die Klasse *ProcessBuilder* erstellten externen Prozess zu starten, implementiert *ProcessBuilder* die Methode *start()*. Die Ausführung von *start()* liefert ein *Process* Objekt zurück, mit welchem der Benutzer die Streams handeln und den externe Prozess zerstören bzw. Zustände des externen Prozess erfragen kann.

Der Aufbau des externen Prozess zum Starten des Avalon-Systems gestaltet sich folgendermaßen (Abbildung 5-14).

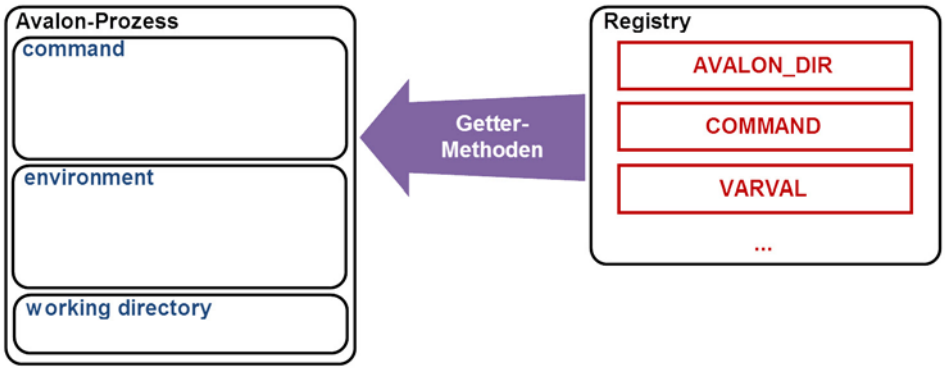


Abbildung 5-14: Prozessaufbau des externen Avalon-Prozess[47]

Die Programm-Argumente und der Programm-Name selber bilden sich aus denen unter dem Schlüssel *COMMAND* hinterlegten Werten in der Registry des Systems. Der Benutzer hat im Settings-Dialog die Möglichkeit, neben der Auswahl des Programm-Namen, standardmäßig *sav.exe*, noch zusätzlich Parameter für das Avalon-System festzulegen. Sämtliche im Settings-Dialog gemachten Einstellungen speichert *JavGo* unter den entsprechenden Schlüsseln in der Registry ab. Aus dem festgelegten Programm-Name und dem Avalon-Verzeichnis wird der absolute Pfad und sogleich erste Eintrag in der *command* Liste zusammengebaut. Der externe Prozess zum Starten des Avalon-Systems braucht den absoluten Programmpfad, um überhaupt starten zu können. Die Folgereinträge in der *command* Liste machen die definierten Parameter aus. Den Abschluss bildet die über das entsprechende *ComboBox*-Element ausgewählte Datei für die Visualisierungskonfiguration (EDV-Datei) und Grundszenerie (WRL-Datei). Dem Abbild der im System definierten Umgebungsvariablen in Form einer *Map*, werden zusätzlich die in der Registry unter dem Schlüssel *VARVAL* abgespeicherten Variablen und Werte hinzugefügt. Der Schlüssel VARVAL korrespondiert mit dem im Settings-Dialog und dort unter dem Punkt *Additional Environment Variable* in Form einer Tabelle visualisierten Variablen und Werte. Darüber hinaus wird zusätzlich noch der Pfad für die Bibliotheken und der Lizenzdatei in Gestalt einer

[47] eigene Darstellung

Umgebungsvariable der *Map* hinzugefügt. Als Arbeitsverzeichnis für den externen Prozess gilt das *bin* -Verzeichnis im Hauptordner des InstantReality-Frameworks.

Das nach dem Ausführen der *start()* Methode erzeugte *Process*-Objekt bietet nur recht eingeschränkte Funktionalitäten, die zudem keinerlei Spezifika in Bezug auf das extern gestartete Programm aufweisen. Damit verbinden sich zwei Probleme, vor denen der Benutzer unweigerlich nach dem Prozess-Start steht. Es besteht erstens keine Möglichkeit über das *Porcess*-Objekt zu erfragen, wann das externe Programm vollständig initialisiert ist. Und zweitens, lässt sich der externe Prozess nur über die *destroy()* Methode per *Hardkill* zerstören. Ein langsames Herunterfahren ist, sofern das externe Programm während des Betriebs nicht weiter per Kommandozeile steuerbar ist, nicht durchführbar. Für die Lösung des ersten Problems wurde folgender Ansatz umgesetzt. Nach dem Start des Avalon-System erfolgt in einer Schleife der Versuch eine Verbindung über das EAI-Interface aufzubauen. Diese Verbindung kommt erst dann zustande, wenn das Avalon-System vollständig initialisiert ist, sodass der geglückte Verbindungsaufbau als Gradmesser für die Initialisierung genutzt werden kann. Das zweite Problem muss leider so hingenommen werden, da das Avalon-System nach erfolgtem Start keinerlei Steuerung per Kommandozeile zulässt. Sowohl beim Start über den SAV als auch über den InstantPlayer wird das Herunterfahren des Avalon-Systems erst über das direkte Schließen des Player-Fensters in Gang gesetzt.

### 5.3.2 SwingWorker

Hinter dem *SwingWorker* verbirgt sich eine generische, abstrakte Klasse die mit der Java Version 1.6 eingeführt wurde, um zeitaufwendige Aufgaben (engl.: Tasks), die gleichzeitig Veränderungen an der grafischen Benutzeroberfläche vornehmen, mit Hilfe von Threads im Hintergrund auszuführen. Der Aufbau der GUI-Applikation erfolgt im sogenannten Event-Dispatch-Thread (EDT), ein Verzicht auf Auslagerung von zeitintensiven Berechnungen und der damit verbunden Abarbeitung im EDT, hätte ein Blocken der gesamten GUI zu Folge. Java legt diesbezüglich in seiner Spezifikation ein Paradigma für Multithreading-Applikationen fest, welches nachfolgend erwähnt werden soll.

- **Zeitintensive Rechenoperationen sollen nicht vom EDT durchgeführt werden**
- **Der Zugriff auf die SWING Komponenten soll nur vom EDT aus erfolgen**

Der Benutzer hat nun die Möglichkeit durch die Erzeugung von sogenannten Worker-Threads, in welche die rechen- und zeitintensiven Aufgaben ausgelagert sind, die oben aufgeführten Vorgaben umzusetzen bzw. einzuhalten. Wie bereits erwähnt, ist die *SwingWorker* Klasse als abstrakt definiert, die Objekterzeugung kann also nur über eine Subklasse von *SwingWorker* erfolgen. Zentral ist hier die abstrakte Methode *doInBackground()*, deren konkrete Implementierung die Hintergrundaufgabe kapselt. Mit dem Starten des Worker-Thread aus dem EDT heraus, erfolgt im Worker-Thread die Ausführung der *doInBackgorund()* Methode des übergebenen

*SwingWorker* Objekts. Mit der Abarbeitung im EDT wird sofort nach der Übergabe des *SwingWorker* Objekts bzw. dem Starten des Worker-Threads per *execute()* Methode fortgefahren. Die *SwingWorker* Klasse bringt neben ihrer abstrakten Definition noch zwei generische Typ-Variablen mit. Die erste Typ-Variabel ist an den Rückgabewert der *doInBackground()* Methode gekoppelt. Auf dieses Endergebnis kann über die ebenfalls durch die *SwingWorker* Klasse implementierten Methoden *done()* und *get()* zugegriffen werden. Die Besonderheit besteht darin, dass beide Methoden vom EDT ausgeführt werden. Die zweite Typ-Variable steht für den Datentyp der erzeugten Zwischenergebnisse. Über die Methode *publish()* der *SwingWorker* Klasse, können innerhalb der Abarbeitung der *doInBackground()* Methode Zwischenergebnisse veröffentlich werden über die der EDT asynchron durch den Aufruf der *process()* Methode zugreifen kann.

Die *JavGo*-Applikation benutzt die *SwingWorker* Klasse um folgende Aktivitäten mit Hilfe von Worker-Threads im Hintergrund abarbeiten zu lassen.

- **Ausgabe der Statusmeldungen des Avalon-Systems in ein Document-Element**
- **Ausgabe der Statusmeldungen des VRML-Parser in eine Document-Elemente**
- **Aufbau der EAI-Verbindung**
- **Aufbau der EAI-Verbindung mit anschließenden Update der bereits in die Session geladenen 3D Objekte**
- **Laden von 3D Objekten in die Session**

Der SwingWorker für die Ausgaben der Stausmeldungen des Avalon-Systems und VRML-Parsers gleicht sich im Aufbau und soll folgend erklärt werden (Abbildung 5-15).

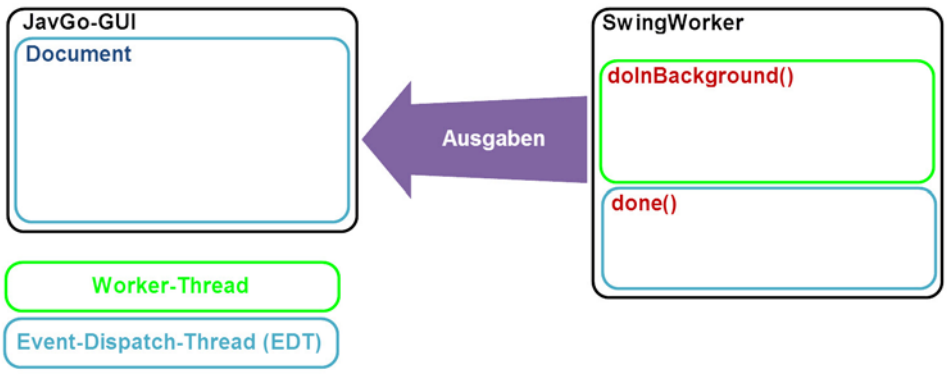


Abbildung 5-15: SwingWorker für die Ausgeben der Statusmeldungen ⇨ Avalon-System, VRML-Parser[48]

Die Prozedur für die Ausgaben der Statusmeldungen auf das entsprechende *Document*-Element in der JavGo-GUI befindet sich in der *doInBackground()* Methode der jeweiligen *SwingWorker* Instanz. Die Abarbeitung der Ausgaben erfolgt dadurch nach dem Starten per

[48] eigene Darstellung

*execute()* Methode durch den Worker-Thread selber. Ein Blocken der Gesamt-GUI während der Ausgabe wird damit verhindert. Die durch die *SwingWorker* Klasse implementierte Methode *done()* wird vom EDT nach beenden des Worker-Thread aufgerufen. Das Beenden erfolgt entweder durch die erfolgreiche Abarbeitung der *doInBackgorund()* Methode oder von Außen über die *cancel()* Methode. Die *SwingWorker* Instanz zur Auslagerung der Statusmeldungen des Avalon-Systems erhält eine Referenz auf das *Document*-Element, in welches die Ausgabe erfolgen soll und eine Referenz des *Stream*-Objekts, in welche die Ausgaben des Avalon-Systems gebündelt werden. In einer Schleife wird der *Stream* fortwährend in der *doIn-Background()* Methode ausgelesen und auf das *Document*-Element ausgegeben. Das Beenden des Vorgangs geht einher mit dem Stoppen des Avalon-System durch den Benutzer. Über die *cancel()* Methode wird der Worker-Thread abgebrochen und durch die dadurch in Gang gesetzte Ausführung der *done()* Methode eine Abbruchausgabe in das *Document*-Element vorgenommen. Das für die Auslagerung der Statusmeldungen des VRML-Parsers zuständige SwingWorker Objekt bekommt neben der Referenz des entsprechenden Document-Elements noch einen

Das Parsing findet immer dann statt, wenn sich die Selektion innerhalb *filesTbl*-Tabelle ändert bzw. wird immer das aktuell markierte 3D Objekt durch den VRML-Parser analysiert. Da die Parsing Vorgang innerhalb der *doInBackground()* Methode abgearbeitet werden soll

Den Aufbau der SwingWorker Instanzen zur Auslagerung des Verbindungsaufbaus über das EAI-Interface und dem Laden von 3D-Objekte in die virtuelle Welt soll folgend erläutert werden (Abbildung 5-16).

