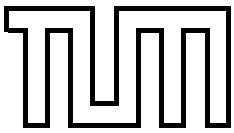
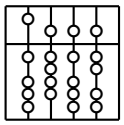


Technische Universität München
Fakultät für Informatik



Systementwicklungsprojekt

Integration eines X-Servers in VR-Systemen

Georgi Nachev, Dimitar Marinov

Aufgabenstellerin:	Prof. Gudrun Klinker, Ph.D.
Betreuer:	Dipl.-Inf. Marcus Tönnis
Abgabedatum:	18. November 2005

Ich versichere, dass ich dieses Systementwicklungsprojekt selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 18. November 2005 Georgi Nachev

München, den 18. November 2005 Dimitar Marinov

Zusammenfassung

Virtual Reality (VR) ist ein Paradigma für die Mensch-Maschine Interaktion. Damit bezeichnet man eine vom Computer geschaffene, interaktive, dreidimensionale Welt, in die eine Person eintaucht. Die Visualisierung einer solchen Umwelt wird durch ein VR-Software-System erreicht, das ist im allgemeinen eine Objekt-orientierte und Event-basierte Lösung zum Verwalten von Szenegraphen. Von allen existierenden Arten von VR-Systemen legen wir Wert in dieser Arbeit auf den immersiven Systemen, die den Sichtpunkt des Benutzers komplett in das Innere einer virtuellen Welt verschieben. Eine solche Verschiebung ist technisch gesehen heutzutage häufig durch Anwendung von einem Head Mounted Display (HMD), oder einer Cave Automatic Virtual Environment (CAVE) erreichbar. Ein HMD ist eine Brille mit eingebautem Computerdisplay, in dem virtuelle Objekte projiziert werden. Eine CAVE dagegen ist ein ganzer Projektionsraum, der von mehreren Personen betreten werden kann. Die virtuelle Objekte werden auf die Wände (und den Fußboden) projiziert und so entsteht eine immersive Illusion. Innerhalb dieses Raumes hat man das Gefühl sich in einer realen Welt zu befinden.

Ein X Window System (X-Server) ist eine Sammlung von Protokollen, Anwendungen und Standards und dient zur Darstellung und Ansteuerung grafischer Benutzeroberflächen vor allem unter Unix-Systemen. Der X-Server kommuniziert mit den Ein-/Ausgabegeräten (Bildschirm, Maus, Tastatur), mit der Grafikkarte und mit dem Betriebssystem und ist die Grundlage für einen grafischen Desktop-Environment.

In dieser Arbeit wird ein verteiltes System entwickelt, das die Oberfläche eines X Window System als Textur in dem virtuellen Raum verschiedener VR-Systeme darstellt. Diese Textur kann auf einem beliebigen Körper abgebildet werden. Auf dieser Weise entsteht ein virtuelles Desktop-Environment, das eine immersive 2D-Benutzerschnittstelle darstellt. Dieses kann als Grundlage für neue Kommunikationsschnittstelle und Interaktionsmöglichkeiten für den Benutzer verwendet werden, der beliebige Anwendungen (wie File-Browser, Multimedia-Software, Web-Browser etc.) direkt in der immersiven Umwelt über ein Eingabegerät starten und bedienen kann.

Abstract

Virtual Reality (VR) is a paradigm for the man-mashine interaction. It designates a computer-created, interactive, three-dimensional world, where a person immerses. The visualisation of such environment is achieved through a VR software system, which in general represents a object-oriented event-based solution for szenegraph management. From all existing kinds of VR-systems, in this work we set the value on the immersive systems, which completely transforms the viewing point of the user inside a virtual world. Technically such translation is achieved nowadays by applying a Head Mounted Display (HMD), or a Cave Automatic Virtual Environment (CAVE). HMD devices are glasses with integrated displays, where the objects are projected. A CAVE in contrast is a whole projection room, which many persons can enter. The virtual objects are projected on the walls (and the floor) arising an immersive illusion.

X Window System (X-Server) is a collection from protocols, applications and standards which are used for displaying and controlling of grafical user interfaces primarily on UNIX-Systems. The X-Server communicates with the I/O devices (display, mouse, keyboard), with the grafic card and the operating system and is the foundation for a grafical desktop environment.

This work presents a distributed system, which displays the user interface of a X window system as a texture in the virtual space of different VR-systems. The texture can be mapped to any shape. That way arises a virtual desktop environment, which is a immersive 2D user interface. This can be used as basis for new kind of communikation interface and interaction with the user, who can execute and control with a input device arbitrary applications (File-Browser, Multimedia-Software, Web-Browser etc.) directly in the immersive environment.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Jahrzehnte nach der Erfindung des Rechners stellt die Software und ihre Schnittstelle zum Menschen immer noch eine Abstraktion der Realität für uns her, mit deren Hilfe wir unsere realen Probleme lösen. Virtual Reality ist kein neuer Begriff (erste Ansätze für Flugsimulatoren vor und während des zweiten Weltkrieges), der uns verspricht, den Abstraktionsgrad zu reduzieren und auf dieser Weise redundante Arbeit zu sparen. Neue Interaktionsmöglichkeiten entstehen und der Benutzer ist nicht mehr auf Bildschirm, Tastatur und Maus angewiesen, hat aber die Möglichkeit, sich frei zu bewegen und mit einer virtuellen Welt direkt zu interagieren, ähnlich wie er mit den Objekten in der realen Welt umgeht. Obwohl all diese Technologien eine sehr intuitive Mensch-Maschine Kommunikation ermöglichen, werden Metapher für einen Schreibtisch - Desktop Environment und Blatt Papier - Text-Dokument wahrscheinlich noch lange in der Computer-Welt und speziell in der Virtual Reality existieren.

In diesem Systementwicklungsprojekt (SEP) konzentrieren wir uns auf das Darstellen eines 2-dimsionalen virtuellen Desktops, so wie wir ihn von dem Rechner zu Hause kennen, frei in dem virtuellen Raum. Auch wenn sowas auf dem ersten Blick degressiv erscheint, sehen wir in einer solchen Kommunikationschnittstelle die Möglichkeit zur Integration einer aus dem PC-Umfeld langfristig bekannten Mensch-Maschine-Schnittstelle. Diese kann als Grundlage für neue und innovative User Interface - Paradigmen dienen, die sich der ganzen Fülle an Möglichkeiten von Layouting, Styles und Action-Handling der klassischen WIMP-Elemente [37] bedienen können.

Das Projekt entstand nach den Gesprächen mit Herrn Stefan Augustiniack, Abteilung Entwicklung und Konstruktion der BMW AG München. Insofern sind alle Requirements des Systems reelle Anforderungen, die aus der täglichen Arbeit mit virtuellen Modellen und immersiven VR-Systemen im Hause BMW AG entstanden sind. Da die Anwender nicht komplett mit den Interaktionsmöglichkeiten in einer CAVE zufrieden waren, wurde nach Wege gesucht, wie diese Schnittstelle verbessert werden kann, bzw. wie eine neue geschaffen werden kann. Dadurch entstand die Idee, die Oberfläche eines X Window Systems als Textur in dem virtuellen Raum dieser CAVE darzustellen und die Inhalte (Anwendungen) dieses virtuellen Desktops über das CAVE-Eingabegerät zu steuern. In unserem SEP beschäftigen wir uns nur mit der Darstellung eines solchen virtuellen Desktops, das die Grundlage für die gewünschte neue Mensch-Maschine Schnittstelle darstellt. Die Interaktion mit dem Desktop soll im Rahmen eines Nachfolgeprojektes zwischen BMW AG und TU München realisiert werden.

2 Kontext dieser Arbeit

Um ein besseres Verständnis für die Anforderungen und den Aufbau des Systems zu ermöglichen, werden wir in diesem Kapitel den Kontext dieses Projektes vorstellen. Es werden die Technologien und Projekte aus der Sicht unserer konkreten Aufgabestellung präsentiert.

2.1 Virtual Reality Überblick

Virtual Reality (VR) ist ein alter Begriff, der sehr vielfaltig verwendet wird und für den es mehrere artverwandte Definitionen gibt. Unserer Meinung nach beschreibt die folgende Definition am besten die Dualität des Begriffs:

Virtual Reality ist einerseits eine den menschlichen Sinnen vorgetäuschte, künstlich erzeugte Umgebung, die eine Simulation verschiedener Eigenschaften unserer Realität darstellt, andererseits aber auch das Fachgebiet, das sich mit deren Erzeugung beschäftigt. Diese Umgebung ist eine vom Computer geschaffene, interaktive, dreidimensionale Welt, in die eine oder mehrere Personen eintauchen. Dadurch wird es möglich, Objekte als real zu erfahren, die es nicht mehr, noch nicht, oder nicht erreichbar gibt.

Virtual Reality ist aus verschiedenen Sichten [7]:

- Kommunikationstheoretisch - der Bereich der Kommunikation, welcher in synthetischen Räumen stattfindet und den Menschen als gleichberechtigten, integralen Bestandteil eines digitalen Systems versteht.
- Technisch - die Gesamtheit von Hard- und Software, welche dem Benutzer einen ihn einbeziehenden drei- oder mehrdimensionalen Ein-/Ausgaberaum zur Verfügung stellt, in dem er zu jedem Zeitpunkt mit autonomen Objekten in Echtzeit interagieren kann.

Als zentralen Aspekt kann man die Realitätsnähe der vom Computer erzeugten Daten ansehen. Um eine Realitätsnähe zu erzeugen, müssen immer komplexere Daten und deren Wechselwirkung untereinander verarbeitet und visualisiert werden.

Anders als bei der Arbeit an einem konventionellen Computer-Bildschirm, wo sich der Benutzer mit Darstellungen auf einem stark beschränkten, zweidimensionalen Medium begnügen muss, hat man in einem Virtual Reality System die Möglichkeit, virtuelle Objekte

so zu präsentieren, dass sie von den Menschen als real wahrgenommen werden. Eine realitätsnahe Wahrnehmung der virtuellen Umgebung wird von einer Reihe von Faktsehenoren geprägt:

- großformatige Projektion - die Ränder des Gesichtfeldes sollten innerhalb der Darstellung liegen.
- stereoskopische Darstellung - erst dadurch wird der optische Sinn voll ausgenutzt, Geometrien dreidimensional wahrzunehmen.
- Positionsverfolgung (Head Tracking) - zur Anpassung der Perspektive an den Betrachterstandpunkt.
- intuitive Interaktionsmöglichkeiten mit virtuellen Objekten.
- Immersion - der Effekt, wenn der Benutzer sich psychisch vollständig in der virtuellen Welt befindet, oder anders gesagt - ´´eingetaucht“ ist. Das kann auch durch Vortäuschen anderer Sinne als nur der Gesichtssinn erreicht werden - akustische Täuschung, Tastsinn etc.

Abbildung 2.1: Anwendungsbereich Flugsimulation. Zur Verfügung gestellt von [13].

Und auch wenn wir technisch immer noch von einem perfekten Illusion einer virtuellen Welt sehr entfernt sind, ist das Spektrum an möglichen Anwendungen denkbar groß. Einsatzgebiete sind vor allem große Forschungseinrichtungen und Industrieunternehmen, die diese Technologie in folgenden Bereichen verwenden:

- Industrielle Fertigung wie z.B. Montagesimulation, Digital MockUp, Ergonomieuntersuchungen, Fertigungsanlagen-Entwurf, Digitale Fabrik
- Design - in der Automobilindustrie (Interior- und Exterioruntersuchungen), Innenaustattungen in der Architektur
- Ausbildung und Training - Flugsimulation - Abb. 2.1, Bedienung von Anlagen
- Medizin - Planung und Training von Operationen
- Naturwissenschaften - Visualisierung von wissenschaftlichen Daten z.B. in der Chemie, Physik, Biologie, Geologie. Siehe Abb. 2.2.

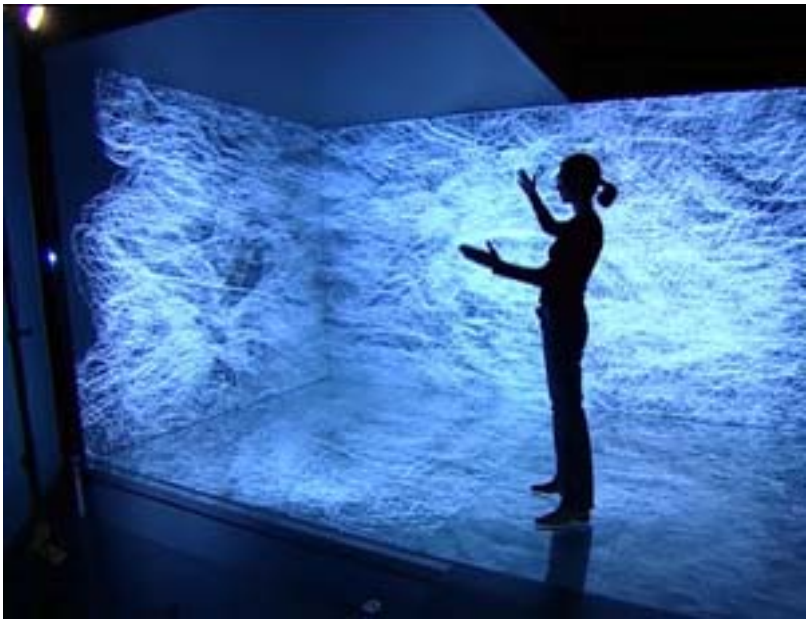


Abbildung 2.2: VR Erlebnis - Visualisierung wissenschaftlicher Daten in einem VR-System. Zur Verfügung gestellt von [25].

2.2 VR Hardware

Wir werden zwei Technologien vorstellen, die eine immersive Virtual Reality erzeugen - Head Mounted Display (HMD) und Cave Automatic Virtual Environment (CAVE). Unsere Lösung ist speziell für Benutzung in einer CAVE gedacht, kann aber ebenso auch in Verbindung mit einem Head Mounted Display benutzt werden.

2.2.1 Head Mounted Display

Head-Mounted Displays oder VR-Helme (kurz HMD) sind Geräte, die auf dem Kopf getragen werden und durch zwei eingebauten Displays (heutzutage LCDs) unmittelbar vor den Augen des Benutzers ein virtuelles Computerbild erzeugen. Ein am Head-Mounted-Display montierter Tracker ermittelt die Kopfposition und -orientierung und aus diesen Daten werden die Bilder berechnet, die auf den beiden Displays visualisiert werden. Über integrierte Lautsprecher werden die Klänge der virtuellen Welt wiedergegeben. Auf diese Weise wird der Benutzer von der realen Welt abgekapselt, sodass er von der realen Umwelt fast nichts wahrnimmt und das Gefühlt bekommt, in der virtuellen Welt eingetaucht zu sein.

Abbildung 2.3: Eine Reihe von verschiedenen HMDs. Zur Verfügung gestellt von [23].

2.2.2 Cave Automatic Virtual Environment

Die Cave Automatic Virtual Environment (kurz CAVE) erzeugt eine immersive Illusion mittels stereoskopischer Bilder, die auf die Wände (evtl. auch den Fußboden) eines würfelförmigen Projektionsraumes projiziert werden, den mehrere Leute gleichzeitig betreten können siehe Abb. 2.4. Ein Audio-Surround-System ist für das Vortäuschen des Gehörsinns des Benutzers zuständig. Der Bergriff wird sowohl für die Technologie als auch für das kommerzielle Produkt verwendet, das von der Firma Fakespace Systems Inc. momentan angeboten wird.

