

Ein Kommunikationsmechanismus für verteilte virtuelle Umgebungen

– Diplomarbeit –

Bearbeitung: Marko Meister (4/91/A)
e-mail: marko.meister@informatik.uni-weimar.de,
WWW: <http://www.uni-weimar.de/~meister1/>

Betreuung: Prof. Dr. Charles Albert Wüthrich
Dr. rer. nat. Bernd Schalbe

Beschreibung: An den Lehrstühlen Graphische Datenverarbeitung und vernetzte Medien der Fakultät Medien der Bauhaus-Universität Weimar wird zur Zeit ein Projekt zu verteilten virtuellen Umgebungen durchgeführt. Als Teil dieses Projektes sollen im Rahmen der Diplomarbeit die kommunikationsrelevanten Module konzipiert und implementiert werden. Dabei sind im Einzelnen folgende Aspekte zu berücksichtigen:

- Analyse existierender Systeme und Methoden
- Erarbeitung eines prinzipiellen Ansatzes zur Objektkommunikation in verteilten virtuellen Umgebungen
- Implementation der Architektur in einer Prototyplösung

Reg.-Nr.: I/97/07

Bearbeitungszeitraum:
2. Dezember 1996 - 13. Juni 1997

Mein besonderer Dank gilt Katrin Schmidt, Norbert und Brigitte Meister, Charles Wüthrich, Bernd Schalbe, Günther Schatter, Grit Thürmer, Holger Regenbrecht & Claudia Ott, Jan Springer, der i-group bzw. der Besetzung des [atelier, virtual, Jens-Uwe Wagner, Dietmar Bratke, Christiane Zeug und allen anderen, ohne die diese Arbeit nicht so geworden wäre wie sie ist.

Inhalt

1	Einleitung	5
1.1	Virtuelle Umgebungen	6
1.2	Verteilte virtuelle Umgebungen	9
1.3	Zur Motivation der Arbeit	9
1.4	Zur Konzeption der Arbeit	10
2	Bestandsanalyse existierender Systeme.....	12
2.1	Kommunikationsmechanismen	12
2.1.1	Hierarchische Client/Server Modelle	13
2.1.2	Punkt-zu-Punkt Kommunikation.....	13
2.1.3	Hybride Ansätze	14
2.2	Verteilungsmechanismen	14
2.3	Informationen in verteilten virtuellen Umgebungen	15
2.4	Spezielle Anforderungen an verteilte virtuelle Umgebungen	16
2.5	Beschreibung der Systeme.....	17
2.5.1	DIVE	17
2.5.2	VEOS	18
2.5.3	Distributed Virtual Environment System	19
2.5.4	VUE.....	20
2.5.5	Rubber Rocks	20
2.5.6	VR-DECK	20
2.5.7	BrickNet	21
2.5.8	NVR	23
2.5.9	MASSIVE	23
2.5.10	AVIARY	25
2.5.11	NPSNET.....	26
2.5.12	RING	26
2.5.13	MR-Toolkit Peer Package	27
2.5.14	Waves / Hidra.....	28
2.5.15	Demand Driven Geometry Transmission.....	29
2.6	Zusammenfassung	30
3	Lösungsansatz	32
3.1	Unabhängige Welten	34

3.2 Autonome Objekte	37
3.2.1 Objektinformigramme	39
3.2.2 Objektabonnement.....	39
3.2.3 Übergang der Objektkontrolle.....	40
3.2.4 Objekttransport.....	42
3.2.5 Zusammenfassung	43
3.3 Kommunikationsprinzip.....	43
3.4 Objektinteraktion.....	45
3.5 Konsistenz der Welten	48
3.6 Optimierung der Ein- und Ausgabe.....	50
4 Implementierung.....	51
4.1 Voodie Prototyp	52
4.2 Graphische Ausgabe.....	54
4.3 Eventdatenstruktur.....	55
4.4 Netzwerkmodule / Eventtransfer.....	56
4.5 Wichtige Ereignisse (Events) in Voodie	57
4.6 Objektabonnement.....	59
4.7 Übergabe der Objektkontrolle	60
4.8 Objektimplementation in Voodie	61
4.8.1 Klassenbeschreibung	63
4.9 Funktionsbeschreibung des Prototypen.....	67
5 Zusammenfassung / Ausblick	68
6 Literaturverzeichnis.....	71
Anhang A Glossar	A-1
Anhang B Mögliche Events in Voodie.....	B-1

*Wann wenn nicht jetzt,
Wo wenn nicht hier ...*

*Wann?
aus: „Blinder Passagier“,
Rio Reiser (1987)*

1 Einleitung

Als Konrad Zuse in den dreißiger Jahren seine ersten elektromechanischen Rechenmaschinen konstruierte, hatte er wohl bei weitem keine Vorstellung von dem, was auf der Basis solcher Geräte und deren Weiterentwicklungen einmal alles möglich sein würde. Zuse hegte am Anfang lediglich den Wunsch, aufwendige statische Berechnungen zu automatisieren (/ZUS93/).

Ivan Sutherland stellte 1965 seine Idee vom ultimativen Display vor (/SUT65/) und läutete damit das Zeitalter der im Speicher eines Computers vorhandenen, virtueller Welten ein. Der Begriff Welt impliziert dabei neben anderen Aspekten (wie z.B. Realismus) auch Räumlichkeit im Sinne einer dreidimensionalen Darstellung.

Wieder stand am Anfang eine Vision. Es dauerte einige Jahre, bis Sutherlands Idee vom Bildschirm als Fenster zu einer realitätsnahen virtuellen Welt im

Computer Wirklichkeit wurde. Schritt für Schritt wurden die Bilder auf dem Computerbildschirm realistischer und lebendiger.

Auch die Entwicklung von Computerspielen zeigt die rasante Entwicklung auf diesem Gebiet. Am Anfang der Entwicklung standen textorientierte Systeme, die durch die Eingabe von Befehlen über die Tastatur bedient wurden und deren Ausgaben lediglich aus Botschaften bestanden, die der Spieler lesen mußte. Durch den Einsatz bewegter zweidimensionaler Grafiken erhöhte sich der Unterhaltungswert der Spiele enorm. Eingaben konnten über intuitiv bedienbare Geräte (z.B. Joystick) vorgenommen werden, die Ausgaben waren durch ihren grafischen Charakter ganzheitlicher aufnehmbar. Heute ist prinzipiell mit jedem Personal Computer die realistische Darstellung dreidimensionaler Objekte möglich. Flugsimulatorsoftware, die in jedem Kaufhaus erhältlich ist und auf jedem Heim-PC läuft, war bis vor einigen Jahren den Hochleistungsrechnern der militärischen Forschungseinrichtungen vorbehalten.

Diese Entwicklung, hin zu immer anspruchsvollerer Software für den breiten Massenmarkt, ist keineswegs abgeschlossen, sie setzt sich permanent mit immer atemberaubenderem Tempo fort. Wir sind heute an einem Punkt in dieser Entwicklung angekommen, an dem sich der Umgang mit dem Computer grundlegend verändern kann.

1.1 Virtuelle Umgebungen

Begriffsbestimmung

Die Forschung auf dem Gebiet virtueller Umgebungen (VU) ist noch sehr jung. Trotzdem oder gerade deshalb erfuhr diese Technologie in den letzten Jahren ein sehr großes Interesse durch die Medien. Leider existieren aber immer noch Schwierigkeiten, den Begriff genau abzugrenzen und eine allgemeingültige Definition vorzunehmen.

Die Konfusion bei der genauen Begriffsbestimmung, ist bereits am vielerorts verwendeten Begriff *virtuelle Realität (VR)* zu erkennen. Dieser Begriff charakterisiert den Gegenstand aus mehreren Gründen weniger gut. Einmal beinhaltet der Begriff virtuelle Realität eine gewisse Widersprüchlichkeit, da Virtualität und Realität Worte mit gegensätzlicher Bedeutung sind. Zum anderen wird unter virtueller Realität meist nur die möglichst realistische Nachbildung der Realität mit Hilfe einer Computersimulation verstanden. Zudem ist durch das große Medieninteresse und die damit verbundene Verwässerung der Bedeutung, der wissenschaftliche Charakter dieses Terminus verlorengegangen. Auf die Benutzung des Begriffs virtuelle Realität wird deshalb in dieser Arbeit weitestgehend verzichtet werden. In Übereinstimmung mit den meisten wissenschaftlichen Abhandlungen zu diesem Thema wird statt dessen der Begriff virtuelle Umgebung verwendet.

In der Literatur ist keine eindeutige Definition für virtuelle Umgebungen zu finden. Die verschiedenen Wissenschaftler versuchen, sich dem Gegenstand auf unterschiedliche Art zu nähern.

Kalawsky definiert virtuelle Umgebungen in /KAL93/ so:

In a virtual environment the human is immersed in a computersimulation that imparts visual, auditory and force sensations. The computer simulation can present conventional real-world environments without modification or entirely new environments where different (or no) physical laws exist. The human operator is allowed to interact with components of the virtual environment through his/her responses being sensed appropriately and coupled into the virtual environment simulation.

Vince (/VIN95/) versteht unter virtuellen Umgebungen lediglich

... a 3D data set describing an environment based upon real-world or abstract objects and data.

Eine andere Definition wurde 1992 bei einem Workshop an der University of North Carolina (/NSF92/) aufgestellt:

By virtual environments, we mean real-time interactive graphics with three-dimensional models, when combined with a display technology that gives the user immersion in the model world and direct manipulation.

Regenbrecht liefert in /REG94/ neben der Darstellung von Grundlagen, Einsatzgebieten und möglichen Auswirkungen, auch eine Definition des Begriffs virtuelle Realität:

Virtual Reality ist der Bereich der Kommunikation, welcher in synthetischen Räumen stattfindet und den Menschen als gleichberechtigten, integralen Bestandteil eines digitalen Systems versteht.

Es ist 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.

Es ist offensichtlich, daß sich die Definitionen des Begriffs virtuelle Umgebung / virtuelle Realität teilweise sehr stark voneinander unterscheiden. In dieser Arbeit werden virtuelle Umgebungen folgendermaßen charakterisiert:

[D1] Eine **virtuelle Umgebung** ist ein Computersystem, mit dessen Hilfe autonome, dreidimensional modellierte und dargestellte Entitäten (Objekte der virtuellen Umgebung) in Echtzeit¹ interagieren können. Ein Nutzer stellt da-

¹ Unter Interaktion in Echtzeit wird in diesem Zusammenhang verstanden, daß Eingaben des Nutzers vom System verzögerungsfrei bearbeitet werden, die Reaktionszeit des Systems also so kurz ist, daß der Nutzer sie nicht wahrnehmen kann.

bei ebenfalls eine solche Entität dar und ist damit integrierter Bestandteil der Umgebung.

Weitere Aspekte und Anforderungen an virtuelle Umgebungen sind denkbar (z.B. soziale und psychologische Aspekte einer künstlichen, nichtrealen Welt, der Einfluß anderer Sinneswahrnehmungen wie Fühlen und Schmecken auf das Erleben virtueller Umgebungen). Eine detaillierte Aufstellung und Untersuchung dieser Probleme ist aber nicht Gegenstand dieser Arbeit.

Einsatz-
gebiete

Eine virtuelle Umgebung, also eine synthetisch generierte Umwelt, mit Hilfe von Computern modelliert und dargestellt, und unter Verwendung spezieller Ein- und Ausgabegeräte für den Menschen nutzbar gemacht, kann den Umgang mit Computern nachhaltig verändern. Es ist abzusehen, daß virtuelle Umgebungen die Benutzung von Computern mindestens ebenso gravierend verändern, wie die Einführung graphischer Benutzeroberflächen dies getan hat.

Wenn virtuelle Umgebungen die reale Umgebung, in der wir uns tagtäglich bewegen, in Form und Funktion realistisch nachbilden, leuchtet die einfache Bedienbarkeit sofort ein, da der Benutzer auf seinen reichen, in der realen Welt gesammelten Erfahrungsschatz zurückgreifen kann.

Für den Benutzer ist der Computer hauptsächlich ein Hilfsmittel, mit dem er² Aufgaben besser und schneller durchführen kann, als dies auf konventionelle Art und Weise möglich wäre. Er will sich bei seiner Arbeit auf die Verwirklichung seiner Ziele konzentrieren und nicht auf die Bedienung der Maschine.

Aber nicht nur die intuitive Benutzbarkeit dreidimensionaler Objekte dient als Motivation für die Beschäftigung mit virtuellen Umgebungen. Es fällt schwer, sich auch nur annähernd die vielfältigen Einsatzmöglichkeiten virtueller Umgebungen vorzustellen. An dieser Stelle sei nur beispielhaft an Architektur (Besichtigung von Bauwerken, bevor sie gebaut werden), Medizin (realistische Darstellungen unzugänglicher Bereiche des menschlichen Organismus zum Zwecke besserer Diagnostik), Raum- und Luftfahrt (Simulation komplizierter und/oder gefährlicher Situationen auf der Erde) und Chemie (interaktive Simulation und Visualisierung komplexer Molekülstrukturen) gedacht.

² Männliche Bezeichnungen, wie z.B. *der Benutzer* sind in diesem Fall, wie auch in der gesamten Arbeit, nicht in der sprachlich eng gefaßten Bedeutung von *ein Benutzer = eine männliche Person* zu verstehen. Die deutsche Sprache weist in dieser Hinsicht einige Unzulänglichkeiten auf, deren Behebung kein triviales Problem ist (/PUS84/). Auch das Pronomen *man* wird in diesem Zusammenhang nicht als Super-Maskulin sondern als Bezeichnung für jeden Menschen (also jede Frau und jeder Mann) benutzt. Es ist allerdings auch möglich, ohne die Benutzung dieses „schlimmen“ Wortes auszukommen, wenn die Sprache unter dem Gesichtspunkt der Gleichberechtigung der Geschlechter bewußt eingesetzt wird. Pusch sensibilisiert den Leser in /PUS84/ für genau diese Problematik, sie stellt jedoch auch fest, daß dieser bewußte Einsatz der Sprache aufgrund der oben erwähnten Unzulänglichkeiten große Schwierigkeiten bereitet.

1.2 Verteilte virtuelle Umgebungen

Begriffsbestimmung Regenbrecht (/REG94/) greift den durch Gibson (/GIB87/) geprägten Begriff des **Cyberspace** auf und definiert ihn als Erweiterung seines VR-Begriffs:

... Cyberspace (ist) ein VR-System ..., das in ein internationales Computernetzwerk eingebunden ist.

Dieser Begriff umschreibt in etwa, was in dieser Arbeit unter verteilten virtuellen Umgebungen verstanden werden soll.

[D2] **Verteilte virtuelle Umgebungen** sind virtuelle Umgebungen, die mehr als einem Nutzer die Möglichkeit bieten, gemeinsam in eine virtuellen Umgebung einzutreten. Im Vordergrund steht die Möglichkeit der Interaktion der verschiedenen, geographisch getrennten Benutzer miteinander und den anderen Objekten der gemeinsamen virtuellen Umgebung.

Durch den Einsatz vernetzter Computersysteme kann aber auch eine Erhöhung der verfügbaren Rechenleistung erreicht werden, indem Rechnerverbünde parallel und gemeinsam als lose gekoppeltes System eine Aufgabe lösen (Parallelverarbeitung). Auch bei verteilten virtuellen Umgebungen wird die Verteilung zum Zweck der Leistungssteigerung eingesetzt. Diese Entwicklungsrichtung spielt in dieser Arbeit aber eine untergeordnete Rolle.

Einsatzgebiete Vernetzte Computer spielen eine immer wichtiger werdende Rolle bei der Kommunikation und Interaktion der Menschen, die die Computer benutzen. Das Internet ist hierfür ein sehr gutes Beispiel (/MEI96/). Auch die intensive Forschungstätigkeit auf dem Gebiet der computergestützten Zusammenarbeit (Computer Supported Collaborative Work – CSCW) spiegelt diese Entwicklung wider.

Virtuelle Umgebungen als Systeme, in denen der Nutzer integrierter Bestandteil ist, legen den Gedanken nahe, die Zusammenarbeit verschiedener, geographisch getrennter Nutzer zu unterstützen.

Aber auch die Zusammenarbeit von Menschen, die sich am selben Ort befinden kann durch den Einsatz verteilter virtueller Umgebungen erleichtert werden. Vorstellbar sind Diskussionen über in Planung befindliche Produkte am virtuellen Modell ebenso wie die gemeinsame Bearbeitung komplexer Aufgaben in einer virtuellen Umgebung.

1.3 Zur Motivation der Arbeit

Natürlich ist viel von dem bisher zu virtuellen und verteilten virtuellen Umgebungen Gesagten, heute noch eine Vision. Noch fehlt es an (preiswerter) Hardware, die ein unbeschwertes Erleben der virtuellen Umgebung ermöglicht. Es werden Ein- und Ausgabegeräte benötigt, die der Anforderung, intuitiv mit der virtuellen Umgebung interagieren zu können, gerecht werden. Auch die Bandbreiten, die mit den heutigen Weitverkehrsnetzwerken zur Verfügung stehen,

reichen bei weitem noch nicht aus, Gibsons Vision vom Cyberspace umzusetzen. Ein weiteres Problem stellt die Software dar, da neue Technologien immer auch neue Probleme mit sich bringen, die durch neue Software gelöst werden müssen.

Es ist festzustellen, daß sich virtuelle Umgebungen bei weitem noch nicht in dem Stadium befinden, in dem sie so eingesetzt werden könnten, wie optimistische VR-Enthusiasten dies von Zeit zu Zeit suggerieren wollen.

Gerade deshalb ist es aber notwendig, Forschung auf diesem Gebiet zu betreiben, um Probleme zu erkennen und entsprechende Lösungsansätze zu finden.

Die rasante Entwicklung der Rechen- und Grafikleistung in den letzten Jahren zeigt, daß die notwendige Hardware relativ schnell für viele Benutzer zur Verfügung stehen kann (/HEN96/, /KAT96/). Auch die Entwicklung neuartiger Ein- und Ausgabegeräte, welche die speziellen Anforderungen virtueller Umgebungen erfüllen, schreitet rasch voran. Die Leistungsfähigkeit der Computernetzwerke wird, nicht zuletzt durch den Internetboom der letzten Jahre, stetig verbessert. Daneben erfolgt natürlich auch intensive Forschungs- und Entwicklungstätigkeit auf dem Softwaresektor.

Es gilt, die neue Technologie der virtuellen Umgebungen in nutzbringende Anwendungen und Systeme zu integrieren, um Vor- und Nachteile am praktischen Beispiel abschätzen und untersuchen zu können. Die Beschäftigung mit virtuellen Umgebungen und speziell mit verteilten virtuellen Umgebungen ist deshalb eine lohnende Aufgabe.

Das Projekt **Virtual ObjectOriented Distibuted Interactive Environment (Voodie)**, welches zur Zeit an der Fakultät Medien der Bauhaus-Universität durchgeführt wird, widmet sich dem Thema der verteilten virtuellen Umgebungen aus genau diesen Gründen. Die vorliegende Arbeit soll dabei die kommunikationsrelevanten Aspekte verteilter virtueller Umgebungen untersuchen. Daneben müssen in dieser Arbeit aber auch die anderen Aspekte in Betracht gezogen werden, um die Integration des behandelten Teilaspektes in das Gesamtprojekt zu ermöglichen.

1.4 Zur Konzeption der Arbeit

Die Arbeit spricht in erster Linie Fachleute auf dem Gebiet der Informatik an, die sich mit verteilten virtuellen Umgebungen beschäftigen wollen. Ziel der Arbeit ist, Mechanismen zu entwickeln, mit deren Hilfe die speziellen, kommunikationsbezogenen Anforderungen verteilter virtueller Umgebungen erfüllt werden können. Damit soll eine Basis für weitere Forschungsarbeiten auf diesem Gebiet geschaffen werden.

Am Anfang steht die Untersuchung der Spezifik verteilter virtueller Umgebungen und eine Bestandsanalyse existierender Systeme. Darauf aufbauend werden die Mechanismen entwickelt. Als Ergebnis wird ein Prototyp vorgestellt, mit dem die entwickelten Mechanismen auf ihre praktische Anwendbarkeit hin untersucht werden können. Dieser Prototyp soll als Basis für die weitere Projekt-tätigkeit dienen. Dabei sollten die gewonnenen Erfahrungen als Ausgangspunkt

für eine grundlegende Überarbeitung der Implementierung benutzt werden.

Im Anhang der Arbeit findet sich ein Glossar, in dem wichtige Begriffe des untersuchten Gebietes erklärt werden. Diese wichtigen Begriffe werden an der Stelle ihres ersten Auftretens erklärt und sind an diesen Stellen *besonders ausgezeichnet*. Teilweise werden in der Arbeit auch Begriffe aus den Originalquellen, zum besseren Verständnis des Zusammenhangs, angeführt. Die Hervorhebung solcher Begriffe erfolgt durch **diese Darstellung**.

Alle in der Arbeit verwendeten Marken- und Firmennamen sind eingetragene Warenzeichen ihrer Eigentümer.

Allein machen sie dich ein ...

*aus: „Keine Macht für Niemand“,
Ton Steine Scherben (1972)*

2 Bestandsanalyse existierender Systeme

Die Idee, verteilte virtuelle Umgebungen zu schaffen, ist nicht neu. Es existieren verschiedene Systeme und Forschungsarbeiten, in denen mit unterschiedlichen Methoden versucht wird, sich dem Gegenstand zu nähern. In diesem Abschnitt werden einige Arbeiten näher vorgestellt. Zuvor erfolgt eine Aufstellung möglicher Kommunikations- und Verteilungsmethoden und allgemeiner Anforderungen an verteilte virtuelle Umgebungen. Dies ermöglicht eine Systematisierung der Lösungsmethoden.

2.1 Kommunikationsmechanismen

Verteilte virtuelle Umgebungen bestehen aus verschiedenen Stationen, die auf den Austausch von Informationen angewiesen sind, um die Konsistenz der gemeinsamen Umgebung zu gewährleisten. Jede Station bietet dabei eine eigene Sicht auf die gemeinsame virtuelle Umgebung. Prinzipiell sind verschiedene

Strategien für die Lösung des entstehenden Kommunikationsproblems vorstellbar.

2.1.1 Hierarchische Client/Server Modelle

Ein System für verteilte virtuelle Umgebungen kann als Client/Server System ausgelegt werden. Ein zentraler Server hält die Informationen der virtuellen Umgebung vor und gibt diese Informationen an Clients (Teilnehmer) weiter.

Der Server fungiert in einem solchen System auch als Synchronisationsinstanz. Alle Teilnehmer erhalten über den Server die gleichen Informationen über die virtuelle Umgebung. Um den notwendigen Netzwerkverkehr zu verringern, können Methoden zur bedingten Nachrichtenweiterleitung eingesetzt werden. Dadurch erhalten die jeweiligen Stationen nur relevante Daten.

Client/Server Systeme besitzen immer eine zentrale Instanz, die prinzipiell als Beschränkung des Systems angesehen werden muß. In den Systemen werden Filtermechanismen und die Verteilung von Aufgaben auf mehrere zentrale Server dazu benutzt, um diesen Nachteil auszugleichen.

Ein Weg, die große Kommunikationslast, die beim Server entsteht, zu verringern, ist der Einsatz von *Broad-* und *Multicast*³. Beim Broadcast wird dabei eine Nachricht einmalig abgeschickt und von allen Stationen empfangen, beim Multicast wird die einmal gesendete Nachricht von einer Gruppe von Empfängern empfangen. Broadcast hat dabei den Nachteil, daß sehr schnell Grenzen erreicht werden, da jede Station alle Nachrichten empfängt und auswerten muß (auch diejenigen, die nicht relevant sind). Multicast besitzt den Nachteil, daß die Einteilung in Multicastgruppen statisch erfolgt und somit eine dynamische Zuordnung von Stationen zu Multicastgruppen schwierig ist. Ein weiteres Problem sind sichere Netzwerkverbindungen. Um solche Verbindungen zu erhalten, ist es notwendig, daß der Empfänger den Erhalt eines Datenpaketes bestätigt, also an den Absender eine Bestätigungsnachricht schickt. Bei Broad- bzw. Multicast führt dies sofort zu einer erhöhten Netzwerkbelastung, die durch den Einsatz dieser Techniken eigentlich verhindert werden sollte.

2.1.2 Punkt-zu-Punkt Kommunikation

Systeme, bei denen die Kommunikation auf Punkt-zu-Punkt Verbindungen zwischen gleichberechtigten Stationen basieren, besitzen nicht den Nachteil einer zentralen Instanz. Ein Mangel solcher Systeme ist aber, daß theoretisch ein vollständiger Kommunikationsgraph zwischen allen Stationen notwendig ist. Dadurch wächst die Anzahl der notwendigen Verbindungen quadratisch zur Anzahl der beteiligten Stationen. Die sich ergebenden Einschränkungen für Systeme, die einen solchen vollständigen Kommunikationsgraph benutzen, liegen auf der Hand, wenn der entstehende Netzwerkverkehr betrachtet wird.

Es ist aber denkbar, daß Filtermechanismen eingesetzt werden, um redundante oder nicht benötigte Verbindungen zu vermeiden.

³ Als weiterführende Literatur zu Multicast sei an dieser Stelle auf /LOO91b/ verwiesen.

Auch bei solchen Systemen ist es möglich, Broad- bzw. Multicastmechanismen anzuwenden. Die oben beschriebenen Vor- und Nachteile gelten analog.

Kommunikation über Punkt-zu-Punkt Verbindungen hat den Vorteil, daß prinzipiell alle Module gleichberechtigt sind. Alle notwendigen Funktionen sind bei jedem Teilnehmer vorhanden, Es existiert keine zentrale Instanz, die zum Engpaß werden könnte.

2.1.3 Hybride Ansätze

Weiterhin ist es möglich, die oben beschriebenen Methoden zu kombinieren. Solche hybriden Ansätze könnten z.B. so aussehen, daß mehrere Server eine gewisse Anzahl von Clients verwalten. Zwischen den Clients und dem Server existiert die oben beschriebene Client/Server Struktur. Die Server untereinander sind über Punkt-zu-Punkt Verbindungen gekoppelt. Andererseits ist aber auch vorstellbar, daß ein zentraler Server als Verbindungsmanager fungiert. In einem solchen System würden zwischen den Teilnehmern Punkt-zu-Punkt Verbindungen bestehen, die durch einen zentralen Server verwaltet werden. Mit solchen Ansätzen ist es möglich, die Nachteile einer bestimmten Art der Kommunikation abzuschwächen und damit insgesamt die Performance zu erhöhen.

2.2 Verteilungsmechanismen

Die Verteilung der Daten in verteilten virtuelle Umgebungen kann durch zwei Merkmale kategorisiert werden. Ein Merkmal ist der Ort, an dem die Daten verwaltet werden. Dieser Ort ist entweder zentral oder dezentral. Das zweite Merkmal ist die Art der Replikation der Daten. Die Daten können entweder als vollständige Replikation bei allen Teilnehmern vorliegen, oder es erfolgt lediglich die Replikation der Daten, die aktuell von der jeweiligen Station benötigt werden.

Danach können vier Verteilungsmechanismen unterschieden werden. Die einzelnen Mechanismen sind in Tabelle 1 dargestellt.

	vollständige Replikation (VRep) (multi-user-same-content)	Replikation nach Bedarf (RepNB) (multi-user-different-content)
zentrale Datenverwaltung (ZeD)	zentrales Datenmodell mit vollständiger Replikation auf allen Stationen (ZeD-VRep)	zentrales Datenmodell mit Replikation der Daten nach Bedarf der Stationen (ZeD-RepNB)
dezentrale Datenverwaltung (DeD)	dezentrales Datenmodell mit vollständiger Replikation auf allen Stationen (DeD-VRep)	dezentrale Datenverwaltung mit Replikation der Daten nach Bedarf der Stationen (DeD-RepNB)

Tabelle 1: Verteilungsmechanismen

Die einzelnen Verteilungsmechanismen besitzen verschiedene Vor- und Nachteile. **Multi-user-same-content** (/SIN94/) Systeme gehen von einer vollständigen Replikation aller Objekte der virtuellen Umgebung auf allen beteiligten Stationen aus. Dies führt zu einer hohen Belastung der Netzwerkverbindungen

zwischen den Stationen, da jede Änderung einzelner Objekte an alle beteiligten Stationen weitergegeben werden muß. In Systemen, die keine vollständige Replikation der Objekte erfordern, muß aber ein Mechanismus vorhanden sein, mit dessen Hilfe entschieden werden kann, welche Objekte repliziert werden und welche nicht. Die zentrale Verwaltung der Objekte ermöglicht eine zentrale und damit einfacher umzusetzende Interaktionserkennung. Gleichzeitig ergibt sich aber die Notwendigkeit, alle Informationen an einer zentralen Stelle zu verwalten. Dies kann leicht zu einer Überlastung führen. Dezentrale Datenverwaltung impliziert die gleichmäßige Aufteilung der Informationen auf die Teilnehmer der verteilten virtuellen Umgebung. Das Problem der Überlastung kann dadurch gelöst werden. Wenn aber an keiner Stelle die vollständigen Informationen der Umgebung vorhanden sind, erhöht sich die Komplexität der Interaktionserkennung.