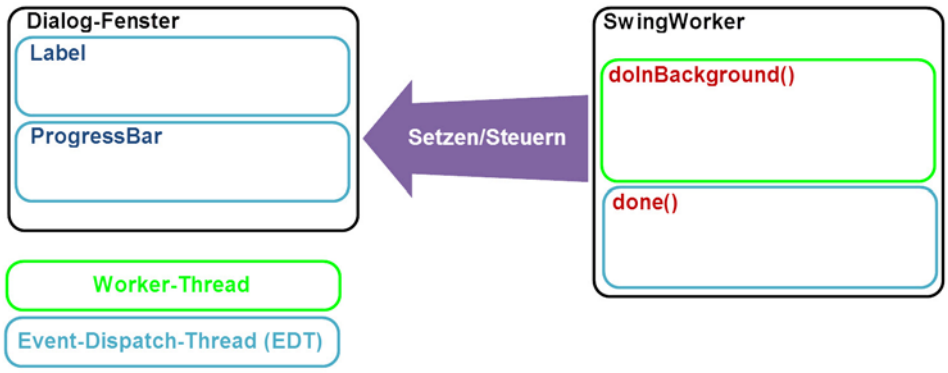


Abbildung 5-16:SwingWorker ⇨ EAI-Verbindungsaufbau / Update der Session / Laden von 3D Objekten[49]

[49] eigene Darstellung

## 5.4 Verbindung über das EAI-Interface

### 5.4.1 Verbindungsaufbau

Wie bereits im Punkt 2.1.6 erwähnt, implementiert das Avalon-System für die bidirektionale Kommunikation das EAI-Interface als externe Schnittstelle. Um die Services nutzen zu können, liefert InstantReality in seinem Paket für die entsprechende Programmiersprachen-Anbindung die jeweiligen Bibliotheken mit aus. Die für die Java-Anbindung nötigen Klassen und Methoden kapseln sich in dem Archiv *instantreality.jar*, welches als *Referenced Libraries* in das JavGo-Projekt eingebunden wurde. Die Verbindung zum Avalon-System über das EAI Interface stellt sich einfache TCP/IP Verbindung dar, welche zur Laufzeit aufgebaut werden kann. Sobald das Avalon-System über das jeweilige grafische Frontend gestartet wurde, kann standardmäßig über das lokale Port 4848 des Laufzeitsystems die Verbindung aufgebaut werden (Abbildung 5-16).

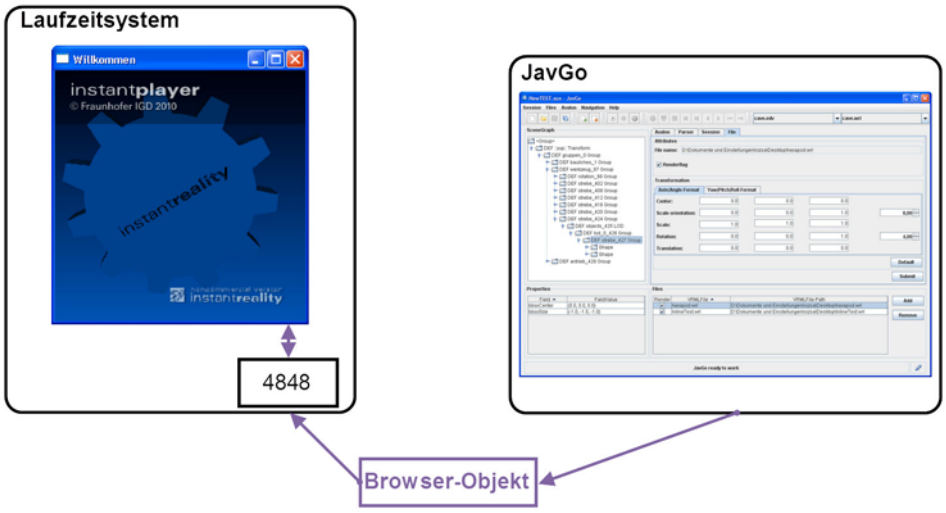


Abbildung 5-17: EAI Verbindung zum Avalon-System

Die erfolgreiche Verbindung über das EAI Interface von der externen Applikation zum Avalon-System wird durch ein Objekt vom Typ *Browser* symbolisiert. Das *Browser*-Objekt ist als Interface definiert, sodass sämtliche Funktionalitäten nur als leere Methodenrümpfe implementiert sind. Für eine direkte Objekterzeugung ist das *Browser* Interface demnach nicht vorgesehen. Die konkrete Implementierung erfolgt durch die Objekterzeugung mit Hilfe der *BrowserFactory* Klasse und dort über die Methode *getBrowser()*. Als Parameter verlangt die *getBrowser()* Methode neben dem Verbindungsport, den als IP-Adresse aufgelösten Host-Name des Rechners auf dem das Avalon-System läuft. Um die Auflösung des Host-Name in seine IP-Entsprechung zu gewährleisten, implementiert Java über die Klasse *InetAddress* die statische Methode *getByName()*. Unabhängig davon, ob nun der Host-Name oder direkt die IP-Adresse übergeben

wird, erfolgt die Rückgabe der IP-Adresse. Sowohl die Werte für den Host-Name als auch für das Verbindungsport bilden sich aus den in der Registry hinterlegten Einstellungen, die über den Settings-Dialog der JavGo-Applikation vorgenommen wurden. Am Ende eines erfolgreichen Verbindungsaufbaus steht die Erzeugung eines *Browser*-Objekts, andernfalls wird im Instanzierungsprozess eine Ausnahme (engl.: Exception) geworfen, die vom Programmierer innerhalb des Quellcodes behandelt werden muss.

### 5.4.2 Einfügen und Löschen von VRML-Modellen

Das Prinzip, wie die einzelnen 3D-Objekte in den Szenegrafen eingebunden werden, wurde im Punkt 3.2 und Punkt 5.2 weitestgehend erläutert. Die programmiertechnische Komponente kam zu dem grob ebenfalls in Punkt 5.2 zur Sprache. Die Präzisierung der Umsetzung des Einfügen und Löschen von 3D-Modellen aus der Session und damit aus den Szenegrafen der visualisierten Welt, soll nun in diesem Punkt erfolgen.

Sowohl für das Entfernen als auch für das Laden von 3D-Modellen in die Grundszenerie, muss auf der einen Seite die javaseitige Repräsentation, also das Session-Objekt und auf der anderen Seite der Szenegraf im Avalon-System aktualisiert und abgeglichen werden. Das Session-Objekt besitzt zur Speicherung der geladenen 3D-Modelle das dynamische Feld *files*, welches wiederum Elemente des Typs *VrmlFile* aufnehmen kann. In den einzelnen VrmlFile-Objekten ist die Objektvariable *id* definiert, die deckungsgleich mit dem Element-Index ist, unter welchem das betreffende VrmlFile-Objekt im dynamischen Feld *files* hinterlegt ist. Die objekteigene *id* Variable macht also den Identifikator aus, sowohl für das VrmlFile-Objekt im *files*-Feld des Session-Objekts, als auch für das entsprechende 3D-Modell im Szenegraf der virtuellen Welt. Die für das Laden und Entfernen der 3D-Modelle aus der Session und damit aus dem Session-Objekt nötigen Methoden sind im Session-Objekt selber implementiert. Beim Eintrag wird das übergeben VrmlFile-Objekt in das Feld *files* eingetragen und die *id* Variabel mit den Element-Index über die Methode *updateID()* abgeglichen. Beim Entfernen hingegen wird über die entsprechende Getter-Methode vom übergeben VrmlFile-Objekt der Wert der Variable *id* erfragt, um darüber das richtige Element aus dem *files* Feld zu löschen.

Die für den Abgleich des Szenegrafen im Avalon-Systen nötigen Methoden sind in der Superklasse *SceneNode* implementiert. Das aus einer Subklasse von *SceneNode* erzeugte Session-Objekt ruft diese Methoden innerhalb der entsprechenden lokal definierten Methoden für das Laden und Entfernen auf.

Wie bereits erwähnt, erfragt das Session-Objekt über die ererbte und konkret implementierte Methode *parentNode()* eine Referenz auf den Eltern-Knoten, in dem Fall den Wurzelknoten *DEF scene Scene* der visualisierten Welt. Der Zugriff erfolgt über die bestehende EAI-Interface Verbindung, welche durch das Browser-Objekt symbolisiert wird. Über die Methode *getNode()*, die als Parameter die DEF-Bezeichnung des Knotens verlangt auf den zugegriffen werden soll, kann direkt über das Browser-Objekt die Referenz auf den Wurzelknoten erzeugt werden. Der

eigentliche Eintragevorgang in den Szenegrafen ist komplett in der Superklasse *SceneNode* und dort in der Methode *updateEAI()* implementiert. Im Punkt 5.2 wurde bereits erwähnt, dass die einzelnen 3D-Modelle in der Form: *DEF File ID Transform Transform{children[DEF File ID Inline Inline{exportNameSpace TRUE url [" absoluter Pfad des 3D-Objekts "]}]}* in den Szengrafen eingebunden werden sollen.

## 5.5 VRML-Parser

Der in *JavGo* verwendete VRML-Parser ist Teil des VRML97 Browsers *VRwave*. Folgend soll kurz auf die Entwicklungsgeschichte von *VRWave* eingegangen werden.