Das System wurde in dem Electronic Visualitaion Lab an der University of Illinois at Chicago entwickelt und zum ersten Mal auf der SIGGRAPH'92 Konferenz demonstriert.

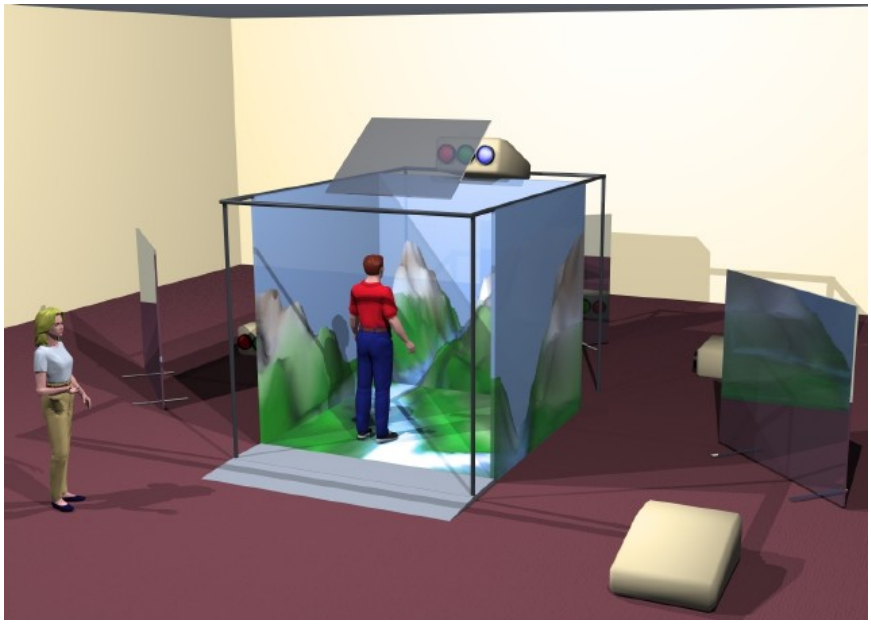


Abbildung 2.4: Die Aufbau einer CAVE. Zur Verfügung gestellt von [10].

Auch hier ist ein Tracking-System notwendig um die genaue Kopfposition und -orientierung des Benutzers zu ermitteln. Obwohl die CAVE von mehreren Anwender gleichzeitig betreten werden kann, ist nur der Kopf des Leaders der Gruppe getrackt. Alle anderen können die virtuelle Welt nur aus seiner Perspektive erleben. Das System kann mit verschiedenen Eingabegeräten ausgerüstet werden, wie z.B. Datenhandschuhe. Diese Geräte dienen der Ermittlung der Fingerstellung und können für Interaktion mit virtuellen Objekten verwendet werden.

Die stereoskopische Bilder werden über Rückwandprojektion generiert. Die Bilder der Rückwandprojektoren werden sehr oft um Platz zu sparen durch Spiegel auf die Wände der CAVE gebracht. Man unterscheidet zwischen aktive und passive Stereoprojektion, in beiden Fällen trägt der Benutzer eine Stereo-Brille.

- Aktive Stereoprojektion - Es ist nur ein Beamer pro Wand notwendig, der aber mit doppelter Frequenz arbeitet (120Hz) und Bilder für die linke und die rechte Auge alternierend generiert. Diesentsprechend muss der Benutzer speziellere Brille (Shutter-Glasses) tragen, die dann mit derselben Frequenz auf einen Anblick von der entsprechenden Auge umschaltet.
- Passive Stereoprojektion - Für jede Wand werden zwei Beamer gesteuert, ein Beamer pro Auge. Die Beamer müssen synchronisiert sein. Auch hier sind Shutter-Glasses notwendig, wenn aber die ggf. eingebauten Spiegel die Polarisation des Lichtes beibehalten, kann mit geeigneten Beamern auch mit einer Pol(-arisations-)brille anstatt Shutter-Glasses gearbeitet werden.

Leider wird das Eintauchen bei dieser Technologie nur zum Teil realisiert, da der Benutzer immer noch Bezugspunkte zur realen Welt hat, wie z.B. die Ecken des Raumes, die man noch wahrnehmen kann. Außerdem kann er sich nicht beliebig in dem virtuellen Raum bewegen, da er schnell an die Grenzen des realen Raumes, der CAVE stößt - Abb. 2.5.

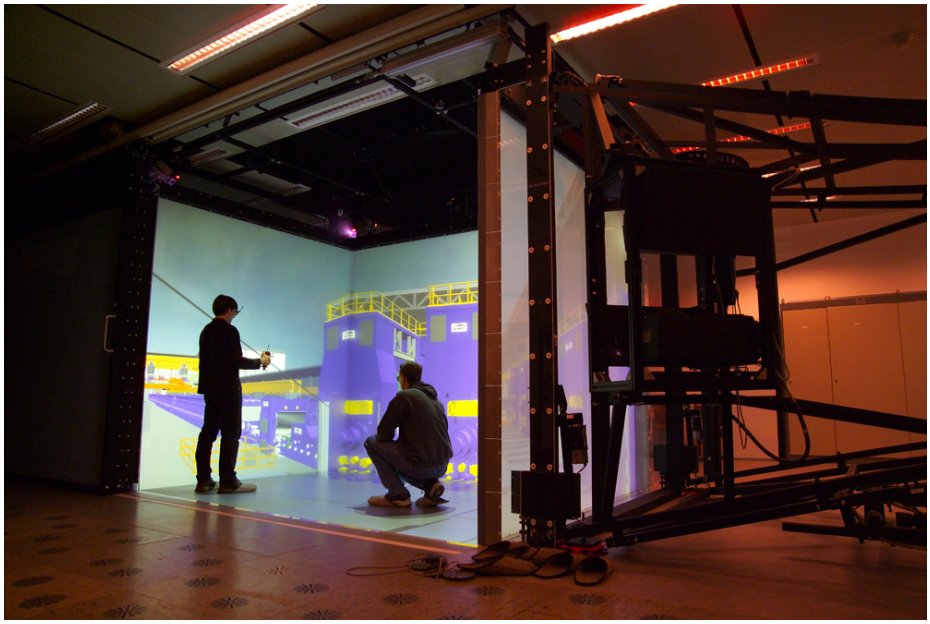


Abbildung 2.5: Anwender in einer CAVE - Bezugspunkte zur realen Welt sind vorhanden. Zur Verfügung gestellt von [8].

2.3 VR-Software-Systeme

Zur Erzeugung der Virtual Reality wird speziell dafür entwickelte Software-Systeme, die effizient komplexe 3D-Welten berechnen und visualisieren können. Fast all diese Systeme arbeiten auf dem Prinzip des Szenengraphen - alle Objekte der VR Welt (Szene) werden in einer hierarchischen Graphenstruktur gehandelt und bearbeitet. Die Knoten des Szenegraphs sind von verschiedensten Typen, wie z.B. Geometrische Primitiven, Materialien, Texturen, Lichtquellen, Transformationen. Auf dieser Weise sind alle notwendigen Informationen für eine problemlose Visualisierung dieser Szene in der Graph-Struktur enthalten. Das ist aber nicht der größte Vorteil eines solchen Systems, sondern die Geschwindigkeit mit der die Events abgearbeitet werden. Ein Event, das in einem Knoten generiert wird, oder an diesem übergeben wird, wird von dem Knoten bearbeitet und dann nur an seinen Kindern weitergeleitet. Das ergibt eine Performanz im Sinne, dass die Events nur von diesem Teil des Graphen (Baumes) bearbeitet werden, für den sie relevant sind.

Wir werden 5 verschiedene VR-Software-Systeme präsentieren, die mehr oder weniger für unsere prototypischen Lösung eine Rolle gespielt haben.

2.3.1 TGS Inventor

TGS Inventor ist ein objektorientiertes Framework zur Programmierung von dreidimensionalen Welten, das die Inventor - Spezifikation implementiert. Es ist entwickelt von SGI und ist de facto Standard für 3D Visualisierungssoftware in dem wissenschaftlichen und technischen Bereich geworden.

TGS Inventor bietet eine Sammlung von Bausteinen an, aus denen relativ einfach leistungsfähige Applikationen zusammengestellt werden können. Es baut auf OpenGL auf und kann von dem Benutzer auch modifiziert und erweitert werden, damit alle Applikationsanforderungen möglichst genau erfüllt werden - Abb. 2.6.

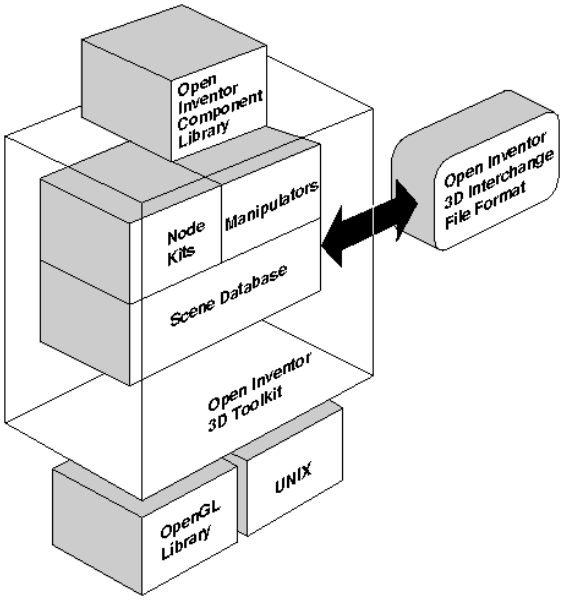


Abbildung 2.6: TGS Inventor Architektur. Zur Verfügung gestellt von [39].

Das in C++ geschriebene Framework verfügt über eine Reihe von grafischen Primitiven (shape, group, engine objects), interaktiven Manipulatoren (handle box, trackball) und Komponenten (material editor, light editor, examiner viewer, siehe Abb. 2.7) und bietet die Komfort und Leistungsfähigkeit eines vollständig Objekt-orientierten und Event-basierten System an.

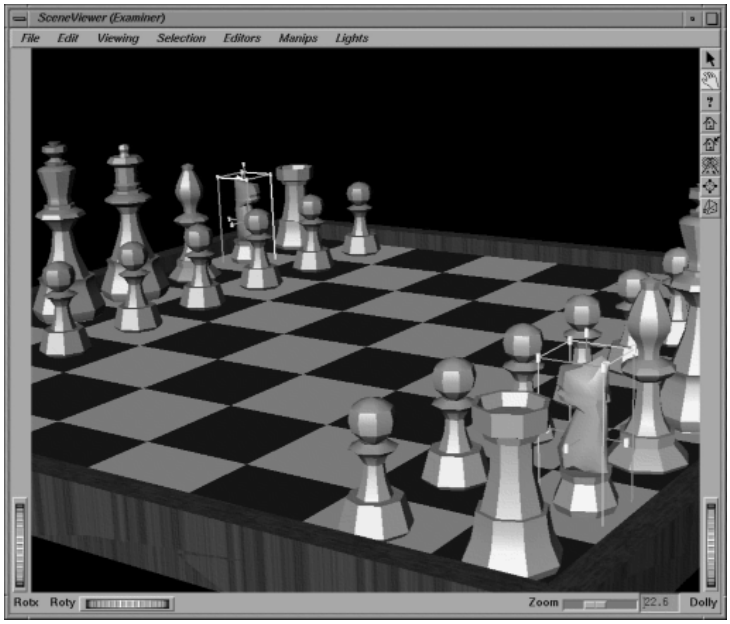


Abbildung 2.7: TGS Inventor Examiner Viewer. Zur Verfügung gestellt von [39].

TGS Inventor ist deswegen zum Standard geworden, weil er zuerst Lösung für die drei wichtigsten Anforderungen eines 3D-Visualisierungssystems angeboten hat:

- Repräsentation von Objekten - Grafiken sollten als modifizierbare Graphikobjekte verwaltet werden, und nicht nur als eine Menge von Zeichenprimitiven.
- Interaktivität - Zusätzlich zur graphischen Darstellung der Objekte muss es eine Möglichkeit zur Interaktion mit den Objekten der Szene angeboten werden. D.h. es ist eine Verwaltung von Ereignissen (Events) notwendig, mit deren Hilfe die Darstellung der Szene und die Eigenschaften der Objekte modifiziert werden können.
- Flexible und erweiterbare Architektur - Ist eigentlich eine Eigenschaft jedes guten Software-System.

2.3.2 Coin3D

Coin3D ist ein 3D Rendering-Toolkit entwickelt von Systems in Motion AS, der vollkommen kompatibel mit TGS Inventor ist und auch die Inventor - Spezifikation implementiert. Eine schnelle Entwicklung von Darstellung, Import und Interaktion mit 3D-Objekten ist angeboten, sowie Effizienzsteigerung durch die Möglichkeit, dass Coin3D und OpenGL Code in derselben Applikation gleichzeitig benutzt werden können. Andere Features sind Support von 3D-Sound, 3DTextures, paralleles Rendering über mehrere Prozessoren. Die verwendete Szenegraph-Datenstruktur macht das Framework ideal zum Benutzen in wissenschaftlichen und technischen Bereichen.

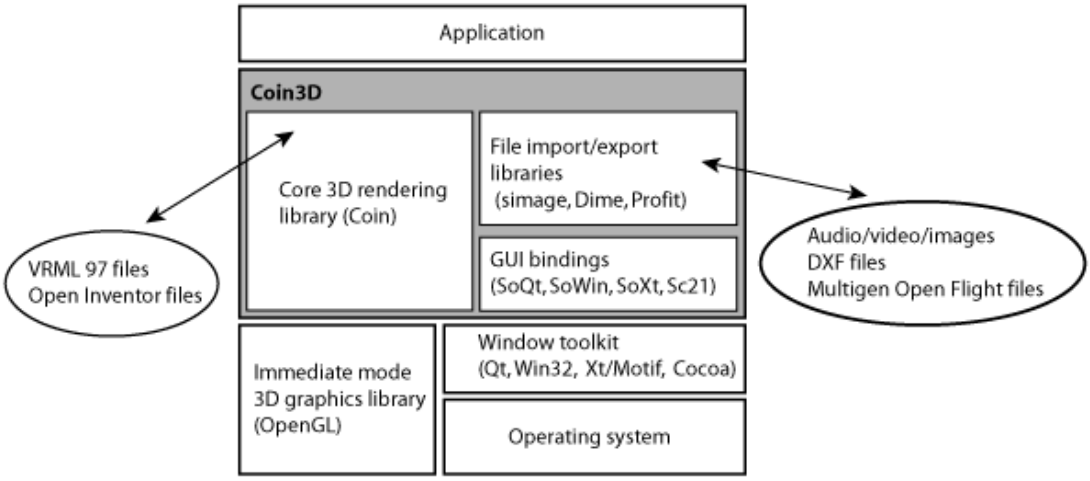


Abbildung 2.8: Coin3D Architektur. Zur Verfügung gestellt von [3].

Um Benutzerinteraktion in der gewünschten Applikation zu integrieren, und da die Software selbst als Rendering-Bibliothek realisiert worden ist, muss auch eine andere GUI-Bibliothek angebunden werden. Solche so genannte GUI-Bindings sind: Microsoft Windows GUI (SoWin), Trolltech's QT (SoQt), Xt/Motif (SoXt), und Mac OS X GUI (Sc21).