Um komplexe Welten zu erzeugen, sollte ein DeD-RepNB Mechanismus eingesetzt werden. Prinzipiell ist ein solcher Verteilungsmechanismus am besten geeignet, große verteilte virtuelle Umgebungen zu unterstützen. Diese Art der Verteilung erfordert aber auch den größten Aufwand bei der Implementierung und erfordert den Einsatz von Mechanismen, die noch nicht vollständig entwickelt sind.

2.3 Informationen in verteilten virtuellen Umgebungen

In verteilten virtuellen Umgebungen existieren verschiedene Arten von Informationen, die über Kommunikation zwischen den Teilnehmern ausgetauscht werden müssen. Es können folgende Informationsarten unterschieden werden:

[K1] Statisches Aussehen der gemeinsamen Welt.

Die Stationen, die an einer gemeinsamen virtuellen Umgebung beteiligt sind, müssen Informationen austauschen, die die Umgebung beschreiben. Angenommen, einige Nutzer wollen sich in einem virtuellen Büro treffen, um eine Besprechung abzuhalten. In diesem Fall ist es notwendig, daß alle Teilnehmer die gleichen Informationen über das Aussehen des Büros (der Umgebung) erhalten. Diese Informationen sind relativ langlebig, ein neuer Teilnehmer der Umgebung muß diese Informationen lediglich einmal erhalten. Problematisch ist dabei allerdings die relativ komplexe Struktur dieser Informationen, welche zu großen Datenmengen und damit zu langen Übertragungszeiten führt.

[K2] Dynamische Veränderungen der gemeinsamen Welt.

Veränderungen in der gemeinsamen Welt müssen ebenfalls (möglichst zeitsynchron) an alle Teilnehmer übertragen werden, um die Konsistenz der Welt sicherzustellen.

Ein Spezialfall dynamischer Änderungen in der verteilten Umgebung ist die direkte Kommunikation zwischen einzelnen Teilnehmern.

Die Einteilung in statische und dynamische Informationen ist nicht starr, sondern hängt vom Systemdesign ab.

Ausgehend von dieser Einteilung, kann die auftretende Netzlast und die Flexibilität einer verteilten virtuellen Umgebung, in Abhängigkeit von den statischen bzw. dynamischen Informationen, dargestellt werden. Aus der Abhängigkeit läßt sich schlußfolgern, daß die Definition statischer und dynamischer Informationen unter dem Gesichtspunkt der Netzlast und der notwendigen Systemflexibilität erfolgen muß (Abbildung 1).

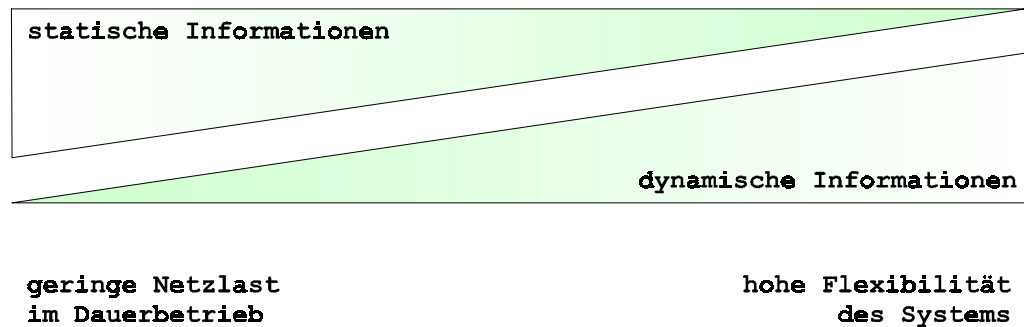


Abbildung 1: Abhängigkeit von Eigenschaften des Systems von der Informationsaufteilung

2.4 Spezielle Anforderungen an verteilte virtuelle Umgebungen

Wie bereits gesagt, werden an virtuelle Umgebungen folgende Anforderungen gestellt:

- [A1] Dreidimensionale Darstellung und Modellierung der Umgebung.
- [A2] Interaktive Manipulation in Echtzeit.
- [A3] Nutzer als integraler Bestandteil der Umgebung.

An verteilte virtuelle Umgebungen werden, neben den allgemeinen Anforderungen, zusätzlich folgende Forderungen gestellt:

- [B1] Jedem Beteiligten muß eine verzögerungsfreie Sicht auf eine konsistente, gemeinsame Welt ermöglicht werden.
- [B2] Jedem Beteiligten muß die Möglichkeit der Interaktion mit der Welt (bzw. mit den autonomen Objekten der Welt) und den anderen Teilnehmern geboten werden.
- [B3] Es sollten keine prinzipiellen Beschränkungen in Bezug auf die Anzahl der beteiligten Nutzer vorhanden sein.
- [B4] Es sollten keine prinzipiellen Beschränkungen in Bezug auf die Komplexität der virtuellen Umgebung vorhanden sein.

Es ist leicht einzusehen, daß die Erfüllung dieser Ansprüche einen erheblichen Kommunikationsbedarf zwischen den Computern (Stationen), über die die Nutzer in die virtuelle Umgebung eintauchen, hervorrufen.

Werden die Probleme der virtuellen Umgebungen für Einzelnutzer beiseite gelassen, so ist der entstehende Kommunikationsaufwand das größte Problem bei verteilten virtuellen Umgebungen. Die Anforderungen erfordern eine Reduzierung des Verkehrs im Netzwerk, da dieses sehr schnell zum Engpaß bei verteilten Anwendungen wird. Weiterhin ist noch nicht klar, welche Netzwerkbelastung wirklich auftritt, da nur wenige Aussagen zur quantitativen Beschaffenheit der in virtuellen Umgebungen verwendeten Daten bekannt sind. Prinzipiell sollten alle bekannten Medien (3D Geometrie, akustische Informationen, bewegte Bilder usw.) integriert werden. Dies führt aber sicherlich nicht zu geringeren Ansprüchen an die Übertragungskanäle.

Deshalb ist es erforderlich, die Kommunikation auf das absolute Minimum zu begrenzen. Zusätzlich ist es notwendig, dafür Sorge zu tragen, daß keine Überlastung einzelner Teile des Systems erfolgt, bzw. die Belastung auf alle Teilnehmer im System gleichmäßig verteilt wird, da sich sonst die überlasteten Stationen als Engpaß des Systems entpuppen.

2.5 Beschreibung der Systeme

Schwerpunkt bei der Untersuchung waren Systeme, die verteilte virtuelle Umgebungen für mehrere Nutzer zur Verfügung stellen. Neben den oben beschriebenen Kriterien zur Kommunikation und Verteilung, wurde besonderer Wert auf die Beantwortung der folgenden Fragen gelegt:

- (1) Ist das System eine Applikation oder ist es eher als Basis (Toolkit, Programmbibliothek) für andere Anwendungen ausgelegt?
- (2) Gibt es im System eine Beschränkung hinsichtlich der Größe und Komplexität der Welt(en), bzw. der Anzahl der gleichzeitig agierenden Nutzer in ein und derselben Welt und wo liegen die Ursachen für die etwaigen Beschränkungen?
- (3) Werden vom System Mechanismen unterstützt, die eine effektive Nutzung der Ressourcen ermöglichen und welche Methoden werden benutzt?
- (4) Auf welche Art und Weise erfolgt die Definition der Bestandteile (Objekte) der virtuellen Umgebung? Lassen sich vollkommen neue Bestandteile dynamisch in das System integrieren?
- (5) Läßt das Systemdesign prinzipielle Erweiterungen zu?

2.5.1 DIVE

DIVE (**D**istributed **I**nteractive **V**irtual **E**nvironment) wurde am Swedish Institute of Computer Science (SICS) entwickelt und 1993 von Carlsson/Hagsand vorgestellt (/CAR93/). Das System basiert auf UNIX als Betriebssystem und den Internet Protokollen zur Kommunikation. DIVE steht für nichtkommerzielle Nutzung frei zur Verfügung.

DIVE besteht grundsätzlich aus Welten (**worlds**) und Prozessen (**processes**). Prozesse können in DIVE entweder Applikationen sein, oder Menschen die in der virtuellen Umgebung dargestellt werden. Die Daten der virtuellen Umge-

bung werden in einer Art verteilter Datenbank abgelegt. Jeder DIVE-Prozeß besitzt eine Kopie dieser Datenbank. Die Konsistenz der Daten wird über Nachrichten (**messages**) und Ereignisse (**events**) sichergestellt, die die einzelnen Prozesse austauschen. Jede Veränderung wird über Broadcast⁴ Messages an alle anderen Prozesse weitergeleitet. Die gesamte virtuelle Umgebung ist in Welten (**worlds**) unterteilt. Die einzelnen Welten sind unabhängig voneinander. Der Nutzer kann allerdings dynamisch von einer Welt in eine andere übergehen. In diesem Falle werden die Daten der neuen Welt vollständig repliziert. Informationen legt DIVE in Objekten ab. Objekte sind die Elementarbausteine der replizierten Datenbank.

Als Benutzeroberfläche werden in DIVE verschiedene Konzepte eingeführt. Ein Benutzer z.B., wird durch ein **Body-Icon** dargestellt. Die Navigation und Fortbewegung in der virtuellen Umgebung erfolgt über Fahrzeuge (**vehicles**).

Durch das Prozeßkonzept von DIVE ist es möglich, Applikationen einzubinden. Diese Applikationen können unter Umständen sogar unabhängig vom Benutzerinterface laufen. Applikation und Benutzerschnittstelle kommunizieren über die replizierte Datenbank.

DIVE diene als Basis für weitere Forschungsarbeiten auf dem Gebiet der verteilten virtuellen Umgebungen, z.B. Einsatz von 3D-Audio (/POP93/).

Durch die Art der Kommunikation und die Aufteilung in vollständig replizierte Welten ist das System in der Anzahl der gleichzeitig in einer Welt agierenden Personen beschränkt, weil bei jedem Teilnehmer eine vollständige Kopie der replizierten Datenbank vorliegen muß. Auch die Größe der einzelnen Welten ist begrenzt, da immer alle Zustandsänderungen aller Objekte in dieser Welt an alle Prozesse weitergegeben werden.

2.5.2 VEOS

Die Virtual Environment Operating Shell (VEOS) wurde ab 1990 am Human Interface Technologie Laboratory der University of Washington in Seattle entwickelt (/BRI94/). Das Projekt wurde 1994 abgeschlossen.

VEOS ist, wie der Name bereits sagt, ein Ansatz, der in Richtung Betriebssystemaufsatz für virtuelle Umgebungen geht. Dieser Ansatz ist sehr interessant, wenn in Betracht gezogen wird, daß Applikationen für virtuelle Umgebungen den Umgang mit Computern sicher nachhaltig verändern werden und deshalb andere Anforderungen an ein Betriebssystem stellen, als heutige Anwendungen dies tun. Im VEOS Projekt wurde intensive theoretische Vorarbeit auf diesem Gebiet geleistet.

VEOS besteht aus einem Kernel, den Modulen **FERN** (eine Art Verteilungsmanager), **SENSORLIB** (Device driver) und dem Renderer **IMAGER**. Prinzipiell kann eine VEOS Welt in einen Interpretationsteil, in einen Modellierungsteil und einen Darstellungsteil aufgespalten werden. Der VEOS Kernel

⁴ In einer neueren Version des DIVE Systems wird auch Multicast unterstützt.

stellt einfache Möglichkeiten zum Datentransport zur Verfügung (NANCY – eine Variante des Linda Parallel Database Model). Die Programmiersprache, in der VEOS Applikationen geschrieben werden, ist Lisp.

VEOS basiert wie DIVE auf direkter Kommunikation zwischen einzelnen Teilnehmern der virtuellen Umgebung. Dies führt sehr schnell zu Performanceproblemen, wenn die Anzahl der partizipierenden Prozesse steigt.

Methoden zur Ressourcenschonung sind nicht eingebaut – es werden immer alle Informationen allen anderen Teilnehmern mitgeteilt.

Die Nutzung der Daten-Programm-Identität von Lisp ermöglicht es, interessante Wege bei der Verteilung und Kommunikation einzuschlagen. Nachteilig wirkt sich das Kommunikationsmodell aus, da es nicht skaliert, sondern sehr schnell Grenzen erreicht.

2.5.3 Distributed Virtual Environment System

Das Distributed Virtual Environment System (**dVS**) der britischen Firma Division Limited, ist nach Grimsdale /CRI91/ ein Betriebssystem für Applikationen in virtuellen Umgebungen. Es stellt eine verteilte Datenbank und entsprechende Manipulationsmechanismen als Basis der Kommunikation zur Verfügung. Ein dVS besteht aus Akteuren (**actors**), Elementen (**elements**) und Instanzen (**instances**). Akteure sind diskrete Prozesse, die in ihrer Gesamtheit die virtuelle Umgebung steuern. Akteure werden eingesetzt, um Aufgaben in der virtuellen Umgebung zu erfüllen. Solche Aufgaben reichen von der Überwachung von Eingabegeräten bis zur Animation von dynamischen Objekten. Elemente sind einfache Objekte, die von den Akteuren gesteuert werden, z.B. Lichtquellen (**lights**) oder Objekte der Umgebung (**EnvObjects**). Instanzen sind die Datenstrukturen, die die aktuell gültigen Daten einer Umgebung enthalten. Akteure generieren in einer Umgebung Instanzen von Elementen. Akteure halten (**hold**) nun bestimmte Instanzen. D.h. der Akteur betreut, überwacht und verändert den Status (**state**) der Instanzen, die von ihm gehalten werden.

Akteure können den Status der lokalen Instanzen verändern. Wenn aber der Status aller im verteilten System vorhandenen Kopien verändert werden soll, dann muß der Akteur dies explizit veranlassen (**update**). Ein spezieller Akteur (**director**) übernimmt die Verteilung dieser Nachrichten und Ereignisse.

Das System ist als kommerzielles Produkt konzipiert und unterliegt entsprechenden Lizenzbestimmungen.

Es existiert ein API (**VL**), das es ermöglicht, Akteure zu implementieren. Auf einem höheren Level ist auch ein Toolkit (**VCTools**) vorhanden, mit dem komplexere Aufgaben auf höherem Niveau gelöst werden können.

Das System bietet Unterstützung für die Parallelisierung der einzelnen Aufgaben des VR-Systems. Die Unterstützung mehrerer Nutzer ist auch vorgesehen, es fehlen aber spezielle Methoden, mit denen die Probleme größerer verteilter Umgebungen gelöst werden können.

2.5.4 VUE

Das Veridical User Environment (VUE) wurde am IBM T. J. Watson Research Center New York entwickelt (/APP92/). VUE ist als Client/Server System ausgelegt. Das System ist nicht als Interaktionssystem für mehrere Benutzer konzipiert, sondern als verteiltes System zur Visualisierung sehr großer, dynamischer Datenmengen. Prozesse kommunizieren über Nachrichtenaustausch miteinander. Es existieren sogenannte Device-Server, die über einen Dialogmanager von den eigentlichen Applikationen getrennt sind. VUE ist als Basis für VR-Applikationen anzusehen. Die strikte Trennung von Devices, Kommunikation und Applikationen bietet ein Grundgerüst für die Implementation virtueller Umgebungen. Im System sind allerdings keine Mechanismen vorgesehen, die eine effektivere Ressourcennutzung ermöglichen würden. Das System ist aber ohnehin nicht für die Interaktion vieler Nutzer, bzw. zur Implementation großer virtueller Umgebungen gedacht.

VUE diente als Basis für weitere Forschung auf dem Gebiet der verteilten virtuellen Umgebungen am IBM Research Center.

2.5.5 Rubber Rocks

Rubber Rocks wurde am IBM T. J. Watson Research Center, Yorktown Heights, New York entwickelt (/COD92/).

Es kombiniert flexible Objektsimulatoren mit einer VR Benutzeroberfläche. Das System basiert auf einer Client/Server Architektur. Ein Dialogmanager in Rubber Rocks fungiert als Bindeglied zwischen Nutzer und virtueller Welt. Der Dialogmanager kommuniziert mit Device- und Simulationsservern, um Informationen über die virtuelle Welt zu erhalten. Sollen mehrere Personen in der virtuellen Umgebung agieren, werden mehrere Dialogmanager eingesetzt. Diese Dialogmanager tauschen untereinander Informationen aus (z.B. die Position eines Nutzers).

Das System ist eher eine flexibel gestaltete Applikation als ein Toolkit. Die Trennung von Inhalt (**content**) und Darstellung (**style**) war ein wichtiges Paradigma der Entwickler. Als Inhalt wird dabei die Funktionalität der Bestandteile des Systems und als Darstellung deren Benutzerinterface angesehen.

Die Entwickler schlagen den Einsatz von komplexen Ereignissen (**higher-level events**) vor. Dies ist ein Ansatz, durch den die notwendige Kommunikation reduziert werden könnte. Interessant ist auch die Integration von physikalischen Gesetzen und Objektsimulatoren.

Rubber Rocks ist nicht ohne weiteres auf viele Nutzer und große Welten erweiterbar, da es die exponentiell wachsende Informationsmenge nicht in geeigneter Weise reduzieren kann. Es fehlt an prinzipiellen Mechanismen, die eine solche Reduktion ermöglichen könnten.

2.5.6 VR-DECK

Das Virtual Reality Distributed Environment and Construction Kit (VR-DECK) wurde am IBM T. J. Watson Research Center, Yorktown Heights, New

York, entwickelt (/COD93/). Die Entwickler ließen Erkenntnisse, die mit dem Rubber Rocks System gesammelt wurden, in das System einfließen.

VR-DECK benutzt Module (**modules**) als Grundbausteine der virtuelle Umgebung. Diese Grundbausteine schließen Objekte, Operationen, Funktionen und Nutzer ein. Die Module kommunizieren über Ereignisse (**events**), die sie entweder erzeugen oder verarbeiten. Die Verarbeitung der Ereignisse basiert dabei auf Regeln (**rules**), die in C++ geschrieben werden.

Welten in VR-Deck bestehen aus Modulen, die Ereignisse austauschen. Wenn zwei Module sich miteinander verbinden, so wird durch das Laufzeitsystem sichergestellt, daß nur diejenigen Ereignisse vom sendenden Modul zum empfangenden übertragen werden, die vom empfangenden Modul auch bearbeitet werden. Dieser Filtermechanismus stellt sicher, daß keine unnötigen Daten über das Netzwerk übertragen werden. VR-DECK vereinfacht die Erstellung einer virtuellen Umgebung durch eine intuitiv benutzbare Benutzeroberfläche. Es ist möglich, neue Module mit neuen Regeln zu erstellen und in die VR-DECK Bibliothek zu integrieren.

Ressourcenschonung kann durch das Verändern der Regeln der einzelnen Module erreicht werden.

Problematisch ist die Erzeugung von großen Welten. Es ist denkbar, daß durch die zentrale Regelauswertung des Systems die Rechenleistung eines einzelnen Computers bei großen Welten nicht ausreicht, um alle Regeln schnell genug auszuwerten und die resultierenden Ereignisse an die einzelnen Modulen weiterzuleiten.

Ein anderer Nachteil des Systems ist, daß beim Start eines Moduls alle anderen Module auch gestartet werden müssen. Zum Einen ergibt sich daraus das Problem, daß eine Applikation auf einer fremden Maschine gestartet werden muß (also die nötigen Rechte vorhanden sein müssen), zum Anderen müssen eben immer alle Module auf allen Maschinen ausgeführt werden, was den dynamischen Eintritt neuer Nutzer in die Umgebung unmöglich macht.

2.5.7 BrickNet

Das BrickNet Toolkit wurde am Institute of Systems Science der University of Singapore entwickelt (/SIN94/).

Es unterstützt die grafische, funktionale und netzwerkspezifische Modellierung virtueller Umgebungen. Dabei wird von virtuellen Welten ausgegangen, die Objekte gemeinsam benutzen (**shared objects**).

Virtuelle Welten in BrickNet funktionieren objektorientiert, bestehen also aus Objekten, die ihre gesamte Funktionalität selbst enthalten (Grafik, Verhalten, Netzwerk). Objekte können zur Laufzeit dynamisch erzeugt und gelöscht werden, Attribute sind veränderbar. Die BrickNet Welten müssen nicht zwangsläufig alle den gleichen Inhalt haben. Jede virtuelle Welt verwaltet ihre eigenen Objekte. Diese Objekte können u.U. gemeinsam von verschiedenen, verteilten Welten benutzt werden. Die Kommunikation zwischen den Welten (**Clients**) geschieht über eine Client/Server Architektur. Server fungieren dabei als Ob-

jektanforderungsverteiler (**Object Request Broker**). Objekte in BrickNet sind ebenfalls Clients. Diese Objekte stellen sich über die Server anderen Clients zur Verfügung. Dabei ist auch eine Zugriffsrechteverwaltung mit eingebaut. BrickNet wurde in C und mit Hilfe der Starship (/LOO91/)

Programmiersprache implementiert. Wenn ein Objekt über das Netzwerk verteilt benötigt wird, so wird der Starship Quelltext übertragen. Auch Objektaktualisierungen werden im Starshipcode übertragen und beim entfernten Rechner ausgeführt. Dabei übernimmt der BrickNet Server die Aufgabe der Verteilung, Weiterleitung und Filterung solcher Nachrichten.

Wesentlich an BrickNet ist, daß die verschiedenen Benutzer nicht zwangsläufig die gleiche virtuelle Umgebung vor sich haben, obwohl dies durchführbar wäre. Dadurch kann die Komplexität der Welt, die auf der Station eines Teilnehmers vorhanden ist, begrenzt werden.

BrickNet Clients kommunizieren nicht direkt miteinander, sondern immer nur über den Server. Ein Client in BrickNet besteht aus mehreren Schichten. Die Interaktionsunterstützungsschicht (**interaction-support Layer**) ist sozusagen die VR-Oberfläche. Sie besteht aus verschiedenen Modulen (**bricks**), die verschiedenen Aufgaben übernehmen (Gerätetreiber - **device brick**, grundlegende 3D Grafik - **geometry brick**, usw.). Die zweite Schicht ist die VR-Wissenschicht (**VR Knowledge Layer**). Diese Schicht verwaltet die Objekte einer virtuellen Welt. Dabei kommen Gesetze (**laws**) zum Einsatz, die verschiedenes Verhalten der Objekte hervorrufen. BrickNet geht in der Wissensschicht von verschiedenen Klassen aus, die die einzelnen Wissenskomponenten enthalten. Solche Klassen sind die Geometrie- und Weltklassen (**Solid- and World Classes**), die Überwachungs- (**Controller-**) und Aktionsklassen (**Action Classes**) und die Objekt- und Abgleichskontrolle (**Object- and Updatemanagement**)

Bei der Objektverwaltung ist das Konzept des derzeitigen Besitzers (**current owner**) eines Objektes interessant. Dieses Konzept verhindert Inkonsistenzen durch gleichzeitiges Verändern eines Objektes durch mehrere Clients. Um Besitzer eines Objektes zu werden, wird eine entsprechende Anforderung an den Server gesendet. Wenn niemand das Objekt besitzt, vergibt der Server die Besitzrechte und sperrt das Objekt für andere Clients solange, bis das Objekt wieder freigegeben wird.

In der Kommunikationsschicht wird die Kommunikation des Clients mit dem Server abgewickelt. Ein Kommunikationsverwalter ist als separater Prozeß ausgelegt.

BrickNet Server bestehen ebenfalls aus verschiedenen Schichten die folgende Aufgaben haben: Clientverwaltung (**Client Management Layer**), Objektverwaltung (**Object Management Layer**), Aktualisierungsanforderungsverwalter (**update request handler**) und Kommunikation (**communication Layer**).

Nachteil des BrickNet Systems ist ganz klar der Server, der zum Kommunikationsflaschenhals werden kann. Die Entwickler schlagen eine Lösung vor, in der die Aufgaben des Servers zusätzlich von den Clients übernommen werden.

Dies führt aber zu komplizierteren Clientprogrammen und könnte die Interaktionsgeschwindigkeit evtl. nachteilig beeinflussen.

Die Netzlast kann in BrickNet durch die Implementation der Objekte beeinflusst werden. Jedes Objekt entscheidet selbst, wie oft es entfernte Kopien aktualisiert, und wie die entfernten Kopien auf diese Aktualisierung reagieren.

2.5.8 NVR

Networked Virtual Reality (NVR) wurde durch Wissenschaftler der Bell Communications Research entwickelt (/BER94/).

NVR stellt ein Software Toolkit dar, welches die Generierung von verteilten (**networked**) virtuellen Umgebungen unterstützt. Das System basiert auf einer Client/Server Architektur. Dabei existiert ein zentraler Kommunikationsserver, über den die Kommunikation der Clients untereinander erfolgt. Alle Clients besitzen eine Kopie der Datenbank, die die virtuelle Umgebung beschreibt. Über Nachrichten, die der Server verteilt, wird die Konsistenz der bei den Clients replizierten Daten sichergestellt. Es wird ein Mechanismus vorgestellt, der bei zu hohen Belastungen (Rechen- bzw. Netzwerkleistung) die Zeit zwischen solchen Nachrichten vergrößert. Dies führt aber zu deutlichem Qualitätsverlust, da die fehlenden Informationen in keiner Weise interpoliert werden. Als weitere Methode die Kommunikation zu verringern, benutzt NVR lokale Objekte (**local objects**). Diese Objekte werden allein von der lokalen Datenbank verwaltet. Denkbare Anwendungen sind hier z.B. Kontrollpanelobjekte, die sich jeder Nutzer lokal anpassen kann, oder auch Objekte, bei denen es nicht auf zeitsynchrones Verhalten bei allen Nutzern ankommt (z.B. ein Objekt welches zur besseren Erkennbarkeit immer um seine eigene Achse rotiert, oder eine Fackel deren Flamme flackert).

Das Problem großer virtueller Welten wird von NVR über getrennte Welten, die über Portale (**portals**) verbunden sind, gelöst. Beim Neueintritt in das System oder beim Übergang in eine andere Welt wird vom Server eine vollständige Kopie der aktuellen Welt geliefert. Der Server hält also von jeder Welt eine Kopie vor und aktualisiert diese Welt permanent.

Dieser Punkt ist problematisch, da dadurch der Server sehr schnell zum Flaschenhals des Systems werden kann.

NVR zielt auf Low-End Hardware ab. Das System ist für heterogene Umgebungen geeignet, es basiert auf TCP/IP und benutzt entfernte Prozeduraufrufe (**remote procedure calls – RPC**⁵) zur Kommunikation.

2.5.9 MASSIVE

Model, Architecture and System for Spatial Interaction in Virtual Environments (MASSIVE) ist ein experimentelles System. Es wurde von Greengalgh und Benford an der University of Nottingham entwickelt und 1995 auf der Conference on Distributed Computing in Vancouver vorgestellt (/GRE95/).

⁵ Siehe /BLO92/ für weitergehende Informationen.

Es benutzt verteilte virtuelle Umgebungen mit dem Ziel der Unterstützung gemeinsamen Arbeitens. Die Entwickler von MASSIVE haben bei ihrer Arbeit großen Wert auf das gemeinsame Arbeiten und auf die Bereitstellung von Schnittstellen für verschiedene Arten der Kommunikation gelegt. Die Entwicklung von MASSIVE wurde von zwei Grundideen getragen: Erstens sollen so viele gleichzeitig agierende Nutzer wie möglich unterstützt werden und zweitens soll Interaktion auch zwischen solchen Nutzern möglich sein, die radikal verschiedene Benutzeroberflächen benutzen bzw. deren Ausstattung sich sehr stark von einander unterscheidet.

