Bastian Kleineidam

Verteilte Protokolle zur effizienten Simulation von Mehrkörpersystemen

Diplomarbeit nach einem Thema von Prof. Dr. Günter Hotz
Fachrichtung 6.2 - Informatik
Universität des Saarlandes, Saarbrücken

Saarbrücken, im September 2000

Hiermit versichere ich an Eides Statt, daß ich diese Arbeit nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Saarbrücken, im September 2000

Für meine Mutter

Danksagung

Danken möchte ich Professor Hotz für das sehr interessante Thema dieser Arbeit. Ausgezeichnete Unterstützung und Betreuung habe ich von Elmar Schömer erhalten. Die Studienkollegen Christian Behrens, Sandro Gamkrelidze, Christian Lennerz, Steffen Lohse und Thomas Warken sorgten für eine angenehme und gelockerte Arbeitsatmosphäre und waren mir mit ihren Anregungen und Diskussionen äußerst hilfreich.

Besonders danken möchte ich meinen Eltern, die mich in meinem Tun immer unterstützt haben.

Inhaltsverzeichnis

1. Einführung

1.1. Problemstellung

Diese Arbeit stellt ein effizientes Protokoll zur interaktiven Teilnahme von mehreren Betrachtern einer Mehrkörpersimulation vor. Dieses Protokoll ermöglichte als Ergebnis die Implementierung einer verteilten Anwendung namens LIVINGSILVIA, einem Mehrbenutzersystem mit interaktiver Teilnahme der Betrachter einer dreidimensionalen Szene.

Aus dem aktiven Gebiet der Simulation werden im folgenden Systeme mit einer endlichen Anzahl von Körpern betrachtet, sogenannte Mehrkörpersysteme. Die verwendeten Simulationsalgorithmen legen Wert darauf, daß die Objekte auf Kräfte und die daraus resultierenden möglichen Kollisionen mit anderen Objekten realitätsnah reagieren. Weiterhin fordern wir von allen verwendeten Algorithmen Echtzeitfähigkeit, so daß wir mit der dargestellten Szene interagieren können.

Die hier vorgestellten Algorithmen beschreiben eine mächtige Schnittstelle zur verteilten, interaktiven Manipulation eines Mehrkörpersystems. Mit dieser Schnittstelle ist es mehreren Benutzern von verschiedenen Rechnern aus möglich, gleichzeitig und interaktiv eine dreidimensionalen Szene zu manipulieren. Ein wichtiges Kriterium ist dabei das Laufzeitverhalten, welches direkten Einfluß auf die Möglichkeiten der Interaktion hat. Weitere Eigenschaften der Algorithmen sind Robustheit gegenüber Kommunikationsstörungen, Flexibilität bei Optimierungs- und Strategie-Parametern sowie Plattformunabhängigkeit in der Implementierung.

Das nächste Kapitel wird zunächst Grundlagen von Mehrkörpersystemen erläutern und im besonderen auf die Darstellung der Körper eingehen. Wir führen den Leser kurz in die Aspekte der Kollisionserkennung ein und stellen die verwendeten Algorithmen vor. Danach betrachten wir interaktive, verteilte Anwendungen.

Kapitel 3 stellt verteilte Protokolle vor, welche naturgemäß in verteilten Systemen zur Anwendung kommen. Es werden die verwendete Struktur erläutert sowie mehrere Kommunikationsprotokolle und ihre Eigenschaften diskutiert. Ausgehend davon werden Paradigmen zur Entscheidungsfindung der einzelnen Aspekte Fehlerbehandlung, Strategien zur Konfliktlösung, Synchronisation und Vorhersagbarkeit beschrieben.

Kapitel 4 erläutert die verwendeten Designmethoden und stellt wichtige Programmstrukturen vor.

Schließlich wird in Kapitel 5 auf die Implementierung der Algorithmen ausführlich eingegangen. Es werden die einzelnen Bibliotheken SILVIA, TRANSWIRE, LIVINGSILVIA und die Oberfläche **slwish** behandelt.

Die Ergebnisse der vorangehenden Kapitel und, ausgehend davon, die möglichen Konsequenzen, Erweiterungen und Optimierungen bilden das letzte Kapitel dieser Arbeit.

1.2. Anwendungsgebiete

Simulation von Mehrkörpersystemen findet sich in vielen Bereichen der Wirtschaft wieder und wird dort als *Virtual Reality (VR)* bezeichnet. Wird die Information auf mehrere Rechner verteilt handelt es sich um *Distributed VR*. Ein Großteil dieser Anwendungen manifestiert sich in Videokonferenzsystemen, virtuellen Mehrbenutzerwelten, Netzwerkspielen und Chatsystemen. Letztere besitzen dabei den größten Marktanteil mit einer Gesamtbenutzerzahl von mehreren hunderttausend Personen und enthalten deshalb die am besten getesteten Algorithmen.

Die Kommunikation mit mehreren Personen gleichzeitig, sei diese als Spiel, persönliches oder geschäftliches Gespräch realisiert, läßt sich sogar als allgemeiner Trend der letzten Zeit beobachten. War bis vor wenigen Jahren noch eine direkte Punkt-zu-Punkt-Kommunikation die meist verwendete Kommunikationsart, sind es heute die elektronische Post (*Email*) und Chatsysteme. Diese erlauben eine Kommunikation von und mit mehreren Benutzern gleichzeitig, eine Eigenschaft, welche sich auch viele Simulationssysteme zu eigen machen. Dies erleichtert den Informationsaustausch in diesen Systemen enorm und ermöglicht verteiltes Arbeiten.

1.2.1. Netzwerkspiele

Computerspiele beinhalten immer realistischere beziehungsweise aufwendigere Simulationsalgorithmen. Die Beliebtheit solcher Mehrbenutzerspiele ist sehr hoch. Hier stellen die Benutzer dieser Spiele hohe Ansprüche an die graphische Darstellung, was sich vor allem in der Modellierung der Szenen niederschlägt. Um eine hohe Interaktivität zu erlangen, werden robuste und fehlertolerante Kommunikationsalgorithmen eingesetzt und die Komplexität der Kollisionsalgorithmen niedrig gehalten.

Das wohl wichtigste Problem ist bei Spielen die Integrität der Daten. Um das Mogeln zu verhindern, werden verschlüsselte Protokolle, Prüfsummen und Zufallswerte eingesetzt. Wir werden in dieser Arbeit den Einsatz von solchen Methoden in Abschnitt 3.4.1 erklären, aber nicht implementieren, da sie für unsere Zwecke der Simulation nicht dienlich sind.

Abbildung 1.1.: Ein Ausschnitt des interaktiven Netzwerkspiels Counter-Strike

1.2.2. Virtual Reality

Die Automobilbranche will mittels Simulation von Montagevorgängen und Fahreigenschaften eines Autos die Produktentwicklung unterstützen. So werden die aufwendige Erstellung von Modellen und teure Fahrtests auf Teststrecken eingespart. Diese Vorteile machen sich auch andere Maschinenhersteller, zum Beispiel der Flugzeugbau, zunutze. Die Simulationsvorgänge müssen komplexe reale Vorgänge simulieren und besitzen deshalb sehr hohe Ansprüche an die Realitätstreue der Algorithmen. Die Beschränkung auf nicht-deformierbare Objekte ist vertretbar, ebenso wie eine Polygonalisierung von gekrümmten Flächen.

Eine ähnliche Entwicklung beobachtet man auch in der Medizin. Hier versuchen Ärzte mittels virtuellen Operationssälen, die notwendige Anwesenheit von operierenden Spezialisten einzuschränken, beziehungsweise sogar ganz aufzuheben. Der Herzchirurg sitzt dabei an einem entfernten Ort und sieht auf einem Bildschirm das Operationsgebiet. Die Werkzeuge werden von ihm mittels einer Simulationsvorrichtung bewegt. Hier sind neben einem zuverlässigen Übertragungsprotokoll vor allem die exakte Steuerung und haptisches Feedback von Bedeutung. Die simulierte Umgebung wird hier mittels Kameras übertragen. Zu Ausbildungszwecken kann man sogar auf die Kamera und den Patienten verzichten und versuchen, den Vorgang der Operation komplett zu simulieren. Hier werden besonders hohe Ansprüche, nicht nur an die wirklichkeitsnahe Simulation von starren, sondern auch von deformierbaren Körpern, gestellt. In diesem Bereich ist momentan keine Interaktion in Realzeit möglich, da der Simulationsaufwand bei deformierbaren Körpern zu hoch ist. Alle Algorithmen mit diesen Eigenschaften benötigen eine lange Vorberechnung, um realistische Ergebnisse zu erzielen.

Weitere Anwendungen finden sich in der Filmbranche zur computergestützten Erweiterung und Erstellung von real wirkenden Filmszenen. In speziellen Räumen gedrehte Szenen werden im Computer aufgearbeitet. Es wurden auch schon komplette Filme im Computer erstellt [Las99]. Die Modellierung einer virtuellen Umgebung ist dabei der erste Schritt, danach werden die Objekte erstellt und animiert. Diese Animation kann nun simuliert werden, wobei hier gilt: je realistischer die Simulation gelungen ist, desto weniger muß diese Szene nachbearbeitet werden.

2. Grundlagen

2.1. Mehrkörpersysteme

Mehrkörpersysteme bestehen aus einer Menge von Körpern mit bestimmten geometrischen und physikalischen Eigenschaften. Weiterhin gelten für die Simulation zeitabhängige Bedingungen, in unserem Falle eine Menge von Bewegungsgleichungen. Wir werden hier nur die verwendeten Eigenschaften erklären. Für eine formale Definition sei auf [War99] verwiesen.
Wir werden Körper im Folgenden auch als „Objekte" bezeichnen.