2.3.3 Virtual Design 2

Virtual Design 2 (VD2) ist ein umfangreiches Werkzeug für den durchgängigen Einsatz in verschiedenen Bereichen der Produktentwicklung, entwickelt von vrcom GmbH und Fraunhofer-IGD, Visualisation and Virtual Reality Department. Die Software ist modular aufgebaut und für Ein- und Ausgabegeräte flexibel konfigurierbar. Funktionen wie die Navigation, die Interaktion und der Aufbau erster Applikationen sind bereits in den Grundmodulen enthalten. Anwendungsorientierte Pakete erweitern die Grundmodule in ihrer Funktionalität für bestimmte Applikationen und Einsatzbereiche.

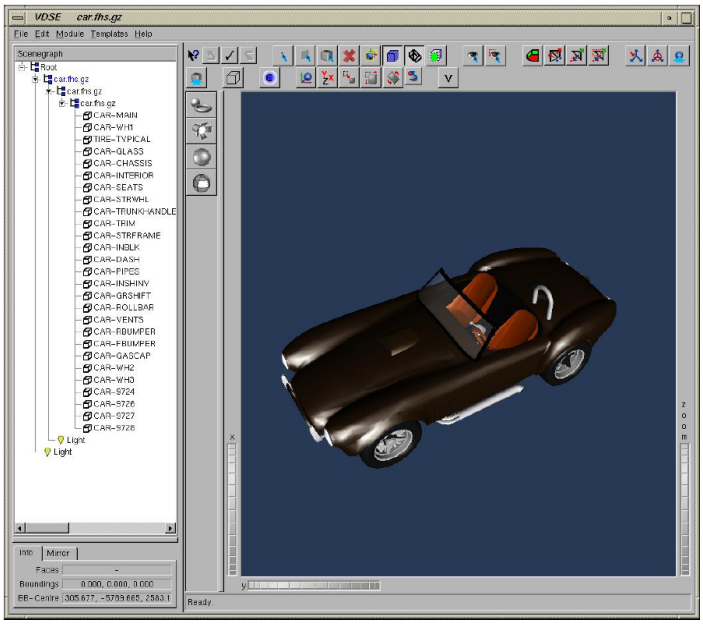


Abbildung 2.9: Virtual Design 2. Zur Verfügung gestellt von [26].

Der VD2-Kernel besteht aus drei Komponenten - Interaktionsmanager, Geräte-Manager und Rendering-Kernel.

Der Interaktionsmanager kontrolliert alle Actions in der virtuellen Umgebung und alle Benutzer-Interaktionen mit der virtuellen Szene. Er ist anpassbar mittels einer Konfigurationsdatei, wo alle statischen und dynamischen Eigenschaften des Systems beschrieben sind.

In dem Rendering-Kernel, der auf OpenGL basiert, wird die ganze Darstellung-spezifische Arbeit des Szenegraphen erledigt.

Das System kann anschließend um weitere Funktionalität erweitert werden. Dies geschieht durch dynamisches Laden von anderen Modulen.

2.3.4 VRED

Ein anderes Virtual Reality System für Visualisierung von komplexen 3D-Szenegraphen ist der Virtual Reality EDitor (VRED) von der Firma AMZ GmbH. Die Software ist basiert auf dem 3D-Renderer OpenSG und bietet Modul-basiertes Design an - Plugins können sogar im laufenden System geladen werden.

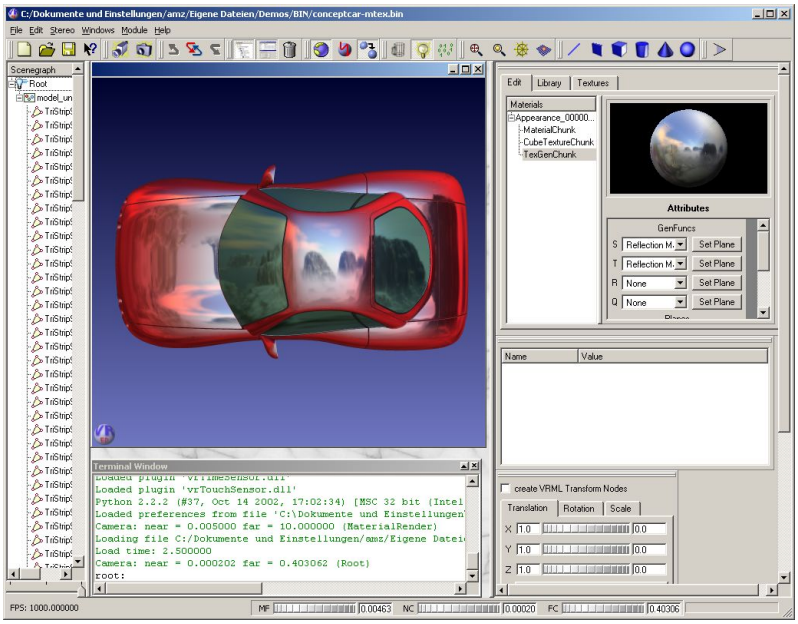


Abbildung 2.10: Virtual Reality EDitor. Zur Verfügung gestellt von [27].

Es werden zwei verschiedenen Versionen angeboten - Lite- und Pro-Version, hier ein paar Features von denen.

VRED Lite Features:

- Interaktive Programmierung mit Python
- VRML 2.0 Loader und Import von Inventor (.iv), Fraunhofer VR-data-exchange format (.fhs) und Stereo Lithography file format (.stl)
- Open Dynamics Engine ODE
- Partikel-System
- Frei definierbare 3D-Menüs und Windows
- Aktive und passive Stereoprojektion
- Multi-Texturierung
- Support von CG- und Phong-Shader
- Web Interface

VRED Pro Features:

- Support von Cluster-Architekturen

- Varianten
- Volumenrendering

2.3.5 OpenSG

Das OpenSG Projekt ist auf der SIGGRAPH'99 Konferenz initiiert, nachdem die Fahrenheit-Initiative von Microsoft, SGI und HP erfolglos beendet wurde. Da die Entwickler der Meinung waren, dass eine oder mehrere Firmen schwierig eine solche fundamentale Lösung zur Welt bringen können, wie ein VR Szenegraph-System, dass alle Anforderungen erfüllt, wurde das Projekt als Open Source Initiative gestartet.

Der größte Vorteil von OpenSG ist die einfache und effiziente Verwaltung von Cluster-Architekturen und Multithread-Datenstrukturen. Das Support von heterogenen Netzwerken, Betriebssystemen und Umgebungen macht das Framework sehr flexibel und die offene Quellcode ermöglicht eine schnelle und einfache Erweiterung um gewünschter Funktionalität.

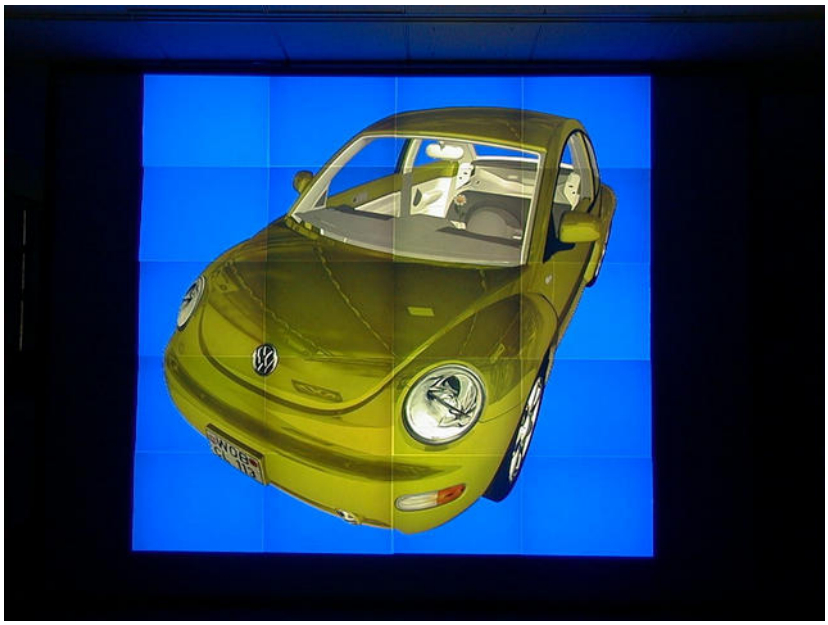


Abbildung 2.11: OpenSG Visualisierung. Zur Verfügung gestellt von [6].

Auch wenn die Hauptidee war, die perfekte Lösung zu finden, hat das System auch seine Nachteile. Vor allem ist das Framework noch in einer früheren Phase der Entwicklung und deswegen sind die einzelnen Komponenten von dem Benutzer selbst in einer Applikation zu binden. Diese Unannehmlichkeit wird aber in den nächsten Versionen sicher gelöst sein.

3 Aufgabenstellung

Die genaue Aufgabenstellung dieses Projektes resultiert aus den Gesprächen, die wir mit Herrn Stefan Augustiniack, Abteilung Entwicklung und Konstruktion BMW AG München und mit unserem Betreuer Marcus Tönnis geführt haben. Insofern sind alle Requirements des Systems einerseits reelle Anforderungen, die aus der täglichen Arbeit mit virtuellen Modellen und immersiven VR-Systemen im Hause BMW AG entstanden sind und andererseits sind sie mit den Forschungsarbeiten der Augmented Reality Research Group mit Leiter Prof. Gudrun Klinker abgestimmt.

Ziel dieses Projektes ist, ein System zu entwickeln, das die Darstellung eines X Window System als Textur in dem virtuellen Raum von verschieden VR-Systemen ermöglicht.

Die dadurch entstehende virtuelle 'Desktop-Environment' kann als neue immersive Benutzerschnittstelle (z.B. in einer CAVE) verwendet werden, indem alle Anwendungen, die von diesem X Window System visualisiert und verwaltet werden, über ein Standard-Eingabegerät (Flystick, Dataglove etc.) gesteuert werden können. Da wir uns in diesem Projekt nur mit der Darstellung eines solchen virtuellen Desktops beschäftigen ist die Interaktion nicht Teil dieser Arbeit und ist als Nachfolgeprojekt zu realisieren.

Alle im Kapitel 2.3 vorgestellten VR-Software-Lösungen sind potenzielle Einsatzgebiete für das System. Es sind prototypische Lösungen für zwei von diesen VR-Frameworks zu erarbeiten, damit verschiedene Tests durchgeführt werden können und eine Abschätzung in Hinsicht auf die Performanz des Systems gemacht werden kann.

Coin3D, als Software, die in der Augmented Reality Research Group zur Visualisierung von 3D-Szenen verwendet wird ist das eine System. Das zweite System wurde von Herrn Stefan Augustiniack bestimmt, da dies in seinem Arbeitsumfeld eingesetzt wird. Am Anfang unseres Projektes war das Virtual Design 2. Im Laufe der Zeit kam aber die Information von der Seite der BMW AG, dass diese Software nicht mehr produktiv in der CAVE im Einsatz ist und deswegen eine Lösung für VRED zu erarbeiten ist.

4 Anforderungsanalyse

Anhand folgendem Szenario wird die Anforderungsanalyse des Systems gemacht:

Scenario: **Materialien und Varianten umschalten**

Actor instances: `Augustiniack:User`

Flow of Events:

1. Herr Augustiniack will ein 3D-Modell von dem Interieur eines Autos in einer CAVE visualisieren und dabei zwischen verschieden Varianten, Materialien und Farben der Mittelkonsole umschalten. Er startet einen X-Server auf dem Rechner, wo die Software für die Variantensteuerung installiert ist, und danach die Software selbst.
2. Anschließend konfiguriert er das XinCave-System für den gestarteten X-Server, und die CAVE. Herr Augustiniack gibt an, auf welchem Körper von dem Szenegraphen die Textur mit der X-Server-Darstellung gemappt wird - eine Ebene, die nicht Teil des 3D-Modells des Autos ist.
3. Danach startet er die XinCave Applikation und die CAVE Visualisierungssoftware VRED.
4. VRED bekommt den für die Erstellung der Textur notwendigen Datenstrom von XinCave.
5. Das Auto-Modell wird in der CAVE geladen und dargestellt, die Textur mit der X-Server Darstellung ist auf der gewünschten Ebene gemappt worden und wird ortsfest in dem virtuellen Raum visualisiert.
6. Zusammen mit Kollegen von anderen Abteilungen seiner Firma geht Herr Augustiniack in die CAVE, um ein Meeting durchzuführen, auf dem entschieden werden muss, welche Varianten der Mittelkonsole des Autos auf einer Präsentation vorgestellt werden.
7. Mittels des CAVE-Eingabegerätes ART Flystick markiert Herr Augustiniack den virtuellen Desktop (die Ebene mit der Textur) und verschiebt ihn so, dass das Auto-Modell nicht von ihm überdeckt wird.
8. Anstatt jedesmal wenn eine Variante umgeschaltet werden muss aus der CAVE auszugehen und die Umschaltung am PC zu machen, bedient Herr Augustiniack die Software direkt von der CAVE über den virtuellen Desktop und spart somit Meetingszeit. Aus dem ART Fystick werden Maus-Events (Bewegungen und Klicks) für den X-Server generiert, die zu den X-Server-System über das Netzwerk übertragen werden.

Im folgendem werden wir detailliert die einzelnen Anforderungen des Systems analysieren, die später als Grundlage für das System Design benutzt werden.

4.1 Functional Requirements

Folgende Stichpunkte verschaffen uns ein Überblick über die Funktionalitätsbereiche des Systems:

Darstellung Das Bild (Frames-Strom) eines X-Servers muss in den VR-Systemen darstellbar sein.

Mapping Von den einzelnen Frames des X-Servers muss eine aktive Textur (ähnlich wie eine Movie-Textur) in den verschiedenen VR-Systemen erzeugt werden. Diese Textur kann der Benutzer auf einem beliebigen Objekt (Körper), wie z.B. eine Ebene oder einen Zylinder mappen.

Bildübertragung Eine Bildübertragung der einzelnen Frames an mehreren Rechner über ein Netzwerk muss möglich sein.

Transformation Die Darstellung muss transformierbar sein, damit der Benutzer die Möglichkeit hat, die zu verschieben, rotieren usw.

Interaktion Da das System als Grundlage für eine neue Benutzerschnittstelle entwickelt wird, muss die Möglichkeit angeboten werden, den X-Server entfernt (aus der CAVE heraus) zu steuern.

4.2 Non-functional Requirements

Eine Reihe von Non-functional Requirements definieren detaillierter eine benutzbare Lösung:

Schnelle Bildcapturing, Übertragung, Rendering - min. 24 Frames pro Sekunde, damit ein fließendes Bild entsteht, das dem Benutzer ein interaktives Gefühl vermittelt.

Kommunikationsbandbreite muss auf max. 100Mbit begrenzt werden, damit ein heute noch handelsübliches 100Mbit-LAN als Kommunikationsnetzwerk zwischen der einzelnen Komponenten ausreicht.

Wunsch-Auflösung von 1024x768 Pixel und 24Bit Farbtiefe ist zu erreichen.

Leichte Portierbarkeit - Da es sehr viele VR Softwaresysteme auf dem Markt gibt muss eine leichte Portierbarkeit zu anderen Viewer möglich sein.