MASSIVE benutzt für die Kommunikation zwischen Benutzern das räumliche Modell der Interaktion (**spatial model of interaction**). Dieses Modell geht von zwei Grundkomponenten aus. Die eine Komponente – Skalierbarkeit (**scalability**) – basiert auf dem Aurakzept. Jedes Objekt in einer virtuellen Umgebung besitzt eine Aura, also einen gewissen räumlichen Einflußbereich. Diese kann für jedes Medium in dem es interagieren kann, verschieden sein. Die Aura legt die Grenze fest, bis zu der das Objekt im jeweiligen Medium mit anderen Objekten interagieren kann. Interaktion kommt nur dann zustande, wenn die Aura zweier Objekte miteinander kollidieren (**Aura collision**). Die Kontrolle, ob eine solche Kollision vorliegt, übernimmt ein Auramanager. Der Auramanager informiert die Objekte, die an einer Kollision beteiligt sind und diese Objekte können dann eine Punkt-zu-Punkt Verbindung zueinander aufnehmen. Die zweite Komponente dieses räumlichen Interaktionsmodells ist die Wahrnehmung anderer Objekte (**awareness**)⁶. Diese ist beim räumlichen Modell in weiter Entfernung weniger stark ausgeprägt als beim direkten Kontakt und kann etwa mit einem Detaillierungsgrad (**Level of Detail, LOD**) verglichen werden. Um die Wahrnehmung zu steuern, wurden Fokus (**focus**) und Nimbus (**nimbus**) eingeführt. Fokus geht immer von Beobachtern aus und gibt an, wie stark der Beobachter sein Wahrnehmung auf ein bestimmtes Objekt konzentriert. Nimbus beschreibt eine Art Sichtbarkeit. Mit dem Nimbus kann ein Objekt steuern, wie stark es selbst von anderen Objekten (Beobachtern) wahrgenommen wird. Wie ein Objekt ein anderes Objekt nun wahrnimmt hängt zum einen vom eigenen Fokus des Objektes ab und zum Anderen vom Nimbus der beobachteten Objekte.

Aura, Fokus und Nimbus können auf drei verschiedene Arten beeinflusst werden. Einmal kann eine implizite Änderung erfolgen. Dies ist z.B. bei einer Bewegung durch die virtuelle Umgebung sinnvoll. Natürlich kann auch explizit Einfluß genommen werden, z.B. indem einfach verschiedene Größen oder Gestalten bzw. enger oder weiter Fokus gewählt werden. Eine weitere Einflußmöglichkeit auf diese Parameter sind Adapter. Adapter sind Objekte, die eine Transformation von Aura, Fokus und Nimbus vornehmen. Ein Beispiel für einen solchen Adapter ist z.B. ein Mikrofonobjekt. Dieses Objekt kann Aura, Fokus und Nimbus anderer Objekte verändern und dadurch bestimmte Funktionen bereitstellen (in diesem Beispiel Verstärkung von akustischen Informationen).

⁶ Der englische Begriff „awareness“ wurde hier mit „Wahrnehmung anderer Objekte“ übersetzt, da diese Deutung des Begriffs den Gegenstand am ehesten umschreibt.

MASSIVE unterstützt die Existenz mehrerer unabhängiger virtueller Welten, die über Portale miteinander verbunden sind. Kommunikation kann graphisch, über Audio oder mit Hilfe eines Textinterfaces bzw. als Kombination dieser Medien, erfolgen.

Das Verteilungsmodell von MASSIVE benutzt typisierte Punkt-zu-Punkt Verbindungen, die eine Kombination aus entfernten Prozeduraufrufen (**remote procedure calls**, /BLO92/), gemeinsamen Attributen (**shared attributes**) und Strömen (**streams**) sind.

Interaktionen können nur dann zustande kommen, wenn zwei Voraussetzungen erfüllt sind: Die Objekte müssen mindestens ein gemeinsames Interface besitzen und sie müssen nah genug beisammen sein. Beide Voraussetzungen spiegeln sich im Konzept des räumlichen Austausches (**spatial trading**) wider. Ein Auramanager übernimmt die Aufgabe, Aurakollisionen gleicher Interfaces verschiedener Objekte festzustellen und die Objekte davon in Kenntnis zu setzen.

Die zentrale Kollisionsfeststellung durch den Auramanager ist als Nachteil von MASSIVE anzusehen, da zentralisierte Funktionalitäten in jedem Fall einen Engpaß darstellen können.

2.5.10 AVIARY

Das AVIARY System wurde von der Communications Research Group am Department of Computer Science der University of Nottingham entwickelt und in /SNO94/ vorgestellt.

AVIARY kann als Sammlung autonomer, miteinander kommunizierender Objekte angesehen werden, die unabhängig voneinander ausgeführt werden. Alle Objekte exportieren ein Interface, mit dessen Hilfe sie in einem gemeinsamen Kommunikationsmedium durch den Austausch von Nachrichten kommunizieren. Solche Objekte können sowohl Gegenstände oder Objekte (**artifacts**) der virtuellen Umgebung sein, als auch Dienste anbieten (z.B. Kollisionsfeststellung).

Das System ermöglicht die Einrichtung von verschiedenen Welten, in denen auch verschiedene Gesetze gültig sein können. Die Kommunikation zwischen AVIARY Objekten geschieht über Nachrichtenaustausch (**message passing**). Es ist möglich, neue Nachrichtentypen zur Laufzeit zu erzeugen.

Die Objekte in AVIARY können von verschiedener Art sein. Es gibt Objekte, die als Applikationen ausgelegt sind, die also ohne weiteres ausführbar sind (**heavyweight objects**) und es gibt Objekte, die – ähnlich den Objekten aus Standard Programmiersprachen – weniger Overhead besitzen, somit leichter (**lightweight objects**) sind und auf eine einfache Art und Weise zwischen Prozessoren hin und her bewegt werden können.

Nutzer werden von AVIARY auf die gleiche logische Ebene gestellt wie Applikationen.

Erkennung von Interaktionen zwischen Objekten (hier Kollisionsfeststellung) erfolgt bei AVIARY über eine Umgebungsdatenbank (**Environment Database**). Diese Datenbank wird von den Objekten über räumliche Änderungen

informiert und informiert ihrerseits die Objekte, wenn eine Kollision festgestellt wurde. Die Objekte müssen dann geeignete Maßnahmen einleiten. In AVIARY wird zur zeitlichen Synchronisation das Konzept einer globalen Zeit (**world-time**) eingeführt.

Schonender Benutzung von Netzwerkressourcen wurde Beachtung geschenkt, indem z.B. nur diejenigen Bestandteile (**artifacts**) der virtuellen Umgebung Synchronisationsereignisse erhalten, die sich in einem bestimmten räumlichen Umfeld um den Betrachter befinden.

2.5.11 NPSNET

NPSNET wurde für militärische Simulationen entwickelt (/MAC94/).

Es benutzt sowohl das IEEE 1278 distributed interactive simulation (**DIS**) application protocol (/DIS/), als auch das IP Multicast network protocol (/LOO91b/). Ein weiterer wesentlicher Punkt des NPSNET Systems ist die Verwendung eines **Dead Reckoning** Verfahrens zur Verminderung der Netzwerkbelastung.

Dead Reckoning bezeichnet eine Methode zur Verringerung der notwendigen Datenübertragung zum Abgleich der Stationen. Um eine flüssige Darstellung der Bewegung eines Objektes zu erhalten, müssen ca. 25 Bilder pro Sekunde dargestellt werden. Wenn Objekte über ein Netzwerk abgeglichen werden müssen, ist es somit notwendig, die Informationen zum Abgleich 25 mal pro Sekunde über das Netzwerk zu übertragen. Dies führt zu einer hohen Belastung des Netzwerks. Außerdem kommt die Bewegung eines Objektes ins Stocken, wenn bei der Übertragung der Abgleichsinformationen Verzögerungen auftreten. Wird Dead Reckoning eingesetzt, so kann die Bewegung eines Objektes anhand der aktuellen Position, der Geschwindigkeit und der Beschleunigung des Objektes bei jedem Teilnehmer berechnet werden. Durch diese lokale Berechnung ist die Flüssigkeit der Darstellung unabhängig vom Eintreffen neuer Objektinformationen. Zusätzlich kann der Zeitraum zwischen zwei Synchronisationsimpulsen erhöht werden. Dadurch erfolgt eine Entlastung des benutzten Übertragungskanal. Für die Berechnung der Bewegung eines Objektes anhand o.g. Attribute existieren verschiedene Lösungen. Neben der Lösung, die in NPSNET benutzt wurde, sei an dieser Stelle noch auf /SIN95/ verwiesen.

Das DIS Protokoll unterstützt und verlangt eine volle Replikation aller Objekte in der virtuellen Umgebung und es verlangt auch, daß die replizierten Objekte immer den aktuellen Status des Originals kennen. Selbst durch den Einsatz von Mechanismen wie Dead Reckoning ist ein Ansatz mit dem DIS Protokoll nicht allgemeingültig anwendbar, da durch die vollständige Replikation aller Objekte sehr schnell die Grenzen der zur Verfügung stehenden Hardware und Netzbandbreite erreicht werden.

2.5.12 RING

RING wurde von den AT&T Bell Laboratories, Murray Hill, New York als Client-Server System für virtuelle Umgebungen mit mehreren gleichzeitig agierenden Nutzern konzipiert(/FUN95/).

Die Kommunikation wird über Server abgewickelt. Dabei werden aber nicht alle Statusänderungen aller Objekte an alle Clients weitergeleitet, sondern es werden nur Nachrichten weitergeleitet, die für den Empfänger wirklich notwendig sind.

Diese Notwendigkeit errechnet der Server über einen Sichtbarkeitsalgorithmus. Clients werden nur über Statusänderungen von Objekten informiert, die für sie selbst sichtbar sind. Es existiert für jedes Objekt (**entity**) genau ein Client, der dieses Objekt verwaltet. Wenn ein Objekt für einen anderen Client interessant wird, dann wird eine Kopie des Objektes angelegt. Die Statusänderungen werden durch Nachrichten (**messages**) übertragen. Die Server fungieren dabei als Filter für Nachrichten.

Problematisch wirkt sich dabei aus, daß jede Kommunikation über den Server abläuft und beim Server die räumlich Aufteilung der gesamten Umgebung bekannt sein muß.

Das RING System benutzt verschiedene Methoden, um die Ressourcen effizient auszunutzen. So wurde z.B. eine Erweiterung vorgeschlagen (**multiresolution update**), die dafür sorgt, daß die Position von Objekten, die räumlich weiter vom Betrachter entfernt sind, mit einer geringeren zeitlichen Auflösung aktualisiert wird als Objekte, die sich dicht beim Betrachter befinden.

2.5.13 MR-Toolkit Peer Package

Das Minimal Reality (MR) Toolkit wurde an der University of Alberta entwickelt (/SHA92/, /SHA93a/).

MR ist eine Programmbibliothek, mit der es möglich ist, Applikationen für virtuelle Umgebungen zu entwickeln. MR basiert dabei auf einer Client/Server Architektur, bei der die Gerätetreiber als Server fungieren. Von diesen Servern rufen die Applikationen dann die benötigten Daten ab. MR stellt eine Softwarebasis für die Entwicklung von Einzelplatz VR-Anwendungen dar.

Peer Package ist eine Erweiterung des MR Systems und wurde an derselben Einrichtung entwickelt (/SHA93b/).

Mit Peer Package ist es möglich, MR-Applikationen miteinander über ein Netzwerk zu verbinden. Dies macht das MR-System zum Multi-User VR System. Die Kommunikation der einzelnen Stationen basiert auf Punkt-zu-Punkt Verbindungen. Die Netzwerktopologie ist dabei ein vollständig konnektierter Graph, d.h. jede Applikation besitzt Verbindungen zu jeder anderen Applikation.

Durch diese Vorgehensweise wächst der Kommunikationsbedarf quadratisch mit der Anzahl der beteiligten Nutzer. MR Peer Package ist deshalb nicht in der Lage, die Probleme großer, verteilter virtueller Umgebungen mit vielen Nutzern zu lösen.

Die Implementation basiert auf UDP, einem verbindungslosen Transportprotokoll /LOO91a/. MR Peer Package besitzt keine Methoden zur Unterbindung der Nachteile von UDP. Diese Nachteile liegen darin, daß der Empfang von

Paketen nicht bestätigt wird. Außerdem kann sich die Reihenfolge der Pakete durch die Übertragung ändern, so daß ein Paket A, welches früher als ein Paket B abgeschickt wurde, später beim Empfänger ankommt, als das Paket B. Deshalb eignet es sich nur für den Einsatz in schnellen LANs, in denen davon ausgegangen werden kann, daß keine Verzögerungen beim Routen der Pakete auftreten und in der Regel alle Pakete ankommen. Als Demonstration wurde ein Multi Player Handball vorgestellt.

Mit einem weiteren Zusatzmodul, dem Umgebungsmanager (**Environment Manager, EM**) werden einige der Probleme in Angriff genommen (/WAN95/). Netzwerkverkehr wird durch verschiedene Methoden (u.a. lokale Simulation, Nachrichtentransfer nur an relevante Empfänger) verringert. Weiterhin werden verteilte Objekte und Zugriffsrechte benutzt. Der EM vereinfacht aber nicht nur die Arbeit mit verteilten Systemen mit mehreren Nutzern, sondern auch Einzelplatzsysteme profitieren von dieser Systemerweiterung.

2.5.14 Waves / Hidra

Die Waves (**Waterloo virtual environment system**) Architektur wurde von Rick Kazman (University of Waterloo, Canada) vorgestellt (/KAZ93a/).

Die Architektur geht davon aus, daß eine virtuelle Umgebung aus einer Anzahl Nachrichtenverwaltern (**message managers**) besteht, die die Kommunikation zwischen Stationen (**hosts**) realisieren.

Die Nachrichtenverwalter leiten Nachrichten von einer Station zu einer anderen und fungieren dabei auch als Nachrichtenfilter. Eine Station kann einem Nachrichtenverwalter explizit mitteilen, daß sie nur bestimmte Nachrichten empfangen will.

Die Stationen sind Prozesse, die **Objoids** simulieren. Objoids ist ein Kunstwort, welches bei Waves für Objekte in der virtuellen Umgebung steht. Ein Objoid besteht aus einem internen Status, exportierten Attributen und einer Anzahl ausführbarer Verhaltensvorschriften, mit denen der interne Status des Objoids dezentral und autonom aktualisiert werden kann. Virtuelle Welten in Waves enthalten nichts außer solchen Objoids. Die Stationen stellen den Objoids Serviceleistungen wie Simulation, Interaktionserkennung und -auflösung (**interaction detection and resolution**) und Visualisierung zur Verfügung.

Objoids können auf einfache Weise von einer Station zu einer anderen übertragen werden, da die gesamte Beschreibung eines Objoids in Parameter überführt werden kann. Diese Parameter können über das Netzwerk an eine andere Station übertragen werden. Dort existiert das Objoid als Kopie (**clone**) weiter. Es ist in der Lage, seinen internen Status eigenständig zu ändern und seinen Zustand mit dem Originalobjoid abzugleichen.

Leider wird in den Quellen nichts über die Implementation und die Parameterisierung der Objoids ausgesagt. Dieser Umstand ist bedauerlich, da gerade die Darstellung der Verhaltensvorschriften als Parameter der Objoids eine komplexe Aufgabe zu sein scheint. Es ist denkbar, daß die Lösung des Problems durch eine Art Interpreter bewerkstelligt wurde, mit dessen Hilfe die

parametrisierten Verhaltensprogramme ausgeführt werden können. Der Autor des Systems, Rick Kazman, teilte auf Anfrage mit, daß das Projekt nicht mehr weiterverfolgt wird und deshalb keine weiteren Informationen verfügbar seien.

Die Verhaltensvorschriften der Objoids enthalten auch explizite Verhaltensmodelle (**explicit behavioral models**). Diese Modelle erlauben es, zukünftiges Verhalten von Objoids vorherzusagen. Damit können z.B. Verzögerungen beim Nachrichtentransfer ausgeglichen werden.

Interaktionserkennung wird in Waves durch separate Stationen durchgeführt. Dabei simulieren diese Stationen eine Anzahl Objoids. Die Interaktionserkennung und -auflösung erfolgt über separate Vorschriften. Die Objoids interagieren also nicht selbst, sondern Interaktionen werden von der Implementation der Objoids getrennt definiert. Dabei besteht eine Interaktionsdefinition aus zwei Teilen: einem Interaktionserkennungsteil und einem Interaktionsauflösungsteil. Die Interaktionserkennung erfolgt zentral, also nur an einer Stelle im Gesamtsystem. Der Engpaß, der sich dadurch ergeben könnte, wird in Waves dadurch gelöst, daß die Erkennung verschiedener Arten von Interaktionen auf verschiedenen Stationen (Interaktionserkennungsstationen) durchgeführt werden kann.

Die Waves Architektur wurde mit dem Prototypen HIDRA (**H**igly interactive **d**istributed **r**eal-time **a**rchitecture) implementiert (/KAZ95/, /KAZ93b/).

Leider ist dieser Prototyp nicht frei verfügbar und das Projekt wird derzeit nicht weiterentwickelt.

2.5.15 Demand Driven Geometry Transmission

Schmalstieg und Gervautz vom Institut für Computergrafik der technischen Universität Wien stellen verschiedene Methoden zur Minimierung der Netzwerkbelastung in verteilten virtuellen Umgebungen vor (/SCH95/, /SCH96/). Ein Prototyp ist als Client/Server System ausgelegt. Der Server hält dabei die Daten der gesamten Umgebung vor. Clients erhalten vom Server die Daten der für sie relevanten Objekte. Die Relevanz wird dabei durch das räumliche Umfeld des Clients festgelegt (**Area of Interest, AOI**).

Hauptaugenmerk legten die Autoren auf die Optimierung des Transports von Geometriedaten. Dabei werden für Objekte Datenstrukturen eingesetzt, die verschiedene Detaillierungsgrade (**Level of Detail, LOD**) enthalten. Wird ein Objekt für einen Client interessant, so wird das Modell des Objektes in der geringsten Auflösung an den Client übertragen. Ein Prefetch-Mechanismus sorgt dafür, daß die Übertragung der Daten vor der eigentlichen Visualisierung abgeschlossen ist. Die AOI um den Client besteht aus verschiedenen Zonen, denen jeweils ein LOD des Objektes zugeordnet ist. Bewegt sich der Client näher an das Objekt heran (bzw. das Objekt an den Betrachter), so werden entsprechend feinere Geometrieinformationen vom Server geladen.

Die Autoren schlagen einen Serververbund vor, um die Skalierbarkeit des Systems zu gewährleisten. Jeder Server ist dabei für einen gewissen Teil der virtuellen Umgebung zuständig.

Es existieren Objekte, die lediglich aus statischen Geometrieinformationen bestehen, Objekte, die zusätzlich statische Animationen beinhalten können und Objekte, deren Verhalten in einer Art Skript festgelegt werden kann.

Dabei werden die Skripts hauptsächlich dazu benutzt, die Reaktion auf asynchrone Ereignisse zu definieren. Bei dieser Art der Objekte wird weiterhin unterschieden, ob das Skript durch den Server oder durch den Client interpretiert werden soll. Weiterhin sind externe Applikationen möglich, die über Clients in die virtuelle Umgebung eingebunden werden können.

Das System von Schmalstieg und Garvautz besitzt den Charakter eines unvollständigen Prototypen. Tests mit vielen Nutzern und großen Umgebungen stehen noch aus und die verwendeten Mechanismen müssen noch optimiert werden.

2.6 Zusammenfassung

Neben der in der Beschreibung der einzelnen Systeme erfolgten Darstellung der Vor- und Nachteile, soll an dieser Stelle nochmals eine zusammenfassende Aufstellung wichtiger Parameter der Systeme erfolgen (Tabelle 2). Dabei wurde die anfänglich durchgeführte Systematisierung zugrunde gelegt. Der Prototypcharakter der meisten Systeme erschwert eine genaue Gegenüberstellung.

Es zeigt sich, daß sich der Einsatz einer Client/Server Architektur nur dann für den Einsatz in verteilten virtuellen Umgebungen eignet, wenn diese Architektur entweder in Verbindung mit Punkt-zu-Punkt Kommunikation eingesetzt wird oder ein hierarchisches Client/Server System aufgebaut wird. Punkt-zu-Punkt Kommunikation stellt einen flexibleren Ansatz dar, ist aber nur in Verbindung mit Methoden zur Netzlastbegrenzung sinnvoll.

System	Kommunikationsmethode	Verteilungsmethode ⁷	Netzlastbegrenzung	max. Nutzeranzahl	Komplexität der Welt
DIVE	Punkt-zu-Punkt	DeD-Vrep	Filtermechanismen	in einer Teilwelt < 10,	durch Einteilung in unabhängige Teilwelten hoch
VEOS	Punkt-zu-Punkt	DeD-Vrep	keine	keine Angaben, wahrscheinlich < 10	durch Kommunikations- und Verteilungsmethode beschränkt
dVS	Punkt-zu-Punkt	ZeD-Vrep	keine	keine Angaben, wahrscheinlich < 10	durch Kommunikations- und Verteilungsmethode beschränkt
VUE	Client/Server	ZeD-RepNB	keine	1 (Verteilung wird zum Zweck der Parallelisierung der Aufgaben eingesetzt)	hoch
Rubber Rocks	Client/Server	ZeD-Vrep	higher-level Events vorgeschlagen	keine Angaben, wahrscheinlich < 10	gering (durch Kommunikations- und Verteilungsmethode beschränkt)
VR-DECK	Punkt-zu-Punkt	ZeD-RepNB	Filtermechanismen (spezielle Regeln)	keine Angaben, wahrscheinlich < 10	mittel
BrickNet	Client/Server	ZeD-RepNB	Filtermechanismen	keine Angaben, wahrscheinlich < 10	mittel (Stationen verwalten nur benötigte Daten)
NVR	Client/Server	ZeD-Vrep	verzögerter Abgleich bei Überlastung, lokale Objekte	keine Angaben, wahrscheinlich < 10	durch Einteilung in unabhängige Teilwelten hoch
MASSIVE	hybrid	ZeD-Vrep	spatial model of interaction	keine Angaben, wahrscheinlich < 20	durch Einteilung in unabhängige Teilwelten hoch
AVIARY	hybrid	ZeD-RepNB	Filtermechanismen	keine genauen Angaben, wahrscheinlich < 10	keine genauen Angaben, durch Prototypcharakter wahrscheinlich gering
NPSNET	Punkt-zu-Punkt	DeD-Vrep	Dead Reckoning, Multicast	max. 300 (10 Mbit Ethernet LAN)	hoch (Datenmodell bei allen Stationen initial vorhanden)
RING	Client/Server	ZeD-RepNB	bedingte Nachrichtenweiterleitung	bis zu 1024	hoch (durch Aufteilung in dynamische Sichtbarkeitsräume)
MR	Punkt-zu-Punkt	DeD-Vrep	Erweiterungen (EM) vorgesehen.	keine Angaben, wahrscheinlich < 10	gering (durch Kommunikations- und Verteilungsmethode beschränkt)
Waves / Hidra	hybrid	DeD-RepNB	Filtermechanismen Abgleich nach Bedarf	keine Angaben, wahrscheinlich < 100	hoch
Demand Driven Geometry Transmission	hybrid	ZeD-RepNB	bedingte Nachrichtenweiterleitung	keine Angaben (Prototyp)	hoch (durch Art der Verteilung)

Tabelle 2: Systematisierung der untersuchten Systeme

⁷ vgl. Abschnitt 2.2 Verteilungsmechanismen (ab Seite 14).

*Am Ende des Tunnels wartet das Licht
aber hier draußen gibt's kein zurück ...*

*Bleib wo Du bist
aus: „Ton Steine Scherben IV“
Ton Steine Scherben, (1981)*

3 Lösungsansatz

In diesem Abschnitt wird die Beschreibung eines speziellen Lösungsansatzes für eine verteilte virtuelle Umgebung erfolgen.

Eine verteilte virtuelle Umgebung, wie sie das Voodie System unterstützen soll, läßt sich in verschiedene Problemkategorien einteilen:

Objektdefinition

Für die Konstruktion von virtuellen Umgebungen ist die Definition von Objekten⁸ und deren Verhalten die wesentliche Aufgabe. Wird die Objektdefinition vom eigentlichen Kern getrennt, lassen sich die Objekte einfacher und flexibler erstellen und testen. Im Prototyp besitzen die Objekte nur

⁸ Ein Objekt ist eine autonome, dreidimensional darstellbare Entität der virtuellen Umgebung.

minimales Verhalten. Dieses Verhalten wird durch die Implementation der jeweiligen Objektklasse festgelegt.

Objektkommunikation

Die Kommunikation zwischen einzelnen replizierten Objekten ist die wesentliche Grundlage für den Aufbau einer verteilten virtuellen Umgebung. In dieser Arbeit werden entsprechende Mechanismen vorgestellt.

Objektinteraktion

Eine wesentliche Aufgabe des Systems wird die Feststellung und Auflösung von Objektinteraktionen sein, da erst die Interaktionen zwischen Objekten eine virtuelle Umgebung wirklich nutzbar machen. Es werden grundsätzliche Ansätze zur Lösung des Interaktionsproblems vorgestellt.

Objekttransport

Der Transport von Objekten von einer Station zu einer anderen muß gelöst werden, damit die Mechanismen zur dezentralen Objektverwaltung effizient einsetzbar sind. Die Möglichkeit des Objekttransportes muß bei der Art der Objektdefinition beachtet werden.

Verteilungsstrategie

Die Bestandsanalyse hat gezeigt, daß die Netzwerkbelastung ein nicht zu unterschätzender Faktor für verteilte virtuelle Umgebungen ist. Neben Mechanismen zur Objektkommunikation müssen deshalb auch Überlegungen zur Verteilung der Informationen und der Rechenlast auf die beteiligten Stationen angestellt werden.

Ein- und Ausgabe

Die Visualisierung und die Anbindung von VR spezifischen Eingabegeräten (z.B. System zur Positionsfeststellung – *Trackingsystem*, 6 *DOF*⁹ Eingabegeräte) ist eine weitere, für die Akzeptanz des Systems, sehr wesentliche Aufgabe.

In dieser Arbeit können bei weitem nicht alle Mechanismen, die für ein solches System notwendig sind, bis ins letzte Detail vorgedacht, geschweige denn implementiert werden. Deshalb sind viele der beschriebenen Ideen lediglich als Lösungsvorschläge zu betrachten, die als Basis für weitere Forschungstätigkeit dienen können.

In Abbildung 2 sind die einzelnen Problemkategorien zusammenfaßt. Dabei sind die Aspekte, die im Prototyp implementiert wurden, dunkel dargestellt.

⁹ DOF steht für degrees of freedom und bezeichnet die Freiheitsgrade die ein Eingabegerät besitzt. Ein Eingabegerät besitzt 6 DOF wenn Bewegungen in jede Raumrichtung und Drehung um die drei Achsen möglich sind.

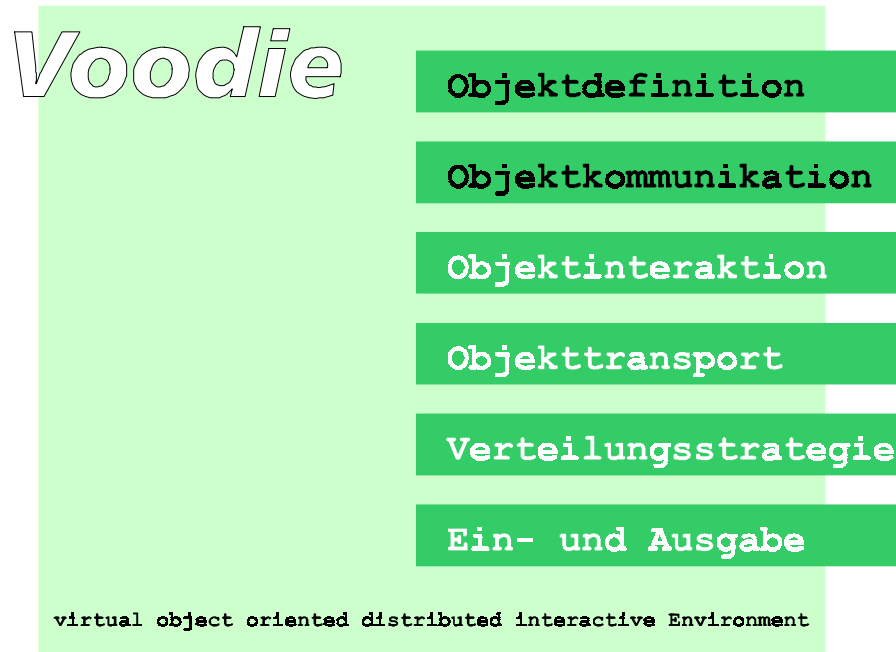


Abbildung 2: Übersicht über das Voodie System

3.1 Unabhängige Welten

Der Begriff Welt wird als Synonym für die virtuelle Umgebung eines Teilnehmers benutzt. Ein Teilnehmer sieht seine Umgebung als (seine virtuelle) Welt.

zentrale
Objektver-
waltung

Die Systeme, bei denen eine zentrale Verwaltung der virtuellen Umgebung vorgenommen wird, haben das Problem, daß sich jeder Teilnehmer mit dieser zentralen Instanz verbinden muß, um Daten mit ihr auszutauschen.

Bei einer zentralen Objektverwaltung muß an irgendeiner Stelle etwas wie eine zentrale Datenbank vorhanden sein, in der die Informationen der Welt unabhängig von den Betrachtern aufbewahrt und ggf. auch bearbeitet (Objekte mit zeitlich veränderten Eigenschaften, Simulationen) werden.