2.1.1. Geometrische Eigenschaften

Wir betrachten in dieser Arbeit durch konvexe Polygone begrenzte Polyeder im $\mathbb{R}^3$. Objekte, welche diese Eigenschaften nicht besitzen, werden trianguliert [Sch99]. Beim Entwurf mechanischer Teile spielen Kugel-, Zylinder- und Kegelflächen eine große Rolle. Aus diesem Grund versuchen aktuelle Ansätze, Freiformflächen zuzulassen, jedoch befindet sich die Implementierung der Kollsisionsalgorithmen für diese gekrümmten Objekte noch in der Entwicklung, so daß über die Möglichkeiten der Echtzeitfähigkeit der Algorithmen noch keine Aussagen gemacht werden können.

Die Position und Ausrichtung eines Körpers wird durch die Position eines *beliebigen aber festen Punktes* o des Körpers beschrieben, sowie durch eine *Rotationsmatrix* R. Die Variablen o und R beschreiben die Transformation des lokalen Koordinatensystems der Objekte in das Weltkoordinatensystems. Besitzt das Mehrkörpersystem zwei gleich aufgebaute Objekte, lassen sich die geometrischen Daten dieser Objekte im lokalen Koordinatensystem beschreiben und müssen deshalb lediglich einmal gespeichert werden.

Weiterhin dürfen Körper nicht zu einem Flächen- oder Liniensegment oder zu einem Punkt entartet sein. Dies ist vor allem bei booleschen Operationen zwischen den Körpern zu beachten [Beh00].

2.1.2. Physikalische Eigenschaften

Alle auf einen Körper angewandten Transformationen müssen längen- und winkelerhaltend sein. Wir beschränken uns damit auf *starre, nicht deformierbare* Körper.

Um eine physikalische Simulation überhaupt durchführen zu können, müssen *Masse* m, *Schwerpunkt* c und *Trägheitsmatrix* I des Objekts bekannt sein. Diese lassen sich für Polyeder leicht berechnen, wenn die Dichte als homogen angenommen wird.

2.1.3. Bewegung

Das eben diskutierte Mehrkörpersystem soll nun dynamisch werden. Auf die Objekte einwirkende Kräfte beschleunigen die Objekte und versetzen diese in Bewegung, so daß eine Konfigurationsänderung von geometrischen und physikalischen Parametern abhängig ist sowie den wirkenden Kräften $f_1, \dots, f_n$ (einschließlich deren Angriffspunkte an den Objekten).

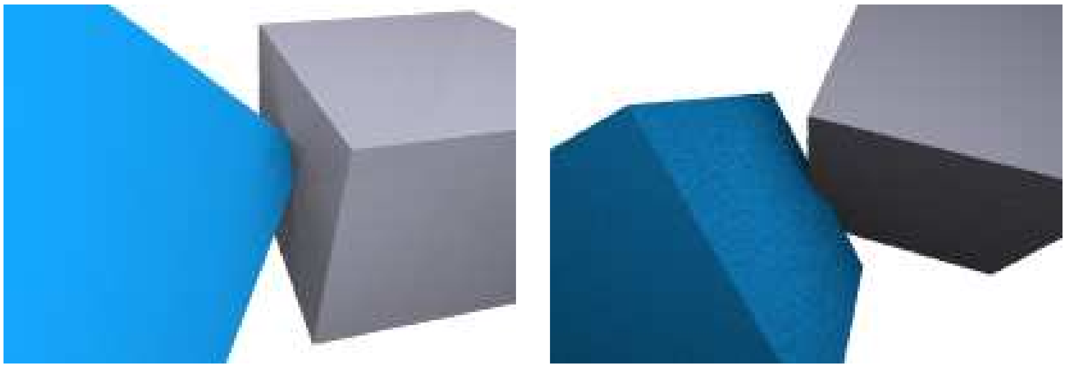
Da die Konfiguration eines Mehrkörpersystems jetzt zu einem gewissen Zeitpunkt definiert wird, sind die bestehenden Konfigurationsparameter Position und Ausrichtung zeitabhängig. Diese zeitabhängigen Parameter sind Geschwindigkeit v und Winkelgeschwindigkeit ω.

Bei einer interaktiven Simulation können wir die Kräfte, die auf die Körper wirken, nicht vorhersagen. Deshalb ist es uns nicht möglich, eine kontinuierliche Kräftefunktion von vornherein anzugeben. Da wir also die Kräfte nur zu gewissen Zeitpunkten kennen, wird die Simulation auch nur zu diesen Zeitpunkten die neuen Konfigurationswerte ausrechnen. Die Simulation ist demnach eine *diskrete Simulation*.

2.2. Kollisionserkennung

Bei bewegten Körpern tritt das Problem auf, daß sich die Objekte durchdringen können. Da wir physikalische Realitätstreue gefordert haben, ist dies eine illegale Konfiguration des Mehrkörpersystems, und wir bezeichnen eine solche Durchdringung zweier Körper als Kollision. Verfahren zur Erkennung und Vermeidung solcher Kollisionen nennt man Kollisionserkennungsverfahren.

Die Voraussetzung, daß die Objekte Polyeder sind, reduziert die Anzahl der Fälle einer Kollision auf zwei. Ist bei einer Durchdringung eine Ecke eines Objektes in die Fläche eines anderen eingedrungen, nennen wir dies *Ecke-Fläche-Kollision*. Durchdringen sich zwei Kanten, fassen wir die Kollision als *Kante-Kante-Kollision* auf. Treten beide Fälle gleichzeitig auf, behandeln wir die Kollision als Ecke-Fläche-Kollision.



Ecke-Fläche Kollision Kante-Kante Kollision

Abbildung 2.1.: Kollisionsarten von konvexen Polyedern

Da wir im ungünstigsten Fall jedes Objekt mit jedem auf Kollision testen müssen, besitzen die Algorithmen zur Kollisionserkennung quadratische Laufzeit. Es können zwar Optimierungen mittels Hüllkörperhierarchien vorgenommen werden, doch verbessert dies nicht die Worst-Case-Laufzeit der Algorithmen. Hinzu kommt noch, daß wir alle Punkte und Kanten eines Objektes ablaufen müssen, was unter Umständen sehr viele sind.

Diese Algorithmen bilden also eine erhebliche Hürde für die Echtzeitfähigkeit, und wir werden bei der Implementierung der Datenstrukturen sehr darauf achten müssen, daß eine Iteration über Objekte und die Iteration über alle Ecken und Kanten eines Objektes effizient möglich sind.

2.2.1. Kollisionsverfahren

Hat man mit einer der obigen Methoden eine Kollision erkannt, stellt sich die Frage, wie man die Konfiguration des Systems wieder legalisiert. Den Vorgang der Legalisierung bezeichnet man als Kollisionsverfahren. Im Bereich der Kollisionsverfahren kennen wir zwei Verfahren, die sowohl echtzeitfähig sind, als auch ein hohes Maß an Realitätsnähe bieten. Sie beruhen auf unterschiedlichen physikalischen Modellen und werden entsprechend als *zwangsbasierte* und *impulsbasierte Simulationsmethode* bezeichnet.

2.2.1.1. Zwangsbasierte Simulation

Bei der zwangsbasierten Methode bewirken Kollisionen die Entstehung bestimmter Kräfte, welche verhindern, daß sich Körper durchdringen. Diese Kräfte zwingen die Körper praktisch wieder auseinander, so daß die Kräfte auch als Zwangskräfte bezeichnet werden und die Bedingungen, die diese aufrechterhalten, als Zwangsbedingungen.

Dieses Prinzip ist nicht nur bei Kollisionsverfahren anwendbar, doch da wir uns hier auf Kontaktsituationen beschränken, sprechen wir auch von Kontaktkräften, die auf die Körper wirken. Jede dieser Kontaktkräfte wirkt auf einen bestimmten Punkt an der Oberfläche des Körpers, so daß wir zu einer vollständigen Beschreibung einer Konfiguration nicht nur die Kräfte, sondern auch deren Ansatzpunkt benötigen.

Da wir eine diskrete Simulation betreiben, muß bei den verwendeten algorithmischen Lösungen beachtet werden, daß sich Objekte temporär durchdringen können. Ein solcher Fall tritt auf, falls zu den diskreten Zeitpunkte $t_1 < t_2$ keine Objekte kollidieren und es gibt einen Zeitpunkt t' mit $t_1 < t' < t_2$, zu dem eine Kollision stattgefunden hat. Ein Algorithmus, der die Kollision zum Zeitpunkt t' nicht beachtet, modelliert das physikalische Verhalten des Mehrkörpersystems nicht korrekt.

2.2.1.2. Impulsbasierte Simulation

Die impulsbasierte Simulation betrachtet Kontakte als Kollisionen in einem einzigen Kontaktpunkt. Kollisionen sind hier von sehr kurzer Dauer, und zwar sind die Zeitintervalle so klein, daß mehrere Kontakte eines Körpers als aufeinanderfolgende Kollisionen modelliert werden können. Ebenso wird ein ständiger Kontakt, zum Beispiel durch Gleiten eines Objekts auf einem anderen, als eine Folge von einzelnen, zeitlich aufeinanderfolgenden Kontakten gehandhabt.
Jeder Kontakt bewirkt die Entstehung zweier entgegengesetzter Impulse, welche den Kontakt schließlich auflösen.