VR-Software-Systeme Eine Lösung für Coin3D und VRED ist zu erarbeiten.

5 Existierende Lösungen

Die Idee, ein X Window System in einem VR-System zu integrieren ist nicht neu, dementsprechend existiert eine Reihe von Projekten, die sich mit dem Thema beschäftigen. Wir haben versucht eine Lösung zu erarbeiten, die möglichst performant ist, hohe Bildqualität und hohe Frame-Raten bietet und somit eine bessere Alternative dieser Projekte ist.

5.1 X11 in Virtual Environments

In seiner Arbeit [31] stellt Phillip Dykstra als erster die Idee vor, einen X Server in VR einzusetzen. Die Lösungsansätze, die er in dem Design seines Systems verwendet, sind nach wie vor bis heute noch aktuell - Shared Memory und Darstellung des virtuellen Desktops als Textur. Bei der Shared Memory handelt es sich um einen Bereich im Arbeitsspeicher, der von mehreren Prozessen zur Kommunikation genutzt werden kann. Prozesse können in diesem Speicherbereich Daten schreiben oder auch lesen.

Allerdings haben die Tests, die auf einem SGI IRIS 4D Workstation durchgeführt worden sind, nur 1 Frame pro Sekunde Bildrate gezeigt, bei einer Auflösung des X11 Servers von 512x512 Pixel. Mit dieser Frame-Rate sind kein fließendes Bild und keine Interaktion mit dem Benutzer möglich.

5.2 Windows on the World

Ein anderes Projekt [32], das ungefähr um dieselbe Zeit wie die Arbeit von Phillip Dykstra in Entwicklung war, erarbeitet eine Lösung für Augmented Reality - das Prototyp benutzt ein See-Through Head Mounted Display um über einen X Server Anwendungen als Fenster in der Ansicht des Benutzers zu visualisieren. Es werden drei verschieden Arten von Fenster unterstützt - mit fester Position in der Umwelt, HMD-gebunden und Objekt-gebunden. Auf dieser Weise wird eine so genannte *X-basierte Augmented Reality* präsentiert. Das ist eine Lösung, die weniger mit unsrem Wunsch-System zu tun hat, die aber ein Beweis dafür ist, dass unser System nicht nur in Virtual Environments benutzt werden kann, sondern auch in der Augmented Reality eine Anwendung finden kann. Leider ist die Leistung des Systems relativ gering um eine echte Benutzerschnittstelle zu ermöglichen - es wird über Frame-Raten von ungefähr 10 Bilder pro Sekunde berichtet.

5.3 ARWin

Die von Dykstra in [31] geäußerte Meinung, dass ein X-Server generell ein guter Ansatz für Lösung des Problems *Starten von Anwendungen in einem Virtual Environment* ist, hat das ARWin-Projekt [30] inspiriert. Es wird einen 3D Augmented Reality Desktop System präsentiert, das auf dem Basis von zwei Servern funktioniert - Event Manager und Display Manager, die zusammen die Funktionalität von einem X Window System und einem Window Manager System abdecken. Über VNC und Texture-Mapping wird dem Benutzer die Möglichkeit gegeben, verschiedene Applikationen in seiner augmentierten Umgebung zu starten und zu bedienen. Sowohl hier als auch in [32] ist als Nachteil die Beschränkung auf einem Benutzer zu erwähnen. Für kollaborative Anwendungen, welche die produktiv angewendeten VR-Systemen sind, ist aber eine solche Begrenzung undenkbar.

5.4 XiVE

Eine Rendering-Library mit dem Namen XiVE [28], die frei zum Downloaden zur Verfügung steht, haben wir gefunden erst nachdem wir fast fertig mit der Implementation unseres Systems waren. Diese Bibliothek bietet eine ähnliche Lösung zum unseren Problem - einzelne Fenster aus dem X Server können in einem virtuellen Raum dargestellt und gesteuert werden. Auch hier sind aber die Darstellungsraten zu niedrig - ungefähr ein Bild pro Sekunde bei einer Auflösung von 1024x768 Pixel und 100% Prozessorauslastung. Für das *Grabben* der einzelnen Frames wird hier die Funktion XGetImage() von dem X-Library benutzt, mit dem wir auch Tests durchgeführt haben, die aber schlechte Ergebnisse gezeigt haben - siehe Kapitel System Design.

5.5 VEWL

Virtual Environment Windowing Library (VEWL) [35] ist eine Objekt-orientierte API für Erstellen von immersiven 3D-Menüs. Alle VEWL Widgets in der Bibliothek sind als eine Qt-ähnliche Struktur implementiert und benutzen für die Visualisierung SGI Performer Aufrufe. Da aber nur eine begrenzte Anzahl von Widgets unterstützt wird und keine Möglichkeit angeboten wird, beliebige Anwendungen darzustellen, ist diese Library nicht für das Erstellen eines umfassenden virtuellen Desktop Environments geeignet.

5.6 Interaktive Visualisierung von Displayinhalten in VR

Das Thema dieser Diplomarbeit [34] ist das dynamisches Darstellen von beliebigen Displayinhalten in einem Virtual Reality System. Interessant hier ist die Untersuchung nach den prinzipiell möglichen Wegen zur allgemeinen Lösung des Problems - über Videosignal, externe Software, Emulation, Streaming. Eine Abschätzung wie gut diese Lösungsalternativen die Anforderungen des Systems erfüllen, ergibt unseren Erwartungen nach, dass ein externer Prozess Bild-Capturing - Übertragung - Bild-Mapping die effizienteste Alternative ist. Allerdings erfüllt die prototypische Lösung, die für das zentrale Steuerdisplay in dem 7er

BMW entwickelt wird, nicht die Requirements unseres Projektes. Die Performanz des Systems ist abhängig von dem verwendeten Bildformat für die einzelnen Frames und hat eine obere Grenze von ungefähr 3FPS bei einer Auflösung von 1000x1000 Pixel und volle Auslastung einer 100Mbit LAN-Verbindung.

6 System Design

In diesem Kapitel bilden wir das Design des Systems. Zuerst werden kurz die Ziele präsentiert, die wir durch das Design erreichen wollen. Dann wird das System in Subsysteme aufgeteilt und jedes Subsystem wird einzeln betrachtet. Verschiedene Möglichkeiten werden untersucht und verglichen. Anhand der Anforderungen und der Designziele wird die am besten geeignete Alternative ausgewählt.

6.1 Ziele

- Performanz – Möglichst hohe Frameraten bei möglichst niedriger CPU-Belastung, damit das System auch auf älterer Hardware laufen kann.
- Kleinere Netzbandbreite – Die Übertragung der Daten im Netzwerk muss möglichst kleine Bandbreite erfordern, so dass das System in allen gängigen Netzwerken eingesetzt werden kann, ohne andere Anwendungen zu verlangsamen.
- Hohe Bildqualität – Das Bild muss im Zielsystem ohne Flimmern und sichtbare Qualitätsverluste visualisiert werden.
- Skalierbarkeit – Bessere Hardware muss auch bessere Ergebnisse (Frameraten, Auflösung, Anzahl der angeschlossene Zielsysteme) erlauben. Das starten weiterer Viewer darf das Gesamtsystem nicht verlangsamen.
- Erweiterbarkeit – Neue Funktionalitäten (z.B. ein neues VR-System integrieren) müssen leicht zu realisieren sein.
- Wiederverwendbarkeit – Die einzelnen Komponenten müssen entkoppelt und unabhängig sein, damit sie auch für andere Zwecke leicht wiederverwendet werden können.

6.2 Subsystem Decomposition

Das System wird in fünf Subsysteme zerlegt.

1. Bildcapture – Extrahiert die einzelnen Bilder aus dem X-Server
2. Steuerung – Übergibt die Steuersignale dem X-Server

3. Kommunikation

   Bildkommunikation – Transportiert Bilddaten zu alle Viewerrechner

   Steuerungskommunikation – Stellt einen Rückkopplungskanal zur Benutzerinteraktion zur Verfügung.

4. Darstellung – Visualisiert die Bilddaten im Viewerapplikation.

5. Eingabe – Bindet diverse Eingabegeräte (z.B. ART FlyStick) zum System ein.

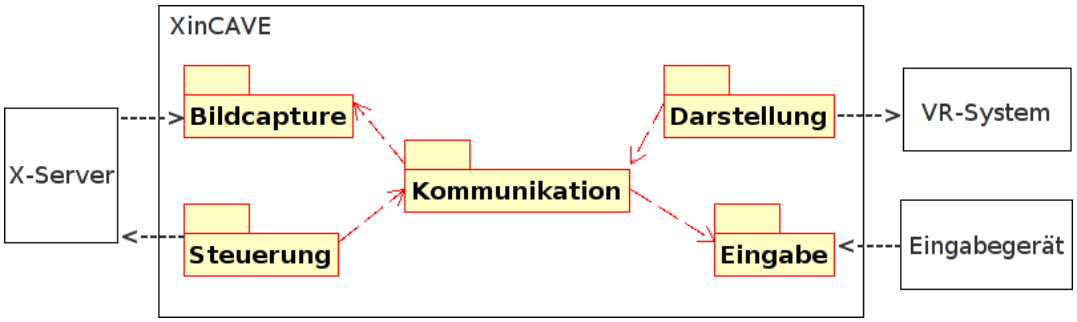


Abbildung 6.1: Subsystem Decomposition

Im Rahmen dieses Projekts werden nur Subsysteme 1, 3 und 4 entwickelt. Die Subsysteme Eingabe und Steuerung bleiben für ein Nachfolgeprojekt.

Für jedes Subsystem gibt es alternative Möglichkeiten zur Realisierung. Die Entscheidung welche Alternative für ein Subsystem ausgewählt wird hat einen großen Einfluss auf das ganze Systemdesign. Abbildung 6.1 zeigt wie die einzelnen Subsysteme verbunden sind.

Da alle Andere Subsysteme mit dem Kommunikationssubsystem gekoppelt sind, hat es den größten Einfluss. Deswegen haben wir zuerst ein Konzept für die Kommunikation entwickelt.

6.2.1 Kommunikation

Für diese Applikation sind sehr große Datenmengen im Kommunikationsnetzwerk zu erwarten. Zum Beispiel bei eine Auflösung von 1280x1024 Pixel und 24Bit Farbtiefe werden fast 4MB Rohe Daten pro Bild generiert. Damit man ein fließendes Video erreicht müssen mindestens 24 Bilder pro Sekunde im Viewerprogramm dargestellt werden. Falls die diese Rohe Daten direkt über das Netzwerk übertragen werden, macht das mindestens 755Mbit/s Bandbreite. Weiter muss das Videostrom an alle Viewerrechner übertragen werden, was die nötige Bandbreite weiter erhöhen kann, wenn 1:1 Verbindungen umgesetzt werden. Deswegen ist es eine echte Herausforderung die Bilddaten von dem Sourcerechner nach alle Viewerrechner ohne oder mit möglichst geringe Qualitätsverluste und eine angemessene Framerate zu übertragen. Wir haben drei verschiedene Lösungen untersucht.

1. Socketbasiert
2. VNC-basiert
3. X-Protokoll basiert

Die Lösungen sind komplementär, deswegen muss nur eine ausgewählt und realisiert werden. In den folgenden Absätze werden die Lösungen und ihre Vor und Nachteile detailliert erklärt.

6.2.1.1 Socketbasierte Kommunikation

Die Bilddaten werden über Sockets von ein Serverprogramm an mehrere Klientprogramme geschickt.

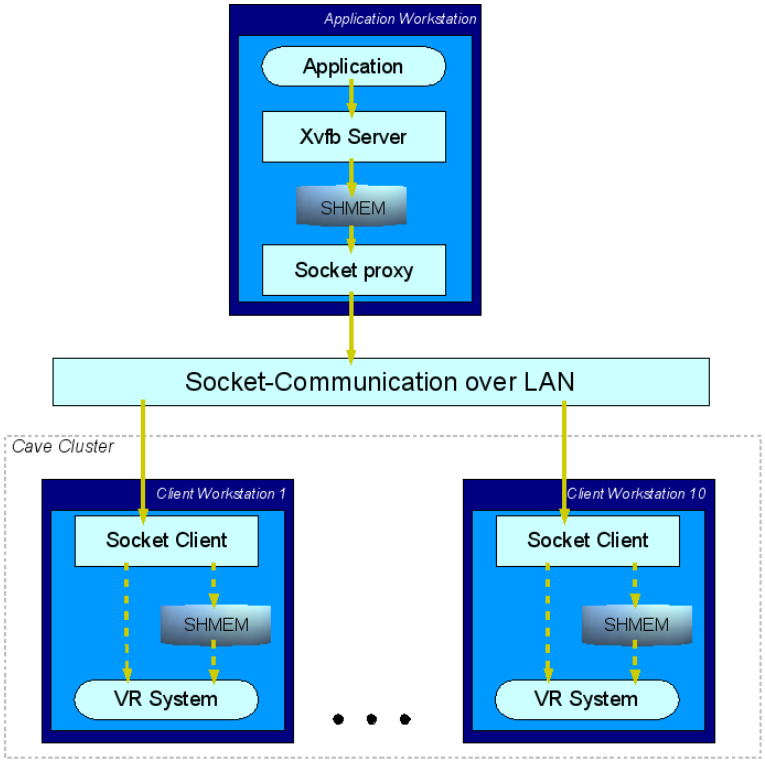


Abbildung 6.2: Socketbasierte Kommunikation

Die Klient-und Serverprogramme sind von uns zu implementieren. Damit die nicht funktionale Anforderung von 100Mbit maximale Bandbreite erfüllt wird müssen verschiedene Techniken zum Reduzieren der Datenmenge angewendet werden. Eine leicht zu implementierende Technik ist Multicast-Sockets zu verwenden. Damit wird nur einen Datenstrom für alle Klientrechner benötigt. Zusätzlich müssen die Bilddaten mindestens mit einem 8:1

Verhältnisse komprimiert werden, damit die 755Mbit/s ins 100Mbit Netzwerk passen. Zur Komprimierung gibt es zwei Hauptalternativen. Komprimierung der einzelnen Bilder (z.B. PNG, JPEG) oder Komprimierung des Videostroms (z.B. MPEG4). Für beide Alternativen gibt es bereits freie Libraries und Tools die die Komprimierverfahren Implementieren. Um Bilder zu Komprimieren ist [12, ImageMagick] eine Möglichkeit und für Video kann das mencoder Tool von dem [14, MPlayer] Projekt verwendet werden.

Mit dem Zusätzlichen Schritt der Komprimierung (und Dekomprimierung) wird die Anforderung für die Bandbreite erfüllt, damit entsteht aber ein anderes Problem - die Performanz. Um Videoströme oder Bilder mit 24 fps und eine Auflösung von 1280x1024 Pixel zu Komprimieren und Dekomprimieren braucht man sehr leistungsfähige Rechner.

Außer der Bildübertragung muss das Kommunikationssubsystem auch ein Benutzerinteraktionskanal realisieren. Dazu gibt es wieder zwei Hauptalternativen - eigene Socketbasierte Client-Server Implementierung oder Verwendung von bestehenden Projekten. Das Projekt [17, x2x] zum Beispiel bietet die Möglichkeit an, einen X-Server mit der Eingabegeräte eines anderen X-Servers zu steuern. Das Projekt unterstützt auch alle X Authentizierungsmethoden. Andererseits ist eine eigene Implementierung mit viel zusätzlichen Aufwand verbunden, ohne irgendwelche Vorteile zu bieten.