Problematisch ist natürlich, wie die Teilnehmer und die zentrale Datenbank einerseits nur minimal miteinander kommunizieren sollen, andererseits aber auch alle Teilnehmer ein einheitliches, synchrones und konsistentes Bild der Welt (oder des Teiles der Welt, der für sie relevant ist) haben sollen. Dies führt in jedem Fall zu einer prinzipiellen Begrenzung des Systems durch die zentrale Instanz.

dezentrale
Objektver-
waltung

Um die Skalierbarkeit einer verteilten virtuellen Umgebung zu gewährleisten, sollte neben dem Einsatz von Methoden zur Verringerung der Netzwerklast und des Kommunikationsaufwandes, vor allem über die dezentrale Verwaltung der Umgebung nachgedacht werden. Jeder zentralisierte Ansatz beinhaltet immer die Gefahr der Entstehung eines Engpasses. Demgegenüber besteht bei dezentralen Ansätzen das Problem der Synchronisation und der Konsistenz der gemeinsamen Umgebung.

Trotzdem ist ein dezentraler Ansatz bei der Objektverwaltung besser geeignet, die Anforderungen an verteilte virtuelle Umgebungen zu erfüllen, da synchronisierte, dezentrale Datenhaltung zwar komplizierter umzusetzen ist, in jedem Fall aber besser skaliert als eine zentrale Verwaltung der gesamten virtuellen Umgebung.

Deshalb wird folgende Lösung vorgeschlagen:

Für jeden Teilnehmer existiert eine eigene virtuelle Umgebung (eine eigene Welt), die prinzipiell unabhängig von den Welten anderer Teilnehmer ist. Jeder Teilnehmer besitzt eine eigene Objektverwaltung, die direkt an seinen **Avatar**¹⁰ gekoppelt ist. Diese Objektverwaltungsinstanz ist für die Verwaltung der Objekte zuständig, die sich in einem gewissen räumlichen Einzugsbereich um den Avatar des Teilnehmers befinden. Diese Vorgehensweise begründet sich auf der Annahme, daß nur die Objekte für einen Teilnehmer relevant sind, die sich eben in einer gewissen endlichen Entfernung um ihn herum befinden (vgl. AVIARY, ab Seite 25 und /SNO94/), er diese Objekte also sehen kann: Der Begriff „Sehen“ kann in diesem Zusammenhang als spezielle Interaktion verstanden werden. Greenhalg und Benford beschreiben in MASSIVE (/GRE95/) ein räumliches Interaktionsmodell, bei dem davon ausgegangen wird, daß jede Art von Interaktion an räumliche Nähe gekoppelt sein muß. Die Idee eines räumlichen Einzugsgebietes ist also auch auf andere Interaktionsformen anwendbar.

Wenn sich nun die räumlichen Einzugsbereiche zweier Teilnehmer überlappen, besteht die Möglichkeit, daß ein Objekt von den zwei Teilnehmern gemeinsam genutzt wird, d.h. in den Welten beider Teilnehmer synchron existiert. Das bedeutet, beide Teilnehmer können mit diesem Objekt interagieren. Gleichermaßen wird durch solche Objekte die Kommunikation zwischen den Teilnehmern möglich.

Praktisch wird dabei das Objekt von einem der Teilnehmer generiert, der andere Teilnehmer abonniert das Objekt. Durch ein solches Abonnement wird eine Instanz des Objektes in der Welt des Abonnenten erzeugt. Diese Instanz kommuniziert selbständig mit der Hauptinstanz des Objektes und synchronisiert sich dadurch mit dieser.

Die Objektverwaltung jedes Teilnehmers behandelt dabei alle Objekte (lokale und abonnierte) gleich. Sie generiert Nachrichten und Ereignisse, mit denen die Objekte über Interaktionen (z.B. Kollisionen) mit anderen Objekten informiert werden.

Abbildung 3 und Abbildung 4 sollen das Prinzip der unabhängigen Welten verdeutlichen. In Abbildung 3 ist die Sicht auf die gesamte virtuelle Umgebung dargestellt. Für keinen der Teilnehmer A, B, C und D ist diese Gesamtsicht auf die virtuelle Umgebung möglich.

¹⁰ Avatare sind spezielle Objekte der virtuellen Umgebung, die zur Repräsentation des Nutzer in der virtuellen Umgebung benutzt werden.

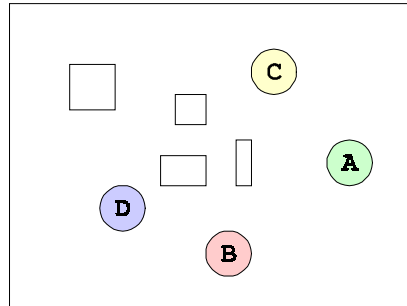


Abbildung 3: Sicht auf die gesamte virtuelle Umgebung

Abbildung 4 zeigt die Erscheinungen der virtuellen Umgebung für die einzelnen Teilnehmer. Für jeden Teilnehmer ist lediglich ein gewisser Teil der gesamten Umgebung sichtbar.

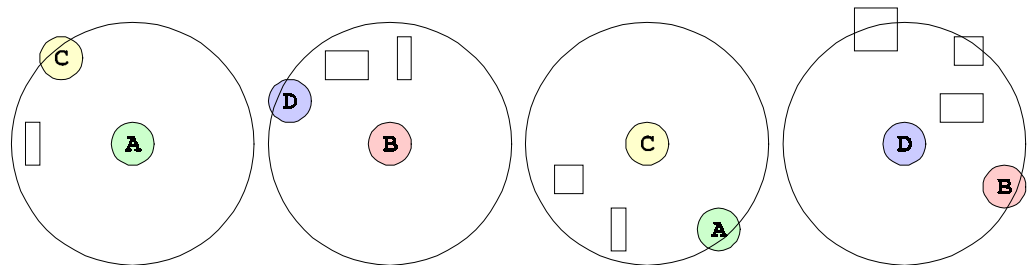


Abbildung 4: Die Welten, wie sie sich für die einzelnen Teilnehmer (A-D) darstellen

In Abbildung 5 wird die entsprechende Kommunikationstopologie, die zwischen den Teilnehmern besteht, dargestellt. Es ist erkennbar, daß z.B. zwischen den Teilnehmern A und D keine Verbindung besteht, da diese Teilnehmer sich nicht sehen können. Die Topologie kann sich aber jederzeit ändern, z.B. wenn sich der Teilnehmer A in den Sichtbereich des Teilnehmers D hinein bewegt. Dann muß auch eine Verbindung zwischen den Teilnehmern A und D hergestellt werden.

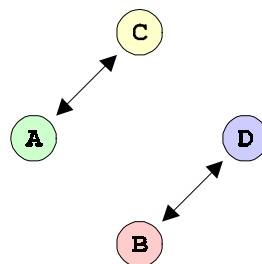


Abbildung 5: Kommunikationstopologie für die dargestellte virtuelle Umgebung

Wichtig ist bei diesem Ansatz eine Definition der Position der Teilnehmer. Diese Definition muß global sein. Die Positionen der Teilnehmer (und ihrer Welten) muß aber nicht unbedingt an einer zentralen Stelle verwaltet werden. Jeder Teilnehmer könnte sich z.B. im Mittelpunkt seines eigenen (lokalen) Ko-

ordinatensystems befinden und die Objekte anderer Teilnehmer in sein eigenes Koordinatensystem einbauen (also die Positionen entsprechend transformieren).

Kommt ein globales Koordinatensystem zum Einsatz, so sind die räumlichen Relationen der Teilnehmer zueinander festgelegt. Eine zentrale Verwaltung wird aber auch in diesem Fall überflüssig, wenn Situationen, in denen Objekte verschiedener Teilnehmer denselben Platz im virtuellen Raum einnehmen, durch geeignete Maßnahmen (z.B. Transformationsvorschriften) verhindert werden.

3.2 Autonome Objekte

Sämtliche Bestandteile einer virtuelle Umgebungen können als autonome, dreidimensional darstellbare Entitäten angesehen werden. Diese Entitäten werden auch als Objekte der virtuellen Umgebung bezeichnet.

Prinzipiell ist es in jeder verteilten virtuellen Umgebung notwendig, die Daten der beteiligten Stationen zu synchronisieren. Die implementierten Kommunikationsmechanismen gehen von der Grundidee aus, daß ein Objekt, das von mehreren Stationen gemeinsam benutzt werden soll, auf jeder dieser Stationen als separates, eigenständiges Objekt vorhanden ist. Jede eigenständige Welt besteht aus solchen autonomen Objekten. Diese Objekte können selbständig ihre Eigenschaften ändern. Dies geschieht durch Mechanismen, die objektspezifisch sind (Simulationsobjekte).

Die Kommunikation zwischen einzelnen Objekten erfolgt durch das Senden und Empfangen von Ereignissen. Ein Ereignis (Event) beinhaltet eine bestimmte Information, die von einem Absender an einen Empfänger geschickt wird.

Objektgruppen
Objekte verschiedener Stationen können miteinander verbunden sein, sich also synchronisieren. Prinzipiell erfolgt die Synchronisation des gesamten Systems über solche verbundenen Objekte. Ein gemeinsam benutztes Objekt ist im Voodie System als Gruppe verteilter, voneinander abhängiger Objekte anzusehen, die ihre Erscheinung selbst bestimmen, aber durch eine (für diese Gruppe) zentrale Instanz synchronisiert werden. Im folgenden wird jede Menge solcherart verbundener Objekte als *Objektgruppe* bezeichnet.

Abonnementmechanismus
Um verbundene Objekte bzw. Objektgruppen einsetzen zu können, wird ein *Abonnementmechanismus* eingeführt.
Die Synchronisation innerhalb einer Objektgruppe wird über den Austausch von Synchronisationsereignissen¹¹ erreicht. In jeder Gruppe abhängiger Objekte existiert genau ein *Masterobjekt*. Dieses Objekt besitzt die Kontrolle über die Objektgruppe, d.h. über dieses Objekt erfolgt die Synchronisation. Alle anderen Objekte der Gruppe sind *Slaveobjekte*. Das Masterobjekt schickt bei Bedarf an alle registrierten Slaveobjekte Synchronisationsereignisse.

¹¹ Im Prototyp werden Nachrichten in Form von Ereignissen ausgetauscht. Das hier angesprochene Synchronisationsereignis taucht in der Prototypimplementierung als **UPDATE** Event auf.

Die Slaveobjekte können auch als Abonnenten des Masterobjektes angesehen werden.

Der Mechanismus sieht weiterhin die Möglichkeit vor, daß ein Abonnent selbst wieder Abonnenten besitzen darf. Dadurch ist prinzipiell jedes Objekt abonnierbar, egal ob dieses Objekt Masterobjekt ist oder nicht. Ein Slaveobjekt kann also seine Synchronisationsevents direkt vom Masterobjekt der Gruppe beziehen, oder aber es erhält seine Synchronisationsevents von einem anderen Slaveobjekt. Das Objekt welches die Synchronisationsereignisse sendet, wird dabei als *Abomasterobjekt* bezeichnet.

Abonnementketten

Durch diese Erweiterung des Mechanismus können *Abonnementketten* aufgebaut werden. In solchen Abonnementketten sendet ein Masterobjekt Synchronisationsereignisse an ein Slaveobjekt. Dieses Slaveobjekt sendet das Synchronisationsereignis an seine Abonnenten, diese ggf. an ihre Abonnenten usw. Der Mechanismus stellt dabei sicher, daß am Anfang jeder Abonnementkette das Masterobjekt der Gruppe steht.

Abbildung 6 zeigt Master-, Slave-, Abomaster- und Abonnentobjekte nochmals im Zusammenhang. In der Abbildung sind weiterhin Abonnementketten und direkte Abonnierung zu sehen. Wesentlich ist bei der Abbildung, daß die Objekte eine Objektgruppe bilden, also alle eingezeichneten Objekte das gleiche Objekt auf verschiedenen Stationen darstellen. Im Abschnitt 3.3 (ab Seite 43) werden die Vor- und Nachteile des Einsatzes von Abonnementketten bzw. direkter Abonnierung näher beschrieben.

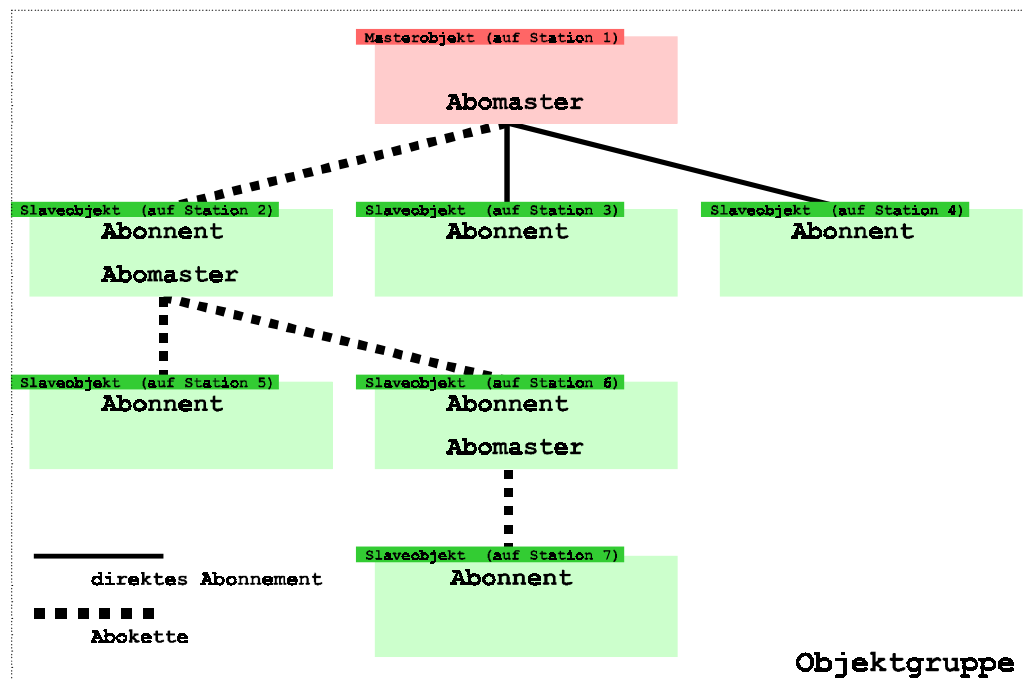


Abbildung 6: Zusammenhang zwischen Masterobjekt und Slaveobjekten bzw. Abomaster- und Abonnentobjekten.

Die Abhängigkeit der Slaveobjekte vom Masterobjekt kann von sehr eng bis sehr lose variieren. Eine sehr enge Abhängigkeit wäre z.B. gegeben, wenn das

Masterobjekt jede Änderung seiner Attribute an die Slaveobjekte weitergibt. Eine lose Abhängigkeit tritt dann auf, wenn die Slaveobjekte ihr Verhalten durch Verhaltensvorschriften autonom bestimmen können und das Masterobjekt nur in Ausnahmefällen Synchronisationsereignisse senden muß.

Die Festlegung, wie häufig Synchronisationsereignisse auftreten, hängt von der Art des Objektes ab und ist Bestandteil des Objektverhaltens.

3.2.1 Objektinformigramme

Ein Problem beim Ansatz der verbundenen Objekte ist der Austausch von Informationen über die Existenz und Lokalität von Objekten in anderen Welten.

Definition
der OI

Deshalb werden *Objektinformigramme (OI)* eingeführt. Solche OI beschreiben, ähnlich wie ein URI (/LEE94a/), die Position eines Objektes im Netzwerk. Neben dieser netzwerkbezogenen Information enthält ein OI auch andere allgemeine, statische Daten. OI könnten auch als kleine statische Varianten eines – von der Informationsmenge her gesehen – größeren Objektes angesehen werden. Die OI können nun sehr einfach und schnell von einer Station an eine andere übertragen werden. Damit ist es möglich, daß ein Teilnehmer gewissermaßen eine Vorschau auf die Welt eines anderen Teilnehmers erhält.

Ein OI enthält Informationen über die Maschine auf der sich die Hauptkomponente des Objektes befindet und eine Objektidentifizierung (analog URI). Weiterhin beinhaltet es Informationen zur Art des Objektes (Typ) und es sollte mindestens eine allgemeine, statische Geometriebeschreibung (Form, Größe, Position) enthalten sein.

3.2.2 Objektabonnement

Es ist nun folgendes Szenario vorstellbar:

Ein Teilnehmer kennt lediglich die Adresse einer zweiten Station im Netzwerk, die Objekte zur gemeinsamen Nutzung zur Verfügung stellt. Die rufende Station stellt nun eine Verbindung her und fordert in einem ersten Schritt die OI aller freigegebenen (abonnierbaren) Objekte an.

Die OI werden auf jeder Station von einem eigenständigen Modul, dem OI-Verwalter bzw. *OI-Objekt* verwaltet.

Nachdem die OI übertragen wurden, kann in einem zweiten Schritt die rufende Station nun Abonnentin eines oder mehrerer Objekte werden. Ein Abonnent erzeugt auf seiner lokalen Station Kopien der entsprechenden Objekte. Diese Kopien tauschen mit einer Hauptinstanz (gewissermaßen das Original, das Masterobjekt) Daten zur Synchronisation aus. Diese Kopien könnten auch als entfernte oder abonnierte Komponenten des Masterobjektes bzw. als Slaveobjekte bezeichnet werden (vgl. Clones in Waves / Hidra, ab Seite 28).

Der Abonnementmechanismus wird im Rahmen der Beschreibung der Prototypimplementierung noch detaillierter vorgestellt werden (Abschnitt 4.6, ab Seite 59).

3.2.3 Übergang der Objektkontrolle

Natürlich ist die starre Aufteilung in Haupt- und Abonnementkomponente nachteilig. Die Grundidee bei der Einführung der dezentralen Objektverwaltung war, daß auf jeder Station nur die Objekte vorhanden sind, die zur Visualisierung der unmittelbaren Umgebung des Teilnehmers notwendig sind. Bewegt sich ein Objekt aus dem Sichtbereich eines Teilnehmers, so sollte es gelöscht werden. Das Objekt kann aber nicht gelöscht werden, wenn es das Masterobjekt einer Objektgruppe ist, da sonst keine Synchronisation mehr erfolgen würde.

Die Festlegung, wer in einer Objektgruppe Master- bzw. Slaveobjekt ist, wird deshalb nicht statisch getroffen, sondern ist dynamisch änderbar. Die Kontrolle über ein Objekt muß vom jeweiligen Masterobjekt an ein entferntes Slaveobjekt übertragen werden können. Um diese Übertragbarkeit zu erreichen, beinhaltet jedes Objekt die Funktionalitäten sowohl des Master- als auch des Slaveobjektes. Wenn zwei Teilnehmer ein Objekt gemeinsam benutzen, dann existiert dieses Objekt also bei beiden Teilnehmern in der gleichen Art und Weise.

Durch die Identität von Master und Slaveobjekten ist es möglich, die Übergabe der Objektkontrolle durch den Austausch von Ereignissen zu realisieren. Wenn das Masterobjekt gelöscht werden soll, übergibt dieses Objekt die Objektkontrolle an einen seiner Abonnenten und sendet entsprechende Nachrichten¹² an alle anderen Abonnenten. Die Abonnenten stellen dann eigenständig eine Verbindung mit dem neuen Masterobjekt her.

Es sind allerdings auch Fälle denkbar, in denen die Objektkontrolle nicht übertragen werden kann¹³. Die Implementierung der Objekte muß diese Fälle entsprechend behandeln.

Wenn zusätzlich alle Abonnenten Kenntnis voneinander hätten, wäre auch ohne eine explizite Nachricht des Masterobjektes ein Übergang der Koordination realisierbar. Ist die Kommunikation mit dem Hauptobjekt nicht mehr möglich, so übernimmt das nächstjüngere Objekt diese Aufgabe. Diese Vorgehensweise bringt aber zusätzlichen Kommunikations- und Verwaltungsaufwand mit sich, da alle Mitglieder einer Objektgruppe die Objekte der gesamten Objektgruppe kennen müssen. In der jetzigen Implementierung ist diese Möglichkeit deshalb nicht vorgesehen. Ein Objekt kennt lediglich seine Abonnenten und sein Abomasterobjekt. Die Slaveobjekte besitzen keine Informationen darüber, welches Objekt das Masterobjekt der Objektgruppe ist.

Es existieren verschiedene Gründe, den Übergang der Objektkontrolle von einem Masterobjekt zu einem Slaveobjekt zu ermöglichen:

¹² Eine solche Nachricht ist im Prototyp als ein `MASTER_ID = <id>` Event implementiert.

¹³ Eine Übernahme der Kontrolle ist z.B. dann nicht möglich, wenn das Objekt an ein bestimmtes Eingabegerät an einer bestimmten Station gekoppelt ist (z.B. Avatar an Headtrackingsystem)

- [G1] Das Masterobjekt sollte die Kontrolle der Objektgruppe an ein Slaveobjekt abgeben, wenn es sich nicht mehr im unmittelbaren Einzugsbereich einer Station befindet.

Es sind durchaus bewegliche Objekte denkbar, die sich selbständig durch die virtuelle Umgebung bewegen. Hier ist leicht einzusehen, daß die Objektkontrolle sinnvollerweise an ein anderes Objekt der Objektgruppe übergeben werden sollte, wenn sich das Masterobjekt sehr weit von der Station entfernt hat, auf der es residiert und deshalb dort nicht mehr sichtbar ist. Ein solches, unsichtbares Objekt muß von dieser Station nicht mehr verwaltet werden. Hat das Masterobjekt die Objektkontrolle abgegeben, wird es zu einem Slaveobjekt und kann aus der Welt dieses Teilnehmers gelöscht werden. In den Welten anderer Teilnehmer existiert dieses Objekt aber weiterhin.

An dieser Stelle tritt das Problem auf, daß Fälle konstruierbar sind, in denen ein Objekt für keinen der Teilnehmer sichtbar ist. Die Objektkontrolle kann in diesem Fall nicht übergeben werden, da keine weiteren Slaveobjekte vorhanden sind. Solche Masterobjekte residieren dann solange auf ihrer aktuellen Station weiter, bis sie in den Sichtbarkeitsbereich einer anderen Station eintreten. Diese Station abonniert dann dieses Objekt. Nun kann Objektkontrolle übertragen werden und das Objekt wird aus der Welt des Teilnehmers, für den es unsichtbar ist, entfernt.

- [G2] Das kontrollierende Objekt sollte die Kontrolle abgeben, wenn es gelöscht werden soll.

Diese Forderung ist dann sinnvoll, wenn ein Objekt nicht vollständig aus der virtuellen Umgebung entfernt werden soll, sondern lediglich von einer der beteiligten Stationen nicht mehr benötigt wird, bzw. diese Station aus der virtuellen Umgebung ausscheidet.

- [G3] Kontrollierendes Objekt sollte immer dasjenige Objekt sein, welches den meisten Interaktionen mit anderen Objekten ausgesetzt ist.

Es ist abzusehen, daß die Interaktion zwischen verschiedenen Objekten durch die jeweiligen Masterobjekte, bzw. die Stationen auf denen das Masterobjekt residiert, realisiert wird. Slaveobjekte geben Interaktionswünsche an ihr jeweiliges Masterobjekt weiter (ggf. über verschiedene Abomasterobjekte). Das Masterobjekt einer Objektgruppe sollte deshalb bei dem Teilnehmer residieren, in dessen Sichtbereich die meisten Objekte vorhanden sind. Es ist wahrscheinlich, daß bei diesem Teilnehmer die meisten Interaktionen auftreten werden. Dadurch kann der bei der Interaktion anfallende Kommunikationsaufwand und die sich ergebende Verzögerung gering gehalten werden.

- Erweiterung
der OI
(Redirection) Das Konzept der OI muß mit der Einführung des Übergangs der Objektkontrolle auch entsprechend erweitert werden. Wenn ein Teilnehmer mit Hilfe eines OI auf ein Objekt zugreifen will, wird eine Verbindung zur entsprechenden Station aufgebaut. Der Interaktionsdämon des Teilnehmers (der Maschine), auf den der OI verweist, leitet die Anforderung an das entsprechende Objekt weiter. Wenn das Objekt nicht mehr existiert – weil es sich z.B. aus dem Sichtbarkeitsbereich des Teilnehmers heraus bewegt hat und deshalb gelöscht wurde – muß eine Art Redirection-Mechanismus angewandt werden. Der Interaktions-

dämon hält zu diesem Zweck über die Lebenszeit aller Objekte hinaus Informationen darüber bereit, wo das Objekt (oder besser ein Objekt der entsprechenden Objektgruppe) zu finden ist. Mit „alle Objekte“ sind hierbei die Objekte gemeint, von denen zumindest einmal ein OI gesendet wurde. Dieser Service wird als dynamische Liste von Objektinformigrammen ausgelegt, da die OI alle notwendigen Informationen enthalten. Ein Redirect könnte dann durch das Senden eines aktualisierten OI erfolgen. Der Service sollte zeitlich begrenzt sein.

Wenn ein entfernter Teilnehmer nun ein Objekt abonnieren will, das sich nicht mehr auf dieser Maschine befindet, kann durch den Redirection-Mechanismus ein entsprechend aktualisierter Verweis zurückgesandt werden. Das Slaveobjekt, welches das Abonnement gewünscht hat, sollte nun in der Lage sein, ein entsprechendes Objekt zu finden. Wenn keine Informationen zu einem Objekt vorliegen, muß ein entsprechendes Fehlerereignis generiert werden bzw. das Slaveobjekt, von dem der Abonnementimpuls ausging, reagiert nach einer gewissen Zeit eigenständig¹⁴.

3.2.4 Objekttransport

Die Erzeugung eines Objektes auf einer Voodie Station kann als Transport dieses Objektes über das Netzwerk angesehen werden. Für diesen Transport sind zwei Wege denkbar:

- [A1] Es werden vorgefertigte Bibliotheken mit Standardobjekten benutzt.
- [A2] Es wird Objektcode¹⁵ von einer Station zu einer anderen übertragen.

Die Bibliotheksvariante hat den Vorteil, daß lediglich der Typ des Objektes und ggf. Initialisierungsparameter übertragen werden müssen. Nachteilig ist, daß diese Art der Objekterzeugung voraussetzt, daß auf allen Stationen die entsprechenden Objektbibliotheken vorhanden sind.

Die zweite Variante ermöglicht ein sehr flexibles System, welches unproblematisch erweiterbar ist.

Interpretierte Objekte haben weitere Vorteile:

- [B1] Umgebungs- und Objektdaten werden von den Verwaltungsmodulen getrennt.
- [B2] Objekte unbekannter Klassen können zur Laufzeit generiert werden, da der Transport zur Laufzeit interpretierbarer Daten über das Netzwerk sehr einfach möglich ist.

¹⁴ Das Slaveobjekt, welches die Abonniierung ausgelöst hat, erwartet nach dem Senden des entsprechenden Abonnement Ereignisses (**SUBSCRIBE** Event), daß das Abomasterobjekt Synchronisationsimpulse sendet. Wenn das Abomasterobjekt auf einer anderen Station zu finden ist und der Redirection Mechanismus zum tragen kommt, sollte in endlicher Zeit ein entsprechendes Ereignis beim Slaveobjekt ankommen. Wenn kein Event beim Slaveobjekt ankommt, so kann das Objekt davon ausgehen, daß der Abomaster nicht mehr existent ist. und sich selbst löschen.

¹⁵ Objektcode steht in diesem Zusammenhang für die Instruktionen, die ausgeführt werden, um das Verhalten eines Objektes zu bestimmen.

Zur Umsetzung wird eine Möglichkeit benötigt, Voodie Objekte geeignet zu beschreiben.

Diese Objektbeschreibung muß nicht nur die Attribute eines Voodie Objektes beinhalten, sondern vor allen Dingen eine Beschreibung des Objektverhaltens und die Funktionalitäten zur Eventbehandlung. Erst dann ist eine wirklich flexible Arbeit mit dem System möglich. Als Lösungsmöglichkeiten für dieses Problem sollten folgende Wege in Betracht gezogen werden:

[C1] Objektdefinition in einer interpretierten Programmiersprache¹⁶.

Vorteil: Interpretierte Programmiersprachen sind als bekannte und erprobte Programmiersprachen existent. Die Integration eines Interpreters sollte relativ einfach möglich sein.

Nachteil: Es müssen wahrscheinlich systemspezifische Erweiterungen definiert werden, die eine effiziente Implementation der Objekte ermöglicht.

[C2] Objektdefinition in einer eigens zu entwickelnden Beschreibungssprache.

Vorteil: Die Beschreibungssprache kann so definiert werden, daß sie den Anforderungen des Voodie Systems optimal genügt.