Von 1992 bis 1994 wurde am Institute for Information Processing and Computer Supported New Media (IICM) an der Universität in Graz ein Browser für interlinked 3D Modelle namens Harmony 3D Scene Viewer entwickelt. Aufgabe des Browsers war es, 3D Inhalte für das Hyper-G System nutzbar zu machen. Hyper-G ist die Bezeichnung eines ebenfalls am IICM in Graz entwickelten Hypermediasystems.

Darunter versteht man ein Informationssystem in dem verschiedenste Daten, seien es Texte, Bilder, Ton- und Videoclips miteinander über Links verknüpft sind, um so die Informationen zu bündeln und für den Nutzer leicht zugänglich zu machen.

Aus Mangel an brauchbaren Formaten, um die 3D Hypermedia Modelle für die Anzeige im Browser zu beschreiben, entwickelte das IICM 1992 ein eigenes Dateiformat, dem sogenannten Scene Description Format (SDF), basierend auf dem ASCII Datenoutput der Wavefront's Advanced Visualizer Software, welche zur damaligen Zeit für die Modellierung der 3D Modelle vom IICM genutzt wurde. Im Dezember 1992 konnte der Harmony 3D Scener Viewer auf der Eurpean Conference on Hypertext Technology (ECHT) in Milan vorgestellt werden. Damit war es erstmals gelungen, eine Verbindung zwischen 3D Modellen und Hyperlinks für das Internet herzustellen. Mit der Entwicklung von VRML 1.0 und der darauffolgenden Spezifizierung, begann man im April 1995 in einem Gemeinschaftsprojekt mit dem National Center for Supercomputing Applications (NCSA) und der Universität von Minnesota, VRweb zu entwickeln. Dabei handelt es sich um einen auf der Basis des Harmony 3D Scene Viewer's entworfenen VRML 1.0 Browsers. Neben der Anzeige von in VRML 1.0 geschriebenen 3D Modellen, sollte mit VRweb ebenfalls dessen Sourcecode der Öffentlichkeit zugänglich gemacht werden. Im Juli 1995 konnte die erste Version von VRweb publiziert werden. Im Sommer 1996 begann man damit, den VRweb Browser für VRML 2.0 auszubauen, zusätzlich sollte der bisher in C++ geschriebene Browser nach Java portiert werden. Am 3. Februar 1997 konnte schließlich die erste Version unter der neuen Bezeichnung VRwave veröffentlicht werden. (vgl. Wagenbrunn 1998, S. 65)

Wie bereits erwähnt, ist der VRML97 Parser namens parse world (pw) Teil des *VRwave* Browsers. Der Parser ist aber von den Entwicklern so konzipiert, dass er völlig eigenständig / losge-

löst von *VRwave* genutzt werden kann(vgl. Wagenbrunn 1998, S. 67). Aus diesem Grund, publiziert das IICM auf dem eigenen Server, *VRwave* und den *pw* Parser gesondert und stellt den pw Parser unter die GNU Library General Public License (LGPL).

Der *pw* Parser Sourcecode liegt auf den IICM Server in zwei verschieden Archiven vor. Der Entwickler hat die Möglichkeit, den Sourcode bereits in vorkompilierten Javabytecode .class Dateien zu downloaden oder als reine .java Quellcode Dateien.

Da eine ausreichende API Dokumentation von Seiten der IICM fehlt, wurde für das *JavGo*-Projekt der *pw* Parser Sourcecode in Form von reinen Java Quellcode Dateien genutzt, umso die Möglichkeit zu haben, den Quellcode der betreffenden Klasse in Klartext einzusehen. Letztendlich wurde aus den Java Quellcode Dateien des pw Parsers ein Jar-Archiv(*pw.jar*) angelegt, welches in das JavGo-Projekt als *Referenced Libraries* integriert wurde.

Die folgende Abbildung (Abbildung 5-16) zeigt die interne Ordnerstruktur des *pw.jar* Archivs.

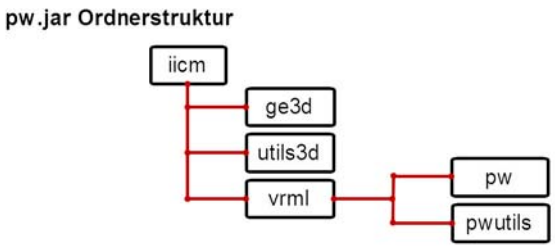


Abbildung 5-18: Interne Ordnerstruktur des Jar-Archives *pw.jar*[50]

Zentral ist hier der Ordner *pw*. Sämtliche Klassenrepräsentationen für die unterschiedlichen VRML 2.0 Knotentypen und Felddatentypen, sowie die *VRMLparser*- und *StrTokenizer* Klasse für das Einlesen aus dem *InputStream* und Zwischenspeichern des Eingelesen, sind in dem Ordner *pw* gruppiert.

[50] eigene Darstellung

### 5.5.1 Parsen einer VRML-Datei - Erzeugen des GroupNode

Den Hauptbestandteil des *pw* Parsers bildet die Klasse *VRMLparser*. Sie ist dafür zuständig, die VRML-Daten von einem Java *InputStream* zu lesen und daraus den Szenegrafen zu bilden.

Zur Instanzierung eines *VRMLparser* Objekt's stehen zwei Kontruktoren bereit:

- **VRMLparser(InputStream input)**
- **VRMLparser(InputStream input, ParserOutput poutput)**

Als Minimalanforderung für die Instanzierung, wird die Übergabe eines Objekt's vom Typ *InputStream* erwartet. Zusätzlich lässt sich über den erweiterten Konstruktor ein *ParserOutput* Objekt übergeben, um mit dessen Hilfe Statusmeldungen (Errors, Warnings, etc.), die während des Parsingprozesses auftreten können, abzufangen und individuell auszugeben.

Intern erfolgt die Objekterzeugung aber ausschließlich über den Konstruktor, der sowohl das *InputStream* Objekt, als auch das *ParserOutput* Objekt erwartet. Wird zum Beispiel dem Konstruktor nur der *InputStream* übergeben, macht dieser nichts weiter, als zusätzlich ein sogenanntes *DefParserOutput* Objekt zu erzeugen, um mit diesen beiden Objekten den Konstrukor für die eigentliche Instanzierung erneut aufzurufen. So stellt sich auch die Standardbenutzung des *pw* Parser dar, in der die Statusmeldungen der Parsers auf der Konsole ausgegeben werden.

Auch wenn der *pw* Parser intern mit einem *InputStream* Objekt arbeitet, so sollte man erwähnen, dass die Klasse *InputStream* als abstrakt definiert ist und als solches nicht instanziert werden kann. *InputStream* fungiert eher als Basisklasse für alle die speziellen *InputStreams*, die Java anbietet. Deshalb wird dem Konstuktor der *VRMLparser* Klasse nicht explizit ein *InputStream* Objekt übergeben, sondern eine Instanz einer Subklasse von *InputStream*. JavGo verwendet dafür ein *FileInputStream* Objekt, welches wiederum ein File Objekt als Parameter übergeben bekommt. Das File Objekt repräsentiert letztendlich die eigentliche VRML-Datei.

Nachdem nun die beiden Objekte an den *VRMLparser* Kontruktor übergeben wurden, erfolgt eine weitere Objekterzeugung innerhalb des Konstruktors, indem das *InputStream* Objekt an den Konstruktor der Klasse *StrTokenizer* weiterdelegiert wird. Die programmiertechnische Umsetzung, den *InputStream* einzulesen mit darauffolgender Abspeicherung des Eingelesenen, ist in der Klasse *StrTokenizer* implementiert. Es handelt sich also nicht um den javaeigenen *StringTokenizer*.

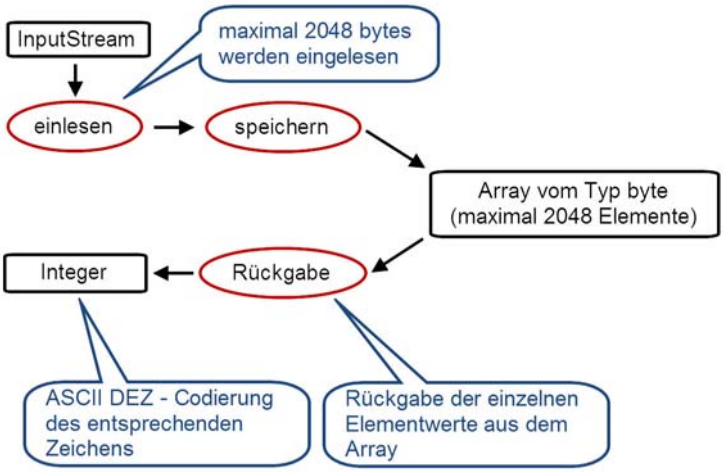


Abbildung 5-19: Abarbeitung der Methode *getChar()* in der *StrTokenizer* Klasse[51]

In obiger Abbildung (Abbildung 5-19) ist schematisch die Abarbeitung der in der *StrTokenizer* Klasse implementierten Methode *getChar()* zu sehen. Über diese Methode wird das eigentliche Einlesen des *InputStreams* mit anschließender Zwischenspeicherung des Eingelesenen vorgenommen. Bei jedem Einlesevorgang werden maximal 2048 Bytes des *InputStreams* gelesen und in einem *Byte Array* mit einer Maximalanzahl von 2048 Elementen gespeichert. Als Rückgabewert wird der Dezimalcode des entsprechenden Zeichens geliefert. Bei jedem Aufruf der *getChar()* Methode inkrementiert sich eine Integervariable, die als Zeiger fungiert. Über diesen Zeiger wird immer der nächste Elementwert aus dem *ByteArray* ausgelesen und zurückgegeben. Ist das Ende des Arrays erreicht, wird der ganze Einles- und Abspeicherungsvorgang erneut wiederholt. Die *StrTokenizer* Klasse implementiert ebenfalls noch eine Methode *nextChar()*, sie liefert das aktuelle Zeichen ohne ein neues einzulesen bzw. den kompletten Einlese- und Abspeicherungsvorgang anzustoßen.

Über die *readBody()* Methode des *VRMLparser* Objekts erfolgt nach dessen Instanzierung die Initialisierung einer vorher deklarierten Variable vom Typ *GroupNode*. Der im Szenegrafen enthaltene Wurzelknoten wird durch das *GroupNode* Objekt repräsentiert.

Im *pw* Parser sind bereits für alle Knotentypen und Felddatentypen entsprechende Klassenrepräsentationen vorhanden, sodass über die Instanzierung dieser speziellen Klassen die spezifischen Eigenheiten der Knotentypen und der jeweiligen Felddatentypen abgebildet werden kann.