Die Arbeiten [Len99] und [War99] gehen ausführlich auf die Kollisionsverfahren ein. Da wir uns nur für die Ergebnisse der angesprochenen Algorithmen interessieren, betrachten wir die Schnittstellen und nicht deren Implementierung und werden lediglich Eingabe und Ausgabe der entsprechenden Funktionsaufrufe verwenden.

2.3. Interaktivität

Die Bewegungskräfte können in unserem System interaktiv vom Benutzer gewählt werden. Das heißt, die auf die Objekte wirkenden Kräfte sind uns nicht im Voraus bekannt.

2.3.1. Echtzeitsysteme

Die menschliche Hand-Auge-Reaktionszeit beträgt circa 80 Millisekunden. Wir verstehen deshalb unter einem Echtzeitsystem ein Programm, dessen Rechenzeit von der Befehlseingabe durch den Benutzer bis zur Ausführung des Befehls im Rahmen von 100-200 Millisekunden liegt.

2.3.2. Mehrbenutzersysteme

Wollen wir mehrere Benutzer gleichzeitig an einem System agieren lassen, können Konflikte entstehen. Die Benutzer arbeiten im Allgemeinen an unterschiedlichen Rechnern und können diese Konflikte weder vermeiden noch antizipieren. Es wird also vom Algorithmus verlangt, solche Konfliktsituationen aufzulösen.

Deshalb benötigen verteilte interaktive Systeme ein Kommunikationssystem, um die einzelnen Prozesse miteinander kommunizieren zu lassen. Zentraler Bestandteil der Kommunikation ist das Protokoll, mit dem Nachrichten ausgetauscht werden. Das Protokoll regelt die bei mehreren Benutzern auftretenden Konflikte und stellt notwendige Synchronisationsmechanismen sowie Fehlerkorrekturmöglichkeiten bereit.

Wir wollen uns im nächsten Kapitel mit den Anforderungen eines solchen Protokolls in verteilten Systemen beschäftigen.

3. Verteilte Protokolle

Wir wollen hier Begriffe und Notwendigkeiten von verteilten Protokollen erläutern. Zunächst werden verteilte Systeme im Allgemeinen betrachtet und danach besonders im Hinblick auf eine Client/Server Struktur.

Daraufhin wird auf die Kommunikation dieser Systeme und deren Protokolle näher eingegangen. Die verschiedenen Netzwerkprotokolle werden erklärt und danach auf die Erfordernisse von verteilten Systemen untersucht.

Die letzten Abschnitte beinhalten die verwendeten Lösungsansätze, mit denen die uns gestellte Aufgabe der interaktiven Simulation schließlich implementiert wird.

3.1. Verteilte Systeme

Wir betrachten zunächst die Definition eines *verteilten Systems*:

Definition 3.1 (Verteiltes System). Ein verteiltes System besteht aus einer Menge von unabhängigen Prozessen, welche miteinander kommunizieren können. □

Die beteiligten Prozesse laufen üblicherweise auf räumlich getrennten Rechnern, wir wollen jedoch auch zulassen, daß Prozesse auf ein und derselben Maschine rechnen. Nicht jeder Prozeß auf einem Rechner kann allerdings Teil eines verteilten Systems sein. Als *abhängig* (und demnach nicht verteilt) betrachten wir Prozesse, welche sich einen gemeinsamen Adreßraum teilen. Solche Prozesse werden auch als *Threads* bezeichnet.

Ob Prozesse nun auf getrennten Rechnern laufen oder nicht, die Auswirkung von Kommunikationsstörungen ist ein wichtiger Bestandteil jedes verteilten Systems und muß entsprechend berücksichtigt werden. Als Kommunikationsstörungen zählen Signalverzögerungen, Übertragungsfehler, fehlgeschlagene Übertragungen und überzählige (beispielsweise doppelte) Übertragungen von Daten. Wir wollen uns in späteren Abschnitten mit diesen Problemen beschäftigen.

Wir haben nun zwei wichtige Entscheidungen zu treffen – die Wahl des Kommunikationsprotokolls und der Struktur des verteilten Systems. Beide Entscheidungen treffen wir in den nächsten beiden Kapiteln.

3.2. Protokolle

Nachrichten werden in unserem verteilten Modell über ein Netzwerk ausgetauscht. Um eine Netzverbindung zwischen zwei Prozessen aufzubauen, benötigen wir Informationen über das Protokoll, die lokale Adresse, den lokalen Prozess, die entfernte Adresse und den entfernten Prozess. Statt Adresse benutzen wir häufig auch das Wort Rechner, denn wir gehen von der (vereinfachten) Annahme aus, daß unsere Rechner eine einzelne, eindeutige Adresse besitzen, und zwar eine gültige Internetadresse, die sogenannte *IP-Nummer*. Wie wir sehen werden, wird eine Abstraktion mittels *Sockets* benutzt, um den lokalen und entfernten Prozess auf einem Rechner eindeutig zu identifizieren. Jeder Socket ist eindeutig einem bestimmten Prozess zugeordnet.

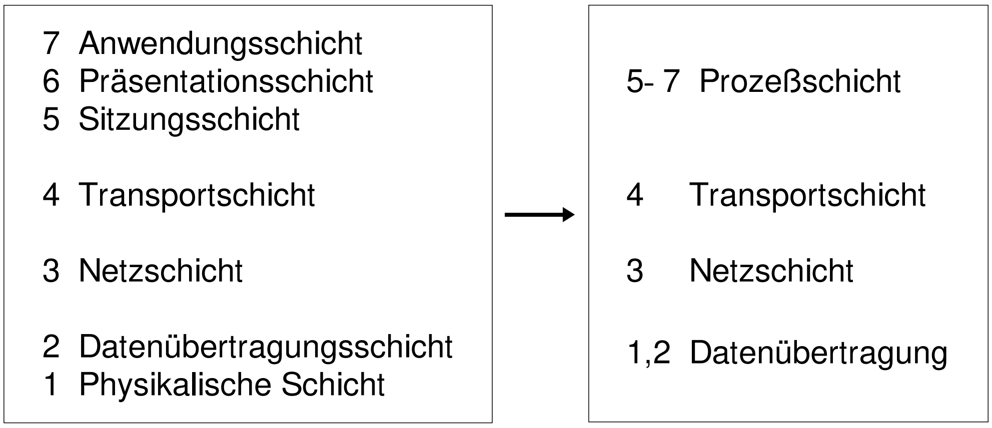


Abbildung 3.1.: Die OSI-Schichten (links) und ein vereinfachtes Modell (rechts)

Zwei Teilnehmer eines vernetzten Systems müssen sich also zunächst über das Kommunikationsprotokoll einigen. Um nicht jedesmal ein komplettes Protokollschema von der Hardwareebene bis hin zur Anwenderebene spezifizieren zu müssen, werden bestehende Protokolle in aufeinander aufbauende Schichten aufgeteilt. Die *International Standard Organization* (ISO) entwickelte dazu das *open systems interconnection* (OSI) Modell zur Rechnerkommunikation, wie in Abbildung 3.1 dargestellt. Die Zusammenfassung der ersten drei und letzten zwei Schichten zu Prozeß- und Datenübertragungsschicht ist eine sinnvolle Vereinfachung. Zum einen werden die obersten drei Schichten aus Implementierungssicht nicht unterschieden, zum anderen sind die untersten zwei Schichten beide hardwareabhängig und werden dort ebenfalls nicht unterschieden. Wir werden diese Vereinfachungen in den folgenden Ausführungen benutzen, um die verwendeten Protokolle in diese Schichten einzuordnen.

Wir beschränken uns in dieser Arbeit auf die Netz- und Prozeßschicht. Protokolle der Prozeßschicht werden wir auch als *Applikationsprotokolle* bezeichnen. Wie wir sehen werden, ist die Wahl des Protokolls ein wichtiger Faktor in der Implementierung von

Fehlerkontrolle und Reaktionsfähigkeit eines interaktiven Systems.

3.2.1. Netzprotokolle

Die implementierten Netzwerkmodule verwenden alternativ zwei Protokollarten der Netzschicht: TCP oder UDP.

Beide Netzwerkprotokolle verwenden eine Abstraktion der Verbindungsschnittstelle, *Socket* genannt. Ein Socket ist ein dateiähnliches Objekt, welches einem Prozess ermöglicht, eingehende Verbindungen zu akzeptieren, ausgehende Verbindungen aufzubauen und Daten zu senden und zu empfangen. Bevor zwei Rechner eine Verbindung aufbauen können, muß ein solches Socketobjekt erzeugt werden.

Der Rechner, der eine Verbindung annimmt, muß dem zuvor erstellten Socketobjekt eine *Portnummer* zuweisen. Eine Portnummer ist eine 16 Bit große natürliche Zahl im Bereich von 0 bis 65535, wird vom Betriebssystem verwaltet und dient dazu, den Prozess, der diesen Socket verwaltet, eindeutig zu identifizieren. Die Portnummern 0 bis 1023, welche nur von Systemadministratoren genutzt werden können, werden normalerweise von häufig benutzten Serverprogrammen belegt. Beispielsweise belegt ein HTTP-Server typischerweise die Portnummer 80.

In Abschnitt 3.3 sehen wir, wie die Netzwerkverbindungen in einer Client/Server Umgebung aufgebaut und genutzt werden.