Nachteil: Der Aufwand, eine Beschreibungssprache und einen entsprechenden Interpreter zu entwickeln, ist relativ hoch.

3.2.5 Zusammenfassung

Die verteilte virtuelle Umgebung besteht aus Objekten. Diese Objekte sind einzelnen Stationen zugeordnet. Nutzer treten in die Umgebung über Stationen ein. Jede Station kann lediglich die Objekte der verteilten Umgebung darstellen, die aktuell auf dieser Station vorhanden sind. Objekte können repliziert werden. Die Replikation erfolgt über einen Abonnementmechanismus. Bedingte Abonnieung kann mit Hilfe der Objektinformigramme realisiert werden. Replizierte Objekte bilden eine Objektgruppe. Dabei existiert immer ein Objekt, welches die Objektkontrolle besitzt. Replizierte Objekte müssen sich synchronisieren. Die Synchronisation ist Aufgabe der Objekte, d.h. ein Objekt entscheidet selbst, ob und wieviel Synchronisationsnachrichten (**UPDATE** Events) es an andere (replizierte) Objekte sendet.

3.3 Kommunikationsprinzip

Die Kommunikation im System erfolgt auf der Basis von Punkt-zu-Punkt Kommunikation zwischen den Stationen bzw. zwischen den Objekten. Diese Art der Kommunikation wird verwendet, weil sie am besten für ein dezentral organisiertes System geeignet ist. Wie bereits beschrieben, ist in diesem Fall aber eine Beschränkung der Anzahl der Punkt-zu-Punkt Verbindungen notwendig. Im Folgenden soll ein Ansatz vorgestellt werden, der eine solche Begrenzung ermöglicht.

¹⁶ z.B. Java (/LIN96/), Starship (/LOO91/), EXPRESS (/EXPR/) oder andere.

Event-routing Wird eine Umgebung betrachtet, die aus n Stationen besteht, jede dieser Stationen jeweils ein Objekt verwaltet und alle Objekte auf allen anderen Stationen als Replikanten (Slaveobjekte) vorliegen, so können die folgenden Extremszenarien unterschieden werden.

Fall A: Wenn jedes Slaveobjekt seine Synchronisationsimpulse direkt vom Masterobjekt erhält, sind pro Synchronisationstakt von jeder Station $(n-1)$ Synchronisationsereignisse an $(n-1)$ andere Stationen zu schicken. Insgesamt existieren also $\frac{1}{2}n(n-1)$ Punkt-zu-Punkt Verbindungen zwischen den n Stationen. Abbildung 7 zeigt die entsprechende Verbindungstopologie.

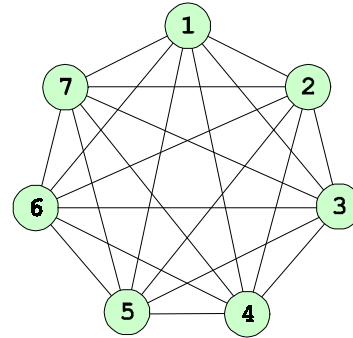


Abbildung 7:
Verbindungstopologie im Fall A

Fall B: Wenn jedes Masterobjekt nur ein Slaveobjekt mit Synchronisationsereignissen versorgt, bestehen im günstigsten Fall lediglich $(n-1)$ Punkt-zu-Punkt Verbindungen zwischen den Stationen. Dieser Fall wird dann erreicht, wenn jede Station nur Verbindungen zu maximal zwei anderen Stationen aufbaut. In diesem Fall werden die Synchronisationsereignisse über eine gewisse Anzahl von Stationen geroutet (Abonnementkette). Die Anzahl der Zwischenstationen ist dabei maximal $(n-2)$. Abbildung 8 stellt die Verbindungstopologie für diesen Fall dar.

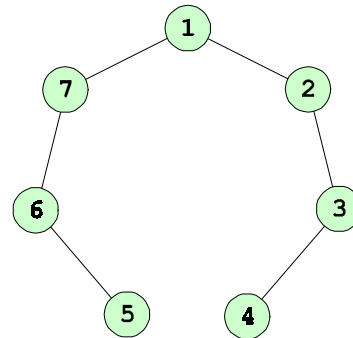


Abbildung 8:
Verbindungstopologie im Fall B

Wird davon ausgegangen, daß der maßgebliche Aufwand bei der Versendung der Synchronisationsereignisse darin liegt, die Verbindung herzustellen, und daß es weiterhin möglich ist, mehrere Synchronisationsereignisse über eine bestehende Verbindung abzusenden, bedeutet das Szenario im Fall B eine wesentliche Schonung der Netzwerkressourcen. Problematisch ist bei Fall B allerdings, daß die Synchronisationsereignisse u.U. über $(n-2)$ Stationen geroutet werden müssen. Die Laufzeit der Ereignisse erhöht sich dadurch direkt proportional zur Anzahl der Zwischenstationen. Dies führt natürlich auch zu einer Begrenzung der beteiligten Stationen.

Praktisch bietet sich deshalb eine Lösung an, die die Vorteile beider Szenarien integriert und dabei die Auswirkungen der jeweiligen Nachteile möglichst gering hält.

Unter Beibehaltung der Forderung nach geringstmöglicher Verbindungszahl zwischen verschiedenen Stationen sind somit folgende Optimierungsziele wesentlich:

- [Z1] minimale Laufzeit von Station zu Station
- [Z2] minimale Anzahl der Zwischenstationen
- [Z3] maximaler Zeitraum zwischen zwei Synchronisationsereignissen

3.4 Objektinteraktion

Im Voodie System ist zur Zeit noch keine Interaktion zwischen einzelnen Objekten möglich. Es ist ein separater Mechanismus zu entwickeln, mit dessen Hilfe Interaktionen erkannt und aufgelöst werden können. In dieser Arbeit kann nur ansatzweise über verschiedene Verfahren nachgedacht werden.

Interaktions-erkennung Alle Objekte können mit anderen Objekten in Interaktion treten. Im einfachsten Fall wird diese Interaktion aus Berührungen bestehen. Die Interaktion zwischen den Objekten erfolgt dabei mit Hilfe einer übergeordneten Instanz. Diese Instanz ist für die Interaktion aller Objekte innerhalb eines bestimmten Raumabschnittes zuständig. Es ist sinnvoll, diesen Raumabschnitt mit dem räumlichen Einzugsbereich des jeweiligen Teilnehmers gleichzusetzen, da dadurch gleichzeitig eine Aufteilung der notwendigen Rechenlast auf die Stationen der Teilnehmer erfolgt.

Interaktionen zwischen zwei Objekten sind prinzipiell nur dann möglich, wenn sich beide Objekte im gleichen Raumabschnitt, d.h. unter der Verwaltung der gleichen Interaktionsinstanz befinden. Interaktion erfolgt also in jeder Welt separat.

Interaktionsdämon (ID) Die Instanz, die die Objekte über Interaktionsereignisse informiert, soll mit *Interaktionsdämon (ID)* bezeichnet werden. Der ID kann auch die Verwaltung der Objekte übernehmen, da Verwaltung (Erzeugung, Zerstörung etc.) und Interaktionserkennung eng miteinander verbunden sind und prinzipiell die gleichen Informationen (z.B. Objektpool) benötigt werden.

Interaktionsereignisse treten z.B. dann auf, wenn sich zwei Objekte berühren. Der ID muß also die Möglichkeit haben, Informationen über Größe und Position der Objekte auszuwerten.

Ein Interaktionsdämon ist für die Erkennung von Interaktionen zwischen Objekten zuständig, die sich in einer gewissen Umgebung um den Teilnehmer befinden. Weil der Teilnehmer in der Lage sein soll, sich in der virtuellen Umgebung zu bewegen, muß konsequenterweise auch der Interaktionsdämon als beweglich definiert werden. Der Raumabschnitt, für den ein ID zuständig ist, ist also kein statisch definiertes Gebiet der virtuellen Umgebung, sondern dieser Raumabschnitt unterliegt dynamischen Änderungen. Für den Teilnehmer bedeutet diese Kopplung, daß keine Grenzen der Welt bemerkbar werden. Der Teilnehmer sieht die Grenzen lediglich in einer gewissen Entfernung (vgl. Aura in MASSIVE, ab Seite 23).

In jeder dieser Welten ist nur das existent, was sich in einem gewissen räumlichen Umkreis um den Akteur abspielt. In gewisser Weise kann dies auch als eine Analogie zu idealistischen Philosophien gesehen werden –

„Nur das sichtbare ist real. Alles andere ist die Idee“

Die gesamte virtuelle Umgebung stellt sich in diesem Zusammenhang als eine Idee dar, ein jeder Teilnehmer sieht lediglich vergängliche (Teil-)Abbilder und Teilaspekte dieser Idee.

Sollte sich ein Masterobjekt aus dem Einflußbereich eines lokalen ID heraus bewegen, existieren zwei Möglichkeiten zum weiteren Verfahren:

- [A] Das Objekt hört auf zu existieren.
Wenn das Objekt Abonnenten besitzt, geht die Koordinierung an einen (z.B. den ältesten) Abonnenten über. Das lokale Objekt wird gelöscht und muß neu abonniert werden, wenn es wieder sichtbar sein soll.
- [B] Das Objekt existiert unabhängig von seiner Position zum ID bis zur expliziten Zerstörung weiter. Dabei geht die Koordinierung nicht an einen Abonnenten über.

Es ist denkbar, daß der Übergang an einen Abonnenten nicht nur durch die Lebensdauer der Abonnenten gesteuert wird, sondern durch andere Mechanismen. Weiter oben wurde bereits ein Übergang in Abhängigkeit von der Position des Objektes vom lokalen ID besprochen. Abonnenten könnten dann die Übernahme vorschlagen, wenn sich das Objekt sehr nahe am eigenen ID befindet. Letztlich entscheidet aber das Masterobjekt, ob es die Kontrolle abgibt oder nicht.

Nachdem nun geklärt ist, wann Interaktionen auftreten können, steht noch die Frage, wie diese Interaktionen erkannt und aufgelöst werden.

Interaktionsauflösung Ein Ansatz wäre die Erweiterung des Interaktionsdämons. Der Interaktionsdämon ist immer mit der Position der Station (also des Nutzers bzw. dessen Avatar) im virtuellen Raum gekoppelt. Um diese Position des Nutzer existiert ein Bereich, in dem alle Objekte (bekannter) entfernter Stationen abonniert werden müssen. Es werden nun dynamische Interaktionsgruppen gebildet, die sich aus den Objekten zusammensetzen, die auf einer Station existieren (Master- und Slaveobjekte). Interaktion kann nur zwischen solchen Objekten stattfinden.

Interaktion wird zusätzlich nur zwischen Masterobjekten erlaubt. Slaveobjekte, mit denen eine Interaktion durchgeführt werden soll, geben den Interaktionswunsch an ihr Masterobjekt weiter. Wenn zwei Masterobjekte interagieren, so geschieht dies, indem beide Objekte von der notwendigen Interaktion informiert werden. Die Objekte können dann miteinander ihre Reaktion abstimmen. Im einfachsten Fall sieht das so aus, daß beide Objekte an das jeweils andere Objekt eine Nachricht mit ihren Interaktionsdaten schicken und das jeweils andere Objekt entsprechend darauf reagiert. Kompliziertere Interaktionsmechanismen sind dabei nicht ausgeschlossen.

Sollte eine Station lediglich ein Objekt von zwei interagierenden Objekten abonniert haben, so geschieht die Interaktion ohne das zweite Objekt, indem nötigenfalls die Objektkontrolle an die Station übertragen wird, auf der beide interagierende Objekte vorhanden sind.

Die Interaktionen werden bei dieser Vorgehensweise durch die Objekte selbst aufgelöst. DIVE (/CAR93/) verwendet diese Methode. Kazman beschreibt in /KAZ93a/ die Schwierigkeiten einer solchen Vorgehensweise. Problematisch ist die Anzahl von Interaktionsmöglichkeiten bei vielen Objekten¹⁷. Jeder Objekttyp muß die Interaktion mit sämtlichen anderen Objekttypen beherrschen. Die jeweiligen Aktionen sind dabei Bestandteil der Objektdefinition. Dadurch ist es nicht möglich, neue Objekte (und damit neue Interaktionen) einzuführen, ohne die alten Objekte zu verändern. Diese Vorgehensweise wird deshalb nicht weiter in Betracht gezogen.

Interaktionsobjekte Die Interaktion verschiedener Voodoo Objekte könnte durch den Einsatz von Interaktionsobjekten ermöglicht werden. Diese Interaktionsobjekte sind nicht-sichtbare, aber abonnierbare Voodoo Objekte. Die Kommunikation von Voodoo Objekten über Interaktionsobjekte geschieht, indem ein Interaktionsobjekt Referenzen auf die Objekte besitzt, die interagieren sollen. Das Interaktionsobjekt prüft dabei permanent, ob eine Interaktion zwischen zwei Objekten vorliegt (z.B. Kollision). Ist dies der Fall, so werden beide Objekte davon benachrichtigt. Interaktionsobjekte residieren dabei auf der Station, auf der auch die beteiligten Objekte vorhanden sind. Durch die Interaktionsobjekte wird die Definition der Interaktionen von der Definition der Objekte getrennt. Neue Objekte (und damit neue Interaktionsmöglichkeiten) können in das System integriert werden, ohne die bereits existierenden Objekte zu verändern. Es müssen lediglich entsprechende Interaktionsobjekte definiert werden.

Nachteilig wirkt sich bei diesem Vorgehen aus, daß für jedes Paar interagierender Objekte ein Interaktionsobjekt benötigt wird. Dadurch steigt die Anzahl der zu verwaltenden Objekte sehr stark an. Bei n Objekten, die alle miteinander interagieren sollen, sind $\frac{1}{2}n(n-1)$ Interaktionsobjekte notwendig.

separate Interaktionsdefinitionen Auch Kazman (/KAZ93b/) schlägt eine von den Objekten unabhängige Definition der Interaktionen vor. Dieser Ansatz ist sinnvoll, weil Objektinteraktionen prinzipiell unabhängig vom Verhalten einzelner Objekte sind. In den Objekten wird nur das Verhalten der Objekte definiert.

Wenn neue Objekte in das System eingeführt werden sollen, ist es notwendig, das Verhalten des Objektes zu spezifizieren und festzulegen, welche Interaktionen mit anderen Objekten möglich sind. Die Spezifikation des Verhaltens und der Interaktionen erfolgt dabei unabhängig voneinander.

Diese Trennung hat den Vorteil, daß die Objekte ohne Änderung weiterverwendbar sind, auch wenn neue Interaktionen mit neuen Objekten notwendig sind.

In Anlehnung an die Implementation wird für die Interaktion von Objekten folgende Lösung vorgeschlagen:

Das Verhalten eines Objektes wird mit Hilfe einer Simulationsschleife und der Eventbehandlungsroutine des Objektes festgelegt. Interaktionen werden separat

¹⁷ Bei n Objekttypen existieren theoretisch $\frac{1}{2}n(n-1)$ Interaktionspaare

spezifiziert. Angenommen es existieren zwei Objekte A und B vom Typ `Type_A` und `Type_B`. Jedes dieser Objekte besitzt ein bestimmtes Verhalten und bestimmte Attribute:

```
Type_A (z.B. sich bewegendes Objekt):
  Attribute:
    Geometrie, Position, Geschwindigkeit
  Verhalten:
    verändere Position in Abhängigkeit von Geschwindigkeit
Type_B (z.B. Wand):
  Attribute:
    Geometrie, Position
  Verhalten:
    bleibe immer an der gleichen Stelle.
```

Die Interaktionsspezifikation für diese Objekte müßte dann Folgendes enthalten:

```
Interaction (Type_A A1 <--> Type_A A2):
  tritt auf:
    (A1.Geometrie(A1.position))
    schneidet/berührt
    (A2.Geometrie(A2.position))
  Reaktion:
    A1: Umkehren(A1.Geschwindigkeit)
    A2: Umkehren(A2.Geschwindigkeit)

Interaction (Type_A A1 <--> Type_B B1):
  tritt auf:
    (A1.Geometrie(A1.position))
    schneidet/berührt
    (B1.Geometrie(B1.position))
  Reaktion:
    A1: Umkehren(A1.Geschwindigkeit)
    B1: nichts machen

Interaction (Type_B B1 <--> Type_B B2):
  nicht relevant
```

Es ist auch denkbar, globale Gesetzmäßigkeiten (z.B. Gravitation) als Interaktion von Objekten mit einem Environmentobjekt aufzufassen. Jedes Objekt wird dabei von der Interaktion mit der Umgebung unterrichtet und die Objekte stellen dann ihr Verhalten auf dieses Umgebungsobjekt ein. Problematisch ist dabei, daß solche Interaktionen permanent auftreten, die Objekte also permanent von diesen Interaktionen unterrichtet werden müßten. Eine Lösung für dieses Problem stellt die separate Definition von Interaktionsmethoden dar, die einmalig vom Objekt angefordert und dann immer wieder durch das Objekt ausgeführt werden.

3.5 Konsistenz der Welten

Um verteilte virtuelle Umgebungen benutzen zu können, ist es notwendig, den beteiligten Teilnehmern eine konsistente Sicht der Umgebung zu bieten. Durch die dezentrale Datenhaltung können Inkonsistenzen auftreten.

Die Konsistenz einer verteilten Umgebung wird in folgenden Fällen verletzt:

[A] Ein Objekt ändert eigenständig seine Eigenschaften.

Diese Inkonsistenz wird durch den Abomechanismus und die damit gewährleistete Synchronisation behoben.

[B] Zwei Welten enthalten nicht die gleichen Objekte.

Diese Verletzung der Konsistenz kann nur über einen Mechanismus gelöst werden, der sicherstellt, daß jede Welt die gleichen Objekte enthält. Dies führt im Extremfall zu einer, auf jeder Station vollständig replizierten Umgebung. Dieser Fall tritt aber in den seltensten Fällen ein. Ein Teilnehmer benötigt für eine konsistente Sicht auf die Welt lediglich die Objekte, die sich in einer gewissen Entfernung von ihm befinden.

Denkbar ist auch eine Zuordnung von Objekten zu Interaktionsgruppen. Zu diesen Gruppen gehören alle Objekte, die sich gegenseitig beeinflussen. Wenn ein Objekt abonniert wird, dann müssen auch alle Objekte der jeweiligen Interaktionsgruppe abonniert werden. Solche Interaktionsgruppen könnten als verkettete Listen von OI implementiert werden. Dieser Ansatz brächte allerdings unnötige Mehrkommunikation mit sich und ist deshalb nicht sinnvoll.

Welten, die lediglich die Objekte der gesamten Umgebung enthalten, die sich innerhalb eines gewissen Bereiches um den Teilnehmer befinden können diese Konsistenzverletzung besser lösen.

Für zwei Teilnehmer der virtuellen Umgebung, die sich im gleichen Raumabschnitt des virtuellen Raumes befinden, müssen die gleichen Objekte sichtbar sein.

Ist dies nicht der Fall, sind prinzipiell Situationen vorstellbar, in denen ein Objekt in unterschiedlichen Welten unterschiedliches Verhalten zeigt. Als Beispiel sei an zwei Welten gedacht, in denen sich ein Objekt (z.B. Ball) bewegt. In der einen Welt könnte dieses Objekt auf ein anderes Objekt (z.B. Wand) treffen und damit seine Flugbahn ändern. Wenn in der anderen Welt nicht die gleichen Objekte vorhanden sind, wäre das Verhalten des Objektes inkonsistent, weil entweder der Ball völlig unmotiviert seine Flugbahn im freien Raum ändert, oder der Ball sich einfach weiterbewegt (und damit in der anderen Welt durch die Wand hindurch fliegt).

Deshalb ist das Abonnement aller Objekte in der unmittelbaren Umgebung eines Teilnehmers zwingend notwendig. Interaktionen (z.B. Kollisionen) sind dann immer schlüssig, da der Interaktionspartner eines Objektes im gleichen Raumabschnitt existent ist. Im ungünstigsten Fall kann sich eine Interaktion zwischen zwei Objekten auf ein einzelnes Objekt eines anderen ID auswirken. Da sich dieses Objekt aber am Rand des Einflußbereiches des ID befindet, ist die scheinbar unmotivierte Veränderung der Objekteigenschaften akzeptabel.

Die automatische Abonnierung aller relevanten Objekte kann über die Untersuchung der OI erfolgen. Die OI müssen dazu Informationen zur Position in der Umgebung enthalten. Eine Station kann dann eigenständig alle Objekte abonnieren, die für sie relevant sind.

dyna-
mische
Optimie-
rung der
Kommuni-
kations-
topologie

Natürlich ergibt sich daraus die Notwendigkeit, daß die OI aller Objekte auf allen Stationen verfügbar sind. Ein klarer Verstoß gegen die Anforderung, möglichst große Welten (viele Objekte) mit vielen Teilnehmern zu unterstützen. Leider ist zu diesem Zeitpunkt noch keine vollständige Lösung für dieses Problem gefunden. Ein Ansatz könnte sein, spezielle OI für die Stationen einzuführen. Diese OI müssen auf allen Stationen bekannt sein. Da die Stationen nur Objekte verwalten, die sich in unmittelbarer Nähe der Station befinden, kann anhand der Positionen der Stationen entschieden werden, ob die OI angefordert werden oder nicht. Natürlich bedeutet diese Lösung nur eine Verschiebung des Skalierungsproblems. Wesentlich eleganter wäre eine Lösung, mit deren Hilfe es möglich wäre, relevante Stationen bzw. Objekte ohne Kenntnis der Gesamtstruktur, nur mit Hilfe der jeweils benachbarten Stationen, ausfindig zu machen. Die dynamische Optimierung der Kommunikationstopologie zwischen den beteiligten Stationen scheint hier ein vielversprechender Ansatz zu sein, der in jedem Fall weiter untersucht werden sollte.

3.6 Optimierung der Ein- und Ausgabe

Die Visualisierung der virtuellen Umgebung sollte mit Hilfe einer separaten Applikation erfolgen. Die Kommunikation der Voodie Applikation mit der Applikation, die die graphische Ausgabe besorgt, wird dabei über Ereignisse abgewickelt.

Display-
sprache

Wenn ein Objekt seine Visualisierung verändern will, muß es demzufolge ein Ereignis an die Visualisierungsapplikation bzw. an das Visualisierungsmodul (Renderer) absetzen. Diese Vorgehensweise stellt aber einen Engpaß dar, da mitunter sehr viele Objekte sehr oft solche Ereignisse absetzen werden. Um keinen Engpaß zu provozieren, sollte vorgesehen werden, die Visualisierungsereignisse so auszulegen, daß diese Events vom Renderer lediglich einmal empfangen werden und dann immer wieder ausgeführt werden. Dieses Vorgehen könnte etwa mit der Registrierung von Callbacks verglichen werden¹⁸.

Diese Methode macht allerdings die Definition einer Displaysprache notwendig. Die Darstellungsereignisse enthalten dann Anweisungen in dieser Sprache und werden vom Renderer ausgeführt.

Ähnlich sieht die Kommunikation der Objekte mit den Eingabegeräten aus. Auch hier empfiehlt es sich, den Kommunikationsaufwand gering zu halten.

In verschiedenen Systemen (Rubber Rocks, ab Seite 20 und /COD92/, Waves / Hydra, ab Seite 28 und /KAZ93a/) wird die Definition von **higher-level Events** vorgeschlagen, um den Kommunikationsaufwand zu verringern.

¹⁸ Callbacks werden z.B. in der GLUT Bibliothek (/KIL96/) benutzt.

*... aber ich werde alles geben,
daß er Wirklichkeit wird!*

*Der Traum ist aus
aus: „Keine Macht für Niemand“
Ton Steine Scherben (1972)*

4 Implementierung

Aufgrund der Systematisierung und der Bestandsanalyse wurden im vorangegangenen Abschnitt Überlegungen angestellt, wie das Kommunikationsproblem in verteilten virtuellen Umgebungen gelöst werden kann. Einige der Überlegungen führten zur Entwicklung von Mechanismen. Diese Mechanismen wurden in einer Prototyplösung implementiert, um ihre Funktionsfähigkeit praktisch zu prüfen.

Wesentliche Ziele bei der Entwicklung des Systems waren Skalierbarkeit und eine flexible, erweiterbare Struktur.

Im Einzelnen realisiert der Prototyp den Eventtransport über das Netzwerk, die Verwaltung der OI, den Abonnementmechanismus, den Übergang der Objektkontrolle, Objektverhalten sowie minimale Visualisierung. Mechanismen wie automatische (positionsabhängige) Abbonierung, Interaktionserkennung und

Interaktionsauflösung sowie Objektcodetransfer wurden noch nicht implementiert.

Der Prototyp stellt einen ersten Ansatz für eine verteilte virtuelle Umgebung zur Verfügung. In einem solchen System müssen neben der Implementation der vorgestellten kommunikationsrelevanten Mechanismen noch andere Probleme gelöst werden. Diese Probleme wurden im Abschnitt 3 (ab Seite 32) dargestellt.

4.1 Voodie Prototyp

Der Voodie Prototyp stellt die Prototypimplementation für die Mechanismen dar. Er besteht aus einer Applikation, die die eigentlichen Mechanismen und die notwendigen Verwaltungsmodule implementiert und einer Visualisierungsanwendung. Die derzeitige Version des Prototypen besteht aus ca. 4500 Zeilen C++ Code und wurde auf einer Indigo² Maximum Impact Grafikworkstation mit TRAM Option der Firma Silicon Graphics Inc. unter Irix 6.2 entwickelt. Der vollständige Quellcode liegt der Arbeit auf einem digitalen Speichermedium bei.

Die Implementierung der Verwaltungsmodule und der Voodie Objekte ist objektorientiert und greift auf Funktionalitäten der Standard C++ Bibliothek (/JOS96/, /STL96/) zurück.

Die Applikation besteht aus einer Reihe eigenständiger Module, die in separaten Threads ablaufen. Dabei wurde zur Implementation die POSIX 1003.1c Threads API (pthreads, /NIC96/) benutzt. Die einzelnen Module kommunizieren dabei über den Austausch von Ereignissen (Events). Dieser Austausch erfolgt über globale, modulspezifische Eventqueues. Bei der Entscheidung, für die Implementierung der einzelnen Module Threads zu verwenden, stand die mögliche Benutzung von Mehrprozessormaschinen an zweiter Stelle. Hauptgrund für den Einsatz von Threads waren softwaretechnologische Überlegungen. Durch den Einsatz von separaten, getrennt ausgeführten Modulen für die Verwaltung, ist eine sehr gute Trennung der einzelnen Module möglich.

Der Prototyp hat verschiedene Aufgaben zu lösen. Eine Aufgabe ist die Verwaltung der Objekte, eine weitere Aufgabe stellt die Kommunikation der Systeme verschiedener Teilnehmer über das Netzwerk dar. Die Visualisierung und die Verarbeitung von Eingaben sind weitere Aufgaben, die durch den Prototyp gelöst werden müssen. Da die einzelnen Module des Systems miteinander kommunizieren müssen, ist auch das Management dieser Kommunikation eine Aufgabe des Prototypen. Ausgehend von diesen Aufgaben wurde das System in folgende Module unterteilt:

Interaktionsdämon

Der Interaktionsdämon (ID) verwaltet Objekte des Voodie Systems. Über ihn werden Objekte generiert und gelöscht.

Net_listener / Net_sender

Diese Module ermöglichen den Transport von Ereignissen von einer Station zu einer anderen.

Renderer / console_listener

Das Renderer Modul verarbeitet Visualisierungsereignisse, console_listener setzt Eingaben in entsprechende Ereignisse um. Im Prototyp fungieren diese Module als Interface zu einer separaten Visualisierungsanwendung. Diese Anwendung verarbeitet die durch das Renderer Modul ausgegebenen Ereignisse weiter und liefert in Abhängigkeit von den Eingaben des Benutzers Daten an das Console_listener Modul.

Distributor

Ein weiteres Modul, der Distributor, verteilt die Ereignisse an die entsprechenden Module und ermöglicht damit die Kommunikation zwischen den einzelnen Modulen.

Voodie Objekte

Objekte im Voodie System werden wie die Verwaltungsmodule als separate Threads implementiert. Dies ermöglicht zum einen eine flexible und abgeschlossene Implementierung der Objekte und zum anderen auch die Implementation von zusätzlichen Verwaltungsmodulen. Im Prototyp wurde z.B. der OI-Verwalter als spezielles Voodie Objekt implementiert.

Abbildung 9 verdeutlicht den Zusammenhang und die Aufgaben der Module.