6.2.1.2 VNC-basierte Kommunikation

Eine andere sehr interessante Möglichkeit für das Kommunikationssubsystem ist die Verwendung des VNC-Protokolls. Ein VNC-System besteht aus zwei Teilen - VNC-Server und VNC-Viewer.

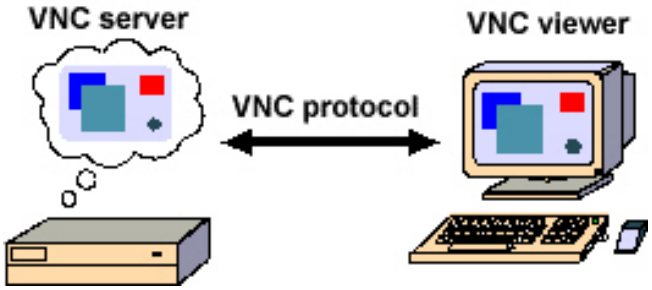


Abbildung 6.3: VNC Architektur. Zur Verfügung gestellt von [4].

Der VNC-Server "betrachtet"das Window-System im Sourcerechner und schickt den Inhalt des Framebuffers an alle zu ihm verbundenen VNC-Viewern. Die VNC-Klienten können Steuerungsereignisse für Maus und Tastatur zum Server schicken. In unserem Fall können wir den VNC-Server ohne Änderung einsetzen. Für die Viewerrechner müssen wir den VNC-Klient so anpassen, dass er die Bilder für Viewerapplikationen zugänglich macht anstatt am Bildschirm zu visualisieren. Einerseits wird die Integration im System wegen der bereits existierende OpenSource Projekte (z.B. [15, RealVNC] und [16, TightVNC]) sehr vereinfacht. Andererseits realisieren diese Projekte einen großen Teil unserer Systemanforde-

rungen und verringern somit den nötigen Implementierungsaufwand. Insbesondere sind folgende VNC-Eigenschaften für unser System interessant:

- Das VNC-Protokoll bietet durch Anwendung verschiedener Komprimierverfahren flexible Möglichkeiten zur Einstellung der Bildqualität/Netzbandbreite. Dabei sind die verwendete Komprimierverfahren speziell für interaktive Anwendungen optimiert (kleine Verzögerungen, vergleichsweise geringe Hardwareanforderungen). Leider leidet aber oft das Bildqualität darunter.
- Die VNC-Server unterstützen eine Basisauthentifizierung. Verschlüsselung ist mit Hilfe von SSH ohne weiteres möglich.
- Das VNC-Protokoll hat auch ein integriertes Interaktionskanal und die VNC-Server realisieren das Steuerungssubsystem vollständig.
- VNC-Server sind nicht auf X und Linux beschränkt, sondern gibt es Implementierungen für jedes gängige Windowsystem und Betriebssystem.

Alle diese Eigenschaften klingen sehr verlockend und heben die VNC-basierte Lösung hervor. Abbildung 6.4 zeigt wie sich diese Lösung in das System integrieren lässt.

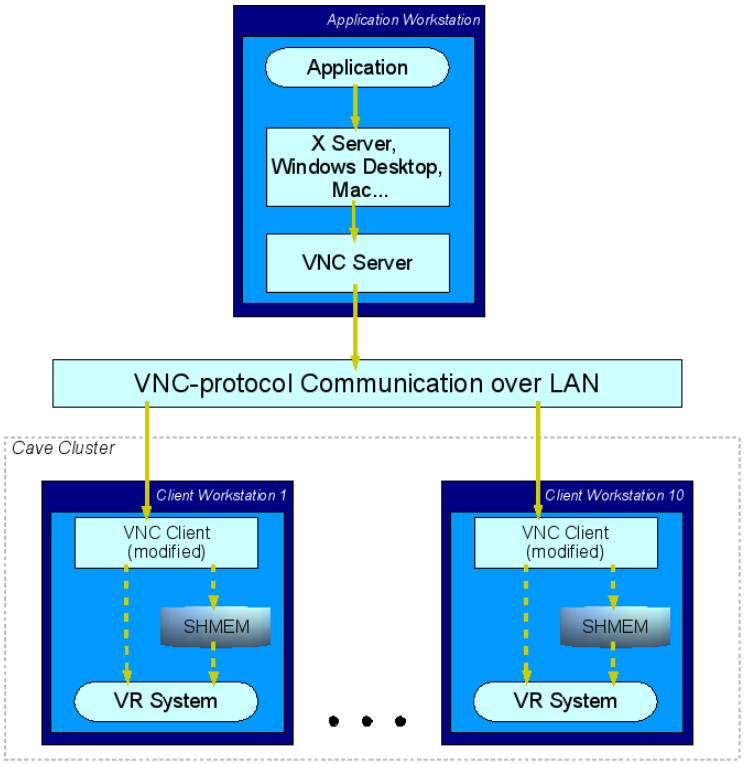


Abbildung 6.4: VNC-basierte Kommunikation

Bei der durchgeführten Experimenten zeigen sich leider auch einige Schwachstellen. Trotzt aller Optimierungen ist Komprimieren/Dekomprimieren mit noch zwei zusätzlichen

Schritten verbunden, was die Latenzzeiten deutlich erhöht und die Frameraten verringert. Das Bildqualität wird dabei auch sichtbar verschlechtert.

6.2.1.3 X-Protokoll basierte Kommunikation

Beim X-Protokoll handelt es sich um ein standardisiertes Protokoll zum Datenaustausch zwischen X-Server und X-Client über ein Netzwerk. Das X-Protokoll unterstützt die Anwendung auf einem X-Client durch die vom X-Server bereitgestellte Darstellung. Durch diese funktionale Trennung - der X-Client für die Anwendung, der X-Server für die Darstellung - sind Client und Server austauschbar; außerdem können entfernte Anwendungen im Netz vom X-Server grafisch unterstützt werden.

Dem Verfahren nach schickt der X-Client eine Anforderung in Form eines Request (REQ) an den X-Server. Dieser bearbeitet die Anfragen nach dem Warteschlangenprinzip und setzt dabei Prioritäten. Die Antwort erfolgt in einem Reply. Die Anfragen an den X-Server können sich mit der Einrichtung eines Fensters befassen, die Darstellung von Text oder Grafik in einem Fenster oder das Abfragen von Server-Eigenschaften.

Von diese Beschreibung werden schon die ersten Vorteile vom X-Protokoll ersichtlich. Text und Vektorgrafik brauchen viel weniger Speicherplatz und damit auch viel kleinere Bandbreite zur Netzwerkübertragung als Rastergrafik. Ein anderes Vorteil bringt sein Alter. Die ersten Versionen des X-Protokolls datieren auf Mitte der 80er Jahre. Mit der Zeit ist das Protokoll sehr reif, stabil und weit unterstützt geworden. Es existieren unzählige Tools, die für alle mögliche Anwendungsfälle angepasst werden können. Beispielsweise sind für unser Fall das [11, dmx] Projekt, das Xvfb (X virtual frame buffer) Tool sehr und die LBX (Low Bandwidth X) Erweiterung interessant.

DMX steht für Distributed Multihead X. Ähnlich wie mit Xinerama ist es mit DMX möglich, mehrere Bildschirme zu einem einheitlichen Display zu verbinden. Im Unterschied zu Xinerama können bei DMX die einzelnen Bildschirme an verschiedenen Systemen in einem Netz hängen. Das Effekt wird erreicht indem man zwischen den normalen X Server und X Clients ein DMX (Proxy) Server geschaltet wird. Alle X Clients verbinden sich zum DMX Server. Für die Clients erscheint dieser er als ein ganz normaler X Server. Er verbindet sich seinerseits zu den eigentlichen X Server und leitet ein Bereich seines Bildes an jeden Client weiter. Für den eigentlichen X Server sieht der DMX Server als ein ganz normaler X Client aus. Die genauen Koordinaten für die weitergeleiteten Bereiche können frei konfiguriert werden und sogar überlappen. Falls man als Koordinaten für alle das ganze Bild konfiguriert werden, erreicht man das Effekt eines Verteilers. Diese Eigenschaft kann im unseren Projekt verwendet werden, das Bild an alle Viewerrechner zu verteilen.

Xvfb steht für X Virtual Frame Buffer. Das Tool wird mit der standard [18, XFree] und auch [19, X.Org] Distribution verbreitet. Xvfb ist ein X-Server, der keine Grafikhardware braucht. Anfangs wurde er zur Testzwecke entwickelt um Programme, die ein X-Server brauchen, aber das Bild unwichtig ist, zu starten. Später wurden viel mehr Anwendungsfälle gefunden. Xvfb hat für unser Projekt eine sehr interessante Eigenschaft. Er kann das sein Bild in eine Datei oder ins shared memory rendern. Deswegen ist er als X-Server auf den Viewerrechner sehr gut geeignet.

Mit Hilfe von DMX und Xvfb kann die Kommunikation unseres Systems wie in Abbildung 6.5 realisiert werden.

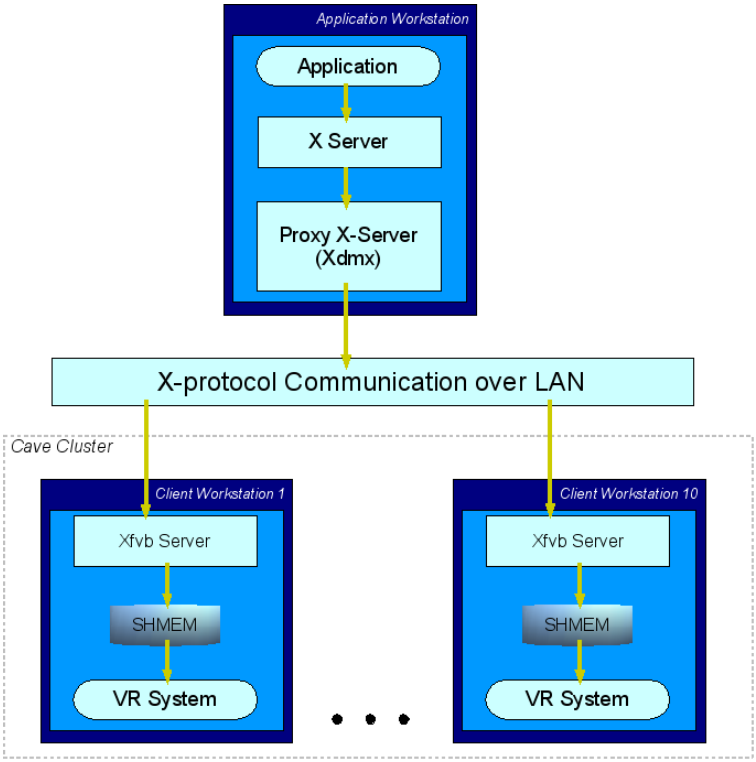


Abbildung 6.5: X-Protokoll basierte Kommunikation

Um die nötige Netzbandbreite abzuschätzen haben wir folgendes Experiment durchgeführt:

Aufbau Auf ein Rechner wurde ein DMX Server mit Auflösung 1600x1200x16 gestartet. Auf zwei weitere Rechner wurden zwei normale X-Server mit der gleiche Auflösung gestartet. Der DMX-Server war so konfiguriert, dass er sein komplettes Bild an den anderen zwei X-Server delegierte. Auf dem DMX-Server wurde ein KDE Session gestartet (kwin, kdesktop, kicker, konqueror, firefox). Mit dem Web-Browser wurde die http://www.wetter.de Webseite (überlastet mit Flash-Animationen) besucht. In eine zweite Phase wurde ein Film (Auflösung 320x240px) mit Mplayer abgespielt.

Messvariablen Gemessen wurde die genutzte Bandbreite an alle Rechner. Die erreichte Framerate wurde abgeschätzt.

Testergebnisse In der erste Phase wurden 12Mbit/s an dem Rechner mit dem DMX-Server und 6Mbit/s an den anderen zwei gemessen. Das Bild war fast fließend mit 20 fps. In der zweite Phase stieg die Bandbreitennutzung auf 80Mbit/s auf dem DMX Rechner und 40Mbit/s auf den anderen zwei. Die Framerate wurde auch deutlich schlechter.

Eine Möglichkeit zur weitere Reduzierung der nötige Netzbandbreite bietet die X-Protokoll Erweiterung LBX. LBX steht für Low Bandwidth X. Das Funktionsprinzip ist einfach. Auf der X-Client Seite wird ein sogenanntes lbxproxy gestartet. Die X-Clients bilden

die Verbindungen zum X-Server über das lbxproxy. Das Proxy optimiert Requests und komprimiert vor allem Rastergrafiken. Die Erweiterung ist für langsame Modemverbindungen gedacht ist aber natürlich vom Typ des Netzes unabhängig.

6.2.1.4 Schlussfolgerung

Allgemein gesehen haben alle drei Lösungen ihre Nachteile. Die Socketbasierte Lösung braucht viel Implementierungsaufwand und eine starke Komprimierung als Zwischenschritt. Das Problem bei der VNC-Lösung ist die Bildqualität. Die X-Protokoll Lösung hat eingeschränkte Portierbarkeit, wegen der Abhängigkeit von einem X-Server. Der Kontext unserer Arbeit macht die Auswahl viel leichter. Das System hat einen X-Server als Bildquelle, deswegen passt die X-Protokoll Lösung in dem Fall optimal. Eine VNC-Lösung würde den Einsatz des Windows oder MacOS Betriebssystems erleichtern, das bringt aber für unsere Anforderungen kein Mehrwert. Mit der X-Lösung wird die beste Performanz und Netzbandbreite erreicht. Der Prozess "Bildcapture" wird im Sourcerechner überflüssig. Das gilt aber nicht für das ganze System. Bildcapture ist jetzt in jedem Viewerrechner nötig. Damit befasst sich der nächste Kapitel.

6.2.2 Bildcapture

Nachdem wir eine Elegante Lösung für den Kommunikationsproblem gefunden haben, müssen wir die Bilddaten auf dem Viewerrechner in einen zur Darstellung passendes Format gewinnen. Dafür gibt es wieder verschiedene Wege. Ausgangspunkt ist der Xvfb Prozess, Ziel sind die unterschiedliche VR-Systeme, insbesondere OpenInventor (Coin3D) und OpenSG (VRED). Alle Zielsysteme basieren auf OpenGL, deswegen wäre jedes von OpenGL unterstütztes Format eine Alternative. Das OpenInventor API unterstützt davon nur die RGB und RGBA Formate, deswegen muss unseres Buldcapture-Subsystem dieses als Ausgabeformat haben.

6.2.2.1 Die Xvfb Ausgabe

Xvfb kann in 3 verschiedene Ausgabemodi gestartet werden. Das Modus wird durch die Kommandozeilen-Optionen bestimmt:

1. Standardmäßig, ohne spezielle Optionen, wird das Bild in einem für den Prozess privaten Framebuffer (alloziert mit malloc()) ausgegeben. Damit ist der Zugriff von externe Prozesse normalerweise nicht möglich.