3.2.1.1. Transmission Connect Protocol (TCP)

Das TCP Protokoll ist ein zuverlässiges, verbindungsorientiertes Protokoll und wird benutzt, einen zweiwegigen Datenstrom zwischen zwei Rechnern herzustellen. Dazu werden Pakete mit Prüfsummen und einer Numerierung versehen. Desweiteren wird für jedes Paket eine Bestätigung erwartet. Bleibt diese aus, wird es erneut versandt. Applikationen können demnach auf Fehlerkontrolle verzichten und müssen lediglich auf etwaige Verzögerungen Rücksicht nehmen.

Der Vorteil der Zuverlässigkeit geht zu Lasten der Protokollgeschwindigkeit. Sowohl der Verbindungsaufbau als auch das Senden eines Datenpakets dauert länger als beim UDP Protokoll, welches im folgenden Abschnitt besprochen wird.

3.2.1.2. Unified Datagram Protocol (UDP)

UDP ist ein verbindungsloses Protokoll, bei dem Rechner einzelne Datenpakete austauschen, ohne zuvor formell eine Verbindung aufzubauen. Im Gegensatz zu TCP ist UDP ein unzuverlässiges Protokoll, da es keinen zusätzlichen Aufwand zur Verwaltung des Verbindungsstatus betreibt. Es können unbemerkt Datenpakete verloren gehen oder falsch beziehungsweise doppelt versandt werden.

3.2.2. Das Applikationsprotokoll

Wir stellen jetzt im Überblick die einzelnen Anfragen und Nachrichten vor, welche eine Mehrkörpersimulation empfangen und senden können soll.

Client-Rechner fragen anfangs nach dem kompletten Datenmodell, das heißt nach den Daten aller in dem Mehrkörpersystem befindlichen Objekte.

Daraufhin spezifizieren die Clienten Bewegungswünsche, welche Änderungen an Position und Ausrichtung eines oder mehrerer Objekte beinhalten.

Beendet der Client sein Programm, schickt er eine Nachricht an den Server, die die aufgebaute Verbindung zum Server wieder schließt.

3.2.2.1. Nachrichten an den Server

- *Anmeldung*
  Mit dem ersten Verbindungsaufbau sendet der Client eine Zeichenfolge, mit der er sich beim Server anmeldet. Die Verbindung wird vom Server nur akzeptiert, wenn die Zeichenfolge des Clienten mit der im Server gespeicherten übereinstimmt. Siehe auch Abbildung 3.3.

- *Laden der Objektdaten*
  Will ein Client eine Szene anzeigen, so erhält er mit dieser Nachricht eine aktuelle Beschreibung aller Objekte.

- *Bewegungswunsch*
  Bewegungswünsche unterteilen wir in Rotations- und Translationswünsche. Alle Objekte der Szene erhalten eine eindeutige Nummer. Die Werte stellen *absolute* Werte dar, nicht etwa Änderungen. Ein Bewegungswunsch besteht aus einer Objektnummer und einem der drei folgenden Datenpakete.

  – *Translation*
    Eine Translation wird durch einen Vektor $\mathbf{v} = <x_1, x_2, x_3> \in \mathbb{R}^3$ übertragen. Die Implementierung verwendet dazu drei Fließkommazahlen mit doppelter Genauigkeit, und die Werte $<x_1, x_2, x_3>$ werden in ebenjener Reihenfolge übertragen.

  – *Rotation um das Objektzentrum*
    Um eine Rotation zu beschreiben, benutzen wir die Achse-Winkel Darstellung. Eine Achse ist ein Vektor $\mathbf{v} = <x_1, x_2, x_3> \in \mathbb{R}^3$, der Winkel $\rho \in \mathbb{R}$ wird in Grad definiert. Sowohl die Achsenwerte als auch der Winkel werden wieder mit Fließkommazahlen doppelter Genauigkeit dargestellt und in der Reihenfolge $<x_1, x_2, x_3, \rho>$ übertragen.

- *Rotation um einen beliebigen Punkt*
  Bei dieser Nachricht wird zusätzlich das Rotationszentrum $\omega = <o_1, o_2, o_3> \in \mathbb{R}^3$ angegeben, so daß das Datenpaket die Form $<x_1, x_2, x_3, \rho, o_1, o_2, o_3>$ besitzt.

- *Maximum der Objektnummern*
  Das Maximum der Objektnummern wird für die Implementierung des VRML Clienten benötigt. Siehe dazu mehr in Abschnitt 5.3.3.

- *Abmeldung*
  Bricht der Client die Verbindung ab, sendet er eine Nachricht, um sich am Server abzumelden. Andernfalls kann dieser nicht entscheiden, ob das Clientprogramm noch aktiv ist.

3.2.2.2. Nachrichten an den Client

Anfragen aller Clienten sind grundsätzlich *Wünsche*. Die Antwort des Servers entscheidet über die endgültige Ausführung.

Der Server schickt hauptsächlich Änderungen von Position und Ausrichtung von Objekten. Auslöser für solche Nachrichten sind die Bewegungswünsche der Clienten und die daran gekoppelte Kollisionsberechnung.

Stellt der Client eine Anfrage zum Laden eines oder mehrerer Objekte, so übermittelt der Server die Objektdaten. Aus Effizienzgründen sind die Objektdaten meist lokal gespeichert, so daß nur eine Referenz auf diese Daten über die Netzwerkverbindung mitgeteilt werden muß.

- *Übermitteln der Weltdaten*
  Die vom Client benötigten Objektdaten werden übermittelt. Die Struktur dieser Daten ist implementierungsabhängig.

- *Bewegungen*
  Vom Server berechnete und ausgeführte Bewegungen werden den Clienten übermittelt. Eine Bewegung besteht aus einer Objektnummer und einer der zwei folgenden Datenpaketen.

  - *Translation*
    Wie oben besitzt eine Translation die Form $<x_1, x_2, x_3>$.
  - *Rotation*
    Die Rotation um das Objektzentrum besitzt wie oben die Form $<x_1, x_2, x_3, \rho>$.

- *Maximum der Objektnummern*
  Mit dieser Nachricht wird das Maximum der Objektnummern, eine natürliche Zahl, übermittelt.

3.3. Client/Server-Algorithmus

Dienstleistungen werden bei einer Client/Server-Architektur von einem einzelnen Rechner angeboten. Diesen Rechner bezeichnet man als *Server*. Im Besonderen verwendet man als Prefix den oder die Dienste, welche der Server anbietet. Zum Beispiel bezeichnet man Rechner, die nach dem HTTP-Protokoll kodierte Anfragen beantworten können, als *HTTP-Server*.

Da der Server Anfragen beantworten können muß, ist er in der Lage, jederzeit Auskunft über den Zustand des zugrundeliegenden Datenmodells zu geben. In unserem Fall kennt er jederzeit Position, Geschwindigkeit und Ausrichtung aller in dem Mehrkörpersystem beteiligten Objekte. Änderungen werden dem Server so schnell wie möglich mitgeteilt, so daß er diese Information weiterverbreiten kann.

Auf der anderen Seite stehen einer oder mehrere *Clienten*. Sie sind es, die Anfragen an den Server stellen und auf Nachrichten warten.

Jeder dieser Client/Server-Algorithmen ist in seinem Grundaufbau gleich strukturiert. Abbildung 3.2 veranschaulicht den Programmfluß des Algorithmus.

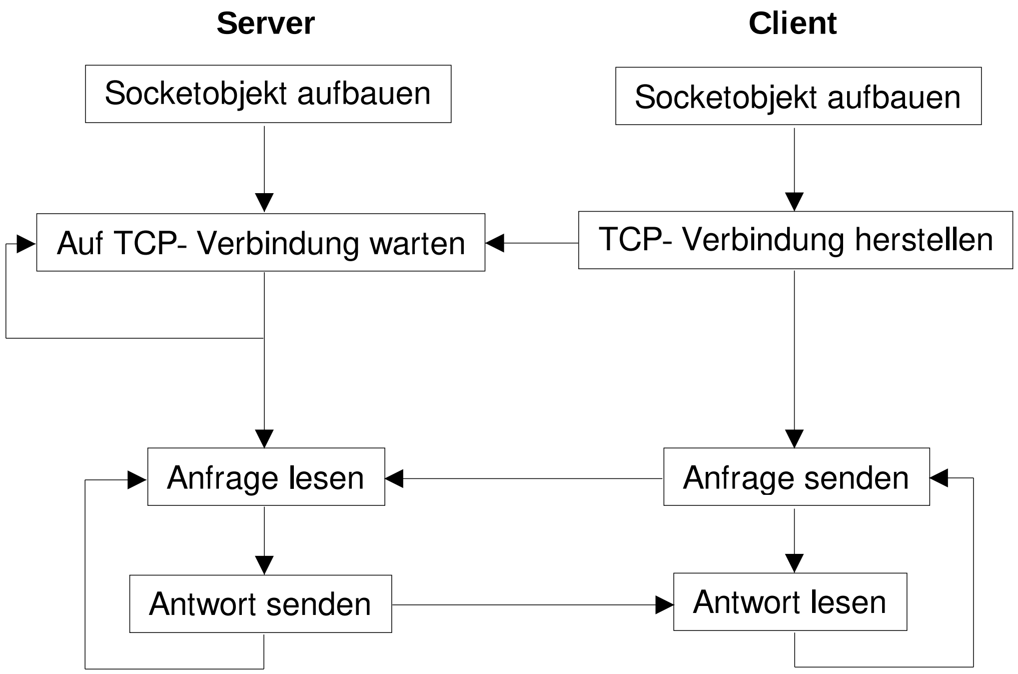


Abbildung 3.2.: Client/Server-Algorithmen

Wie man sieht, enthält der Server-Algorithmus, nachdem eine Verbindung akzeptiert wurde, eine Verzweigung im Programmfluß. Diese Verzweigung kann durch Erzeugen eines neuen, abhängigen Prozesses (*Thread*) geschehen. Bei vielen Anmeldungen werden allerdings sehr viele Threads erzeugt. Eine effizientere Methode ist ein Multiplexverfahren, welches alle bestehenden Netzwerkverbindungen und zusätzlich eine eventuelle Neuanmeldung eines Clienten in schneller Abfolge prüft. Der Server braucht dann nicht nur keine neuen Threads zu erzeugen, sondern kann auch auf Kommunikationsmechanismen zwischen den einzelnen Threads verzichten. Die Implementierung des Algorithmus (siehe dazu Abschnitt 5.3.2) vereinfacht sich dementsprechend.