Die Klassenrepräsentation der verschiedenen Knotentypen sind alle von der zentralen Basisklasse *Node* abgeleitet (Abbildung 5-20). Die knotenspezifischen Felder werden in der von *Node* geerbten Hashtable *subfields* hinterlegt. Bei jeder Instanzierung übernimmt der Konstruk-

[51] eigene Darstellung

tor der jeweiligen Knotentypklasse die Eintragungen der Felder über die ebenfalls von *Node* geerbte Methode *addFields()*. Nach dem Schema Feldname, Felddatentyp und Feldtyp werden der Methode *addFields()* die Daten übergeben. Hinter dem Feldtyp verbirgt sich eine Integer Konstante, die angibt, wie auf das Feld zugegriffen werden kann. Dem Felddatentyp, repräsentiert durch eine Instanz der entsprechenden Klassenrepräsentation, wird diese Konstante in der *addFields()* Moethode zugewiesen, bevor die Eintragung in die Hashtable *subfields* erfolgt. Der Feldname fungiert in der Hashtable demnach als Schlüssel und der entsprechende Wert wird durch eine Instanz des durch eine Klasse repräsentierten Felddatentyps dargestellt. In der *Node* Klasse ist zusätzlich noch die statische Methode *readNode()* implementiert, deren Abarbeitung führt zur Instanzierung der knotentypspezifischen Klassenrepräsentation. Die *readNode()* Methode ist durch ihre statische Deklaration, völlig unabhängig von der *Node* Klasse bzw. dem *Node* Objekt, davon abgesehen ist die *Node* Klasse als *abstract* definiert und als solches nicht Instanzierbar. Die *Node* Klasse implementiert die *readNode()* Methode nur konkret.

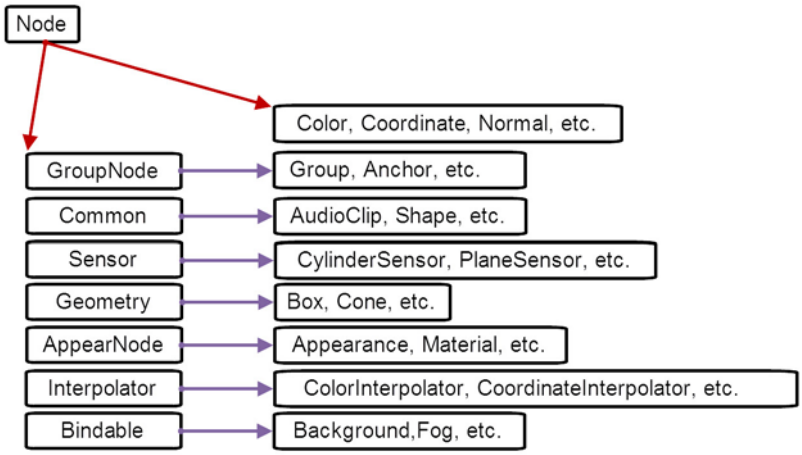


Abbildung 5-20: Vererbungshierarchie der Knoten im *pw* Parser[52]

Für die Klassenrepräsentationen der speziellen Felddatentypen fungiert die Klasse *Field* als Basisklasse (Abbildung 5-21). Die für den Feldzugriff benötigten Konstanten werden in der Basisklasse definiert und an die spezifischen Felddatentyp Instanzen vererbt. Darüber hinaus implementiert die *Field* Klasse die Methode *readtFieldValue()*, die wiederum die als *abtract* definierte Methode *readValue()* aufruft. Jede Subklasse von *Field* ist also dazu verpflichtet, die *readValue()* Methode zu überschrieben und dem Felddatentyp entsprechend einen Auslesemechanismus konkret zu implementieren. Bei den Klassenrepräsentationen für die Multi-valued-field (MF) Felder ist nochmals eine weitere Klasse als Gruppierung vorgeschalten. Die *Multi-Field* Klasse überschreibt die von *Field* geerbte Methode *readValue()* so, dass in einer While Schleife über den Aufruf der Methode *read1Value()* die einzelnen Werte aus der für MF Felder typischen Liste an Werte ausgelesen werden kann.

[52] eigene Darstellung

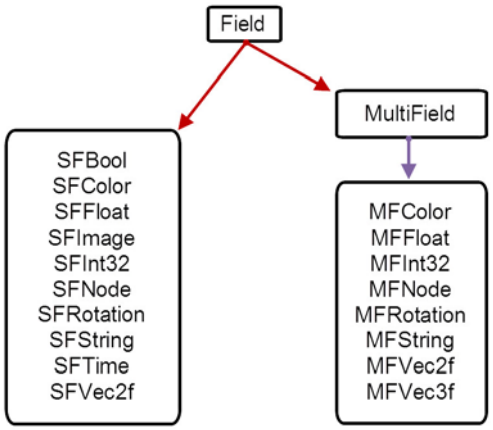


Abbildung 5-21: Vererbungshierarchie der Felder im *pw* Parser[53]

Zur Datenhaltung der eingelesenen Werte, implementieren die Klassenrepräsentationen der MF Felder, wiederum datentypspezifische Felder (Arrays). Zum Beispiel nutzt eine *MFFloat* Instanz zu Speicherung der Floatwerte, ein *FloatArray*. Interessant ist in dieser Hinsicht, der Felddatentyp *MFNode*. Eine über die Klasse *MFNode* erzeugte Instanz implementiert ein dynamisches Feld namens *nodes*, indem wiederum die eingelesenen Knoten abgelegt werden (Abbildung 5-22).

In der folgenden Abbildung (Abbildung 5-22) ist der Aufbau des Wurzelknotens, also des *GroupNode* Objekts dargestellt.

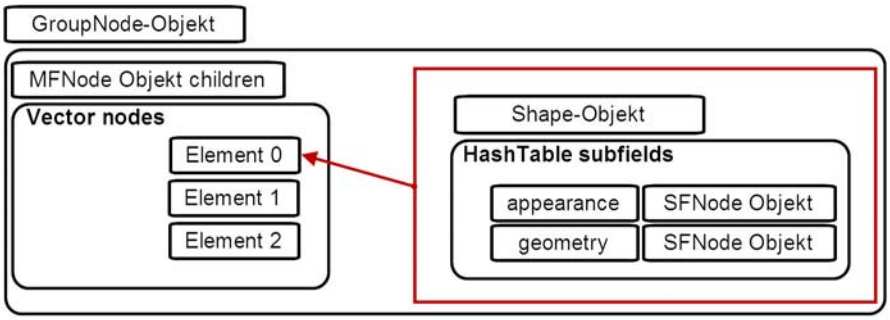


Abbildung 5-22: Aufbau des *GroupNode* Objekts[54]

Die einzelnen Kindknoten des Wurzelknotens werden in einem *Vector* gehalten. Dieser wird über das *children* Objekt implementiert, welches innerhalb der *GroupNode* Klasse als öffentliche Instanzvariable besteht. Jeder Knoten, der als Gruppierungsknoten fungiert und dadurch neben der *Node* Klasse noch zusätzlich von der Klasse *GroupNode* abgeleitet wird, bekommt

[53] eigene Darstellung

[54] eigene Darstellung

durch die *GroupNode* Klasse das *children* Objekt vom Typ *MFNode* vererbt. Wie bereits angemerkt, implementiert jede von der *MFNode* Klasse erzeugte Instanz, das dynamische Feld nodes als Behälter für die eingelesenen Knoten. Der Vorgang des Einlesens und der Ablage der Kindknoten im Vector wird durch die im *GroupNode* Objekt implementierte Methode *readNodes()* abgearbeitet. Sowohl der Einlesevorgang, an dessen Ende das auf Grundlage des eingelesenen Knotentyps erzeugte Knotentyp Objekt als Rückgabewert steht, wie auch die Eintragung des Knotentyp Objekt's in den Vector (Abbildung 5-22), wird von je einer Anweisung explizit in der *readNodes()* Methode übernommen.

Hinter dem Einlesevorgang steht aber weit mehr, als nur eine einfache Einzelanweisung, deshalb soll folgend der Ablauf des Einlesens der Knoten näher erläutert werden. Vorher kurz und vereinfacht dargestellt, das Prinzip, nach dem der *pw* Parser das Einlesen abarbeitet:

- **Einlesen des Knotentyps aus dem InputStream**
- **Instanzierung der entsprechenden Klassenrepräsentation auf der Grundlage des Knotentyps**
- **Setzen der vorinitialisierten Felder mit den tatsächlich vorhanden Feldwerten durch nochmaliges Einlesen aus dem InputStream**

Sowohl das Einlesen des Knotentyps, mit anschließender Instanzierung der entsprechenden Klassenrepräsentation und dem Einlesen der tatsächlich vorhanden Feldwerte, wird in der statischen Methode *readNode()* abgearbeitet. Die Klasse *Node* implementiert die *readNode()* Methode konkret. Der eingelesene Knotentyp wird als Stringvariable an die in der Klasse *NodeNames* konkret implementierte Methode *createInstanceFromName()* weitergeleitet. In der Klasse *NodeNames* sind insgesamt 46 statische Konstanten deklariert, in denen der jeweilige Name des Knotentyps als String enthalten ist. In der Methode *createInstanceFromName()* wird nun über 46 *If*-Anweisungen per Stringvergleich eine Übereinstimmung gesucht. Bei Gleichheit erfolgt die Instanzierung der Klassenrepräsentation, entsprechend des Knotentyps. Bei der Objekterzeugung werden, wie bereits erwähnt, im Konstruktor der entsprechenden Knotentyp Klasse, die für den Knotentyp spezifischen initialisierten Felder in die Hashtable *subfields* eingetragen. Als Rückgabewert liefert *createInstanceFromName()* das auf Grundlage des Stringvergleichs erzeugte Knotentyp Objekt (Abbildung 5-23).