2. Mit der `-fbdir directory` Option werden speichergemappte (mmap) Dateien im *directory* Verzeichnis als Framebuffer verwendet. Der Zugriff von Außen ist über das Dateisystem möglich. Damit ist dieser Modus für Testzwecke sehr gut geeignet, da existierende Programme direkt auf das Dateisystem zugreifen können. Weiter ist unter Linux möglich ein Dateisystem im Hauptspeicher (type tmpfs) zu mounten. Meistens ist so ein Dateisystem unter */dev/shm* bereits gemountet. Damit wird die Verzögerung des langsamen Festplattenspeichers umgegangen und bleibt nur das Overhead der Dateisystemlayer als Nachteil.

3. Mit der `-shmem` Option werden shared memory Bereiche als Framebuffer verwendet. Die shmem id's werden in der Standardausgabe ausgegeben. Damit ist der Zugriff von außen über die SysV shared memory Schnittstelle möglich. Wegen der direkte Speicherzugriff ist bei diesem Modus die beste Performanz und Frameraten zu erwarten.

Die letzten zwei Modi haben XWD (X Window Dump) als Ausgabeformat. Es ist ein Bitmap Format, das wenig verbreitet und sehr wenig dokumentiert ist. Deswegen unterstützen nur wenige Programme dieses Format. Ein Beispiel dafür ist das Tool `convert` von der ImageMagick-Suite. Es kann die XWD Dateien nach RGB Format konvertieren. Damit die durch diese Lösung erreichbare Frameraten abgeschätzt werden, haben wir folgendes Experiment durchgeführt:

Aufbau Auf ein Rechner wurde ein Xvfb durch folgendes Kommando gestartet:

```
Xvfb :1 -screen 0 1024x768x24 -fbdir /dev/shm/
```

Somit wird das Bild (von screen0) des Xvfb Servers im */dev/shm/Xvfb_screen0* gerendert. Danach wurde durch folgendes Kommando das XWD Bild in RGB Format konvertiert:

```
time convert /dev/shm/Xvfb_screen0 sgi:Xvfb_screen0.rgb
```

Messvariablen Durch das Tool `time` wurden die zum Konvertieren nötigen Systemressourcen gemessen.

Testergebnisse

```
real    0m0.357s
user    0m0.223s
sys     0m0.068s
```

Das Experiment zeigte dass das `convert` Tool ungeeignet für unsere Zwecke ist. Das Tool braucht 0.5s um nur ein Bild zu konvertieren, was die maximale Framerate auf 2 fps beschränkt. Ein weiteres Nachschauen im Sourcecode zeigte, dass ImageMagick alle Eingabebilder zuerst im MIFF (Magick Image File Format) Konvertiert und erst davon das Ausgabebild erzeugt. Die Niedrige Framerate ist nicht das einzige Problem bei dieser Lösung. Ein weiterer Nachteil ist, dass die von uns verwendete ImageMagick Version nur XWD Bilder unter 8 Bit Farbtiefe richtig konvertieren könnte.

Alternativ besteht die Möglichkeit eine eigene Implementierung für den Konvertiervorgang zu schreiben, die speziell für die Aufgabe XWD→RGB optimiert ist. Es soll hier noch nach einer existierenden und performanten Lösung gesucht werden.

6.2.2.2 X-Server Screencapture

Unabhängig vom Ausgabemodus des Xvfb Servers gibt es eine weitere Möglichkeit seine Bilder zu erfassen - das Screenshot. Es gibt unzählige Anwendungen, die ein Screenshot eines X-Servers machen. Für unsere Zwecke ist ein Kommandozeilentool ohne GUI wie das `xwd` Tool von dem XFree bzw. XOrg Projekt oder das `import` Tool von dem ImageMagick Suite bestens geeignet.

xwd kann das erfasste Bild nur in eine XWD-Datei speichern. Auf dem ersten Blick scheint diese Möglichkeit keine Vorteile im Vergleich mit XWD Ausgabe im shmem zu haben. Der Unterschied zeigt sich beim Versuch das Bild im RGB Format (mit `convert`) zu Konvertieren. Das Ergebnis sieht bei alle Farbtiefen gut aus. Leider ist die Performanz nicht die Stärke dieser Lösung, da das Bild zuerst in XWD Format erfasst wird und erst dann nach RGB konvertiert wird. Der erste Schritt hat 1s für 1027x768 gedauert und der zweite noch 0.5s.

import zum Gegenteil kann die erfasste Bilder in alle von ImageMagick unterstützte Formate speichern. Das erfasste Bild sieht auch gut aus. Die Performanz haben wir durch folgendes Kommando getestet: `time import -window root -display :1 sgi:capture.rgb`. Hier sind die Ausgaben von `time`:

```
real    0m0.887s
user    0m0.249s
sys     0m0.069s
```

Das Tool braucht ca. eine Sekunde um ein 1024x768 Screen mit 24bit Farbtiefe in RGB Format zu speichern. Jedoch brauchen wir etwas, das uns die gewünschte Framerate von 25Hz gewährleistet.

Die Ergebnisse von den getesteten Tools waren schlecht, deswegen haben wir versucht ein eigenes Tool zur Screenshots zu implementieren. Das ist eine ziemlich leichte Aufgabe, da die "Magie" des Screenshots nur ein einziger Aufruf der `XGetImage()` Funktion im xlib braucht. Das Programm hat unwesentlich schneller als die getestete Tools funktioniert - es könnte 24 Bilder mit 1024x768 Auflösung und 24bit Farbtiefe in interne ZPixmap Buffer für ca. 18 Sekunden capturen. Dieses Ergebnis hat gezeigt dass eine Screenshot-basierte Lösung für unseres Projekt überhaupt nicht geeignet ist.

6.2.2.3 Direkter shmem Zugriff

Abbildung 6.6: convert-Tool mit direktem shmem Zugriff

Von den oben beschriebenen Experimenten der verschiedenen Verfahren sieht man, dass es nur eine einzige Möglichkeit gibt, die alle Anforderungen erfüllen kann. Ein eigenes Tool, das die Bilder aus shmem des Xvfb-Servers richtig ausliest und dann nach RGB konvertiert

ist zu schreiben. Das wird wegen der mangelhaften XWD-Dokumentation keine leichte Aufgabe zum Implementieren sein, gibt aber die Möglichkeit den Code speziell zur diese Konvertierung zu Optimieren und damit beste Ergebnisse zu bekommen. Abbildung 6.6 zeigt, wie die Realisierung auszusehen hat.

Das Tool kann in Form eines Prozesses oder auch als ein Shared Library implementiert werden. Im Fall eines Prozesses muss ein Interprozess-Kommunikationsmechanismus wie (z.B. SysV shmem) verwendet werden, bei ein Shared Library kann der Code direkt im Viewerprozess angebunden werden und ein internes Buffer verwenden.

6.2.3 Darstellung

In diesem Kapitel werden die Designentscheidungen für das letzte Subsystem im Pipeline getroffen - das Darstellungssubsystem. Eingabe ist das RGB-konvertierte Bild aus der Bildcapture-Subsystem. Das Bild muss in verschiedene VR-Systeme auf eine beliebige Figur visualisiert werden. Alle Ziel-VR-Systeme Coin3D (OpenInventor), VRED (OpenSG), VD2 basieren auf OpenGL, deswegen ist das Wiederverwenden von Code möglich, jedoch hat jedes System eine eigene API, die zur Auswahl der Figur und ihre Position im Raum verwendet werden soll. Gemeinsam verwendete Codeteile können in ein shared library implementiert werden, die dann von eine VR-System spezifische Komponente angebunden und verwendet werden.

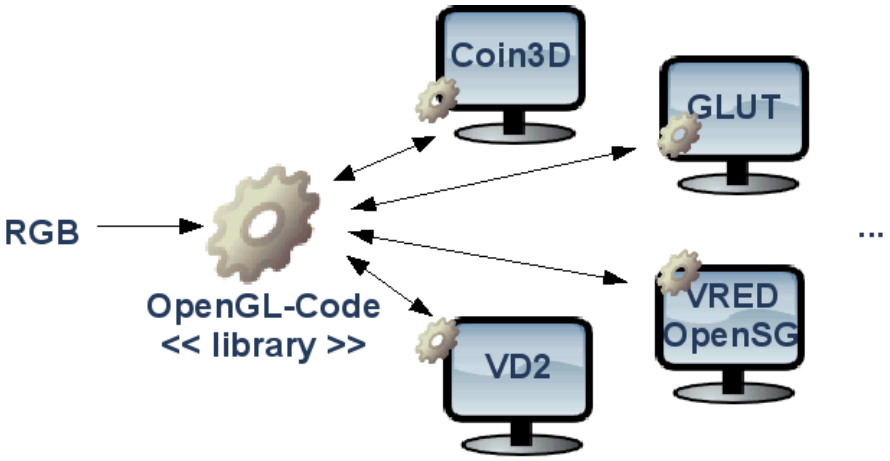


Abbildung 6.7: Darstellungssubsystem-Design

Diese Architektur ermöglicht das leichte Anbinden von neue VR-Systeme. Noch besser - ein VR-System ist sogar nicht erforderlich. Es ist möglich jedes OpenGL basiertes System durch wenig Implementierungsaufwand zu integrieren.

7 Implementierung

Im letzten Kapitel haben wir durch Evaluation verschiedener Werkzeuge und eine Vielzahl von Experimenten ein geeignetes Design für das System erstellt. Viele Komponenten wurden durch existierende OpenSource Tools realisiert, für andere haben wir keine geeignete Tools gefunden. Diese Komponenten müssen von uns implementiert werden. Insbesondere sind das die Bildcapture Komponente, das OpenGL Basis Shared Library und die Einzelne VR-System spezifische Komponenten für das Darstellungssubsystem. Dieses Kapitel befasst sich mit der Beschreibung der Details der Implementierung dieser Komponenten. Als Programmiersprache haben wir uns für C++ aus folgende Gründe entschieden: Einerseits sichert C++ problemlose Integration, weil alle abhängige Systeme auch in C oder C++ implementiert sind, andererseits gibt es viele Codebeispiele, die behilflich sein können und nicht an letzter Stelle ist C++ fast ein Standard in der UNIX-Welt.

7.1 Bildcapture-Komponente

Unsere Implementierung der Bildcapture-Komponente liegt im *ShmXvfbToOpenGL/convert.cpp*. Wir haben uns für einen Prozess und gegen eine Shared-Library entschieden, weil die einzelnen Komponenten auf diese Weise besser entkoppelt werden. Das konvertierte Bild wird in ein shmem-Bereich gerendert. Das shmem-ID wird in der Standardausgabe ausgegeben. Für das Verständnis der Implementierung sind Kenntnisse über den XWD-Bildformat, die OpenGL-unterstützte Bildformate und shmem Zugriff nötig. Im folgenden Abschnitt vermitteln wir das nötige Hintergrundwissen.

7.1.1 XWD-Bildformat

Alle Informationen, die wir über das XWD-Format wissen stammen aus der *<X11/XWDFile.h>* Header. Die binäre Daten bestehen grundsätzlich aus Header-und Datenbereiche. Das Header ist logischerweise am Anfang und ist durch die Struktur `XWDFileHeader` beschrieben. Im Header sind Metadaten für das verwendete Format im Datenbereich enthalten. Unmittelbar nach dem Header folgt der Datenbereich, wo die Pixeldaten gespeichert sind. Die Metadaten im Header sind in MSB first-Format. Das heißt, damit die Daten interpretiert werden, muss zuerst eine Überprüfung gemacht werden, ob der Rechner big-oder little-endian ist und falls little-endian muss die Datenstruktur zuerst gespiegelt werden. Aus dem Header wird die Höhe und Breite des Bildes und die Anzahl Bits pro Pixel ausgelesen. Die Pixel sind im Datenbereich von Links nach Rechts und von Oben nach Unten sequenzialisiert. Die Anordnung der Farben in jedem Pixel

werden durch drei weitere Metadaten bestimmt - die Rot- Grün- und Blau-Maske. Bei der Implementierung haben wir festgestellt, dass diese sich in verschiedene Xvfb Implementierungen unterscheiden. Beispielsweise verwendete die von uns getestete Version von XFree eine BGRA Anordnung und die XOrg Implementierung - RGBA. Eine Besonderheit der XFree Implementierung, die wir experimentell gefunden haben ist, dass das Bild nicht sequenziell im Speicher abgelegt wird, sondern nach jede abgeschlossene Zeile ein Sprung von drei leere Zeilen folgt. Das hat auch die Schwierigkeiten bei der Konvertierung durch ImageMagick erklärt.

7.1.2 OpenGL-Bildformate

OpenGL unterstützt viele verschiedene Bitmap-Formate - RGB, RGBA, BGRA, COLOR_INDEX... Die Namen entsprechen der Anordnung der Farben in ein Pixel. Das Gemeinsame für alle Formate ist, dass die Pixeldaten zeilenweise von Links nach Rechts und von Unten nach Oben sequenzialisiert werden. Weiteste Unterstützung hat das RGBA Format, deswegen haben wir das als Zielformat ausgewählt. Im Unterschied zu XWD gibt es für OpenGL viel Dokumentation (siehe [5, The Red Book]), wo die weitere Details nachgelesen werden können.

7.1.3 System-V shared memory

Shared Memory erlaubt, dass virtuelle Speicherbereiche verschiedener Prozesse gemeinsam genutzt werden. Die System-V API wird ziemlich oft zum Realisieren von Shared Memory Zugriffe genutzt. Das API besteht aus eine Menge Funktionen. In unsere Implementierung werden vier davon genutzt. Die Funktion `shmget()` erlaubt das Allozieren neuer gemeinsame Speicherbereiche. Mit `shmctl()` werden nicht mehr benötigte Speicherbereiche freigegeben (Freigabe erfolgt erst nachdem kein Prozess mehr den Speicherbereich referenziert). Durch die Funktion `shmat()` kann man einen schon allozierten Speicherbereich zum virtuellen Speicher des Prozesses einbinden und mit `shmdt()` wird wieder getrennt. Zum Steuern der Zugriffsrechte und andere Einstellungen wird wieder die Funktion `shmctl()` verwendet. Für weitere Details wird auf [21] und die Man-Seiten der einzelnen Funktionen verwiesen.

7.2 libxgl - OpenGL Basislibrary

Das gemeinsame Library `libxgl` hat zwei wesentliche Aufgaben. Einerseits wird dadurch die Komplexität der OpenGL Aufrufe hinter ein einfaches objekt-orientiertes API versteckt, andererseits werden gemeinsame Codefragmente wiederverwendet. Der Name XGL soll als X in OpenGL interpretiert werden - die Umkehrung von GLX. Unsere Initialimplementierung hat auf die Ausgabe der Bildcapture-Komponente über Shared Memory zugegriffen, dabei waren Synchronisierungsprobleme zu beachten. Um die Synchronisierungsaufgabe zu vereinfachen haben wir uns entschieden die Bildcapture-Komponente als ein Shared-Library umzuschreiben. Durch diese Änderung gehen die Vorteile der gemeinsame Nutzung des Konvertierten Bildes verloren. Das ist für unsere Anwendung kein Nachteil, da im Prinzip

nur ein Prozess pro Rechner die konvertierten Daten verwenden wird. Eine andere Folgerung ist, dass die Trennung von Bildcapture und XGL-Library keine echte Vorteile mehr hat. Deswegen haben wir uns entschieden den Bildcapture-Code auch im XGL-Library zu integrieren. Das Library wird aus zwei verschiedene Sichten erklärt - Black-Box und White-Box. Die Black-Box-Sicht erklärt das API der Library anhand der vorhandene öffentliche Klassen und ihre Methoden. Die White-Box-Sicht befasst sich mit der Interna der Implementierung.