Der Nachteil dieser Methode ist, daß sich unter Umständen die Antwortzeit verzögert (und zwar um den Zeitraum, in dem alle Verbindungen auf anliegende Nachrichten geprüft werden), doch diese Verzögerung liegt im Bereich von wenigen Millisekunden und ist deshalb zu vernachlässigen.

3.3.1. Konflikte und Strategien

Aufbauend auf dem Protokoll der Netzschicht muß das Protokoll der Prozeßschicht verschiedene Aufgaben erfüllen.

Die Strategie der einzelnen Teilnehmer ist ein wesentliches Entscheidungsmerkmal, welches das Prozeßprotokoll berücksichtigt. Arbeiten die Teilnehmer zusammen, das heißt, verfolgen sie ein gemeinsames Ziel, sprechen wir von *Kooperation*. Im anderen Fall *konkurrieren* die Teilnehmer untereinander und verfolgen unterschiedliche Ziele.

Konkurrierende Teilnehmer erwarten eine klare und vordefinierte Auflösung von Konflikten. Ein Konflikt ist dabei folgendermaßen definiert:

Definition 3.2 (Konflikt). Wollen 2 oder mehrere Benutzer gleichzeitig ein oder mehrere Parameter des zugrundeliegenden Systems ändern, nennt man dies einen Konflikt. □

In unserem Fall ist das System ein Mehrkörpersystem, und die Parameter sind in der Regel Bewegungswünsche. Zur Definition von Gleichzeitig siehe Abschnitt 3.3.4.

Eine Verifikation der Nachrichten ist ebenfalls von großer Bedeutung, um unfaires Spiel zu verhindern (siehe 3.4.1). Die Geschwindigkeit, mit der Wünsche zum Server geschickt werden, ist ein entscheidender Faktor bei der Konkurrenzfähigkeit. Auf Funktionen, die unterschiedliche Netzwerkgeschwindigkeiten bei den Nutzern berücksichtigen, kann deshalb nicht verzichtet werden.

Da unsere Mehrkörpersimulation fast ausschließlich in Montagesimulationen eingesetzt wird, benötigen wir hier eine kooperierende Strategie. Aus diesem Grund gehen wir nicht weiter auf die Algorithmen konkurrierender Strategien ein. Kooperierende Strategien können auf aufwendige Verifikation und Konfliktlösung verzichten und können einfachere und schnellere Algorithmen verwenden.

Es bleibt die Frage der Synchronisation der einzelnen Nachrichten und des Verhaltens im Fehlerfall zu klären.

3.3.2. Fehlerkorrektur

Eine der Optimierungen der Übertragungsgeschwindigkeit ist die Verwendung des unzuverlässigen Netzprotokolls UDP. Bei diesem Protokoll werden im Gegensatz zum TCP Protokoll keine Bestätigungen über das erfolgreiche Senden eines Datenpakets verlangt. Demnach muß man sich Gedanken machen, welche Auswirkungen ein verlorenes Datenpaket hat und wie man diesen Fall behandelt. Es folgt zunächst ein Überblick, wie die Gesamtdauer einer Nachrichtenübermittlung in einzelne Zeitabschnitte unterteilt werden kann.

3.3.2.1. Übertragungsgeschwindigkeit

Ein wesentlicher Faktor der Interaktivität ist die Geschwindigkeit, mit der sich abgesetzte Nachrichten zum Server verschicken lassen. Besonders bei konkurrierenden Strategien kann diese Geschwindigkeit entscheidend darüber sein, ob der abgesetzte Befehl überhaupt berücksichtigt wird.

Die Dauer $\delta T(\text{transfer})$ vom Absetzen einer Nachricht bis zum Ausführen auf dem Server kann wie folgt berechnet werden:

$$\delta T(\text{transfer}) = \delta T_C(\text{code}) + \delta T_C(\text{send}) + \delta T_S(\text{sync}) + \delta T_S(\text{decod})$$

wobei T_C Zeit des Client und T_S Zeit des Server Algorithmus spezifiziert. $\delta T(\text{code})$ bezeichnet die Dauer zum Kodieren einer Nachricht, $\delta T(\text{decode})$ die der Dekodierung. Das Versenden benötigt die Dauer $\delta T(\text{send})$. $\delta T(\text{sync})$ ist der Zeitabschnitt zwischen dem Ankommen der Nachricht am Server und dem Bearbeiten des Serveralgorithmus. Je nachdem, wie der Server Algorithmus strukturiert ist, können nämlich ankommende Nachrichten in eine Warteschlange geschrieben und erst später behandelt werden.

3.3.3. Fehlerkontrolle

Übertragungsfehler können mit Hilfe von Prüfsummen, Bestätigungen und Numerierung der Datenpakete im Übertragungsprotokoll kompensiert werden. Der zusätzliche Verwaltungsaufwand geht jedoch zu Lasten der Übertragungsgeschwindigkeit $\delta T_C(\text{send})$.

Eine wichtige Eigenschaft ist die Übertragung von absoluten Positionen und Ausrichtungen. Ein Datenpaket spezifiziert bei dreidimensionalen Szenen eine Translation v und Rotation R eines Objektes. Es werden jetzt nicht etwa Änderungen δv und δR dieser Werte übertragen, sondern die neuen Werte v' und R'. Damit wird jeder Übertragungsfehler durch die erfolgreiche Übertragung eines der nachfolgenden Pakete wieder ausgeglichen. Wir brauchen uns daher nicht um die verschiedenen Fehlerarten zu kümmern.

Wie groß ist nun die Wahrscheinlichkeit $p(\text{Positionsfehler})$, daß ein Objekt zu einem beliebigen Zeitpunkt eine falsche Position und Ausrichtung hat? Nehmen wir an, die Wahrscheinlichkeit $p(\text{Signalfehler})$, daß ein beliebiges Datenpaket nicht oder verfälscht am Client empfangen wird, konstant und unabhängig von der Anzahl der versendeten Datenpakete ist, vergleichbar also mit dem Werfen einer Münze. Man beachte, daß Fehler beim TCP Protokoll *nicht unabhängig* sind, denn TCP ändert bei Auftreten von fehlerhaften Paketen Werte wie die Übertragungsgeschwindigkeit. Ebenso unterscheiden wir nicht die Art des Fehlers. Wir gehen im Folgenden jedoch von Unabhängigkeit aus und betrachten die Werte als untere Schranke.

Es sei $p(\text{Positionsfehler}) = p(\text{Signalfehler}) = \text{konstant}$. Die Wahrscheinlichkeit, daß i fehlerhafte Pakete hintereinander versandt werden, beträgt $p(\text{Signalfehler})^i$. Wie groß ist nun die erwartete Länge L einer Kette fehlerhafter Pakete? Zur besseren Übersicht definieren wir $p := p(\text{Signalfehler})$ und betrachten eine unendliche Folge von Paketen. Die Wahrscheinlichkeit, daß i fehlerhafte Pakete, gefolgt von einem fehlerfreien Paket, auftreten, beträgt $p^i \cdot (1-p)$, und damit ist die erwartete Länge von i fehlerhaften Paketen

$$\begin{aligned}
L &= \sum_{i=1}^{\infty} i(1-p)p^i \\
&= (1-p) \cdot \sum_{i=1}^{\infty} ip^i \\
&= (1-p)\frac{p}{(p-1)^2} \\
&= \frac{p}{1-p}
\end{aligned}$$

Für $p \leq 0.5$ ist $L \leq 1$. Geht man also davon aus, daß mehr als die Hälfte der Pakete fehlerfrei übertragen werden, bekommen wir im Mittel nicht mehr als ein fehlerhaftes Paket in Folge. Dies sagt allerdings nichts über die maximale Länge der Fehlerketten aus. Fehlerketten von großer Länge bewirken lange Verzögerungen bei der Simulation.

3.3.3.1. Vorhersage

Es kommt trotz einer geringen mittleren Fehlerrate vor, daß für längere Zeit fehlerhafte oder gar keine Pakete übertragen werden. Ursachen dafür können Routerausfälle oder elektrische Signalstörungen sein. Um trotzdem noch als Benutzer an dem System weiterarbeiten zu können, kann der Algorithmus *Vorhersagen* über das Ergebnis einer versandten Nachricht machen. Vorhersagen können bindend sein, das heißt, das Ergebnis wird als Endergebnis angenommen.

Kompetitive Algorithmen benutzen normalerweise eine Vorhersage darüber, was der Benutzer wohl als nächstes Steuerkommando eingeben wird. Da wir jedoch in erster

Linie an exakter Simulation interessiert sind, entscheiden wir uns, solche Vorhersagemechanismen nicht in unser System zu integrieren. Sie hätten keinen großen Nutzen.