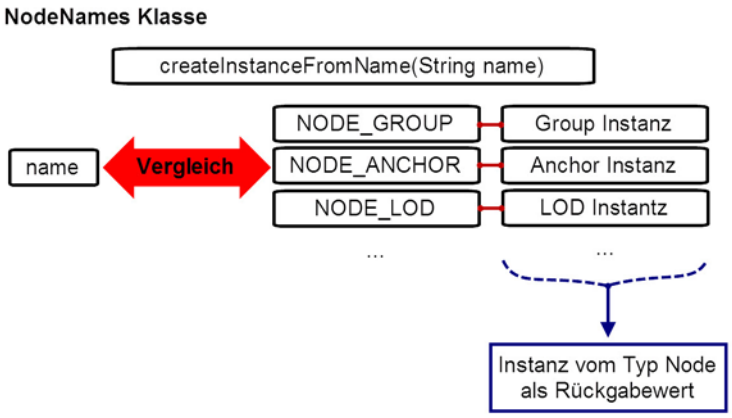


Abbildung 5-23: Die *createInstanceFromName()* Methode in der Klasse *NodeNames*[55]

Nach der Erzeugung des Knotentyp Objekts erfolgt mit einer weiteren Anweisung in der Methode *readNode()*, das Einlesen der tatsächlichen Feldwerte aus dem *InputStream* und die entsprechende Neusetzung der Feldwerte im Knotentyp Objekt. Wie zu erahnen, wird ab diesem Abschnitt im gesamten Parsingprozess ausschließlich mit dem konkreten Knotentyp Objekt gearbeitet. Per Aufruf der *readNodeBody()* Methode werden über die *readFields()* Methode die einzelnen Felddatentyp Objekte aus der HashTable *subfields* des Knotentyp Objekts herausgelesen. Durch die Übergabe des *VRMLpaser* Objekts an die entsprechenden Methoden, besteht immer der Zugriff auf die eingelesenen Werte aus dem *InputStream*. Die *readFields()* Methode implementiert eine *While*-Schleife, die solange läuft, bis im betreffeden Abschitt des *InputStreams* keine Feldnamen mehr existieren. Im Rumpf der *While*-Schleife wird der bei jedem Schleifendurchlauf der jeweils aktuelle Feldnahme als Schlüssel benutzt, um den dazugehörigen Wert, in dem Fall das entsprechende Felddatentyp Objekt, aus der Hashtable *subfields* des konkreten Knotentyp Objekts herauszubekommen. Danach erfolgt über den Aufruf der *readFieldValue()* Methode und der *readValue()* Methode des konkreten Felddatentyp Objekts, das Auslesen der Feldwerte aus dem *InputSream* und der Anschließenden Neusetzung der Feldwerte.

[55] eigene Darstellung

### 5.5.2 Abbilden des GroupNode auf eigene Datenstruktur

Der vom *pw* Parser erzeugte *GroupNode* musste für die Anzeige im *JTree* nochmals auf eine eigene Datenstruktur abgebildet werden. Die Notwendigkeit dafür soll folgend erläutert werden.

Damit die Daten im *JTree* Objekt, also der View, angezeigt werden können, muss das *JTree* Objekt über die im *TreeModel* konkret implementierten Methoden, genaue Aussagen über das jeweilige Knotenobjekt treffen können (Punkt 5.1.3). Wenn festgestellt wurde, dass es sich bei den jeweiligen Knotenobjekt um kein Blatt handelt, dann müssen die Kindknoten Objekte in einer Struktur gruppiert sein, die es ermöglicht, die genaue Anzahl der Kindknoten zu ermitteln bzw. über den Index auf den jeweiligen Kindknoten in der Struktur zuzugreifen. Wie im Punkt 5.5.1 bereits erwähnt, sind die VRML-Gruppierungsknoten im *pw* Parser so implementiert, dass die Kindknoten alle in einem dynamischen Feld hinterlegt sind. Mit dem Zugriff auf dieses Feld bzw. diesen Vector, lassen sich die oben genannten Anforderungen leicht erfüllen. Bei einem Knoten jedoch, der ein oder mehrere Felder des Felddatentyps *SFNode* enthält, stellt sich das Bild anders dar. Die einzelnen Felder eines Knotens und so auch die *SFNode* Objekte, sind alle in einer Hashtable zusammengefasst. Die Zuordnung in einer Hashtable wird über Schlüssel zu den jeweiligen Werten hergestellt. Die Schlüssel bilden in diesem Zusammenhang die Feldnamen und die dazugehörigen Werte die entsprechenden Felddatentyp Objekte. Es ergibt sich damit der Sachverhalt, dass der Zugriff auf die *SFNode* Objekte in der Hashtable und damit auf die Kindknoten, nicht über Indizes erreichen werden kann (Abbildung 5-24).

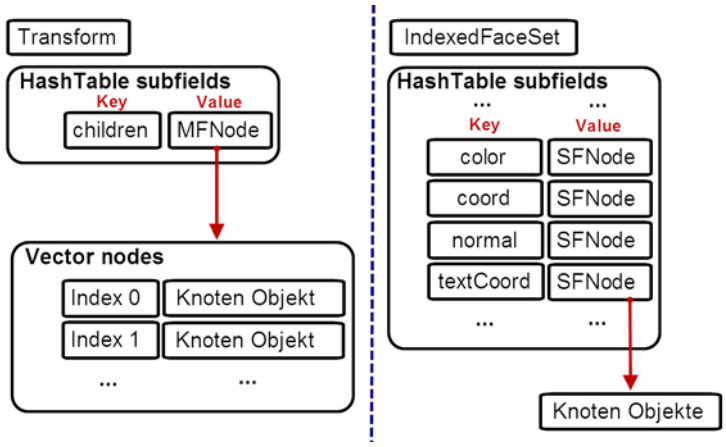


Abbildung 5-24: Problemdarstellung *MFNode* und *SFNode*[56]

Die *JavGo*-Applikation löst dieses Problem, indem jeder einzelne Knoten wiederum durch ein Objekt repräsentiert wird, welches sowohl die im *MFNode* Feld gruppierten Kindknoten als auch

[56] eigene Darstellung

die in den *SFNode* Feldern enthalten Knoten alle in einen Vector *children* gruppiert (Abbildung 5-25). Die vom *pw* Parser gelieferten Knoten werden faktisch jeweils von einem *SGNode* Objekt ummantelt.

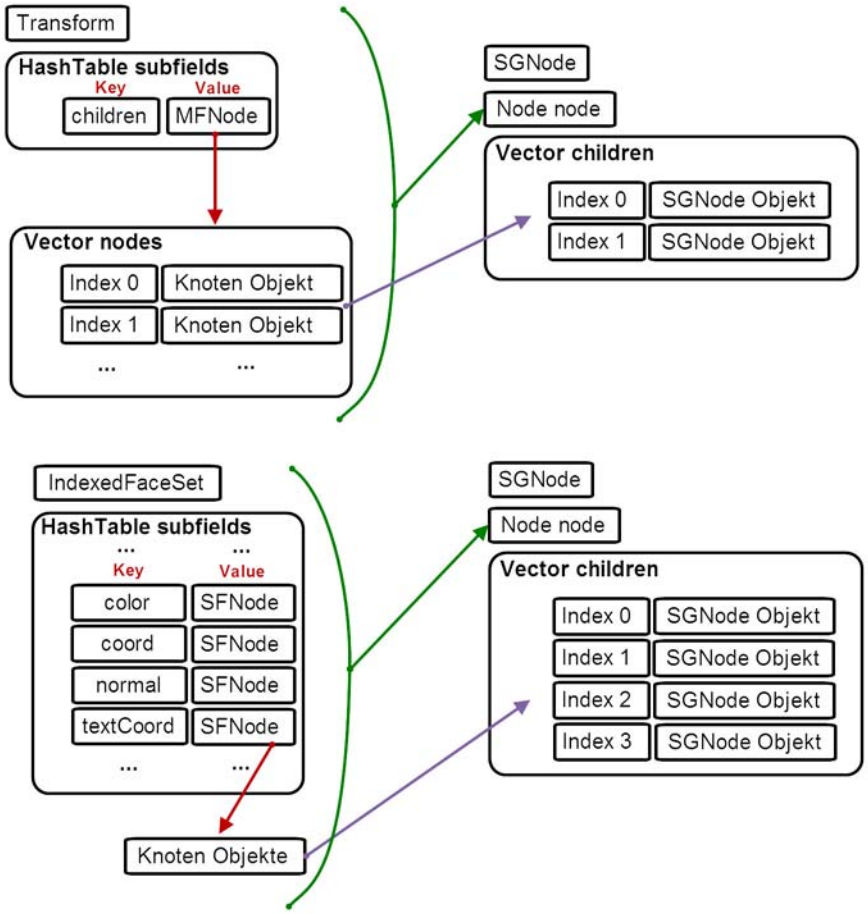


Abbildung 5-25: Abbilden der Knoten auf *SGNode* Objekt[57]

Damit die anderen Informationen, wie die zusätzlichen Felder neben den *MFNode* und *SFNode* Feldern nicht verloren gehen bzw. beibehalten werden, wird der gesamte Knoten nochmals in einer lokalen Instanzavariable *node* in dem jeweiligen *SGNode* Objekt abgespeichert. Sowohl der *children* Vector als auch die *node* Instanzvariable wird per Getter-Methode nach außen geführt.

Die gesamte Neuabbildung des vom *pw* Parser erzeugten *GroupNode* wird per Rekursion vorgenommen. Der Konstruktor der *SGNode* Klasse erhält immer den zu ummantelnden Knoten. In einer While-Schleife wird die *subfields* Hashtable des übergebenen Knotens durchlaufen. Per If Anweisung wird im Schleifenrumpf auf Felddatentyp *SFNode* oder *MFNode* geprüft. Bei

[57] eigene Darstellung

*SFNode* erfolgt die einmalige Neuinstanzierung der *SGNode* Klasse mit der Übergabe des im *SFNode* Feld befindlichen Knoten Objekts. Anschließen findet die Eintragung des erzeugten *SGNode* Objekts in den Vector *children* statt. Ergibt die Prüfung ein Feld vom Felddatentyp *MFNode*, dann erfolgt wiederum in einer While-Schleife die Neuintstanzierung und Eintragung sämtlicher über das *MFNode* Feld gruppierten Kindknoten.

Angestoßen wird der gesamte Abbildungsvorgang auf die eigene Datenstruktur, durch die in der Klasse SGNode implementierte statische Methode *createSGNode()*. Der vom *pw* Parser erzeugte *GroupNode* weißt in Bezug auf alle anderen Gruppierungsknoten, die Besonderheit auf, dass die *subfields* Hashtable leer ist. Der Wurzelknoten besitzt also keine Felder und zwangsläufig auch kein Feld *children*. Die gruppierten Kindknoten werden vielmehr durch eine lokale Variable *children* vom Typ *MFNode* repräsentiert (Abbildung 5-26).

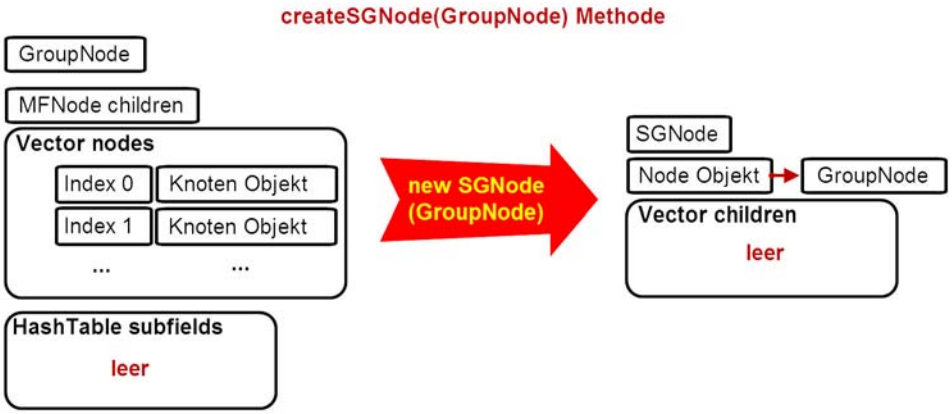


Abbildung 5-26: Erster Arbeitsschritt in der *createSGNode()* Methode[58]

Ein Durchlaufen der Hashtable *subfields* des *GroupNode* Objekts nach einem Feld *children*, würde also kein vernünftiges Ergebnis bringen. Aus diesem Grund implementiert die SGNode klasse, die Methode *createSGNode()*. Mit der Übergabe des *GroupNode* Objekts an die *createSGNode()* Methode, erfolgt die Instanzierung eines *SGNode* Objekts, mit der wiederholten Übergabe des *GroupNode* Objekts an den *SGNode* Konstruktor (Abbildung 5-26). Nach der Erzeugung des *SGNode* Objekts wird in einer While-Schleife über den im *GroupNode* Objekt, und dort über die in der *MFNode* Variable *children* befindlichen Kindknoten iteriert. Jeder Kindknoten wird wieder von einem *SGNode* Objekt ummantelt, bevor es in den *children* Vector des *SGNode* Objekts, welches den *GroupNode* ausmacht, eingetragen wird (Abbildung 5-27).