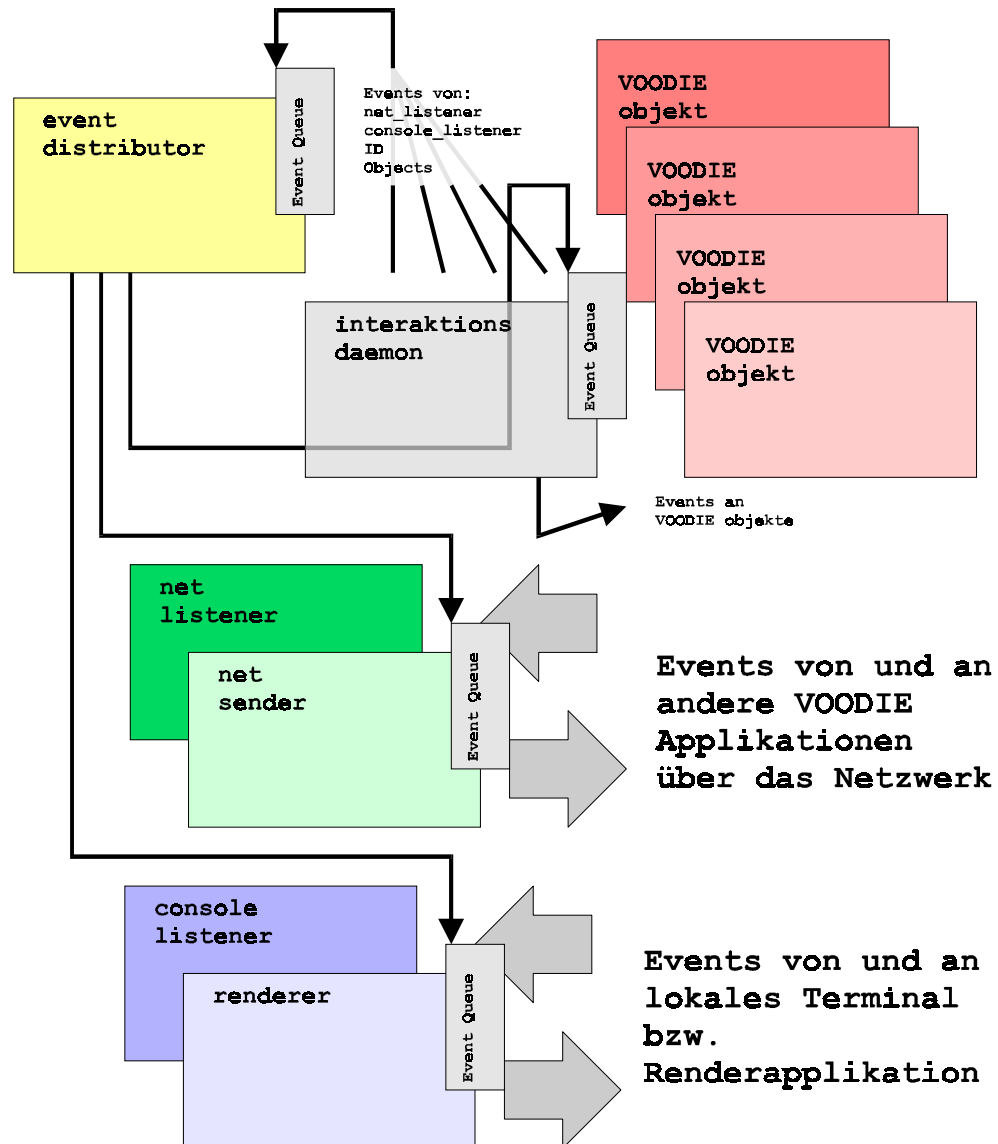


Abbildung 9: Interner Aufbau der Voodie Applikation

4.2 Graphische Ausgabe

Die Implementierung der Mechanismen beinhaltet keine Funktionalität zur Visualisierung der Voodie Objekte. Durch das **Renderer** Modul wird lediglich der Inhalt der **RENDER** Events auf den Standardausgabekanal ausgegeben. Derzeit werden diese Ausgaben von einer eigenständigen Renderapplikation weiterverarbeitet. Für die Implementierung der Renderapplikation wurde das OpenGL API (/NEI93/) und die GLUT Bibliothek (/KIL96/) benutzt. Die Renderapplikation läuft als Koprozess parallel zur Hauptanwendung (Voodie Applikation).

Die Kommunikation der beiden Prozesse erfolgt über eine Duplex Pipe. Das bedeutet, daß auf den Standardausgabekanal (`stdout`) der Voodie Applikation von der Renderapplikation aus zugegriffen werden kann. Der Standardeingabe-

kanal (`stdin`) der Voodie Applikation wird von der Renderapplikation mit Daten versorgt. Abbildung 10 zeigt die Kommunikation der beiden Prozesse.

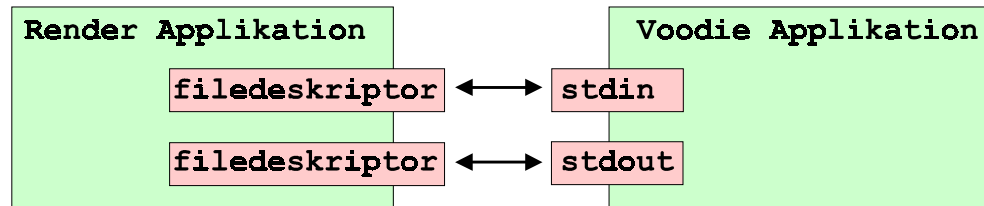


Abbildung 10: Kommunikation zwischen Render- und Voodie Applikation

4.3 Eventdatenstruktur

Die gesamte Kommunikation im Voodie System ist ereignisorientiert. Dabei besitzen die Ereignisse (Events) die Struktur

```

TYPE
FROM_HOST
FROM_MODULE
TO_HOST
TO_MODULE
MESSAGE.
  
```

Diese Struktur ist die derzeit implementierte Struktur für Events. Sie orientiert sich sehr stark an der internen Implementation des Voodie Prototyps.

mögliche
Verbes-
serungen

In der bisherigen Eventdatenstruktur sind keine Objektidentifikatoren enthalten. Sollen Voodie Objekte identifiziert werden, so geschieht dies derzeit über den Messageteil des Events. Diese Identifikatoren können aber fest in die Datenstruktur eingebaut werden, indem die Module auch als Voodie Objekte behandelt werden. Diese Objekte besitzen dann reservierte ObjektIDs und sind nicht abonnierbar. Aufgrund bisheriger Erfahrungen und Tests sollten die Events in der Form:

```

TYPE
SENDER
RECEIVER
MESSAGE
  
```

vorliegen.

Dabei sind `SENDER` und `RECEIVER` jeweils ObjektIDs.

Dadurch könnten sich z.B.: folgende (vorbelegte und damit reservierte) ObjektIDs ergeben:

DISTRIBUTOR	...	Objekt 0
NET_SENDER	...	Objekt 1
NET_LISTENER	...	Objekt 2
RENDERER	...	Objekt 3
CONSOLE	...	Objekt 4
ID	...	Objekt 5
OI_OBJECT	...	Objekt 6

Eine ObjektID kann dann – unter Hinzunahme einer Portnummer – in der Form:

```
HOSTNAME:PORTNUMMER/OBJEKTNUMMER
```

dargestellt werden. Beispielhaft sähen ObjektIDs folgendermaßen aus:

```
parrot.informatik.hab-weimar.de:3490/2  
parrot.informatik.hab-weimar.de:3490/ID  
141.54.12.13/7 (default Port)
```

Wenn diese Darstellung um den Namen des Protokolls erweitert wird, kann eine URL (/LEE94a/, /LEE94b/) konforme Schreibweise für die Identifizierung eines Voodie Objektes benutzt werden:

```
voodie://parrot.informatik.hab-weimar.de:3490/ID
```

4.4 Netzwerkmodule / Eventtransfer

Der Netzwerkverkehr im Voodie System basiert auf UNIX Stream Sockets (/BRO94/, /STE92/, /WRI95/, /RAG93/, /HAL96/).

Stream Sockets stellen ein Interface für die Netzwerkkommunikation zur Verfügung, welches verbindungsorientierte Kommunikation ermöglicht. Dadurch können Datenströme beliebiger Länge von einer Station zu einer anderen übertragen werden. Die Verwendung von Stream Sockets bringt aber auch einen nicht unerheblichen Aufwand für das Verbindungsmangement mit sich, da der Datentransport Methoden zur Fehlerkorrektur beinhaltet. Durch die Benutzung von verbindungsloser, paketorientierter Kommunikation (realisierbar z.B. durch den Einsatz von UDP) wäre eine Performancesteigerung beim Datentransport möglich. In diesem Fall müßte aber die Notwendigkeit einer Fehlerkontrolle und Empfangsbestätigung geprüft werden.

Da die Übertragung der Ereignisse noch optimiert werden muß, diese Aufgabe aber für das Funktionieren des Prototypen nicht relevant ist, wurden im Rahmen der Lehrveranstaltung Rechnernetze II des Studienganges Informatik der Bauhaus-Universität Weimar im Sommersemester 1997 Belegaufgaben zur Optimierung der Ereignisübertragung ausgegeben. Die Ergebnisse dieser Arbeiten werden erst nach Abschluß dieser Arbeit vorliegen. Die Ergebnisse sollten auf jeden Fall auf eine Verwendbarkeit in zukünftigen Voodie Implementationen hin untersucht werden.

Die Netzwerkmodule im Voodie System (`Net_Listener`, `Net_Sender`), haben die Aufgabe, Events von einer Station zu einer anderen zu übertragen. Das Modul `Net_Sender` hat die Aufgabe, Events an eine entfernte Station zu schicken. Das Modul `Net_Listener` empfängt Events, die von anderen Stationen abgeschickt wurden.

Bei der Übertragung der Events wird folgendes „low-level“ Eventtransferprotokoll benutzt (Eventlänge unbegrenzt, Streamvariante):

```

LISTENER : erwarte Verbindung
SENDER   : stelle Verbindung her
LISTENER : sende Begrüßung
              This is VOODIE <Version>
              accepting Events:
              (TYPE FROM_HOST FROM_MODULE
              TO_HOST TO_MODULE MESSAGE end of message)
SENDER   : lese Begrüßung
wiederhole
SENDER   : sende ein Paket (einen Teil, einige Bytes
              der Zeichenkette)
LISTENER : lese ein Paket
LISTENER : sende Empfangsbestätigung
              received <num> Bytes. ok!
SENDER   : erwarte und lese Empfangsbestätigung
LISTENER : hänge das Paket an die bereits
              empfangene Zeichenkette an
bis die gesamte Zeichenkette übertragen wurde.
SENDER   : sende
              end of message
LISTENER : sende Bestätigung, Verbindung abbrechen
SENDER   : lese Bestätigung, Verbindung abbrechen
LISTENER : empfangene Zeichenkette ist Rückgabewert
    
```

Da die Events uncodiert übertragen werden, ist es möglich, ein Event von Hand über das Netzwerk an eine Voodie Station zu übertragen. Dies könnte mit telnet z.B. so aussehen:

```

parrot 102 /usr/people/meister1> telnet 141.54.24.13 3490
Trying 141.54.24.13...
Connected to 141.54.24.13.
Escape character is '^]'.
This is Voodie v0.002, feb,mar 97,(p) by mm
accepting Events: (TYPE FROM_HOST FROM_MODULE TO_HOST TO_MODULE
MESSAGE end of message) ...
DATA LOCAL INPUT 141.54.24.13 ID CREATE_OBJECT CLASS =
voodie_first_object_c end of message
received 77 Bytes. ok!
Connection closed by foreign host.
parrot 103 /usr/people/meister1>
    
```

zeitliche
Ordnung
von Ereignissen

Für die Mechanismen, die auf dem Austausch von Ereignissen basieren, ist es nicht notwendig, die Ereignisse zeitlich zu ordnen. Die derzeit implementierten Mechanismen benötigen eine solche Ordnung nicht. Für in Zukunft zu entwickelnde Mechanismen ist es aber durchaus denkbar, daß eine zeitliche Ordnung der Ereignisse hergestellt werden muß. Lamport schlägt in /LAM78/ einen verteilten Algorithmus vor, der zu diesem Zweck eingesetzt werden könnte.

4.5 Wichtige Ereignisse (Events) in Voodie

Im Voodie System sind sehr viele Events möglich. Einige dieser Events wurden in den vorangegangenen Ausführungen bereits erwähnt. Die meisten dieser Events werden automatisch als Reaktion auf andere Events generiert. Prinzipiell können alle diese Events auch explizit von außen durch einen Operator an das jeweilige Modul bzw. Voodie Objekt geschickt werden. Für die Demonstration der Mechanismen werden aber lediglich die folgenden Events benötigt:

Global Exit

Typ:	EXIT	Name:	GLOBAL_EXIT
From Host	LOCAL	To Host	LOCAL
From Module	INPUT	To Module	DISTRIBUTOR
Message	<error_string>		
Beschreibung	Alle Module beenden, Objekte zerstören, Programm beenden.		

Create Object

Typ:	DATA	Name:	CREATE_OBJECT
From Host	LOCAL	To Host	LOCAL
From Module	INPUT	To Module	ID
Message	CREATE_OBJECT CLASS = <a_valid_class_name>		
Beschreibung	Generiere auf der lokalen Station ein Objekt der Klasse <a_valid_class_name> (CREATE_OBJECT Event).		

Send All OI

Typ:	DATA	Name:	SEND_ALL_OI
From Host	LOCAL	To Host	<TO_HOST>
From Module	INPUT	To Module	ID
Message	SEND_ALL_OI		
Beschreibung	Bitte um Sendung aller OI der Station <to_host> (SEND_ALL_OI Event).		

Typ:	DATA		
From Host	LOCAL	To Host	LOCAL
From Module	INPUT	To Module	OBJECTS
Message	id = <num> STATE = EXIT		
Beschreibung	Weise das Voodie Objekt mit der Identifikation <num> der lokalen Station an, seinen Status auf EXIT zu setzen.		

Subscribe OI

Typ:	DATA	Name:	SUBSCRIBE_OI
From Host	LOCAL	To Host	LOCAL
From Module	INPUT	To Module	OBJECTS
Message	id = 1 SUBSCRIBE_OI = <num>		
Beschreibung	Anweisung an das OI-Objekt, den OI, der sich an der Position <num> der aktuellen OI-Liste befindet, zu abonnieren. (SUBSCRIBE_OI Event, nicht zu verwechseln mit SUBSCRIBE Event).		

Die derzeitige Implementation des graphischen Interfaces erlaubt lediglich das Senden dieser Events. Eine Aufstellung aller möglichen Ereignisse findet sich im Anhang B „Mögliche Events in Voodie“)

4.6 Objektabonnement

Ein Objekt kann nur dann abonniert werden, wenn zumindest sein OI bekannt ist. Im OI sind Informationen zur Klasse des Objektes und die eindeutige Identifizierung (*ObjektID*) enthalten. Das Abonnement eines Objektes geschieht in drei Phasen. In Abbildung 11 sind die Phasen beim Objektabonnement zum besseren Verständnis graphisch dargestellt.

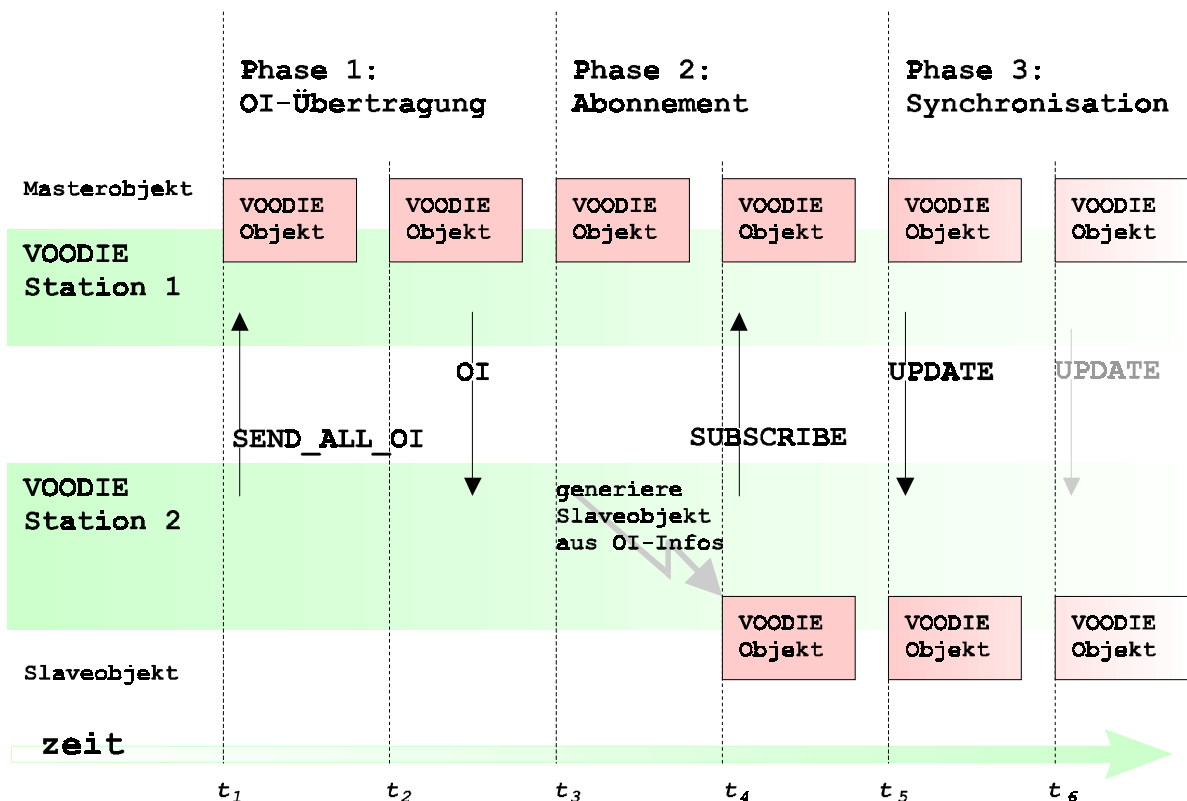


Abbildung 11: Phasen beim Objektabonnement

Phase 1: Die OI werden in der ersten Phase des Objektabonnements übertragen.

Station 2 sendet zu einem Zeitpunkt t_1 ein `SEND_ALL_OI` Ereignis an eine andere Station (Station 1). Diese Station reagiert auf dieses Ereignis, indem sie alle Objekte anweist, einen OI an die erste Station zu senden. Diese OI kommen irgendwann zwischen den Zeitpunkten t_2 und t_3 bei Station 2 an.

Soll ein Objekt (Masterobjekt) der Station 1 von Station 2 abonniert werden, so ergibt sich in einer zweiten Phase folgender Ablauf:

Phase 2: Auf Station 2 wird ein Slaveobjekt generiert (anhand der Daten im OI).

Das Slaveobjekt sendet dann ein **SUBSCRIBE** Event an das Masterobjekt auf Station 1 (ObjektID stand im OI) und wird somit zum Abonnent des Masterobjektes.

- Phase 3: Das Masterobjekt (auf Station 1) sendet nun während der dritten Phase **UPDATE** Events an das neu generierte Objekt Slaveobjekt (auf Station 2), bis es ein **UNSUBSCRIBE** Event vom Slaveobjekt erhält.

Die OI werden durch das OI-Objekt verwaltet. Das Abonnement eines Objektes einer anderen Station kann von außen durch Senden eines **SUBSCRIBE OI = <oi_num>** Events an das OI-Objekt eingeleitet werden. Das OI-Objekt veranlaßt dann die Erzeugung des Slaveobjektes

4.7 Übergabe der Objektkontrolle

Es sind zwei Ausgangssituationen für den Übergang der Objektkontrolle denkbar:

- [A1] Ein Slaveobjekt will die Objektkontrolle übernehmen.

Das Slaveobjekt sendet ein **REQ_OWNER** Event an das Masterobjekt. Das Masterobjekt entscheidet nun, ob es die Kontrolle abgeben will.

Wenn nein, passiert nichts.

Wenn ja, sendet das Masterobjekt ein **SET_OWNER** Event an das Slaveobjekt und wird selbst zu einem Slaveobjekt indem es seinen Status auf **SLAVE** setzt. Das Objekt wird dann automatisch Abonnent seines ehemaligen Slaveobjektes.

Wenn das Slaveobjekt das **SET_OWNER** Event erhält, setzt es seinen Status auf **MASTER** und nimmt das ehemaligen Masterobjekt in seine Aboliste auf.

- [A2] Ein Masterobjekt will die Kontrolle abgeben.

In diesem Fall sendet das Masterobjekt an einen seiner Slaves ein **REQ_REQ_OWNER** Event und bittet damit darum, ein **REQ_OWNER** Event zu schicken. Der weitere Ablauf erfolgt wie unter [A1] beschrieben.

Durch Senden eines **STATE = EXIT** Events an ein abonniertes Objekt wird das entsprechende Slaveobjekt gelöscht und das Abonnement aufgehoben. Dabei wird durch das Slaveobjekt folgendes sichergestellt:

- [B1] Das Slaveobjekt sendet ein **UNSUBSCRIBE** Event an seinen Abomaster¹⁹.

- [B2] Wenn das Slaveobjekt selbst Abonnenten besitzt, dann müssen die Abonnenten von der bevorstehenden Löschung des Objektes informiert werden. Deshalb sendet das Objekt seine *master_id* (die ObjektID des Abomasterobjektes) als **MASTER_ID = <id>** Event an alle seine Abonnenten. Die Abonnenten senden dann ein **SUBSCRIBE** Event an den neuen Abomaster. Damit wird die Unterbrechung evtl. bestehender Aboketten verhindert.

- [B3] Ein Masterobjekt kann nur dann gelöscht werden, wenn es keine Abonnenten besitzt. Ein Masterobjekt mit Abonnenten wandelt sich nach dem Empfang eines **STATE = EXIT** Events selbständig in ein Slaveobjekt

¹⁹ Dies muß nicht unbedingt das Masterobjekt der Gruppe sein.

um.

Dabei geht die Objektkontrolle vom Masterobjekt an ein Slaveobjekt der gleichen Abonnementgruppe über.

4.8 Objektimplementation in Voodie

Da das Voodie System in C++ implementiert wurde, sind systemintern prinzipiell alle Module und die meisten Datenstrukturen C++ Klassen. In der Nutzersicht auf das Voodie System sind Objekte dagegen eigenständig agierende Einheiten, die sich durch Senden von Events an das Rendermodul darstellen können. Diese Einheiten, im folgenden Voodie Objekte genannt, können als Objektgruppen auf verschiedenen Stationen residieren. Die Voodie Objekte einer Gruppe synchronisieren sich dabei über den vorgestellten Mechanismus (Abonnement, Updates etc.).

Implementationsseitig besitzen Voodie Objekte eine Funktion, die Events behandeln kann. Dabei kann nötigenfalls die Eventbehandlungsfunktion der jeweils übergeordneten Basisklasse aufgerufen werden, um allgemeinere Events automatisch behandeln zu lassen.

In der vorliegenden Implementation können die meisten Events durch die Eventbehandlungsroutine der Basisklasse `voodie_base_object_c` behandelt werden. Dies betrifft insbesondere alle Events, die für die Mechanismen (Objektsubscription, Übergabe der Objektkontrolle usw.) notwendig sind. Objekte abgeleiteter Klassen behandeln nur die Events, die in den Basisklassen nicht behandelt werden können, bzw. die nur für das jeweilige Objekt relevant sind.

Die Klasse `voodie_threaded_object_c` und davon abgeleitete Klassen definieren Objekte, die eine Funktion zur Steuerung des Objektverhaltens besitzen. Diese Funktion läuft in einem separaten Thread (also parallel zum Rest der Applikation und anderen Thread-Funktionen) ab. In AVIARY (ab Seite 25 und /SNO94/) werden *lightweight objects* vorgeschlagen. Objekte mit Thread-Funktion können ebenfalls als leichtgewichtige autonome Objekte in diesem Sinne angesehen werden.

Die Implementation eines Voodie Objektes kann dann so erfolgen, als ob keine anderen Aktionen ausgeführt werden müßten. Dies vereinfacht die Implementation der Voodie Objekte sehr stark. Die Eventbehandlungsroutine läuft asynchron zur Thread-Funktion. Konkurrierende Datenzugriffe müssen dabei durch den Einsatz einer (standardmäßig vorhandenen) *Mutexvariable* synchronisiert werden.

Soll z.B. ein Voodie Objekt implementiert werden, welches einen Ball simuliert, so könnte die Implementation im Pseudocode etwa so aussehen:

Simulationsschleife [`thread_function()`]:

```
durchlaufe, solange du lebst:
  wenn du Master bist:
    berechne Position,
      Flugrichtung,
      Geschwindigkeit,
      Beschleunigung
    mit Hilfe der vorherigen Werte,
```

```

        beachte dabei Umwelteinflüsse (Gravitation etc.) und
        andere Objekte (Kollisionen)
    benachrichtige alle Deine Abonnenten von den neuen Werten
        [update_slaves("<all_the_values>");]
    wenn du Slave bist:
        mache gar nichts
    wenn dein Status EXIT ist,
        begehe Selbstmord
        [suicide()]
    generiere in jedem Fall ein Event für das Ausgabemodul
        [enqueue_a_render_event ("Display me and use
        <all_the_values>");]

```

Diese Art der Implementation bedeutet, daß einzig das Masterobjekt Simulationsberechnungen ausführt. Das Verhalten der Slaveobjekte wird nur durch **UPDATE** Events vom Masterobjekt beeinflusst.

Es ist leicht einzusehen, daß dadurch ein sehr starker Kommunikationsaufwand durch die Master-Slave Synchronisation entsteht. Eine etwas andere Implementation löst dieses Problem dadurch, daß auch die Slaveobjekte ihr Verhalten berechnen. Im Pseudocode also so:

```

    durchlaufe solange du lebst:
        wenn du Master bist:
            berechne Position,
                Flugrichtung,
                Geschwindigkeit,
                Beschleunigung
            mit Hilfe der vorherigen Werte,
            beachte dabei Umwelteinflüsse (Gravitation etc.) und
                andere Objekte (Kollisionen)
            wenn Interaktionen mit anderen Objekten aufgetreten sind,
            benachrichtige alle Deine Abonnenten von den neuen
                Werten,
                [update_slaves("<all_the_values>");]
        wenn du Slave bist:
            berechne Position,
                Flugrichtung,
                Geschwindigkeit,
                Beschleunigung
            mit Hilfe der vorherigen Werte,
            beachte dabei Umwelteinflüsse (Gravitation etc.) und
                andere Objekte (Kollisionen)
        wenn dein Status EXIT ist,
            begehe Selbstmord
            [suicide()]
        generiere in jedem Fall ein Event für das Ausgabemodul
            [enqueue_a_render_event ("Display me and use
            <all_the_values>");]

```

Eventbehandlungsroutine [handle_event(event)]:

```

    durchlaufe, wenn ein Event vorliegt:
        lasse die Basisklasse(n) alle Events behandeln,
        die dort behandelt werden können
        behandle unbehandelte Ereignisse wie folgt:
            Event ist Update-event und du bist Slave:
                setze Deine Werte auf die Werte aus
                dem Update Event

                benachrichtige auch alle Deine Abonnenten
                von den neuen Werten
                [update_slaves("<all_the_values>");]
            ansonsten:
                das Event kann nicht behandelt werden.

```

4.8.1 Klassenbeschreibung

Bei der praktischen Implementierung der Voodie Objekte bietet sich die Aufstellung einer Klassenhierarchie an. Basis dieser Hierarchie bildet eine Klasse, deren Instanzen alle grundlegenden Eigenschaften eines Voodie Objektes besitzen. Von dieser Klasse werden weitere, spezialisierte Klassen abgeleitet. Abbildung 12 stellt den Zusammenhang der Klassen in der Hierarchie dar.

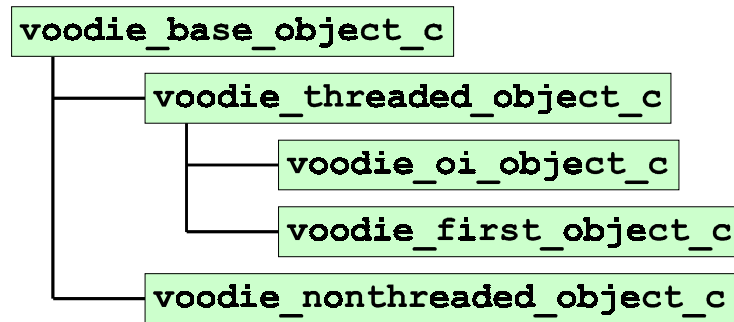


Abbildung 12: Klassenhierarchie

Im folgenden werden die einzelnen Klassen genauer beschrieben.