3.3.4. Synchronisation

Um die einzelnen Nachrichten überhaupt synchronisieren zu können, muß ihnen ein Zeitpunkt zugewiesen werden, zu dem sie gültig sind. Da wir nicht einer konkurrierenden Strategie folgen, wird dieser Zeitpunkt definiert als die Ankunftszeit der Nachricht am Server. Damit entfällt auch das Mitsenden von Zeitstempeln und das Protokoll und die Auswertung der Nachrichtenpakete vereinfacht sich wesentlich.

Die diskrete Simulation der Kollisionserkennung erlaubt es uns, eine intervallartige Gleichzeitigkeit zu definieren. Alle Zeitpunkte von Nachrichten, die in einem gewissen Zeitintervall $\delta T(\text{sync})$ (siehe oben) liegen, werden von den Kollisionsalgorithmen als *gleichzeitig* betrachtet. Dabei kann es auch vorkommen, daß mehrere unterschiedliche Kräfte zur gleichen Zeit am gleichen Objektpunkt wirken. In diesem Fall werden die Kräfte zu einem einzelnen resultierenden Kraftvektor zusammengefaßt.

Die Intervallgröße, in der eingehende Nachrichten als gleichzeitig angesehen werden, liegt üblicherweise im Bereich von $\frac{1}{10}$-$\frac{1}{5}$ Sekunden. Ist das Intervall kleiner, werden aufgrund der Verzögerung der Signalübertragung gestreute Nachrichten nicht mehr zusammengefaßt. Ist es größer, treten zu große Verzögerungen im Algorithmus auf.

3.4. Sicherheit

Vor allem in verteilten Systemen spielt die Sicherheit eine wichtige Rolle. Ein oft gewünschter Aspekt ist die Verschleierung der Informations*übertragung*. Niemand sollte überhaupt bemerken, daß Informationen ausgetauscht werden. Dies ist jedoch sehr schwierig, so daß wir uns hier darauf beschränken, den *Inhalt* der Information zu schützen. Zu diesem Zweck betrachten wir die eindeutige Identifizierung von Benutzern (Authentifikation) und die Verschlüsselung von Daten.

3.4.1. Authentifikation

Zunächst einmal müssen sich Clienten *anmelden*. Wir haben bisher nicht näher spezifiziert, was dies bedeutet. Das erste Datenpaket, das der Client zum Server sendet, beinhaltet außer dem Wunsch zur Anmeldung beim Server ein Paßwort, ohne das die Anmeldung fehlschlagen wird. Die folgende Grafik veranschaulicht den Vorgang.

Das Paßwort läßt sich im Server-Algorithmus frei wählen und muß nach dessen Start allen Clienten bekannt sein, um eine erfolgreiche Anmeldung durchzuführen.

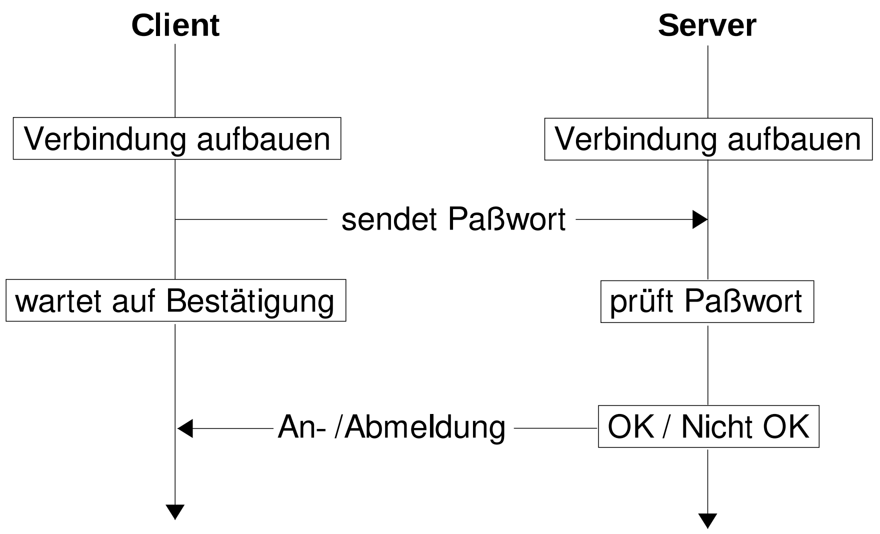


Abbildung 3.3.: Client-Authentifizierung

Gelingt einem nicht authorisierten Benutzer trotzdem die Anmeldung am Server, kann er gefälschte beziehungsweise für ihn vorteilhafte Pakete an den Server senden, um sich einen Vorteil gegenüber anderen Benutzern zu verschaffen. Daß dies gar nicht so abwegig ist, sehen wir zur Zeit in allen aktuellen Netzwerkspielen, in denen Spieler diese Sicherheitslücke regelmäßig ausnutzen.

Die Wirksamkeit gefälschter Wünsche wird bei unserem Serveralgorithmus dadurch verhindert, daß Clienten lediglich Wünsche äußern, aber keine Entscheidungen erzwingen können. Entscheidungen über die Auswirkung eines Wunsches werden im Serveralgorithmus anhand der gewählten Strategie gefällt, welche sich von außen nicht ändern läßt.

3.4.2. Verschlüsselung

Um den Authentifizierungscode nicht im Klartext über die Kommunikationsleitung schicken zu müssen, empfiehlt es sich, diesen zu verschlüsseln.

Wir wollen hier nicht weiter auf Verschlüsselungstechniken eingehen, da diese weder Bestandteil dieser Arbeit sein sollen, noch von dem implementierten Prototyp benötigt werden. Wir verweisen an dieser Stelle auf [Sta98].

4. Programmspezifikation

4.1. Aufbau der Kollisionsbibliothek

Bei der Implementierung wurde das Prinzip *Model-View-Controller* [GHJV95] verwendet. Dabei wird das verwendete Datenmodell (*Model*) von der Oberfläche (*View*) und der Benutzersteuerung (*Controller*) getrennt. Die drei Komponenten interagieren, indem sie sich untereinander Nachrichten über Zustandsänderungen beziehungsweise neue Ereignisse zusenden. Die Verwendung einer objektorientierten Sprache, in der Objekte ebenfalls mittels Nachrichten kommunizieren, macht eine intuitive Implementierung möglich. Abbildung 4.1 zeigt eine Modellierung eines Modells mit drei verschiedenen Benutzeroberflächen. Controller und View sind in der Grafik zu einem Objekt vereinigt.

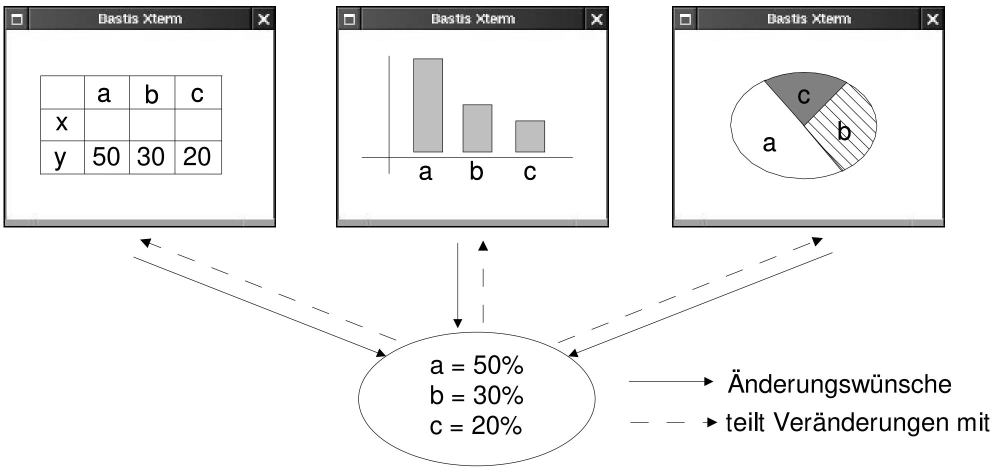


Abbildung 4.1.: Model-View-Controller-Prinzip

Die Modularisierung hat den Vorteil, daß einzelne Module austauschbar sind. In SiLVIA wurden bisher zwei verschiedene Oberflächenarten entwickelt; eine basierend auf der OpenGL Bibliothek [WNDS99], eine weitere mittels der 3D-Modellierungssprache VRML [BCM96]. Abbildung 4.2 veranschaulicht die einzelnen Komponenten und die verwendete Implementierungsumgebung. Bei allen verwendeten Komponenten handelt es sich um plattformunabhängige Standardsoftware. Aus diesem Grund steht die Kollisionsbibliothek SiLVIA momentan für die Betriebssysteme Windows 9x/NT, Linux und Solaris zur Verfügung.