[58] eigene Darstellung

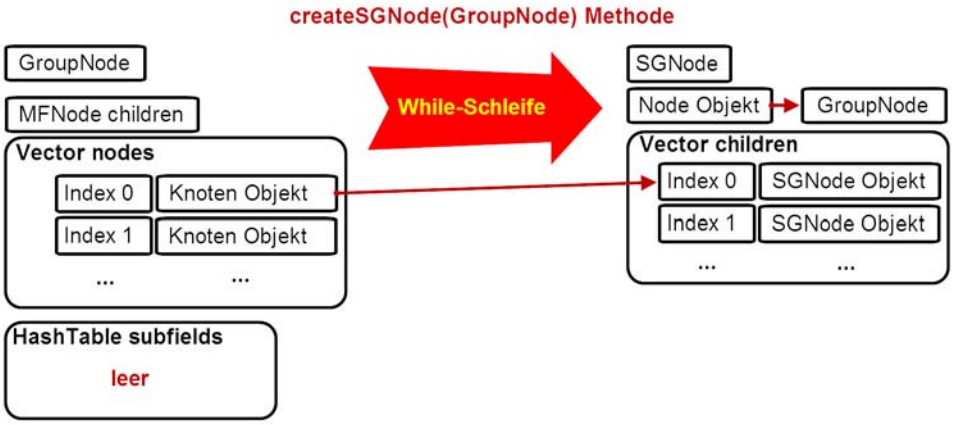


Abbildung 5-27: Zweiter Arbeitschritt in der *createSGNode()* Methode[59]

In folgender Abbildung (5-28) ist nochmals der Ablauf des vom *pw* Parser erzeugten *GroupNode* Objekt bis zum *SGNode* Objekt, welches den *GroupNode* abbildet, schematisch dargestellt. Die statische Methode createSGNode() erstellt als Rückgabewert ein *SGNode* Objekt, was den kompletten Wurzelknoten ausmacht. Alle im Wurzelknoten gruppierten Kindknoten und wieder deren etwaige Kindknoten tauchen als Folge von *SGNode* Objekten in den *children* Vektoren der entsprechenden *SGNode* Objekten, die Kindknoten besitzen, auf. Und dies unabhängig davon, ob die entsprechenden Knoten Objekte vorher in einem Feld vom Felddatentyp *SFNode* oder *MFNode* waren.

[59] eigene Darstellung

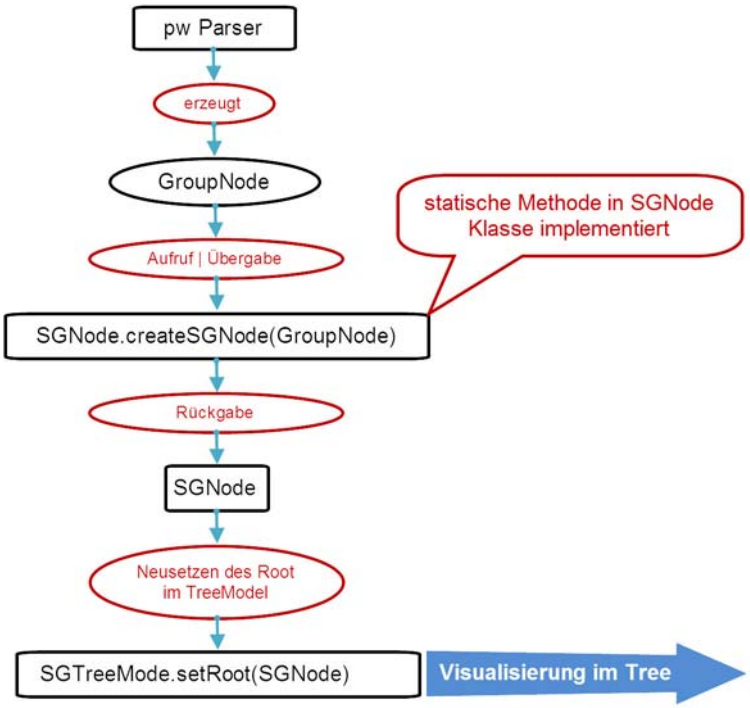


Abbildung 5-28: Die schematische Darstellung des Ablaufs ⇨ *GroupNode* des *pw* Parser ⇨ *GroupNode* ummantelten *SGnode* Objekt[60]

Der von der *createSGNode ()* Methode erzeugte *SGNode* wird dem TreeModel des *JTree* als neuer Wurzelknoten übergeben. Dort zeigt ein spezieller Listener der View, also dem *JTree* Objekt, dass sich die Struktur des Baumes grundlegend geändert hat und die eine vollständige Neuvisualisierung von der View vorgenommen werden soll.

[60] eigene Dartsellung

## 5.6 Modelle

### 5.6.1 Zentrale Klasse für alle Modelle

Wie bereits im Punkt 5.1 beschrieben, verfolgt SWING das MVC Prinzip als Struktur. Wie ein roter Faden zieht sich dieses Prinzip durch den Aufbau jeglicher SWING Elemente. Die JavGo-GUI verwendet mehrere GUI Komponenten, wo dieses Prinzip besonders deutlich wird.

- **Tabel**
- **Combobox**
- **Tree**

Es stellte sich nun das Problem, dass die verschiedenen Modelle in den jeweiligen Klassen zugänglich sein mussten. Ein möglicher Ausweg wäre die statische Deklarierung der einzelnen Modelle gewesen, damit hätten sich aber wieder weitere Nachteile aufgetan.Von jeder Klasse aus wären die Modelle zugänglich gewesen.

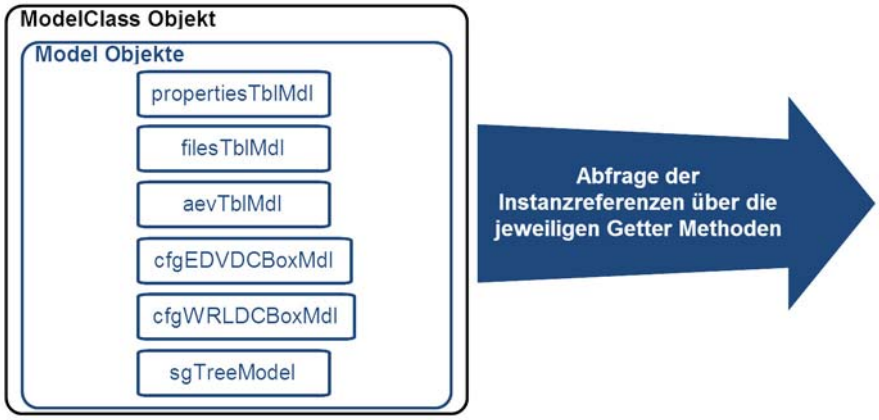


Abbildung 5-29: Zentrale Klasse für die Instanzierung aller Modelle[61]

Das in *JavGo* verfolgte Prinzip sieht folgendermaßen aus. Es stellt ein eine zentrale ModelClass in den Fokus, welche sämtliche Modelle der in *JavGo* verwendeten Tabellen, ComboBoxen und des Tree enthält. Über diese zentrale Klasse soll so zu der Zugriff auf die jeweiligen Modelle möglich werden.

[61] eigene Darstellung

Im Genauen stellt sich dies folgend dar. Im Konstruktor der ModelClass werden die einzelnen Modelle instanziert. Der Zugriff wird über die jeweiligen Getter-Methoden hergestellt, indem über diese die Referenz auf die entsprechende Model Instanz nach außen geführt wird. So wird die ModelClass einmal in der Hauptklasse instanziert und die Referenz auf die ModelClass Instanz an die jeweilige Klasse weitergegeben, die Zugriff auf bestimmtes Model braucht. Über das ModelClass Objekt und dem Aufruf der entsprechenden Getter-Methode, hat nun die Klasse die Möglichkeit, Zugriff auf das entsprechende Modell zu bekommen.

### 5.6.2 Individuell erzeugte Modelle

Sowohl für die Tabellen als auch für den Tree, war es notwendig ein eigenes Model zu erzeugen. Dies ist von Nöten, wenn die Daten, die in der jeweiligen GUI Komponente dargestellt werden mit Hilfe der Standard-Modelle so nicht darstellbar sind oder sonstige Zusätze eingebaut werden müssen.

In der Files Tabelle sollte die erste Spalte zum Beispiel Werte des Typs Boolean anzeigen, die in dem Fall die RenderFlag's der jeweiligen geladenen .wrl Dateien symbolisieren. Im genaueren bedeutet dies, dass in der ersten Spalte Checkboxen visualisiert werden, die je nachdem ob TRUE oder FALSE entweder aktiv oder inaktiv sind. Bei Verwendung des StandardModells („DefaultTableModel") würden die gesamten Daten innerhalb der Tabelle als Objekte des Typs Object aufgefasst, sodass in den einzelnen Zellen nur die Stringrepräsentation des jeweiligen Objekts visualisiert wird.

Zudem sollten in der Files-Tabelle und in der AEV-Tabelle Zusatz-Mechanismen eingebaut werden, die einen Doppelteintrag von gleichen Daten unterbinden. Für die Files-Tabelle ist das Kriterium für einen Doppelteintrag gegeben, wenn .wrl Dateiname und Pfad schon in der Tabelle vorhanden ist. Für die AEV-Tabelle, wenn der Variablenname schon im Model vorhanden ist.