`voodie_base_object_c` ist die Basisklasse für alle Objekte. Objekte dieser Klasse besitzen folgende öffentliche Attribute:

C++ Code	Beschreibung
<code>object_id_c _my_id;</code>	Objektidentifizierung des Objektes
<code>char _class_name[100];</code>	Name der Klasse des Objektes
<code>bool _subscribable;</code>	Wenn diese Variable auf <code>FALSE</code> gesetzt wird, kann das Objekt nicht abonniert werden
<code>object_id_c _master_id;</code>	Diese ObjektID ist die Identifizierung des Abomasterobjektes
<code>object_state_c _state;</code>	Der Status des Objektes kann <code>MASTER</code> , <code>SLAVE</code> oder <code>EXIT</code> sein.
<code>id_list_t _slave_objects;</code>	Eine Liste, in die ObjektIDs der Abonnenten stehen.

Alle Objekte dieser und abgeleiteter Klassen besitzen folgende öffentliche Funktionen:

C++ Code	Beschreibung
<code>voodie_base_object_c (int id);</code>	Konstruktor für das Objekt.
<code>virtual ~voodie_base_object_c;</code>	Destruktor für das Objekt.

<pre>virtual int handle_event (event_c event); virtual void update_slaves (const char* update_string);</pre>	<table border="0"> <tr> <td style="border-left: 1px solid black; padding-left: 5px;">Diese Routine besorgt die Eventbehandlung.</td> </tr> <tr> <td style="border-left: 1px solid black; padding-left: 5px;">... sendet den übergebenen String als UPDATE Event an alle Abonnenten.</td> </tr> </table>	Diese Routine besorgt die Eventbehandlung.	... sendet den übergebenen String als UPDATE Event an alle Abonnenten.
Diese Routine besorgt die Eventbehandlung.			
... sendet den übergebenen String als UPDATE Event an alle Abonnenten.			

Objekte der Klasse `voodie_base_object_c` (und Objekte aller abgeleiteten Klassen) sind in der Lage, auf folgende Events zu reagieren:

```
STATE = EXIT
SEND_OI
REQ_UPDATE ID = <slave_id>
SUBSCRIBE ID = <slave_id>
UNSUBSCRIBE ID = <slave_id>
```

als Masterobjekt:

```
STATE = SLAVE
REQ_OWNER ID = <slave_id>
```

als Slaveobjekt:

```
STATE = MASTER
MASTER_ID = <new_master_id>
SET_OWNER ID = <old_master_id>
REQ_REQ_OWNER
```

`voodie_threaded_object_c`

Die Klasse `voodie_threaded_object_c` implementiert die Funktionalitäten, die ein Objekt mindestens besitzen muß, wenn seine `thread_function()` in einem separaten, unabhängigen Thread laufen soll. Dieser separate Thread dient der Implementation des Objektverhaltens und wird mit Objekt-Thread bezeichnet.

Diese Klasse ist von `voodie_base_object_c` abgeleitet. Folgende Attribute sind zusätzlich vorhanden:

C++ Code	Beschreibung
<code>pthread_mutex_t *_mutex;</code>	diese Mutex wird vom Objekt benutzt, um konkurrierende Zugriffe auf interne Variablen durch die Thread Funktion und die Eventbehandlungsfunktion zu synchronisieren
<code>pthread_t _object_thread;</code>	Datenstruktur für Threaddaten

`voodie_threaded_object_c` besitzt folgende öffentliche Funktionen:

C++ Code	Beschreibung
<code>virtual void run (void);</code>	generiert Thread und startet <code>thread_function()</code>
<code>virtual void thread_function (void);</code>	Diese Funktion kann von abgeleiteten Klassen überladen werden, um das Verhalten des Ob-

Objektes zu implementieren.

`voodie_oi_object_c` Die Klasse, deren Instanzen OI-Objekte sind, heißt `voodie_oi_object_c`. Diese Klasse ist von `voodie_threaded_object_c` abgeleitet. Auf jeder Station muß ein solches OI-Objekte vorhanden sein. Das OI-Objekt besitzt in der jetzigen Implementation die ObjektID 1. Das OI-Objekt hat die Aufgabe, eingehende OI Events in eine Liste einzutragen und regelmäßig zu prüfen, ob die Objekte noch immer existieren. Zusätzlich muß das OI-Objekt Events der Form **SUBSCRIBE OI = <num>** verarbeiten können (also entsprechende Abonnentenobjekte generieren). OI-Objekte selbst können nicht abonniert werden, d.h. die Membervariable `subscribable` wird bei der Erzeugung des OI-Objektes explizit auf `FALSE` gesetzt. Neben den Attributen der Basisklassen verfügt dieses Objekt zusätzlich über eine OI-Liste `oi_list_t _oi_list`.

Die Klasse redefiniert die Thread-Funktion und die Eventbehandlungsfunktion der Basisklasse `voodie_threaded_object_c`.

Objekte der Klasse `voodie_oi_object_c` sind in der Lage, auf folgende Events zu reagieren:

```

OI ID = <id> CLASS = <class_name> STATE = <state_val>
SUBSCRIBE OI = <num>
SHOW_LIST
    
```

Alle anderen Events werden durch die Basisklassen bearbeitet. Dazu genügt es, `voodie_threaded_object_c::handle_event()` vor der eigentlichen Eventbehandlung durch `voodie_oi_object_c` aufzurufen.

`voodie_nonthreaded_object_c` Objekte dieser Klasse besitzen lediglich eine Funktion zur Ereignisbehandlung. Ein Thread-Funktion ist nicht vorhanden. `voodie_nonthreaded_object_c` kann als Basisklasse für Objekte dienen, die kein eigenes Verhalten besitzen (z.B. statische Umgebungsobjekte). Es ist denkbar, daß sehr viele Objekte einer virtuellen Umgebung solche statischen Objekte sind, da die meisten Gegenstände ihr Verhalten nicht eigenständig ändern müssen. Verhaltensänderungen können bei Objekten dieser Klasse über Ereignisse ausgelöst werden.

`voodie_first_object_c` `voodie_first_object_c` bildet die erste Beispielimplementation für eine Klasse, deren Instanzen echte Voodie Objekte sind. Diese Klasse ist von `voodie_threaded_object_c` abgeleitet.

Mit `voodie_first_object_c` wurden Mechanismen, die der Reduktion des Synchronisationsaufwandes dienen können, implementiert. Objekte dieser Klasse besitzen als Stellvertreter für echtes Verhalten eine Variable, die bei jedem Simulationsschritt um eins erhöht wird. Im einfachsten Fall erhöht lediglich das Masterobjekt diese Variable und benachrichtigt den Rest der Objektgruppe davon. Die jetzige Implementierung sieht allerdings vor, daß auch die Slaveobjekte diese Variable verändern können. Das Masterobjekt sendet nicht nach jeder Veränderung eine Synchronisationsnachricht, sondern nur in bestimmten Abständen. Die Slaveobjekte verändern die Variable eigenständig und gleichen den Wert der Variable mit dem Masterobjekt ab, wenn eine

Synchronisationsnachricht ankommt. Durch diese Vorgehensweise kann auch Dead Reckoning implementiert werden.

Die Thread-Funktion dieser Klasse sieht im Pseudocode so aus:

```
while (!suicided) {
    lock_mutex()
    enqueue_a_render_event ("message for RENDERER:
                            Display me and my Attributes");
    if (_state == MASTER)
        i++
    release_mutex()
    if (0 == (i%25))
        update_slaves("i is <value_of_i>");
    if (_state == SLAVE)
        i++
    if (_state == EXIT)
        suicide()
    wait_a_few_moments()
}
```

Die Eventbehandlungsfunktion sieht so aus:

```
// Alle Events, die die Basisklassen behandeln
// können, behandeln
lock_mutex()
handle_status = voodie_threaded_object_c::handle_event(event);
if (EVENT_UNHANDLED == handle_status) {
    if (event == REQ_UPDATE ID = <slave_id>) {
        send_update_to_one_slave(slave_id, "i is <value_of_i>")
        handle_status = EVENT_HANDLED
    }
    if (_state == SLAVE) &&
        (event == „UPDATE i is <value_of_i>“) {
        i = <value_of_i>
        set_attributes_on_update_event()
        update_slaves("i is <value_of_i>")
        handle_status = EVENT_HANDLED
    }
}
release_mutex()
return (handle_status)
```

voodie_first_object_c ist also in der Lage, die Events

```
REQ_UPDATE ID = <slave_id>
UPDATE i is <value_of_i>
```

zu behandeln. Alle anderen Events werden durch die Basisklassen bearbeitet.

4.9 Funktionsbeschreibung des Prototypen

Nachdem der Prototyp gestartet wurde, ist es möglich, verschiedene Ereignisse an die Verwaltungsmodule bzw. Objekte des Prototyp zu senden, und damit die Mechanismen zu testen. Die Ereignisse werden durch die Auswahl aus einem Menü erzeugt. Abbildung 13 zeigt den Prototypen mit aufgeklapptem Ereignismenü.

Es können Objekte der Klassen `voodie_first_object_c`, `voodie_threaded_object_c` und `voodie_nonthreaded_object_c`

generiert werden.

Es ist weiterhin möglich, von einer anderen Station, auf der auch der Prototyp läuft, die OI der dort vorhandenen Objekte anzufordern. Nach der Auslösung eines entsprechenden Ereignisses (`SEND_ALL_OI` Event) durch die Anwahl des Menüpunktes `Events | Request All OI from ...`

erhält das OI-Verwaltungsobjekt (ObjektID = 1) die OI der entsprechenden Objekte und stellt diese in einer Liste dar (Abbildung 14).

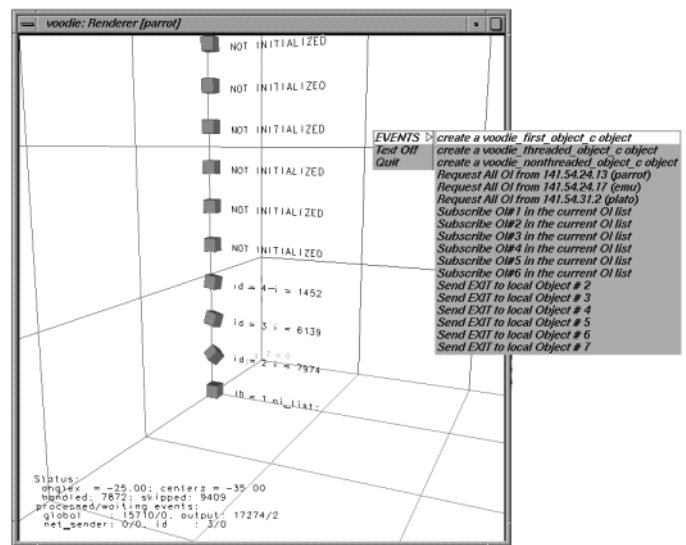


Abbildung 13: Prototyp mit Ereignismenü

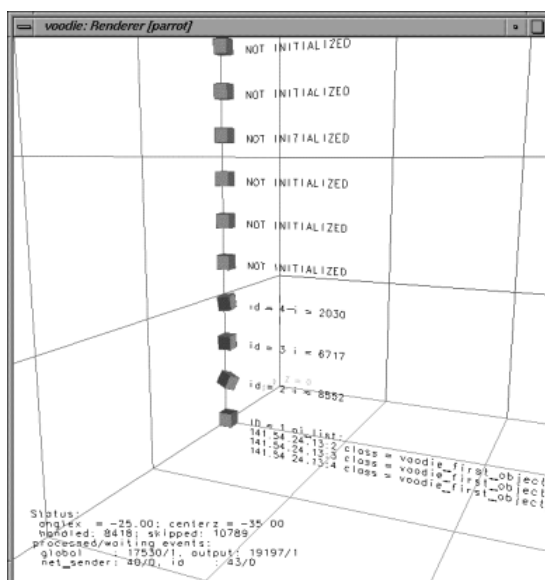


Abbildung 14: Prototyp, OI-Objekt (Objekt id = 1) mit OI-Liste

Das Abonnement dieser Objekte erfolgt dadurch, daß dem OI-Objekt entsprechende `SUBSCRIBE_OI` Events zugesandt werden. Die Übergabe der Objektkontrolle von einem Objekt einer Objektgruppe zu einem anderen, kann getestet werden, indem eine Objektgruppe erzeugt wird und dem Masterobjekt ein `STATE = EXIT` Event geschickt wird. Das Masterobjekt gibt dann die Kontrolle ab, weil es davon ausgeht, demnächst gelöscht zu werden.

Der Prototyp generiert eine Protokoll-datei. In dieser Protokolldatei werden die Aktivitäten des Programmes und Debug Informationen festgehalten. Der Umfang der zu protokollierenden Daten ist dabei über die Kommandozeile einstellbar.

*Doch ich will diesen Weg zu Ende gehn
Und ich weiß wir werden die Sonne sehn ...*

*Wenn die Nacht am tiefsten
aus: „Wenn die Nacht am tiefsten“
Ton Steine Scherben, (1975)*

5 Zusammenfassung / Ausblick

Die Entwicklung eines Systems für verteilte virtuelle Umgebungen stellt eine komplexe Aufgabe dar. In der vorliegenden Arbeit wurden Mechanismen vorgestellt, die als Basis für den Aufbau eines solchen Systems dienen könnten.

Der Abonnementmechanismus realisiert die Möglichkeit, Objekte einer virtuellen Umgebung zu replizieren und damit für mehrere Teilnehmer zur Verfügung zu stellen. Der Mechanismus zum Übergang der Objektkontrolle dient zum Aufbau dynamischer Umgebungen bei den einzelnen Teilnehmern, die jeweils nur relevante Objekte enthalten. Das Konzept der OI wurde eingeführt, um eine vollständige Replikation der virtuellen Umgebung bei allen Teilnehmern zu umgehen.

Die Implementation dieser Mechanismen stellt aber noch kein nutzbares System dar. Der Prototyp ist als Machbarkeitsstudie zu verstehen, mit dessen Hilfe qualitative Aussagen zur Tauglichkeit der Mechanismen für den Einsatz in

verteilten virtuellen Umgebungen gefunden werden sollen. Diese Tauglichkeit wurde bestätigt.

Die Implementierung kann als Grundstein für die Entwicklung eines arbeitsfähigen Systems benutzt werden. Portierungen auf andere Systeme sind notwendig, der Prototyp ist im jetzigen Stadium aber noch nicht für direkte Weiterverwendbarkeit ausgelegt.

Neben der praktischen Umsetzung der Mechanismen wurde mit dem Prototyp auch das Ziel verfolgt, Probleme zu erkennen und abzugrenzen. Diese Probleme wurden im Abschnitt 3 benannt und müssen durch weitere Arbeit am Projekt gelöst werden.

Die am Anfang der Arbeit aufgestellten Kriterien für verteilte virtuelle Umgebungen können auf den beschriebenen Lösungsansatz angewandt werden. Die Lösung stellt sich dann als ein auf Punkt-zu-Punkt Kommunikation basierendes System dar. Diese Art der Kommunikation wurde gewählt, um das System von Anfang an möglichst dezentral aufzubauen. Die Nachteile der Punkt-zu-Punkt Kommunikation werden durch den Einsatz bedingter Abonnierung und dynamische Optimierung der Kommunikationstopologie entschärft. Für diese Punkte steht eine praktische Umsetzung noch aus, es wurden lediglich Ansätze vorgeschlagen.

Der Verteilungsmechanismus des Systems sieht vor, daß die Daten (Objekte) dezentral verwaltet werden und eine Replikation nur bei Bedarf erfolgt. Nach Tabelle 1 (auf Seite 14) wird also ein DeD-RepNB Mechanismus verwendet.

Die Arten der Kommunikation sind durch das System nicht statisch vorgegeben, sondern können durch die Implementation der Objekte dynamisch beeinflußt werden. Dies ermöglicht eine flexible Aufteilung in statische und dynamische Daten, die an die jeweiligen Erfordernisse angepaßt werden kann.

Für die Lösung des Interaktionsproblems wurden Ansätze vorgestellt, von denen die separate Definition von Interaktionen am vielversprechendsten aussieht.

Ob Voodie die speziellen Anforderungen an verteilte virtuelle Umgebungen wirklich unterstützt, muß durch quantitative Untersuchungen an einer voll funktionsfähigen Implementation entschieden werden. Quantitative Untersuchungen sind im jetzigen Stadium der Entwicklung wenig sinnvoll, da der Prototyp erstens unvollständig ist und zweitens Optimierungen an den Modulen notwendig sind. Der Prototyp zeigt aber bereits, daß die Mechanismen durchaus in der Lage sind, mehreren Benutzern eine konsistente und verzögerungsfreie Sicht auf die Objekte zu bieten. Durch die dezentrale Struktur des Systems, in Verbindung mit den oben beschriebenen Mechanismen zur bedingten Abonnierung und zur Optimierung der Kommunikationstopologie, ist auch der Aufbau großer Umgebungen für viele Benutzer möglich.

Die Vorteile des System liegen in seiner dezentralen Struktur und in der Grundidee, die Mechanismen auf der Basis der Anforderung nach Bedarf (Request by demand) aufzubauen. Der Prototyp ist unvollständig, es fehlen einige Mechanismen und die entsprechenden Lösungsvorschläge sind nur in Ansätzen

vorhandenen. Durch die Komplexität der Aufgabe ist aber eine umfassende Lösung im Rahmen einer Diplomarbeit ausgeschlossen.

Um daß System soweit weiterzuentwickeln, daß es unter realen Bedingungen getestet werden kann, müssen die Ein- und Ausgabemodule verbessert, die low-Level Netzwerkkommunikation optimiert und die Mechanismen zur bedingten Abonnie rung umgesetzt werden. Die Integration einer Interaktionserkennung und Auflösung ist ebenso notwendig, wie die Implementation von komplexeren Beispielobjekten.

Wenn es gelingt, die fehlenden Mechanismen umzusetzen, können mit Voodie verteilte virtuelle Umgebungen geschaffen werden, bei denen die Daten vollständig dezentral verteilt sind. Die Größe und Komplexität der gesamten virtuellen Umgebung wird nicht durch prinzipielle Systemeigenschaften begrenzt, sondern durch die Leistungsfähigkeit der verwendeten Hardware, die Güte der Übertragungskanäle und durch die Implementation der Objekte.

Obwohl die vorgeschlagenen Methoden teilweise auch in anderen Systemen Verwendung fanden, ist der beschriebene Ansatz als neue Möglichkeit zur Kommunikation in verteilten virtuellen Umgebungen zu betrachten. Die dezentrale Datenhaltung und das Request by Demand Prinzip sind prinzipiell am besten für den Aufbau komplexer verteilter virtueller Umgebungen geeignet. Die vollkommen dezentrale Organisation einer verteilten virtuellen Umgebung wurde in keinem der untersuchten Systeme umgesetzt.

Durch die separate Definition von eigenständigen Objekten ist die Trennung der Basis- und Servicefunktionalitäten von den eigentlichen Objekten (also dem Inhalt der virtuellen Umgebung) möglich. Wenn diese Trennung im weiteren Projekt fortgeführt werden kann, so ist in Verbindung mit der Umsetzung und Optimierung der Mechanismen ein äußerst flexibles Basissystem für verteilte virtuelle Umgebungen realisierbar.

... *hier is, was ich suche* ...

Im Süden
aus: „Über Alles“
Rio Reiser (1994)

6 Literaturverzeichnis

- /APP92/ P. A. Appino, J. B. Lewis, L. Koved, D.T. Ling, D. A. Rabenhorst, C. F. Cordella: *An Architecture for Virtual Worlds*. Presence, Vol. 1, No. 1, Winter 1992, pp. 1-17
- /BER94/ J. E. Berger, L. T. Dinh, M. F. Masiello, J. N. Schell: *NVR: A System for Networked Virtual Reality*. in Proceedings of the International Conference on Multimedia Computing and Systems, Boston, Massachusetts, May 1994

- /BLO92/ J. Bloomer: *Power Programming with RPC*. O'Reilley & Associates 1992; BIM²⁰: 24.542.- 126
- /BRI94/ W. Bricken, G. Coco: *The VEOS Project*. Presence Vol. 3, No. 2, pp. 111-129, Spring 1994
- /BRO94/ C. Brown: *UNIX Distributed Programming*. Prentice Hall 1994; BIM: 24.542.- 164(2)
- /CAR93/ Carlsson, Hagsand: *DIVE - a Multi-User Virtual Reality System*. in Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS) September 1993, (Seattle, WA), pp. 394-400
- /COD92/ C. Codella, R.Jalili, L. Koved, J. B. Lewis, D. T. Ling, J. S. Lipscomb, D. A. Rabenhorst, C. P. Wang: *Interactive Simulation in a Multi-Person Virtual World*. in Proceedings of CHI '92, pp. 329-334
- /COD93/ C. Codella, R.Jalili, L. Koved, J. B. Lewis: *A Toolkit for Developing Multi-User, Distributed Virtual Environments*. in Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS) September 1993, (Seattle, WA), pp. 401-407
- /CRI91/ C. Grimsdale: *dVS Distributed Virtual Environment System*. Computer Animation, virtual reality, Visualisation 1991; Blenheim Online, Pinner, Middlesex, pp.163-170
- /DIS/ IEEE 1278 series: *Distributed Interactive Simulation*
- /EXPR/ ISO Norm 10303 (STEP); Part 11, Part 21-26
- /FUN95/ T. A. Funkhouser: *RING: A Client-Server System for Multi-User Virtual Environments*. in Proceedings of 1995 Symposium on Interactive 3D Computer Graphics, Monterey CA USA, pp. 85-92
- /GIB87/ W. Gibson: *Neuromancer*. Heyne Verlag München 1987, ISBN: 3-453-05665-5
- /GRE95/ C. Greenhalgh, S. Benford: *MASSIVE: a Distributed Virtual Reality System Incorporating Spatial Trading*. in Proceedings of the 15. International Conference on Distributed Computing, Vancouver, Can 1995
- /HAL96/ B. Hall: *Beej's Guide to Network Programming*. 1995,1996
<http://www.ecst.csuchico.edu/~beej/guide/net>

²⁰ Katalognummer der Bibliothek der Bauhaus-Universität Weimar, Zweigstelle Informatik und Mathematik, Coudraystraße 7, Weimar

- /HEN96/ D. Hennessey, A. Patterson: *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers Inc. ; San Francisco, California; 1990, 1996; ISBN 1-55860-329-8
- /JOS96/ N. Josuttis: *Die C++ Standardbibliothek, Eine detaillierte Einführung in die vollständige ANSI/ISO-Schnittstelle*. Addison-Wesley 1996, BIM: 24.535.8.-110
- /KAL93/ R. S. Kalawsky: *The Science of Virtual Reality and Virtual Environments*. Addison Wesley 1993, ISBN: 0-201-63171-7, BIM: 24.645.- 31
- /KAT96/ R. Katz: *Computer Architecture*. Course in Computer Science; University of California, Berkeley; Lec. 2, p 8.
<http://www.cs.berkeley.edu/~randy/Courses/CS252.S96/CS252.Intro.html>
- /KAZ93a/ R. Kazman: *Making WAVES: On the Design of Architectures for Low-end Distributed Virtual Environments*. in Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS) September 1993, (Seattle, WA), pp. 443-449.
<http://www.cgl.uwaterloo.ca/~rnkazman/HCI-papers.html>
- /KAZ93b/ R. Kazman: *Load Balancing, Latency Management and Separation of Concerns in a Distributed Virtual World*. unpublished Paper, 1993
<http://www.cgl.uwaterloo.ca/~rnkazman/HCI-papers.html>
- /KAZ95/ R. Kazman: *HIDRA: An Architecture for Highly Dynamic Physically Based Multi-Agent Simulations*. International Journal in Computer Simulation, 5, 1995, pp. 149-164.
<http://www.cgl.uwaterloo.ca/~rnkazman/HCI-papers.html>
- /KIL96/ M. J. Kilgard: *OpenGL Programming for the X-Window System* Addison Wesley Developer Press 1996
- /LAM78/ L. Lamport: *Time, Clocks, and the Ordering of Events in a distributed System*. Communications of the ACM; Volume 21, Number 7, July 1978
- /LEE94a/ T. Berners Lee: *Uniform Resource Identifiers in WWW*. RFC 1630; CERN 1994
<http://www.w3.org/pub/WWW/Addressing/rfc1680.txt>
- /LEE94b/ T. Berners Lee, L. Masinter, M. McCahill: *Uniform Resource Locators (URL)*. RFC 1738; 1994
<http://www.w3.org/pub/WWW/Addressing/rfc1738.txt>

- /LIN96/ P. van der Linden: *Java pur, Hintergründe und Entwicklung*. Heinz Heise Verlag 1996, ISBN: 3-88229-086-2, BIM: 24.78.- 273
- /LOO91/ P. L. Loo: *The Starship Manual (Version 2.0)*. ISS TR#91-54-0. Institute of Systems Science. National University of Singapore, Kent Ridge, Singapore 0511; 1991
- /LOO91a/ D. E. Loomer: *Internetworking with TCP/IP, Vol 1, Principles, Protocol and Architecture*. Chapter 11; Prentice Hall 1991 ISBN: 0-13-468505-9, BIM: 24.78.-.138.1
- /LOO91b/ D. E. Loomer: *Internetworking with TCP/IP, Vol 1, Principles, Protocol and Architecture*. Chapter 17; Prentice Hall 1991 ISBN: 0-13-468505-9, BIM: 24.78.-.138.1
- /MAC94/ M. R. Macedonia, M. J. Zyda, D. R. Pratt, P. T. Barham, S. Zeswitz: *NPSNET: A Network Software Architecture for Large Scale Virtual Environments*. Presence, Vol. 3, No. 4, Fall 1994, pp. 265-287
- /MEI96/ M. Meister, Y. Benger: *Vademekum für das Internet*. Bauhaus-Universität Weimar 1996, ISBN: 3-86068-046-3
- /NEI93/ J. Neider, T. Davis, M. Woo: *OpenGL Programming Guide*. Addison Wesley 1993, ISBN: 0-201-63274-8, BIM: 24.737.-35:1
- /NIC96/ B. Nichols, D. Buttlar, J. Proulx Farrwell: *Pthreads Programming*. O'Reilly & Associates 1996, BIM: 24.542.- 217
- /NSF92/ NSF Invitational Workshop, 23.-24. 3. 1992: *Research direction in virtual Environments*. Report of an NSF Invitational Workshop; in Computer Graphics Volume 26; Number 3 August 1992; pp 153-177.
- /POP93/ S.T. Pope, L. E. Fahlen: *The Use of 3-D Audio in a Synthetic Environment: An Aural Renderer for a Distributed Virtual Reality System*. in Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS) September 1993, (Seattle, WA), pp. 176-182
- /PUS84/ L. F. Pusch: *Das Deutsche als Männersprache*. Suhrkamp Verlag Frankfurt/Main 1984, ISBN: 3-518-13324-1
- /RAG93/ S. A. Rago: *UNIX System V Network Programming*. Addison-Wesley 1993
- /REG94/ H. Regenbrecht: *Virtuelle Realität und Design*. Diplomarbeit. Hochschule für Architektur und Bauwesen Weimar – Universität 1994
- /SCH95/ D. Schmalstieg, M. Gervautz: *Towards a Virtual Environment for interactive World Building*. MVD '95, Modelling – Virtual Worlds – Distributed Graphics, Bad Honnef / Bonn, Germany; Nov. 1995

- /SCH96/ D. Schmalstieg, M. Gervautz: *Demand-Driven Geometry Transmission for Distributed Virtual Environments*. Proceedings of EUROGRAPHICS '96, Computer Graphics Forum Volume 15, Number 3, pp. C-421- 432
- /SHA92/ C. Shaw, J. Liang, M. Green, Y. Sun: *The decoupled Simulation Model for Virtual Reality Systems*. Human Factors in Computing Systems, CHI'92, Conference Proceedings, Monterey, California, pp 321-328; May 1992
- /SHA93a/ C. Shaw, J. Liang, M. Green, Y. Sun: *Decoupled Simulation in Virtual Reality with the MR Toolkit*. ACM Transactions on Information Systems 11; ACM SIGCHI ; July 1993
- /SHA93b/ C. Shaw, M. Green: *The MR Toolkit Peers Package and Experiment*. in Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS) September 1993, (Seattle, WA), pp. 463-469
- /SIN94/ G. Singh, L. Serra, W. Png, H. Ng: *BrickNet: A Software Toolkit for Network-Based Virtual Worlds*. Presence Vol. 3, No. 1, Winter 1994, pp. 19-34
- /SIN95/ S. K. Singhal, D. R. Cheriton: *Exploiting Position History for Efficient Remote Rendering in Networked Virtual Reality*. Presence, Vol. 4, No. 2, Spring 1995, pp 169-193
- /SNO94/ D. N. Snowdon, A. J. West: *AVIARY: Design Issues for future Large-Scale Virtual Environments*. Presence, Vol. 3, No. 4, Fall 1994, pp. 288-308
- /STE92/ W. R. Stevens: *Programmieren von UNIX-Netzen, Grundlagen, Programmierung, Anwendung*. Hanser, Prentice Hall 1992
- /STL96/ *Standard Template Library Programmer's Guide*. Silicon Graphics Inc.; 1996
<http://www.sgi.com/Technology/STL/>
- /SUT65/ I. Sutherland: *The ultimate display*. Proceedings of the IFIP Congress; pp. 506-508; 1965
- /VIN95/ J. Vince: *Virtual Reality systems*. ACM Press; 1995, ISBN: 0-201-87687-6, BIM: 24.645.- 37
- /WAN95/ Q. Wang, M. Green, C. Shaw: *EM - An Environment Manager For Building Networked Virtual Environments*. in Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS) 1995, Seattle, WA; Chapter 29, pp. 11-18
- /WRI95/ G. R. Wright, W. R. Stevens: *TCP/IP Illustrated, Volume 2, the Implementation*. Addison-Wesley 1995; ISBN: 0-201-63354-X; BIM: 24.78.- 151:2
- /ZUS93/ K. Zuse: *Der Computer mein Lebenswerk*. Springer Verlag 1993, ISBN: 3-540-56292-3, BIM: 24.00.- 22

Anhang A Glossar

Abomasterobjekt

Jedes Objekt einer Objektgruppe kann Abomasterobjekt sein und eine gewisse Anzahl Abonnenten besitzen. Ein Abomasterobjekt sendet an seine Abonnenten regelmäßig Synchronisationsevents, wenn es selbst das Masterobjekt der Objektgruppe ist, sonst werden empfangene Synchronisationsereignisse einfach an die Abonnenten weitergeleitet.