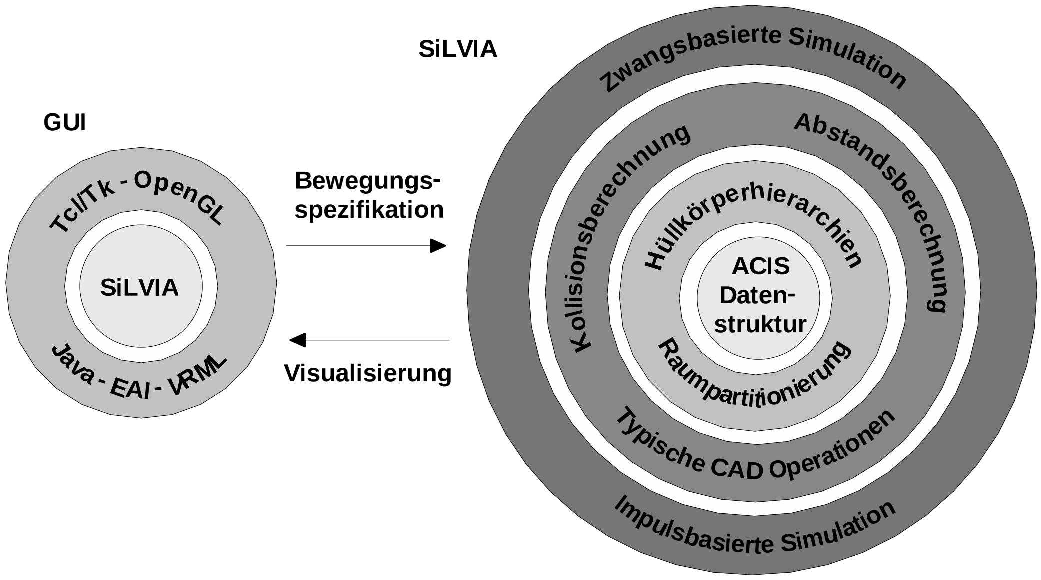


Abbildung 4.2.: Struktureller Aufbau von SiLVIA

4.2. Modellierung dreidimensionaler Körper

Wenden wir uns dem Datenmodell zu. In Kapitel 2.2 haben wir gesehen, daß Algorithmen zur Kollisionserkennung sehr oft auf die Geometriedaten der Objekte zugreifen müssen, im ungünstigsten Fall quadratisch in der Anzahl der Objekte. Deshalb ist der Einfluß eines schnellen Zugriffs auf die Daten groß und muß besondere Beachtung finden. Das Modell muß beliebige dreidimensionale Körper im Raum mit der gewünschten Präzision darstellen können. Zudem wünschen wir uns eine einfache Erweiterbarkeit, um die in 2.1.1 und 2.1.2 genannten Restriktionen der Körper in späteren Projekten teilweise oder ganz aufheben zu können, beispielsweise durch die Einführung von Körpern mit gekrümmten Oberflächen.

Zur Modellierung sind die topologischen als auch geometrischen Daten gespeichert. Eine mögliche Repräsentation ist ein Graph, dessen Ecken und Kanten die topologischen Daten speichern. Diese Darstellung wird auch Drahtgittermodell genannt. Die geometrischen Punktkoordinaten der Ecken werden als Attribute realisiert. Diese Darstellung besitzt zu wenig Informationen, um damit Kollisionen auszurechnen:

1. Es werden keine Oberflächen und Volumina repräsentiert.
2. Die Darstellung eignet sich nur für ebenflächig begrenzte Polyeder, ist also nicht erweiterbar.
3. Die Darstellung ist mehrdeutig, wie Abbildung 4.3 zeigt. Dort sieht man einen kleinen Quader in einen größeren eingebettet. Der kleine Quader kann nun hohl sein oder nicht. Ist er hohl, so kann er sowohl in der x-Achse als auch in der y- oder z-Achse ausgefräst sein. Es gibt also vier unterschiedliche Körper, die das gleiche in der Abbildung dargestellte Drahtgittermodell besitzen.

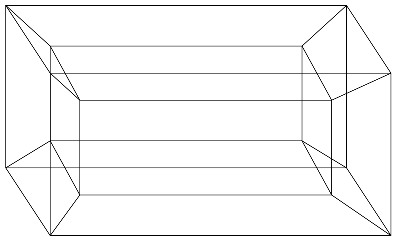


Abbildung 4.3.: Mehrdeutigkeit des Drahtgittermodells

Daher werden in SiLVIA Objekte durch ihren Rand dargestellt. Dies wird als *boundary representation* bezeichnet. Dabei ist der Rand eines Objekts eine Menge von Flächen. Eine Fläche kann wiederum durch ihren Rand, nämlich eine Menge von Kanten, beschrieben werden. Kanten lassen sich durch zwei Ecken darstellen. Damit ergeben sich drei topologische Klassen: Flächen, Kanten und Ecken. Die geometrische Information eines Objekts ist damit nicht an die Ecken allein gebunden wie im Drahtgittermodell, sondern kann an alle topologischen Klassen geknüpft werden. Es können beispielsweise gekrümmte Kanten oder Flächen modelliert werden.

Wir sollten beachten, daß SiLVIA momentan nur Objekte mit bestimmten Eigenschaften zuläßt. Wir können zwar gekrümmte Oberflächen erzeugen, jedoch operieren die Kollisionsalgorithmen ausschließlich auf ebenflächig begrenzten Polyedern. Besitzt ein Objekt nicht diese Eigenschaft, so muß es zunächst in diese Form überführt werden, beispielsweise durch eine Polygonalisierung aller gekrümmten Flächen. Siehe dazu auch [Sch99].

SiLVIA führt verschiedene weitere Attribute von Objekten ein und speichert diese in einer eigenen Objektklasse. Diese Klasse enthält Volumen, Masse, Dichte, Massenschwerpunkt und Namen eines Objekts. Da Objekte verschoben oder gedreht werden können, enthält die Klasse auch die aktuelle Ausrichtung des Objekts und seine aktuelle Geschwindigkeit und Winkelgeschwindigkeit.

Es existieren noch weitere Attribute in der Objektklasse wie Sichtbarkeit und das Unterliegen einer eventuell vorhandenen Gravitation. Diese Attribute sind jedoch grundsätzlich Erweiterungen der bestehenden Algorithmen und haben für die eigentliche Kollisionsberechnung nur eine untergeordnete Bedeutung.

Die Menge der Objekte in einem Mehrkörpersystem läßt sich in *statische* und *dynamische* Objekte unterteilen. Statische Objekte werden im Laufe der Simulation nicht bewegt, so daß lediglich dynamische Objekte Manipulationen unterliegen. Um statische Objekte auszuzeichnen, gibt man ihnen eine unendlich große Masse.

Mehrere Objekte werden nun zu Gruppen zusammengefaßt. Eine Menge von Gruppen bildet eine Szene oder Welt. Die Unterteilung einer Menge von Objekten in Gruppen erlaubt eine einfache Vergabe von Eigenschaften für eine Teilmenge von Objekten. So kann man mehrere Objekte zu einem komplexeren zusammenfassen und die Kollisionserkennung zwischen diesen Objekten, deren relative Ausrichtung untereinander sich nicht ändert, ausschalten. Es ergibt sich eine hierarchische, einfach erweiterbare und mächtige Modellierung eines Mehrkörpersystems.

4.2.1. Speichermanagement

Die C++ Programmiersprache [Str92] gestattet es uns, die Speicherverwaltung unserer Daten selbst vorzunehmen. Der Nachteil ist ein größerer Aufwand bei der Programmierung, jedoch kann eine sorgfältig gestaltete Speicherverwaltung die Effizienz der Algorithmen steigern.

Wir unterscheiden den Zeitpunkt und die Art der Speicherallokation. Letztere läßt sich in C++ durch Überladen der Allokationsfunktionen **new** und **delete** realisieren. Erstere ist die Anwendung der Allokationsfunktionen zu einem bestimmten Zeitpunkt. Wir werden uns bei der Art der Allokation an die Standardimplementierung halten. Eine eigene Speicherverwaltung ist immer erst nach der Fertigstellung eines Programms und nach einer Effizienzmessung sinnvoll. Unsere Implementierung stellt aber einen Prototyp dar, dessen Implementierung kein abgeschlossener Prozeß ist.

Wir werden aber sorgfältig darauf achten, wann wir Speicher allokieren und wann wir ihn wieder freigeben. Der Vollständigkeit halber sei gesagt, daß lokale Variablen *statisch* auf dem Funktionsheap in einem Datensegment allokiert werden. Der Speicher des Heaps wird beim Verlassen der Funktion automatisch wieder freigegeben. Siehe dazu auch den folgenden Abschnitt. Teile der Bibliothek SiLVIA sind in der Programmiersprache C implementiert worden, so daß hier die Allokationsbefehle **malloc** und **free** auftauchen. Die in dieser Arbeit implementierten Algorithmen sind aber ausnahmslos in C++ implementiert.

4.2.1.1. Statische Allokation

Einfache Datentypen in C++ werden statisch allokiert und bei Verlassen der Funktion beziehungsweise des Programms automatisch wieder freigegeben. Zu den einfachen

Datentypen in C++ zählen ganze Zahlen **int** , Fließkommazahlen **double** und **float**, Zeichen **char** und der spezielle Datentyp **void**. Alle aus diesen Datentypen zusammengesetzen Strukturen werden wiederum statisch allokiert. Hierzu zählen Strukturen **struct** und **enum** sowie Klassen **class**. Da statische Typen keine Speicherverwaltung benötigen, setzen wir sie bevorzugt als Datentypen ein.

Ebenso werden Funktionsparameter im lokalen Funktionsrahmen gespeichert. Deshalb ist zu beachten, daß große Objekte (also Klassen und Strukturen) nicht mittels *copy by value*, sondern als Referenzen übergeben werden.

4.2.1.2. Dynamische Allokation

Wir legen fest, daß der Teil des Progamms, welcher Speicher dynamisch allokiert, diesen auch wieder freigeben muß. Ist dieser Teil eine Funktion, so muß allokierter Speicher wieder freigegeben werden, bevor der Funktionsaufruf beendet ist. Ist er eine Klasse, so gebrauchen wir Konstruktor und Destruktor, um Speicher zu verwalten.