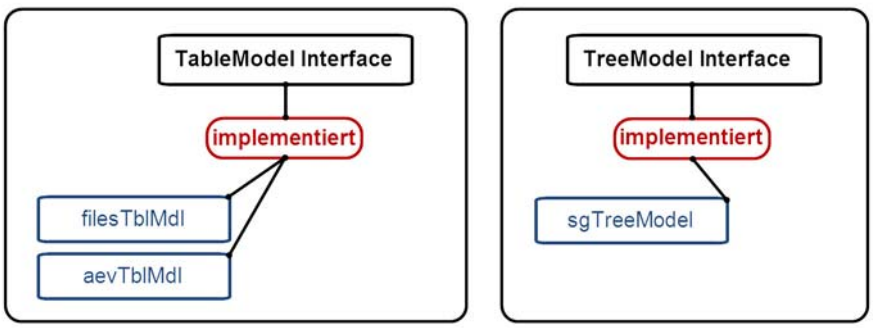


Abbildung 5-30: Individuell erzeugte Modelle

Wie in obiger Abbildung zu sehen, wurden die Modelle mit Hilfe der Implementierung des entsprechenden Interfaces erzeugt.

### 5.6.2.1 AEVTableModel-Klasse

Die AEVTableModel-Klasse bildet das Model für die AEV-Tabelle. In der Tabelle werden die zusätzlichen Umgebungsvariablen für den externen Avalon Prozess gehalten. Die eigentlichen Daten der Tabelle sind in einem Vector („aevEntries“) hinterlegt.

Ein Vector ist ein dynamisches Feld, das je nachdem ob Elemente hinzugefügt oder entfernt werden, wächst oder sich vermindert. Nach der Instanzierung ist der Vector mit einer Kapazität von zehn Elementen vorinitialisiert.

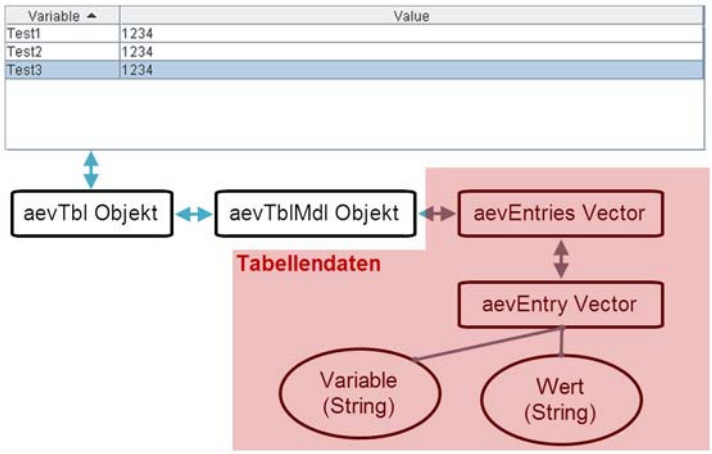


Abbildung 5-31: AEVTabelModel[62]

Der Vector aevEntries nimmt wiederum Vectoren auf. Dabei symbolisieren die einzelnen Vectoren („aevEntry“) die Zeilen in der AEV-Tabelle. Eine Zeile besteht aus dem Variabelnnamen und den dazugehörigen Wert, sodass der aevEntry Vector zwei Elemente des Typs String enthält.

Um nun Doppeleinträge zu vermeiden, ist in der AEVTableModel-Klasse ein weiterer Vector („aevVar“) implementiert, in dem die Variabelennamen sämtlicher in der Tabelle befindlichen Variablen abgespeichert sind. Wie im Punkt 5.6.2 bereits erwähnt sollen Einträge mit gleichem Variablenennamen unterbunden werden. Die Abfrage, ob die einzutragende Variable bereits in der Tabelle vorhanden ist, übernimmt die in die AEVTableModel-Klasse implementierte Methode .isExisting(). Je nachdem ob die Variabel schon im Vector aevVar vorhanden ist, gibt die Methode TRUE oder FALSE als Rückgabewert zurück. Bei jedem Eintrag in die AEV-Tabelle

[62] eigene Darstellung

wird also erst die isExisting() Methode ausgeführt und deren Rückgabewert ausgewertet, der dann darüber entscheidet, ob der Datensatz in das Model eingetragen wird oder nicht.

Noch zu erwähnen ist die Problematik der Sortierung. Die AEV-Tabelle sortiert nach der ersten Spalte (Variablenname) aufsteigend. Hier wird aber nur in der View, also dem JTable-Objekt sortiert, der Datenhalter, in dem Fall das Model der AEV-Tabelle, bleibt davon unberührt. Im Model selber liegen die Daten daher in unsortierter Reihenfolge, so wie sie eingetragen wurden, vor.

#### 5.6.2.2 FilesTableModel-Klasse

Das filesTblMdl Objekt bildet sich aus der Instanzierung der FilesTableModel Klasse. Es symbolisiert das Model, also den Datenhalter, der Files Tabelle. In der Files Tabelle werden sämtliche in die Session geladenen .wrl Dateien aufgeführt. In der folgenden Abbildung ist das ganze schematisch dargestellt.

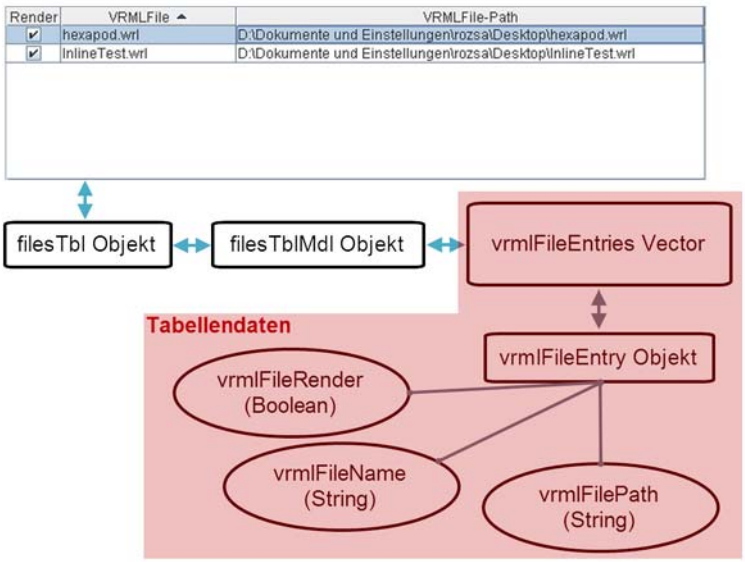


Abbildung 5-32: FilesTableModel[63]

Die eigentlichen Daten der Files Tabelle werden in dem Vector *vrmlFileEntries* gehalten. Dieser besteht wiederum aus Objekten des Typs *vrmlFileEntry*. Jedes dieser Objekte symbolisiert eine Zeile in der Tabelle. Das *vrmlFileEntry* Objekt besteht aus drei Variablen:

[63] eigene Darstellung

- ***vrmlFileRender* ⇨ RenderFlag der .wrl Datei ⇨ Typ: *Boolean***
- ***vrmlFileName* ⇨ Dateiname der .wrl Datei ⇨ Typ: *String***
- ***vrmlFilePath* ⇨ Pfad der .wrl Datei ⇨ Typ: *String***

Zusätzlich besitzt das Objekt noch Getter-Methoden, um die Inhalte der drei Variablen abzufragen bzw. eine Setter-Methode, um das Renderflag zu setzen.

Wie im Punkt 5.6.2 bereits angesprochen, muss die erste Spalte in der Files-Tabelle in der Lage sein, Werte des Typs *Boolean* anzuzeigen. Genauer gesagt, muss die View der Tabelle die Boolean-Werte als solche interpretieren, um sie als Checkboxen zu visualisieren.

Generell baut eine Tabelle die in ihr befindlichen Zellen mit Hilfe eines Cell-Renderes auf. Dabei handelt es sich quasi um eine Vorschrift, wie die Daten in einer Zelle visualisiert werden sollen. Der Cell-Renderer kann in Abhängigkeit von dem Typ der Daten, die visualisiert werden sollen, agieren oder in Abhängigkeit zur einer speziellen Spalte der Tabelle.

Die Tabelle an sich schaut zu erst, ob vom Programmierer ein eigener Cell-Renderer für die jeweilige Spalte registriert wurde, ist dies nicht der Fall ergibt sich das Szenario, wie in folgender Abbildung.

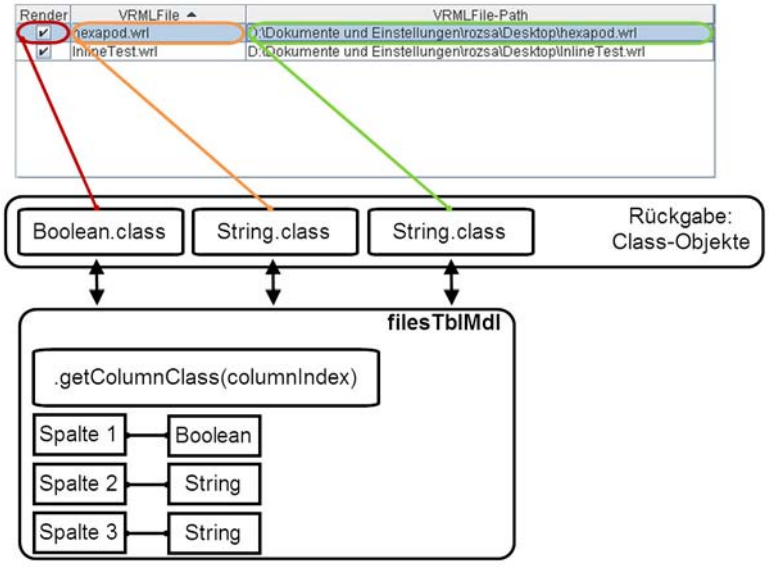


Abbildung 5-33: Cell-Render[64]

Über die Methode *.getColumnClass()* des Models erfragt die Tabelle die Datentypen der Spalten-Zellen. Als Rückgabewert liefert die Methode Class-Objekte. Die Tabelle vergleicht nun die zurückgegebenen Datentypen mit einer Liste von Datentypen, für die schon per default Cell-

[64] Eigene Darstellung

Renderer registriert sind. Für die folgenden Datentypen initialisiert die Tabelle von sich aus schon die entsprechenden Cell-Renderer.

- **Boolean ⇨ Rendering erfolgt mit CheckBoxen**
- **Number ⇨ Rendering erfolgt mit rechts ausgerichteten Label**
- **Double, Float ⇨ Rendering erfolgt mit rechts ausgerichteten Label, in dem der Text per NumberFormat formatiert wird**
- **Date ⇨ Rendering erfolgt über ein Label, in dem der Text per DateFormat formatiert wird**
- **ImageIcon, Icon ⇨ Rendering erfolgt über ein zentriertes Label**
- **Object ⇨ Rendering erfolgt über ein Label in die Stringrepräsentation des Objekts visualisiert wird**

#### 5.6.2.3 SGTreeModel-Klasse

Das für den JTree benutze TreeModel wird durch die Instanz der Klasse SGTreeModel repräsentiert. Wie bereits im Punkt 5.1.3 erwähnt muss jede Klasse, die als Model für den JTree fungieren will, das TreeModel Interface implementieren. Dadurch erhält die Model Klasse die Methodensignaturen über deren konkrete Implementierung das JTree Objekt die Daten im Baum abbilden kann.