7.2.1 Black-Box-Sicht (User API)

Die Schnittstelle der Library besteht aus eine einzige Klasse - `XvfbTexture` mit elf öffentliche Methoden (siehe Abbildung 7.1).

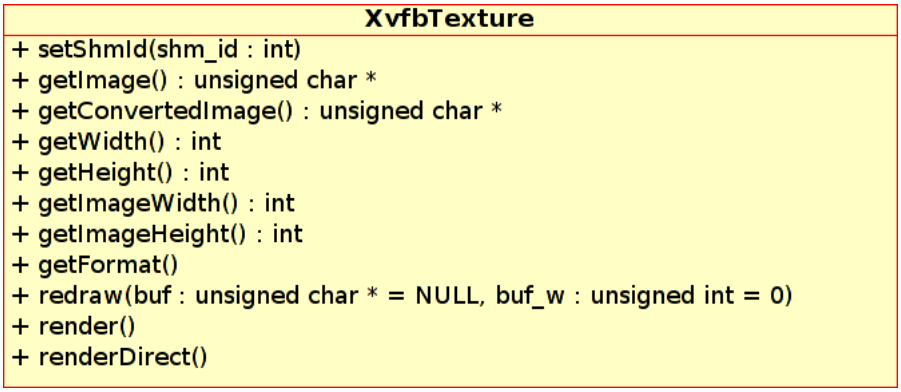


Abbildung 7.1: libxgl API

`setShmId()` ist die Methode, die eine `XvfbTexture` Instanz durch einen Shared-Memory Bereich (re)initialisiert. Übergeben wird als einziger Parameter das ID des Shared-Memory-Bereichs. Diese Methode muss nach Instanziieren immer aufgerufen werden, bevor man die anderen Methoden verwenden darf. Sonst ist das Verhalten des Programms undefiniert!

`getImage()` erlaubt einen direkten Zugriff auf die Pixeldaten des XWD Bildes.

`getConvertedImage()` hat als Rückgabewert ein Pointer auf die durch `redraw()` konvertierte Pixeldaten in OpenGL RGBA-Format. Bevor man diese Methode verwendet, muss mindestens einmal aufgerufen werden.

`getWidth()/getHeight()` gibt die Breite bzw. Höhe der von `render()` und `renderDirect()` verwendeten Textur in Pixel zurück. Die Zahl ist immer eine minimale Zweierpotenz, die größer als der Breite bzw. Höhe des Original-XWD-Bildes. Wenn man sich das Buffer zweidimensional vorstellt, werden die eigentliche Daten unten-links gerendert. Oben und Rechts können Bereiche mit undefinierte Daten entstehen. Dieses Format ist für eine performante OpenGL-Rendering bestens geeignet.

`getImageWidth()`/`getImageHeight()` gibt die Breite bzw. Höhe des Originalbildes in Pixel.

`getFormat()` gibt ein `GLenum` mit dem Format des Originalbildes (GL_BGRA oder GL_RGBA)

`redraw()` erzwingt eine Konvertierung der aktuelle Pixeldaten vom Shared-Memory-Bereich in RGBA Format. Anhand der übergebene Parameter werden verschiedene Buffer zur Ausgabe verwendet. Ohne Parameter wird einen internen Buffer verwendet. Zugriff auf das Ergebnis bekommt man über die Methode `getConvertedImage()`. Als erster Parameter kann ein Pointer auf ein externer Buffer übergeben werden. Der Zweite Parameter bestimmt die Breite (Zeilenlänge) des Buffers. Standardwert ist die Breite des Originalbildes. Damit kann man das Bild in "breitere" externe Buffer richtig konvertieren.

`render()` lädt das konvertierte Bild in eine Textur im Grafikspeicher hoch.

`renderDirect()` ist wie `render()`, aber diesmal wird das Originalbild im Grafikspeicher geladen. Diese Methode ist viel performanter, aber die hochgeladene Textur ist horizontal gespiegelt.

7.2.2 White-Box-Sicht (Interna der Implementierung)

Die ganze Funktionalität ist in die einzige Klasse selbst Implementiert und liegt in *libxgl/src/xvfbtexture.cpp*. Die wichtigsten Methoden sind `setShmId()`, `redraw()` und `uploadTexture()`. Alle Anderen sind entweder einfache Getter und Setter oder basieren auf die drei Methoden.

`setShmId()` ist eine Initialisierungsmethode. Falls vorhanden werden zuerst alte Shared Memory Bereiche getrennt und alte interne Buffer freigegeben. Dann wird das übergebene Shared Memory Bereich zum Prozess eingebunden. Aus dem XWD-Header werden die nötigen Daten ausgelesen und in interne Felder abgespeichert. Ein neuer interner Buffer mit der entsprechende Größe wird angelegt und initialisiert. Die OpenGL Textur Breite und Höhe werden als minimale Zweierpotenz größer als Breite bzw. Höhe des Eingabebildes berechnet. Am Ende werden die Farbmasken neu berechnet.

`redraw()` ist die Methode, die sich um die XWD→RGB Konvertierung kümmert. Die Implementierung ist als eine zwei verschachtelte Schleifen über Zeilen und Pixel realisiert.

`uploadTexture()` ist die Methode, die das Bild im Grafikspeicher über OpenGL-Aufrufe überträgt. Zur bessere Performanz wird zuerst durch `glTexImage2D()` eine Textur mit Zweierpotenz Breite und Höhe vorbereitet und dann durch durch `glTexSubImage2D()` die eigentliche Pixeldaten hochgeladen.

7.2.3 Unit-Tests

Zum Testen der Funktionalität der Library und zum Messen der Performanz haben wir ein einfaches Programm als Unit-Test geschrieben. Der Source-Code liegt im *libxgl/test/testxvfbtexture.cpp*. Das Programm verwendet GLUT für die Verwaltung von Fenster, Eingabeereignisse und Timer. Durch OpenGL-Aufrufe wird ein Rechteck im Fenster gezeichnet und mittels eine `XvfbTexture`-Instanz wird das Bild auf seine Oberfläche gemappt.

7.3 VR-System spezifische Darstellungskomponenten

Die VR-System spezifische Darstellungskomponenten dienen zur Integration des ganzen Systems in verschiedene VR-Systeme. Wie im System-Design beschrieben, sind die APIs dieser Systeme unterschiedlich. Grundsätzlich kann man zwischen zwei verschiedene Arten der Integration unterscheiden.

- Erweiterung des System-APIs – Damit werden neue Komponenten zum System-API (z.B. eine neue Klasse) hinzugefügt, die die nötige Funktionalität realisieren. Vorteil ist die bessere Integration im System. Die Details bleiben vom User versteckt. Für die Realisierung ist ein sehr gutes Verständnis für die Implementierungsdetails des Ziel-Systems nötig. Sourcecode muss am besten vorhanden sein.
- Erweiterung auf Basis der System-API – Die Erweiterung wird in der Useranwendung integriert, die das System-API verwendet. Diese Lösung ist einfacher zu realisieren, da das Ziel-System als Black-Box verwendet wird. Wissen für die Interna der Implementierung sind nicht nötig. Die Nachteile äußern sich in Performanzverluste durch die Verschachtelung der Systeme und schlechte Wiederverwendbarkeit.

Ein Kritischer Punkt für die Performanz dieses Subsystems ist die Übertragung der Textur im Grafikspeicher. Deswegen haben wir im Basislibrary entsprechende Methoden dafür vorbereitet. Die Verwendung dieser Methoden ist meistens nur mit der erste Art Integration möglich, sonst wird das Basislibrary nur zum Konvertieren verwendet.

Im folgenden Abschnitt betrachten wir die Details zur Implementierung der einzelnen Komponenten.

7.3.1 Coin3D

Für die Realisierung dieser Komponente haben wir zuerst die gut Dokumentierte OpenInventor API [3, 39] verwendet und eine Integration der zweiten Art implementiert. Dabei haben wir eine Instanz der `SoTexture2` Klasse verwendet. Die Implementierung hat zwar funktioniert, aber das Bild war statisch. Die Klasse hat die Daten in ein internes Buffer kopiert und nicht das übergebene Buffer verwendet. Deswegen müsste man die Daten für jedes Frame neu setzen. Das Ergebnis war eine sehr schlechte Performanz. Unser Testsystem könnte nur ca. 0,5 fps. erreichen und die Speichernutzung der Applikation stieg ständig sehr schnell. Diese Ergebnisse zeigten uns, dass `SoTexture2` für Video nicht tauglich ist.

Ein Nachschauen im Coin3D Source zeigte, dass `SoTexture2` intern eine `GLImage` Instanz verwendet. Die `GLImage` Klasse verwaltet seinerseits ein internes Buffer.

In ein zweiter Versuch haben wir eine Integration der ersten Art Implementiert. Im ersten Versuch haben wir die Grundlagen der Coin3D Implementierung kennengelernt und jetzt könnten wir leichter das API erweitern. In diesem Schritt hat uns auch das Buch [40, Inventor Toolmaker] sehr geholfen. Zum Zweck haben wir eine neue Klasse hinzugefügt - `SoActiveTexture2`, die von `SoTexture2` ableitet. Die Idee war die schon bekannte `SoTexture2` Schnittstelle mit eine videotaugliche Implementierung zu realisieren. Das könnten wir leider nicht so realisieren, weil die nötigen Felder der `SoTexture2` Klasse als privat definiert sind, und damit nicht zum Erweitern freigegeben. Die Einzige Möglichkeit, die es noch gab, war unsere Klasse direkt von `SoNode` abzuleiten und den Code komplett neu schreiben. Das hat letztendlich gut funktioniert. Die Performanztests haben 20 fps bei eine Auflösung von 1024x768@24Bit ergeben. Dabei haben wir vom `XvfbTexture` die `render()` Methode zur Übertragung der Textur im Grafikspeicher verwendet. Später haben wir die Implementierung weiter optimiert, so dass die effizientere Methode `renderDirect()` verwendet wird. Durch diese Optimierung könnte das System mit 25 fps bei Auflösung 1024x768@24Bit und 70% der CPU laufen. Dabei muss man aber aufpassen, dass die so übertragene Textur horizontal gespiegelt ist. Damit das Bild in der Szene richtig erscheint wird beim Mappen auf die entsprechende Figur nochmal horizontal gespiegelt.

Wir haben die endgültige Implementierung in ein Shared Library namens `libactivetexture` gepackt. Der Code ist im *ActiveTexture2/libactivetexture/SoActiveTexture2.cpp* zu finden. Es wird auch ein Test bzw. Beispielsprogramm geliefert (*ActiveTexture2/test/testActiveTexture.cpp*), das eine `SoActiveTexture2`-Instanz in eine Szene darstellt. Ein Viewer für Inventor-Dateien (.iv) der SoActiveTextuire2 Objekte kennt und eine Beispielsdatei liegen auch im *ActiveTexture2/test*.

7.3.2 VRED (OpenSG)

Die Beziehung zwischen VRED und OpenSG ist im Vergleich zu Coin3D und Inventor ganz anderes. VRED ist ein System, das auf OpenSG aufbaut. OpenSG ist OpenSource, aber VRED ist es nicht mehr. Obwohl es eine veraltete Version von VRED gibt, die OpenSource ist [27], haben wir kein Code dazu im Internet gefunden. Aus diesem Grund werden wir die Realisierung von diese Komponente in zwei Schritte aufteilen. Zuerst wird Code zur Integration unseres Systems in OpenSG geschrieben und im zweiten Schritt wird das Ergebnis in VRED integriert.

7.3.2.1 OpenSG Integration

Auch für diese Komponente haben wir zuerst versucht eine Erweiterung auf Basis der System-API zu implementieren. Wir haben in eine Image-Instanz (*<OpenSG/OSGImage.h>*) unsere Pixeldaten geladen und dann durch ein SimpleTexturedMaterial (*<OpenSG/OSGSimpleTexturedMaterial.h>*) in eine Szene visualisiert. Die erste Version hat zwar Funktioniert, die Performanz war aber wieder schlecht, weil die Pixeldaten vom Buffer im `XvfbTexture` nach dem internen Buffer in der OSGImage-Instanz kopiert

wurden. Später haben wir festgestellt, dass wir auf dem OSGImage internen Buffer direkt zugreifen können. Schnell haben wir den Code so umgeschrieben, dass die `XvfbTexture` durch die `redraw()` Methode direkt im OSGImage-Buffer rendert. Durch diese Optimierung haben wir sehr gute Performanzergebnisse bekommen. Im unseren Testsystem haben wir bei 18 fps eine Auflösung von 1024x768 mit 24Bit Farbtiefe erreicht. Der Code dazu befindet sich in *OpenSG/test/texture2.cpp*.

7.3.2.2 VRED Integration

Die Ergebnisse vom ersten Schritt sind gut, wir haben den nötigen Code und da VRED auf OpenSG basiert haben wir gedacht - es ist eine einfache Aufgabe den Code im System zu Integrieren. Auf einmal standen wir vor einem geschlossenen System, wo kein Code zur Verfügung ist um nachzuschauen, wie die andere Funktionalitäten im System gebunden sind. Noch schlimmer - für VRED gibt es keine Development-Version, die die zur Integration neuer Komponenten nötige Headerdateien beinhaltet. Trotzdem haben wir ein Versuch gemacht, soviel wie möglich Informationen über die VRED-Interna zu finden. Alle VRED-Module sind als Shared-Libraries realisiert. Sie werden dynamisch (zur Laufzeit) vom System geladen. Durch das Tool `ldd` haben wir festgestellt, dass alle Libraries eine Abhängigkeit vom [1, Boost.Python-Library], wahrscheinlich wird dadurch die Python Schnittstelle von VRED realisiert. Die gesammelte Informationen waren ungenügend, deswegen haben wir uns an der Hersteller Firma AMZ gewendet. Die Antwort war dass sie keine Development-Version für kurze so Zeit bereitstellen können. Deswegen bleibt die Integration im VRED als mögliche Erweiterung des Systems in der Zukunft.

8 Ergebnisse und mögliche Erweiterungen

Im Laufe unseres Projektes wurde eine Reihe Tests durchgeführt, sowohl mit verschiedenen Konfigurationen des Systems, als auch einfachere Tests, die uns den Überblick geschafft haben, welcher Ansatz unsere konkreten Anforderungen am besten lösen kann. In diesem Kapitel konzentrieren wir uns auf das Auswerten der Endergebnisse und geben Vorschläge, was an dem System verbessert werden kann und wie die Weiterentwicklung ausschauen kann.

8.1 Auswertung des Systems

Für die meisten Benchmarks, die wir geführt haben, haben wir den Rechner benutzt, auf dem wir unser System entwickelt und implementiert haben. Er hat folgende Konfiguration:

- Prozessor: Pentium III 933MHz
- Grafikkarte: Matrox G550 AGP 32MB DDR
- Auflösung: 1024x768@24 bit

Bei den Tests haben wir immer die dargestellten Bilder pro Sekunde (FPS) und die Prozessor-Auslastung gemessen. In der Abb. 8.1 sind die Ergebnisse dargestellt.