Abonnementmechanismus

Synchronisationsmechanismus zur Replikation von Objekten. Ein Hauptobjekt (Abomasterobjekt) versorgt dabei seine Abonnenten mit den notwendigen Daten.

Abonntentobjekt

Jedes Slaveobjekt einer Objektgruppe ist automatisch Abonnent eines Abomasterobjektes.

Avatar

Stellvertreterobjekt für einen Nutzer in der virtuellen Umgebung, wird für dessen Visualisierung benötigt. In virtuellen Umgebungen ist der Nutzer integraler Bestandteil des Systems. Deshalb muß der Nutzer eine Repräsentation (einen Avatar) im System besitzen.

Broadcast

Broadcast ist eine Methode, bei der alle existierenden Stationen in einem (Teil-) Netz eine einmalig abgeschickte Nachricht empfangen können. Wesentlich ist, daß die Nachricht dabei lediglich einmal gesendet wird und alle Stationen die Nachricht gleichzeitig empfangen. TCP/IP realisiert Broadcastnachrichten über spezielle Broadcastadressen.

Console_listener

Dieses Verwaltungsmodul stellt die Möglichkeit zur Verfügung, über die Tastatur Ereignisse an das System zu übergeben.

Dead Reckoning

Mit Dead Reckoning wird eine Methode bezeichnet, die es ermöglicht, Synchronisationen in verteilten Umgebungen einzusparen. Dabei berechnen zwei autonome Objekte ihr zukünftiges Verhalten aus ihrem aktuellen Zustand. Zur Synchronisation reicht dann in der Regel die Übertragung der Zustandsdaten in einem größeren Zeitraster aus. Muß ein Objekt beispielsweise zur flüssigen Darstellung 25 mal pro Sekunde seinen aktuellen Zustand anzeigen, so sind

ohne Dead Reckoning die Zustandsdaten 25 mal pro Sekunde zu übertragen. Wird Dead Reckoning eingesetzt, kann ein Objekt seinen Zustand auch dann aktualisieren, wenn keine Synchronisation der Zustandsdaten erfolgt. Das Objekt ändert dann seinen Zustand selbständig aufgrund definierter Regeln.

DOF

DOF steht für degrees of freedom und bezeichnet die Freiheitsgrade die ein Eingabegerät besitzt. Ein Eingabegerät besitzt 6 DOF wenn z.B. Bewegungen in jede Raumrichtung und Drehung um die drei Achsen möglich sind. Eine Standardcomputermaus besitzt lediglich 2 DOF.

Event

Events oder Ereignisse sind die Basis der Kommunikation im Voodie System. Ein Ereignis wird als Nachricht von einem Absender an einen Empfänger geschickt. Ereignis und Nachricht stehen also in einem gewissen Zusammenhang. Ereignisse im Voodie System beinhalten einen Ereignistyp und einen Nachrichtenteil, so daß prinzipiell auch von Nachrichten gesprochen werden könnte. Darauf wurde aber aus Gründen der Übersichtlichkeit verzichtet. Absender und Empfänger müssen nicht auf derselben Station residieren. Der Transport der Ereignisse erfolgt, durch die Verwaltungsmodule des Voodie Systems, für Sender und Empfänger transparent.

Interaktionsdämon (ID)

Der Interaktionsdämon ist ein Verwaltungsmodul im Voodie System. Er erzeugt, verwaltet und zerstört Objekte, überwacht und regelt Interobjektkommunikationen (z.B. Kollisionserkennung) und leitet Ereignisse an die entsprechenden Objekte weiter. Jede Station besitzt einen ID. Er ist immer an einen Teilnehmer oder eine Station gekoppelt. Dabei werden vom ID lediglich die Objekte verwaltet, die sich in einer gewissen räumlichen Umgebung um sich selbst befinden. Der ID ist beweglich.

Masterobjekt

Das Masterobjekt einer Objektgruppe ist dasjenige Objekt, welches die Objektkontrolle innehat.

Multicast

Multicast (auch IP-Multicast) ist eine Methode, die es ermöglicht, Daten an mehrere Empfänger gleichzeitig abzusetzen, ohne dabei jeden Empfänger einzeln anzusprechen. Dabei werden aber nicht wie beim Broadcast alle existierenden Empfänger angesprochen, sondern lediglich einige. Welche Empfänger angesprochen werden, wird über Multicastgruppen festgelegt.

Mutex

Mutex ist ein Kunstwort aus Mutual und Exclusion (gegenseitiger Ausschluß).

Eine Mutex wird in Systemen benötigt, bei denen eine Ressource (z.B. Variable, Gerät u.a.) von verschiedenen Modulen abwechselnd benutzt werden soll. Der Teil der Instruktionen, in dem Benutzung der Ressource erfolgt, wird als kritischer Abschnitt (critical section) bezeichnet. Eine Mutex wird dabei von jedem der Module am Anfang eines kritischen Abschnitts gesperrt, so daß kein anderes Modul auf die Ressource zugreifen kann. Am Ende des kritischen Abschnitts wird die Mutex wieder freigegeben.

Net_listener

Dieses Verwaltungsmodul des Voodie Systems regelt den Empfang von Daten über das Netzwerk. Diese Daten werden im NET_LISTENER Modul in Ereignisse umgewandelt und weitergegeben.

Net_sender

Dieses Verwaltungsmodul des Voodie Systems ist für das Senden von Daten über das Netzwerk zuständig. Diese Daten sind generell Ereignisse. Das NET_SENDER Modul setzt die Ereignisse dabei entsprechend um.

Objekt

Allgemein sind Objekte in virtuellen Umgebungen autonom agierende Einheiten, die die Umgebung mit Leben füllen. Sowohl Gegenstände, als auch Avatare sind Objekte einer virtuellen Umgebung.

Aus Anwendersicht ist ein Objekt im Voodie System eine eigenständig agierende Einheit, die durch Senden von **RENDER** Events in der Lage ist, sich darzustellen. Solche Objekte sind Voodie Objekte. Wenn nicht ausdrücklich von der Implementation in C++ geredet wird, bezeichnet der Begriff Objekt ein Voodie Objekt bzw. ein Objekt in der virtuellen Umgebung (z.B. Tisch, Ball, Avatar o.ä.).

Da das Voodie System in einer objektorientierten Programmiersprache implementiert ist, sind Voodie Objekte auch aus der Sicht des Programmierers Objekte. Neben den Voodie Objekten sind aber auch alle Verwaltungsmodule als Objekte implementiert.

Objektabonnement

siehe Abonnementmechanismus.

Objektgruppe

Objekte im Voodie System besitzen u.U. Repräsentationen auf anderen beteiligten Stationen. Mit Objektgruppe ist die Gesamtheit dieser Repräsentationen eines Objektes gemeint. Eine Objektgruppe besteht aus einem oder mehreren Objekten, die voneinander abhängig sind, sich also synchronisieren müssen. Eine Objektgruppe setzt sich aus einem Masterobjekt und keinem oder mehr Slaveobjekten zusammen.

Objektidentifikator (ObjektID)

Jedes Objekt im Voodie System besitzt einen eindeutigen Identifikator. Dieser Identifikator sieht allgemein so aus:

<hostname> : <Objektnummer>

<hostname> ist dabei die Adresse der Station, auf der das Objekt residiert und <Objektnummer> steht für die stationseindeutige Nummer des Objektes

Objektinformigramm (OI)

Ein Objektinformigramm stellt eine Art Abbild eines Objektes dar, welches vorzugsweise dazu benutzt werden soll, die Informationen eines Objektes über das Netzwerk anderen Teilnehmern zur Verfügung zu stellen, ohne das Objekt selbst übertragen zu müssen. Es enthält wichtige Informationen eines Objektes. Derzeit sind diese Informationen der Identifikator des Objektes (ObjektID) und die Klasse (den Typ) des Objektes

In späteren Implementationen sollten weitere Informationen (z.B.: Erzeugungsparameter oder allgemeine Geometrieinformationen) im OI enthalten sein. Mit Hilfe der OI kann ein Teilnehmer entscheiden, welche Objekte für ihn relevant sind und diese Objekte dann abonnieren.

OI-Objekt, OI-Verwalter

Im Voodie System existiert ein spezielles Objekt, welches die Verwaltung der OI übernimmt. Jede Station besitzt ein solches OI-Objekt.

RENDER Event

Objekt einer virtuellen Umgebung müssen die Möglichkeit der Visualisierung besitzen. **RENDER** Events sind Ereignisse, die ein Objekt im Voodie System an das Visualisierungsmodul bzw. die Renderapplikation sendet. Dort wird das Ereignis entsprechend verarbeitet.

Renderer

Dieses Verwaltungsmodul des Voodie Systems ist der Empfänger für **RENDER** Events.

SEND_ALL_OI Event

Ein spezielles Ereignis zur Anforderung aller verfügbaren OI eines Teilnehmers.

Slaveobjekt

Ein Slaveobjekt ist ein Objekt einer Objektgruppe, welches nicht die Objektkontrolle besitzt.

Station

Allgemein versteht man unter einer Station einen Rechner, der Bestandteil

einer verteilten Anwendung ist. In Bezug auf die vorliegende Implementierung ist eine Station die Einheit aus Computer, Netzwerkanschluß und Voodie Applikation. Dabei kann (in der jetzigen Implementierung) auf einem Computer mit einem Netzwerkanschluß nur eine Voodie Applikation laufen.

TCP

Transmission Control Protocol. Verbindungsorientiertes Protokoll, welches auf UDP aufsetzt. TCP stellt Mechanismen für Fehlerkorrektur und Sequenzkontrolle zur Verfügung. Der dadurch entstehende Mehraufwand führt zu geringerem Datendurchsatz.

UDP

User Datagram Protocol. Ein paketorientiertes, verbindungsloses Protokoll zur Datenübertragung. UDP beinhaltet keine Mechanismen zur Fehlerkorrektur. Es ist auch möglich, daß UDP Datagramme in einer anderen Reihenfolge beim Empfänger ankommen, als der Sender sie abgeschickt hat. Mit UDP ist dadurch aber auch ein höherer Datendurchsatz möglich als mit verbindungsorientierten Protokollen (z.B. TCP).

virtuelle Umgebung, verteilte virtuelle Umgebung

In dieser Arbeit wird unter einer virtuellen Umgebung ein System verstanden, welches die dreidimensionale Darstellung und Modellierung der Umgebung, deren interaktive Manipulation in Echtzeit und den Nutzer als integralen Bestandteil dieser Umgebung beinhaltet.

Eine verteilte virtuelle Umgebung ist eine virtuelle Umgebung, die die gleichzeitige Benutzung durch mehrere geographisch getrennte Teilnehmer erlaubt.

Voodie

Virtual objectoriented distributed interactive environment

Name des Prototypen, der den vorgestellten Mechanismus implementiert und des Projektes an der Fakultät Medien der Bauhaus-Universität Weimar, das sich mit verteilten virtuellen Umgebungen befaßt.

Anhang B Mögliche Events in Voodie

Die folgende Aufstellung enthält alle derzeit im Voodie System intern möglichen Events und eine entsprechende Beschreibung. Dabei sind die Events je nach Modul in empfangene und gesendete Events eingeteilt. Dadurch kommen manche Events mehrfach in der Aufstellung vor.

Events an main; Event Distributor:

Typ:	EXIT	Name:	GLOBAL_EXIT
From Host	LOCAL	To Host	LOCAL
From Module	...	To Module	DISTRIBUTOR
Message	<error_string>		
Beschreibung	Alle Module beenden, Objekte zerstören, Programm beenden.		

Typ:	ERROR	To Host	LOCAL
From Host	LOCAL	To Module	DISTRIBUTOR
From Module	NET_LISTENER		
Message	<error_string>		
Beschreibung	Schwerer Fehler im Netzwerkempfangsmodul aufgetreten, Programm wird beendet. Dieses Event tritt z.B. auf, wenn keine Netzwerkverbindungen hergestellt werden können.		

Typ:	ERROR	To Host	LOCAL
From Host	LOCAL	To Module	DISTRIBUTOR
From Module	NET_SENDER		
Message	<error_string>		
Beschreibung	Schwerer Fehler im Netzwerksendemodul aufgetreten, Programm wird beendet. Dieses Event tritt z.B. auf, wenn keine Netzwerkverbindungen hergestellt werden können.		

von main (event distributor):

Typ:	EXIT	To Host	LOCAL
From Host	LOCAL	To Module	DISTRIBUTOR
From Module	DISTRIBUTOR		
Message	...		
Beschreibung	Dieses Event wird vom Distributor selbst in Folge eines		

schweren Fehlers generiert.

Typ: **EXIT**
 From Host **LOCAL** To Host **LOCAL**
 From Module **DISTRIBUTOR** To Module **NET_SENDER**
 Message ...
 Beschreibung Modul **NET_SENDER** beenden.

Typ: **EXIT**
 From Host **LOCAL** To Host **LOCAL**
 From Module **DISTRIBUTOR** To Module **RENDERER**
 Message ...
 Beschreibung Modul **RENDERER** beenden.

Typ: **EXIT**
 From Host **LOCAL** To Host **LOCAL**
 From Module **DISTRIBUTOR** To Module **ID**
 Message ...
 Beschreibung Modul Interaktionsdämon (**ID**) beenden, vorher alle Objekte löschen.

Typ: ...
 From Host ... To Host **<to_host>**
 From Module ... To Module ...
 Message ...
 Beschreibung Alle Events mit einem Wert **!=LOCAL** im **to_host** werden an das Modul **NET_SENDER** weiter geschickt.

Typ: ...
 From Host ... To Host **LOCAL**
 From Module ... To Module ...
 Message ...
 Beschreibung Alle Events mit dem Wert **LOCAL** im **to_host** Feld werden an das Modul, dessen Name im **to_module** Feld steht, geschickt. Events an das Modul **DISTRIBUTOR** werden verarbeitet (siehe *to main*).

an Net_Listener

Das Net_Listener Modul besitzt keine eigene Eventqueue,

kann deshalb keine Ereignisse empfangen.

von Net_Listener:

Typ: **ERROR**

From Host **<from_host>** To Host **LOCAL**

From Module **NET_LISTENER** To Module **RENDERER**

Message **invalid message format, ignoring message**

Beschreibung Unkritischer Fehler im **NET_LISTENER** Modul aufgetreten, Event geht an **RENDERER**.

Typ: **ERROR**

From Host **LOCAL** To Host **LOCAL**

From Module **NET_LISTENER** To Module **DISTRIBUTOR**

Message **<error_string>**

Beschreibung Kritischer Fehler aufgetreten, z.B. keine Netzwerkverbindung möglich.

Typ: **<type>**

From Host **<from_host>** To Host **LOCAL**

From Module **<from_module>** To Module **<to_module>**

Message **<data>**

Beschreibung Aus den empfangenen Datenpaketen werden Events generiert. Dabei wird die Adresse in **to_host** durch **LOCAL** ersetzt. Die Ereignisse werden an den **DISTRIBUTOR** weitergegeben.

an Net_Sender:

Typ: **<type>**

From Host **<from_host>** To Host **<to_host>**

From Module **<from_module>** To Module **<to_module>**

Message **<data>**

Beschreibung Aus dem Event werden Datenpakete generiert, die über das Netzwerk an die Adresse in **to_host** geschickt werden.

Typ: **EXIT**

From Host **LOCAL** To Host **LOCAL**

From Module **DISTRIBUTOR** To Module **NET_SENDER**

Message	...
Beschreibung	Es werden keine weiteren Events mehr verschickt, das Modul wird beendet.

von Net_Sender:

Typ:	ERROR		
From Host	LOCAL	To Host	LOCAL
From Module	NET_SENDER	To Module	DISTRIBUTOR
Message	<error_string>		
Beschreibung	Ein kritischer Fehler ist aufgetreten, z.B. keine Netzwerkverbindung möglich.		
Typ:	ERROR		
From Host	LOCAL	To Host	LOCAL
From Module	NET_SENDER	To Module	RENDERER
Message	<error_string>		
Beschreibung	Ein unkritischer Fehler ist aufgetreten, z.B. Event konnte nicht gesendet werden, weil die Zielstation nicht antwortet.		

an Console:

das **CONSOLE** Modul besitzt keine eigene Eventqueue.

von Console:

Typ:	<type>		
From Host	LOCAL	To Host	<to_host>
From Module	INPUT	To Module	<to_module>
Message	<data>		
Beschreibung	Über den Standardeingabekanal können Events eingegeben werden. Das CONSOLE Modul generiert aus den Eingaben Events und sendet an den DISTRIBUTOR .		

an Renderer:

Typ: ...

From Host	...	To Host	...
From Module	...	To Module	...
Message	...		
Beschreibung	Alle Events werden direkt auf den Standardausgabekanal ausgegeben.		
Typ:	EXIT		
From Host	LOCAL	To Host	LOCAL
From Module	DISTRIBUTOR	To Module	RENDERER
Message	...		
Beschreibung	Dieses Event veranlaßt den RENDERER zusätzlich, seine Arbeit zu beenden.		

von Renderer:

das Rendermodul generiert keine Events.

an ID (Interaktionsdämon):

Typ:	EXIT		
From Host	LOCAL	To Host	LOCAL
From Module	DISTRIBUTOR	To Module	ID
Message	...		
Beschreibung	Dieses Event veranlaßt den Interaktionsdämon, alle Objekte zu löschen und seine Arbeit zu beenden.		

Typ:	DATA	Name:	CREATE_OBJECT
From Host	...	To Host	LOCAL
From Module	...	To Module	ID
Message	CREATE_OBJECT CLASS = <class_name>		
Beschreibung	Der ID generiert ein neues Objekt der Klasse <class_name> .		

Typ:	DATA	Name:	CREATE_ABO_OBJECT
From Host	...	To Host	LOCAL
From Module	...	To Module	ID
Message	CREATE_OBJECT CLASS = <class_name> MASTER_ID =		

<hostname>: <num>	
Beschreibung	Der ID generiert ein neues Objekt der Klasse <class_name> . Dem Objekt wird dann ein MASTER_ID = ... Event geschickt. Dieses Objekt wird dadurch ein Abbonnentobjekt des Objektes, dessen ObjektID hinter MASTER_ID = steht.
Typ:	DATA Name: DESTROY_OBJECT
From Host	... To Host ...
From Module	... To Module ID
Message	DESTROY_OBJECT ID = <num>
Beschreibung	Das Objekt mit der ObjektID <num> wird gelöscht.
Typ:	DATA Name: SEND_ALL_OI
From Host	... To Host ...
From Module	... To Module ID
Message	SEND_ALL_OI
Beschreibung	Wenn der ID ein solches Event erhält, sendet er an alle Objekte ein SEND_OI Event. from_module und from_host werden nicht verändert.

von ID:

Der Interaktionsdämon sendet keine Events, sondern leitet Events für die Objekte direkt weiter, indem die entsprechende `handle_event()` Routine aufgerufen wird.

an OBJECTS (diese Events werden auch vom ID bearbeitet, indem sie an die entsprechenden Objekte weitergegeben werden):

Typ:	DATA
From Host	... To Host ...
From Module	... To Module OBJECTS
Message	id = <num> ...
Beschreibung	Solche Events sendet der ID sofort an das Objekt mit der ObjektID <num> indem die entsprechende <code>handle_event()</code> Routine aufgerufen wird.
Typ:	DATA
From Host	... To Host ...
From Module	... To Module OBJECTS
Message	id = 1 SHOW_LIST

Beschreibung Spezielles Event für das OI-Objekt. Das OI-Objekt wird angewiesen, die aktuelle Liste der OI als Event an den **RENDERER** zu schicken.

Typ:	DATA		
From Host	...	To Host	...
From Module	...	To Module	OBJECTS
Message	id = 1 OI ID = <hostname>:<num> CLASS = <class_name> STATE = <state>		

Beschreibung Spezielles Event für das OI-Objekt: Ein OI ist eingetroffen. Das OI-Objekt nimmt den OI in die OI Liste auf.

Typ:	DATA	Name:	SEND_OI
From Host	...	To Host	...
From Module	...	To Module	OBJECTS
Message	id = <num> SEND_OI		

Beschreibung Das Objekt soll seinen OI schicken. Der OI wird an das Objekt 1 (OI-Objekt) der Station geschickt, deren IP-Adresse als Absender im Event steht (**from_host**).

Typ:	DATA		
From Host	...	To Host	...
From Module	...	To Module	OBJECTS
Message	id = <num> MASTER_ID = <ObjektID>		

Beschreibung Das Objekt mit der ID **<num>** wird aufgefordert, seine **Master_id** (Abomaster) neu zu belegen. Das Objekt muß dafür sorgen, daß

- der alte Abomaster ein **UNSUBSCRIBE** Event erhält,
- der neue Abomaster ein **SUBSCRIBE** Event erhält,
- der Status des Objektes auf **SLAVE** gesetzt wird, wenn dies noch nicht der Fall ist.

Typ:	DATA	Name:	SUBSCRIBE_OBJECT
From Host	...	To Host	...
From Module	...	To Module	OBJECTS
Message	id = <num> SUBSCRIBE id = <ObjektID>		

Beschreibung Das Abomasterobjekt mit der ID <num> wird durch dieses Event aufgefordert, dem Objekt mit der angegebenen ObjektID regelmäßig **UPDATE** Events zu schicken. Zu diesem Zweck wird die ObjektID in die Abonnentenliste eingetragen.

Typ: **DATA**
 From Host ... To Host ...
 From Module ... To Module **OBJECTS**
 Message **id = <num> REQ_UPDATE id = <ObjektID>**

Beschreibung Ein Abomasterobjekt wird durch dieses Event aufgefordert, dem Objekt mit der angegebenen ObjektID einmalig ein **UPDATE** Event zu schicken.

Typ: **DATA**
 From Host ... To Host ...
 From Module ... To Module **OBJECTS**
 Message **id = <num> UPDATE ...**

Beschreibung Durch dieses Event werden Slaveobjekte aufgefordert, die Daten die nach **UPDATE** in der Message stehen auszuwerten und gegebenenfalls seine Attribute neu zu setzen.

Typ: **DATA**
 From Host ... To Host ...
 From Module ... To Module **OBJECTS**
 Message **id = <num> UNSUBSCRIBE id = <ObjektID>**

Beschreibung Ein Abomasterobjekt wird durch dieses Event aufgefordert, das Objekt mit der angegebenen ObjektID aus der Abonnentenliste zu entfernen.

Typ: **DATA**
 From Host ... To Host ...
 From Module ... To Module **OBJECTS**
 Message **id = <num> REQ_REQ_OWNER id = <ObjektID>**

Beschreibung Ein Slaveobjekt wird durch dieses Event aufgefordert, an das Masterobjekt mit der angegebenen ObjektID ein **REQ_OWNER** Event zu schicken.

Typ: **DATA**
 From Host ... To Host ...
 From Module ... To Module **OBJECTS**
 Message **id = <num> REQ_OWNER id = <ObjektID>**

Beschreibung Das Masterobjekt mit der ObjektID `<num>` wird aufgefordert, die Objektkontrolle an das Objekt mit der angegebenen ObjektID zu übergeben. Dies geschieht durch Senden eines **SET_OWNER** Events.

Typ: **DATA**

From Host ... To Host ...

From Module ... To Module **OBJECTS**

Message **id = <num> SET_OWNER id = <ObjektID>**

Beschreibung Das Slaveobjekt mit der ID `<num>` wird aufgefordert, seinen Status auf **MASTER** zu setzen und die angegebene ObjektID in seine Abonnentenliste aufzunehmen.

Typ: **DATA**

From Host ... To Host ...

From Module ... To Module **OBJECTS**

Message **id = <num> STATE = EXIT**

Beschreibung Das Objekt wird aufgefordert, seinen Status auf **EXIT** zu setzen.

- Wenn das Objekt ein Slaveobjekt ist, sendet es ein **UNSUBSCRIBE** Event an seinen Abomaster.
- wenn das Objekt selbst Abonnenten besitzt, sendet das Objekt an alle Abonnenten ein **MASTER_ID = ...** Event danach zersört sich das Objekt selbst.
- Wenn das Objekt ein Masterobjekt ist, begeht es Selbstmord, wenn es keine Abonnenten besitzt.
- Wenn das Objekt Abonnenten besitzt, sendet es an seinen ersten Abonnenten ein **REQ_REQ_OWNER** Event und existiert weiter.

Typ: **DATA**

From Host ... To Host ...

From Module ... To Module **OBJECTS**

Message **id = <num> STATE = MASTER**

Beschreibung Das Objekt wird aufgefordert, seinen Status auf **MASTER** zu setzen.

Typ: **DATA**

From Host ... To Host ...

From Module	...	To Module	OBJECTS
Message	id = <num> STATE = SLAVE		
Beschreibung	Das Objekt wird aufgefordert, seinen Status auf SLAVE zu setzen.		

von *OBJECTS*:

Typ:	DATA		
From Host	...	To Host	...
From Module	...	To Module	OBJECTS
Message	id = <num> UPDATE ...		
Beschreibung	Dieses Event enthält Synchronisationsinformationen für das Slaveobjekt mit der ID <num> .		

Typ:	DATA		
From Host	...	To Host	...
From Module	...	To Module	OBJECTS
Message	id = <num> REQ_UPDATE id = <my_ObjektID>		
Beschreibung	Es wird ein einmaliges UPDATE Event vom Objekt <num> angefordert.		

Typ:	DATA		
From Host	...	To Host	...
From Module	...	To Module	OBJECTS
Message	id = <num> SUBSCRIBE id = <my_ObjektID>		
Beschreibung	Das Objekt mit der ID <num> wird aufgefordert, regelmäßig UPDATE Events an <my_ObjektID> (also das Event sendende Objekt) zu schicken.		

Typ:	DATA		
From Host	...	To Host	...
From Module	...	To Module	OBJECTS
Message	id = <num> UNSUBSCRIBE id = <my_ObjektID>		
Beschreibung	Das Objekt mit der ID <num> wird aufgefordert, keine UPDATE Events mehr zu senden.		

Typ:	DATA		
From Host	...	To Host	...

From Module	...	To Module	OBJECTS
Message	id = <num> REQ_REQ_OWNER id = <my_ObjektID>		
Beschreibung	Das Objekt mit der ID <num> wird aufgefordert, ein REQ_OWNER Event zu schicken.		
Typ:	DATA		
From Host	...	To Host	...
From Module	...	To Module	OBJECTS
Message	id = <num> REQ_OWNER id = <my_ObjektID>		
Beschreibung	Das Objekt mit der ID <num> wird aufgefordert, ein SET_OWNER Event zu schicken.		
Typ:	DATA		
From Host	...	To Host	...
From Module	...	To Module	OBJECTS
Message	id = <num> SET_OWNER id = <my_ObjektID>		
Beschreibung	Das Objekt mit der ID <num> wird aufgefordert, die Objektkontrolle zu übernehmen. Das sendende Objekt (<my_ObjektID>) soll dabei in die Aboliste aufgenommen werden.		
Typ:	DATA		
From Host	...	To Host	...
From Module	...	To Module	OBJECTS
Message	id = <num> MASTER_ID = <my_ObjektID>		
Beschreibung	Das Objekt mit der ID <num> wird aufgefordert, seinen Status auf Slave zu setzen (wenn noch nicht der Fall) und seinen Abomaster zu ändern.		