Folgende Funktionalitäten müssen in der TreeModel Klasse über die konkrete Implementierung der vom TreeModel-Interface festgelegten Methoden gewährleistet werden.

- **Abfrage des Wurzelknoten ⇨ getRoot()**
- **Abfrage, ob es sich bei den jeweiligen Knoten um ein Blatt handelt. ⇨ isLeaf()**
- **Abfrage, wie viel Kindknoten durch den jeweiligen Elternknoten gruppiert werden. ⇨ getChildCount()**
- **Abfrage des entsprechenden Kindknotens ⇨ getChild()**
- **Abfrage, an welcher Stelle sich der Kindknoten in Bezug zu seinem direkten Elternknoten befindet. ⇨ getIndexOfChild()**
- **Zusätzliche noch die Möglichkeit TreeModelListener zu entfernen bzw. hinzuzufügen ⇨ addTreeModelListener() und removeTreeModelListener()**

# 6. Zusammenfassung und Ausblick

Die im InstantReality-Paket enthaltenen grafischen Frontends für das Avalon-System lassen für den Arbeitseinsatz des VRCP wichtige Funktionen vermissen. Infolge einer studentischen Arbeit wurde eine Steuerungs-GUI für die Kommandozeilenversion des Visualisierungsfrontends vom Avalon-System, dem SAV-Player, entwickelt. Die in C++ unter dem Namen *avGo* programmierte Applikation beseitigte den Nachteil, dass sowohl über den InstantPlayer als auch über den SAV nach vorheriger Initialisierung durch eine Visualisierungskonfiguration und Grundszenerie keine weiteren in einer WRL-Datei hinterlegten 3D-Objekte hinzugeladen werden können. Auf Grund der eingeschränkten Portabilität der *avGo*-Applikation, wurde eine Portierung nach Java forciert. Die programmiertechnische Umsetzung dieser Portierung unter Einbindung zusätzlicher Funktionalitäten bildet das Thema dieser Diplomarbeit. Als Endergebnis konnte eine lauffähige in Java umgesetzte Steuerungs-GUI für den SAV-Player präsentiert werden. Auf Grund der begrenzten Zeit mussten jedoch Abschläge bei der Implementierung von Funktionen in Kauf genommen werden. Die *avGo*-Applikation bot die Möglichkeit, die Festlegung der globalen, wie auch objektspezifischen Transformationsparamter, über die zwei verschiedenen Formen Achsen-Winkel bzw. Euler'sche-Winkel bezüglich der Rotation vorzunehmen. In der in Java umgesetzten *JavGo*-Applikation konnte nur die Eingabe über die Achsen-Winkel-Form integriert werden. Da VRML intern aber sowieso die Rotation ausschließlich über die Achsen-Winkel-Form durchführt, sollte dies verschmerzbar sein. Neben der Visualisierung der internen Knotenstruktur der einzelnen geladenen 3D-Objekte mit der Möglichkeit sich die Eigenschaften der einzelnen enthaltenen Knoten-Elemente anzeigen zu lassen, sollten diese Eigenschaften noch durch den Nutzer veränderbar bzw. beeinflussbar sein. Diese Veränderung durch den Nutzer, mit dem anschließenden Abgleich über das EAI-Interface, konnte nicht umgesetzt werden. In diesem Zusammenhang stellt sich wieder das Problem, dass der Abgleich und damit der Zugriff auf den entsprechenden Knoten nur erfolgen kann, wenn eine *DEF*-Bezeichnung innerhalb der VRML-Notierung für das *MFNode*- oder *SFNode*-Element festgelegt wurde. Letztlich müsste schon beim Export des 3D-Modells in das VRML-Format oder bei der Notierung des entsprechenden 3D-Objekts in VRML darauf geachtet werden, dass jeder Knoten, gleich ob *MFNode* oder *SFNode*, eine eindeutige Bezeichnung bekommt, über welche im Nachhinein per EAI-Verbindung zugegriffen werden kann. Da diese Sicherstellung leider nicht im direkten Einflussbereich des Programmierers der *JavGo*-Applikation steht und die Daten einfach so hingenommen werden müssen, wie sie von Dritten hinterlegt wurden, konnte die Funktion, neben dem reinen Zeitmangel, nicht umgesetzt werden. Ein möglicher Workaround wäre hier, schon beim Parsen der Knotenstruktur des entsprechenden 3D-Objekts zu schauen, wo in der Struktur ein Knoten-Element mit DEF-Bezeichnung vorhanden ist und auch nur dort die Funktionalität der Veränderung der Knoteneigenschaften neben der reinen Anzeige eben dieser, anzubieten. Wie bereits erwähnt, wird jedes einzelne 3D-Objekt per Inline-Knoten unter den Wurzelknoten der Grundszenerie eingebunden. Bei jedem dieser Inline-Knoten ist das Feld *exportNameSpace* auf TRUE gestellt. Somit werden die im eingebundenen Teilbaum enthaltenen Knoten-Namen dem Namensraum des Wurzelknotens der virtuellen Welt bekanntgegeben und damit für einen etwaigen Zugriff nutzbar. Für die bidirektionale Verbin-

dung stellt das Avalon-System das EAI-Interface unter der Bezeichnung Avalon External Interface zu Verfügung. Intern arbeitet das Avalon-System auf Basis von X3D, die Kommunikation mit dem auf dem X3D Daten basierenden Szenegraf und den Application-Services wird über das durch X3D Spezifikation implementierte SAI Interface vorgenommen. Leider wird die Funktionalität des SAI Interfaces nicht bis in die Application-Services des Avalon-Systems nach außen geführt, sodass hier immer noch mit dem durch den VRML-Standard definierten EAI Interface gearbeitet werden muss. In der Arbeit wurde die Thematik der Steuerung des externen Avalon-Prozess erläutert. Aufgrund der fehlenden Möglichkeit zur Einflussnahme auf den einmal gestarteten Avalon-Prozess über die Kommandozeile und dadurch über die durch das Process-Objekt bereitgestellten *Streams*, lässt sich der Prozess an sich weder in seinem Status für eine externe Applikation beurteilen noch der Prozess sauber steuern. Hier wäre die Integrierung einer Funktion seitens des Avalon-Systems zur Erfragung, wann das System vollständig initialisiert ist, sehr hilfreich, neben der Möglichkeit per *Command* das System sauber herunterzufahren. Für die Weiterführung des Projekts *JavGo* wären neben der Einbindung der noch fehlenden Eingabemöglichkeit der Rotation über die Euler'schen-Winkel Form und dem Integrieren des Workarounds für die Manipulation von Feldwerten auch zu überlegen, die Visualisierung der Knotenstruktur nicht nur auf das jeweils ausgewählte 3D-Objekt zu beschränken, sondern eine Gesamtanalyse und Ansicht des kompletten Wurzelknotens der Grundszenerie durchzuführen. Denkbar wäre hier dem Nutzer die Option zugeben über eine entsprechende Einstellung zwischen beiden Visualisierungsmöglichkeiten zu wählen. Die Einbindung von sogenannten Visualisierungsfiltern für das Tree-Element, um bestimmte Elemente aus der Anzeige der Knotenstruktur zur besseren Übersicht auszuschließen, wäre ebenfalls eine noch denkbare Erweiterung.

# 7. Literaturverzeichnis

Aschke, Michael: Stereoskopische Navigation in 3D-Datensätzen für die Erweiterte Realität in der Chirugie. - Waabs: GCA -Verlag, 2007

Behr, Johannes: Avalon: Ein skalierbares Rahmensystem für dynamische Mixed-Reality Anwendungen. - 2005. - 164 S. Darmstadt, Technische Universität Darmstadt, Fachbereich Informatik, Dissertation, 2005

Bell, Gavin; Parisi, Anthony; Pesce, Mark: The Virtual Reality Modeling Language: Version 1.0 Specification. URL: <http://www.web3d.org/x3d/specifications/vrml/VRML1.0/index.html>, verfügbar am 01.06.2010

C.Burdea, Grigore; Coiffet, Philippe: Virtual Reality Technilogy: Second Edition. - 2. Aufl. Hoboken: John Wiley & Sons, Inc., 2003

Dähne, Patrick: Entwurf eines Rahmensystems für mobile Augmented-Realtiy-Anwendungen. - 2007. - 169 S. Darmstadt, Technische Universität Darmstadt, Fachbereich Informatik, Dissertation, 2008

Hausstädter, Uwe: Der Einsatz von Virtual Reality in der Praxis: Handbuch für Studenten und Ingenieure. - 2. Aufl. Berlin: Rhombos Verlag, 2010

H. Wagenbrunn, Karl: Rendering and External Authoring Functionality for VRwave. - 1998. - 135 S. Graz, University of Technology, Institute for Information Processing and Computer Supported New Media (IICM), Master's Thesis, 1998

Krüger, Guido; Stark, Thomas: Handbuch der Java-Programmierung: SWING Grundlagen. URL: <http://stl-www.htw-saarland.de/syst-lab/java/hdjp/k100229.html#sectlevel3id035001001>, verfügbar am 22.12.2010

Luis, Dirk; Müller, Peter: Jetzt lerne ich Java: Der einfache Einstieg in die Internetprogrammierung. - 1. Aufl. München: Markt und Technik Verlag, 2000

N.Matsuba, Stephan; Roehl, Bernie: VRML - Das Kompendium: Einführung - Arbeitsbuch - Nachschlagewerk. - 1. Aufl. Haar bei München: Markt & Technik, 1996

Riege, Kai: Mehrbenutzerinteraktion in einem projektbaserten Two-Viewer System. - 2006 - 97 S. Weimar, Bauhaus-Universität Weimar, Fakultät Medien, Diplomarbeit, 2006

S. Ziezold, Hendrik: Automatische generierte Laufmuster für virtuelle Charaktere. – 2006 – 84 S. Koblenz, Universität Koblenz Landau, Fachbereich Informatik, Diplomarbeit, 2006

Steger, Daniel: Motion Capture mit optisch-magentischem Trackingsystemen in VR-Applikationen. - 2004. - 139 S. Chemnitz, Technische Universität Chemnitz, Fakultät für Informatik, Diplomarbeit, 2004

Stricker, Didier; Bockholt, Ulrich: Augmented Reality: Neue Anwendungsfelder durch innovative Technologien, Darmstadt 2006, INI-GraphicsNET, Nr. 02|2006

The Virtual Reality Modeling Language: International Standard ISO/IEC 14772-VRML97. URL: <http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97/>, verfügbar am 01.06.2010

# 8. Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Chemnitz, den 31.01.2011