Es ist zu beachten das bei dem ersten Test - **Screenshot** keine Übertragung und Darstellung der einzelnen Frames stattfindet. Die gemessenen Werte (FPS und CPU Load) gelten nur für das ´´Grabben“ der Screenshots. Da Ergebnisse sehr schlecht sind - nur 1,5 Bilder pro Sekunde bei 100% Auslastung des Prozessors, lohnte es sich nicht ein Test für das Gesamtsystem zu machen.

Bei den anderen 4 Tests haben wir das komplette System getestet. Es ist zu erkennen, dass wenn wir die Coin3D API für das Darstellen des X-Server-Bildes als Textur benutzen, die gemessenen Werte weiterhin sehr schlecht sind. Die OpenSG API dagegen zeigt eine viel bessere Geschwindigkeit - 18 FPS. Das ist auch der Grund, warum wir die Darstellung im OpenSG nicht weiterentwickelt haben und uns auf eine Verbesserung der Coin3D Implementierung konzentriert haben. Die Idee, das X-Server-Bild in Coin3D über die OpenGL API darzustellen hat sich als sehr nützlich erwiesen und die Verbesserung der Ergebnisse ist deutlich zu erkennen - 20 FPS, oder eine 40 mal schnellere Darstellung im Vergleich zu Coin3D API. Wir haben geschafft, dieses Ergebnis noch weiter zu optimieren, indem wir für die Textur direkt die ´´Rohdaten“ von dem shared memory verwenden und ein Kopieren in

den Arbeitsspeicher vermeiden. Das ist der Test **OpenGL API (no copy)**, wo eine optimale Geschwindigkeit von 25FPS und eine niedrigere Prozessorauslastung (70%) zu messen ist.

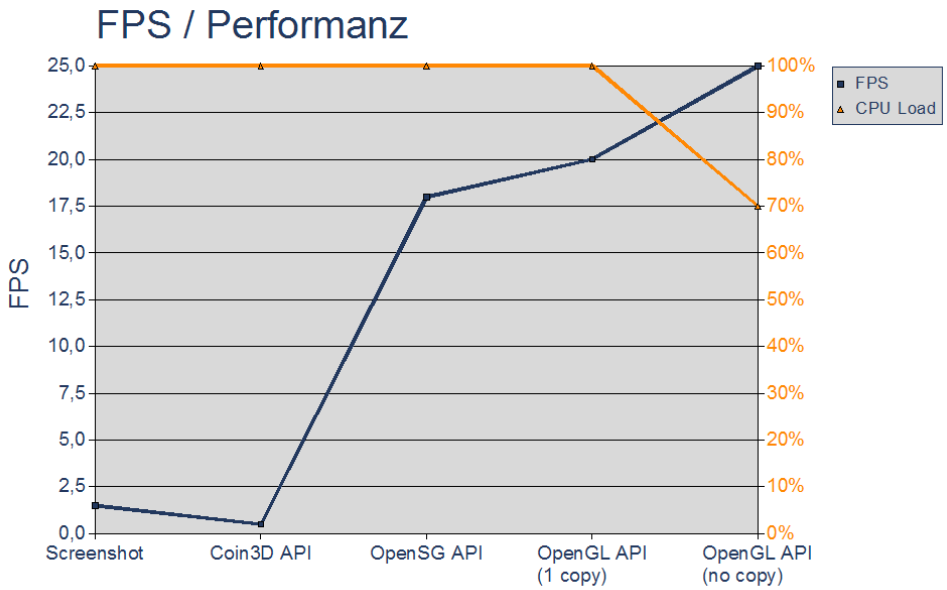


Abbildung 8.1: FPS- / Performanz-Messung bei den verschiedenen Konfigurationen des Systems

Mit zwei anderen Rechner haben wir den Testfall **OpenGL API (1 copy)** wiederholt, um zu überprüfen, wie sich das System auf besser ausgestatteten Computer verhält und eine Aussage treffen zu können, wie stark Hardwareabhängig unsere Lösung ist. Der erste Rechner war mit der folgenden Konfiguration:

- Prozessor: AthlonXP 2GHz
- Grafikkarte: Nvidia GeForce FX 5600
- Auflösung: 1024x768@24 bit

Der zweite Rechner war das Desktop-Systems von Herrn Augustiniack in BMW:

- Prozessoren: 2 Intel Xeon 3.2GHz
- Grafikkarte: Nvidia Quadro FX 3000/AGP/SSE2
- Auflösung: 1024x768x24

Die Ergebnisse sind in Form von drei Diagramme in Abb. 8.2 zu sehen. Bei einer ungefähr linearen Steigerung der Prozessorleistung fällt die Prozessorauslastung auch linear.

Daraus ist die Schlussfolgerung zu ziehen, dass unsere Implementation konstante Ressourcen verwendet, die für das Lösen des Problems notwendig sind. Und da die Rechner von dem CAVE-Cluster ähnlich konfiguriert sind wie das Desktop-System von Herrn Augustiniack, wird unsere Anwendung für sie keine große Belastung darstellen und neben dem VR-Software-System problemlos laufen.

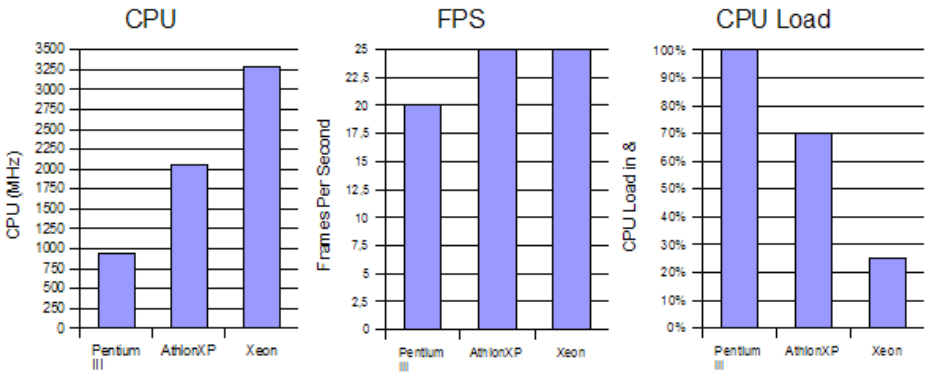


Abbildung 8.2: Vergleich der Ergebnisse von den einzelnen Test-Rechner

8.2 Future Work

Am Anfang unserer Ausarbeitung haben wir darauf hingewiesen, dass diese Arbeit als Grundlage für eine neue Kommunikationsschnittstelle entwickelt wird, dementsprechend gibt es Stellen in dem Projekt, an denen eine Weiterentwicklung notwendig ist. In diesem Kapitel geben wir einige Anregungen, was optimiert werden muss und wie die Funktionalität des System ausgebaut werden kann.

Um eine optimale Performanz im OpenSG zu erreichen, ist die Darstellung der Textur ähnlich wie in Coin3D auf OpenGL-Ebene zu implementieren. Dadurch werden auch bei langsameren Rechnern Geschwindigkeiten von ca. 25FPS erreicht, was ein fließendes Bild ermöglicht. Die Bibliothek-Architektur der Lösung macht die Implementierung außerdem benutzerfreundlicher.

Als nächstes muss die OpenSG-Lösung in VRED integriert werden. Dafür sind Informationen von den VRED-Entwickler AMZ GmbH notwendig, wie VRED-Plugins realisiert und benutzt werden. Diese Informationen sind nicht frei verfügbar und müssten von der Firma angefordert werden. Erste Gespräche haben wir schon geführt und die Firma hat sich bereit erklärt,die zur Verfügung zu stellen.

Eine Automatisierung beim Starten und Stoppen des Systems wird die Komplexität reduzieren, die Benutzerfreundlichkeit erhöhen und auch den Benutzer, die sich gar nicht mit dem System auskennen, ermöglichen es für ihre CAVE-Präsentationen zu verwenden. Am leichtesten ist das mittels Shell-Scripten zu realisieren. Auch das Konfigurieren des Systems kann vereinfacht werden, indem eine Konfigurationsdatei verwendet wird, oder eine interaktive GUI angeboten wird.

Ein großer Baustein, der noch zu entwickeln ist, ist die Benutzerinteraktion mit dem X-Server direkt in der CAVE. Über das CAVE-Eingabegerät muss es möglich sein, die Software, die in der X-Server läuft, ähnlich wie mit einer Maus zu steuern und zu bedienen. Bestimmte Events von den Eingabegerät und Tracking-System müssen abgefangen werden, Mausevents generiert werden und über das X-Protokoll zurück an den X-Server geschickt werden. Ganz wichtig hier ist zu entscheiden, wann der Benutzer Objekte aus der Szene behandeln will und wann er mit dem X-Server interagieren will. Dafür ist ein Konzept zu entwickeln, das eine intuitive Benutzerinteraktion sicherstellt. In [33] und [36] sind 2 Ansätze für das Lösen dieses Problems vorgestellt.

9 Zusammenfassung

Im Rahmen dieses Systementwicklungsprojektes wurde ein verteiltes System entwickelt, das die Oberfläche eines X Window System als Textur in dem virtuellen Raum von 2 VR-Software-Systemen darstellt - Coin3D und OpenSG. Damit haben wir eine Grundlage für ein neues virtuelles Desktop-Environment geschaffen, die so dem Benutzer eine Kommunikationsschnittstelle anbieten kann.

Verschiedene Strategien zur Realisierung des Systems wurden analysiert und Lösungen für die beiden VR-Software-Systeme implementiert, getestet und ausgewertet. Als die beste Kommunikationsalternative hat sich das X-Protokoll bewährt, das mehrere Vorteile im Vergleich zu den anderen Kommunikationsmöglichkeiten angeboten hat.

Für die effiziente Darstellung des X-Server-Bildes als Textur in Coin3D, war eine Implementierung auf der OpenGL-Ebene notwendig, die OpenSG API dagegen hat eine ausreichende Geschwindigkeit gezeigt. Durch weitere Optimierung wurde die Performanz des Systems erhöht und die Prozessorauslastung reduziert.

Alle Tests haben gezeigt, dass unsere Lösung konstante Rechner-Ressourcen verwendet und problemlos auf Rechner ausgeführt werden kann, auf denen auch VR-Software-Systeme laufen. Dadurch ist die Möglichkeit unseres Systems in einem CAVE-Cluster zu benutzen, sichergestellt.

Literaturverzeichnis

[1] *Boost.Python C++ library Homepage*. http://www.boost.org/libs/python/doc/index.html.

[2] *CAVE Homepage*. http://www.evl.uic.edu/pape/CAVE/.

[3] *Coin3D Homepage*. http://www.coin3d.org/.

[4] *Grafische Benutzeroberflächen im Netz*. http://archiv.tu-chemnitz.de/pub/1998/0015/data/vnc.html.

[5] *OpenGL Programming Guide or "The Red Book"*. http://fly.srk.fer.hr/~unreal/theredbook/.

[6] *OpenSG Homepage*. http://www.opensg.org/.

[7] *Professur Informatik in der Architektur, Bauhaus-Universität Weimar*. http://www.uni-weimar.de/architektur/InfAR/lehre/Course01/.

[8] *Rechen- und Kommunikationszentrum, RWTH AAchen*. http://www.rz.rwth-aachen.de/.

[9] *TGS Inventor Homepage*. http://oss.sgi.com/projects/inventor/.

[10] *The Advanced Visualization Laboratory Homepage*. http://www.avl.iu.edu/.

[11] *The Distributed Multihead X Project Homepage*. http://dmx.sourceforge.net/.

[12] *The ImageMagick Suite Homepage*. http://www.imagemagick.org/.

[13] *The Link Foundation Homepage*. http://www.binghamton.edu/home/link/link.html.

[14] *The MPlayer Homepage*. http://www.mplayerhq.hu/.

[15] *The RealVNC Homepage*. http://www.realvnc.com/.

[16] *The TightVNC Homepage*. http://www.tightvnc.com/.

[17] *The x2x Project Homepage*. http://x2x.dottedmag.net/.

[18] *The XFree86 Organisation Homepage*. http://www.xfree86.org/.

[19] *The X.Org Foundation Homepage*. http://www.x.org/.

[20] *UML Unified Modeling Language Homepage*. http://www.uml.org/.

[21] *Unix Multi-Process Programming and Inter-Process Communications (IPC) - Shared Memory*. http://users.actcom.co.il/~choo/lupg/tutorials/multi-process/multi-process.html#shmem.

[22] *Viewtec Homepage*. http://www.viewtec.ch/techdiv/vr_info_d.html.

[23] *Virtual Realities Inc. Homepage*. http://www.vrealities.com/.

[24] *Virtual Reality im LRZ*. http://www.lrz-muenchen.de/services/peripherie/vr/.

[25] *VR-Labor Rechenzentrum, Technische Universität Carolo-Wilhelmina*. http://www.tu-braunschweig.de/rz/services/visualisierung/vrlabor.

[26] *VRCOM Homepage*. http://www.vrcom-online.de/.

[27] *VRED Homepage*. http://www.vred.org/.

[28] *XiVE Homepage*. http://staff.science.uva.nl/~robbel/XiVE/.

[29] B. BRÜGGE and A. H. DUTOIT, *Object-Oriented Software Engineering. Conquering Complex and Changing Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.

[30] S. DIVERDI, D. NURMI, and T. HOERER, *ARWin - A Desktop Augmented Reality Window Manager*, Proc. ISMAR '03, (2003).

[31] P. DYKSTRA, *X11 in Virtual Environments: Combining Computer Interaction Methodologies*, Proceedings IEEE Symposium on Research Frontiers in Virtual Reality, (1993).

[32] S. FEINER, B. MACINTYRE, M. HAUPT, and E. SOLOMON, *Windows on the World: 2D Windows for 3D Augmented Reality*, Proc. UIST '93, (1993).

[33] C. GEIGER, L. OPPERMANN, and C. REIMANN, *3D-Registered Interaction-Surfaces in Augmented Reality Space*, Proc. Second IEEE International Augmented Reality Toolkit Workshop, Tokyo, (2003).

[34] S. GÄMPERLE, *Interaktive Visualisierung von Displayinhalten in VR*, Diplomarbeit, (2003).

[35] D. LARIMER and D. A. BOWMAN, *VEWL: A Framework for Building a Windowing Interface in a Virtual Environment*, Proceedings of INTERACT: IFP TC13 International Conference on Human-Computer Interaction, (2003).

[36] H. RENKEWITZ, C. WINKELHOLZ, and T. ALEXANDER, *An alternative interaction technique using a markerless input device for Mixed Reality*, Tagungsband zum 1. Workshop Virtuelle und Erweiterte Realität der GI-Fachgruppe AR/VR, (2004).

[37] B. SCHNEIDERMAN, *Designing the User Interface, Edition 3*, Addison Wesley, 1997.

[38] B. STROUSTRUP, *The C++ Programming Language*, Addison-Wesley Pub Co, 2000.

[39] J. WERNECKE, *The Inventor Mentor: Programming Object Oriented 3D Graphics with OpenInventor, Release 2*, Addison Wesley, 1994.

[40] J. WERNECKE, *The Inventor Toolmaker: Extending OpenInventor, Release 2*, Addison Wesley, 1994.

[41] M. WOO, J. NEIDER, T. DAVIS, D. SHREINER, and O. A. R. BOARD, *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*, Addison-Wesley Pub Co, 1999.