Erwähnenswert ist, daß Konstruktoren *nicht* vererbt werden. Es werden aber die Konstruktoren der Basisklasse aufgerufen. Gleiches gilt für Destruktoren.

4.2.2. Ausnahmebehandlung

Wir verwenden Ausnahmen zur Fehlerbehandlung in unseren Programmen. Eine Funktion wirft im Fehlerfall eine Ausnahme aus (**throw**) und erlaubt somit den aufrufenden Kontexten, die Ausnahme aufzufangen (**catch**) und den Fehler zu behandeln. Somit wird dem Programm die Möglichkeit gegeben, auf einen Fehler angemessen zu reagieren.

4.3. Die Benutzeroberfläche

4.3.1. Das Visitor Pattern

Eine wichtige Rolle bei der Benutzerfreundlichkeit eines Programms nimmt das Speicherformat der Daten ein. SiLVIA verwendet keine eigenen proprietären Speicherformate, sondern unterstützt Standardformate für 3D-Objekte:

- VRML [BCM96]
  Die Virtual Reality Modelling Language ist das bevorzugte Speicherformat von SiLVIA.

- ACIS SAT [Spa99]
  Das SAT.Format ist ein von der Firma Spatial Technology entwickeltes Dateiformat.
- OFF [Ros89]
  Ein etwas älteres Format ohne Unterstützung gekrümmter Oberflächen, jedoch wird es in einigen Anwendung noch verwendet.
- Povray [Hal00] (nur schreiben)
  Dieses Format wird von der Raytracer-Software PovRay benutzt.
- PostScript [Ado99] (nur schreiben)
  PostScript dient zur Druckausgabe.
- JPEG [Wal91] (nur schreiben)
  Mit dieser Exportfunktion lassen sich animierte Szenen erstellen, indem die Einzelbilder aneinandergefügt werden.

Das Datenmodell besitzt eine nach dem *Visitor Pattern* [GHJV95] modellierte Schnittstelle. Ein Visitor erlaubt es, eine Funktion auf einer Menge von Objekten auszuführen. Mit dem Visitor ist es deshalb möglich, neue Funktionen für eine Menge von Objekten zu definieren, ohne die Elemente dieser Objekte zu ändern. Wir verwenden diese Eigenschaft, um einfach neue Dateiformate zum Export oder Import in SiLVIA zu unterstützen.

4.3.2. Die Kamera-Analogie

Die Sichtweise einer Szene bedient sich der Analogie einer Kamera. Man kann Sichtwinkel, Position und Ausrichtung einstellen, um eine Szene zu betrachten.
Abbildung 4.4 stellt die möglichen Parameter der Kamera dar.

Die Kameraklasse beinhaltet Variablen zur Bestimmung des sichtbaren Bereichs einer Szene, in unserem Fall die Sichtbarkeitspyramide (Frustum), und zur Bestimmung von Vergrößerungsfaktor, Skalierungsvektor, Position und Ausrichtung.

Kamerafunktionen wie Vergrößern, Skalieren, Rotieren und Bewegen wurden in einer eigenen Klasse gespeichert. Dies vereinfacht das Benutzen mehrerer Kameras erheblich. Zudem können Kameraeinstellungen besser abgespeichert werden.

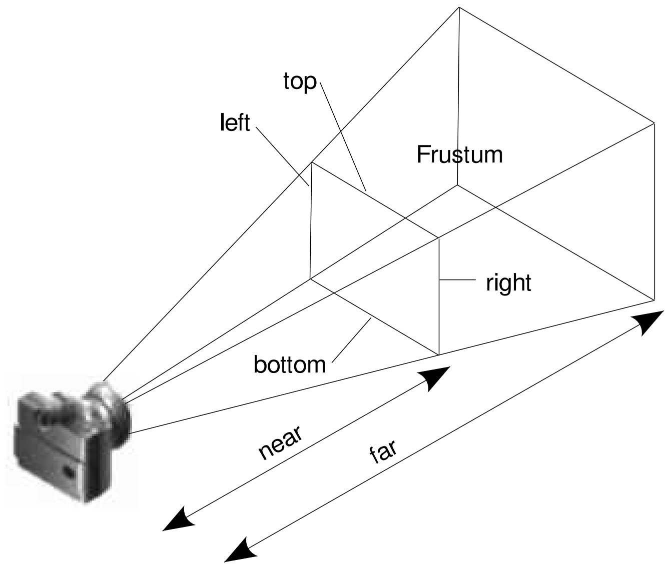


Abbildung 4.4.: Parameter der Kamera

5. Implementierung

Im Rahmen der Arbeit wurden die bis hierhin besprochenen Algorithmen implementiert. Die Implementierung gliedert sich in drei Komponenten. Die erste ist die Kollisionsbibliothek *SiLVIA* und beinhaltet Funktionen zur Verwaltung von Mehrkörpersystemen und zur Kollisionserkennung.

Die zweite Komponente beinhaltet die verwendeten Netzwerkalgorithmen und besteht zum einen aus der Bibliothek *TransWire*, welche die Client/Server-Programmierung erheblich vereinfacht sowie aus der Bibliothek *LivingSilvia*, welche TransWire benutzt, um den Server-Algorithmus zu implementieren. Darüberhinaus wurde die Standardoberfläche **slwish** von SiLVIA ebenfalls mit Netzwerkfähigkeiten ausgestattet.

Die dritte Komponente beinhaltet Beispieldaten und -anwendungen, welche im Augenblick aus einer einfachen nicht interaktiven Demonstration **simple**, einer Triangulationsdemo **approximation**, einer CAD Oberfläche **CadView** mit CSG-Eigenschaften und schließlich der interaktiven Demonstration **slwish** besteht. Zudem beinhaltet LivingSilvia noch eine Java/VRML-Oberfläche, die mit Internetbrowsern wie Netscape oder Internet Explorer bedient wird. Wir erläutern die Gemeinsamkeit aller Oberflächen und gehen ausführlich auf die Implementierung von **slwish** und der Java/VRML-Oberfläche ein.

5.1. Die Bibliothek SiLVIA

5.1.1. Das Datenmodell

Die Spezifikation sieht die Repräsentation des *Randes* eines Objektes vor. Topologische und geometrische Daten werden dabei getrennt. Das ACIS-SAT-Format [Spa99] realisiert diese Eigenschaften und wurde deshalb in SiLVIA als Implementierung verwendet. Es werden zudem weitere topologische Klassen eingeführt. Mehrere Flächen (*Faces*) werden zu Schalen (*Shells*) zusammengefaßt, mehrere Schalen zu Klumpen (*Lumps*) und mehrere Klumpen zu Körpern (*Bodies*). Grafik 5.1 veranschaulicht die gesamte ACIS-Datenstruktur mit topologischen und ihren dazugehörigen geometrischen Klassen.

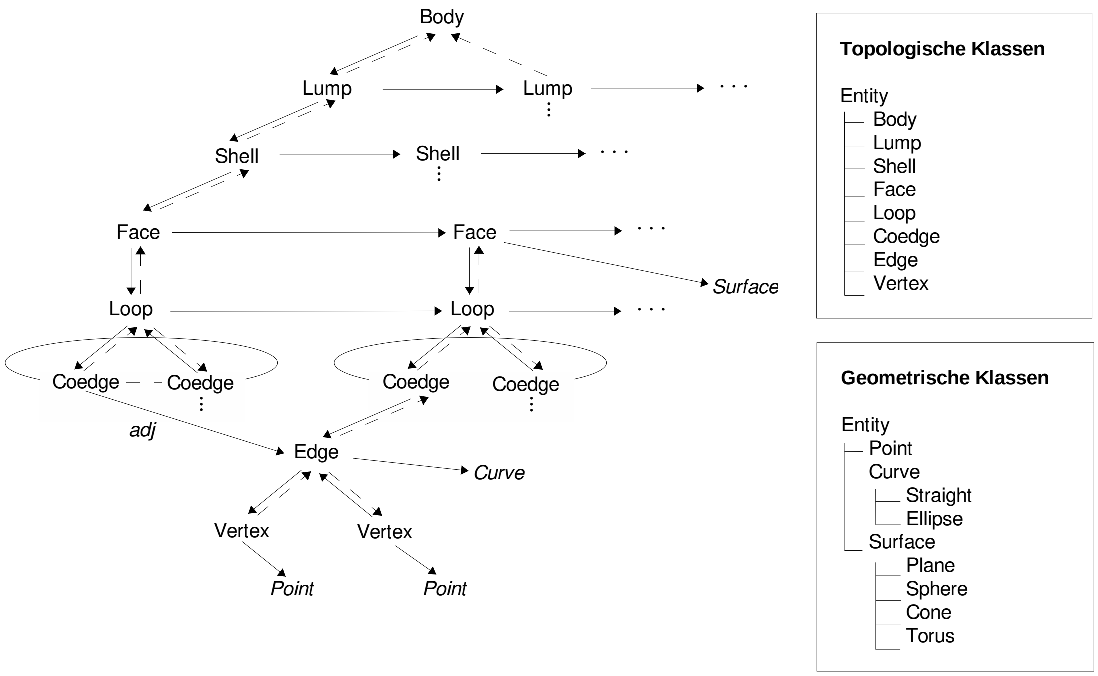


Abbildung 5.1.: Die ACIS-Struktur von SiLVIA

Um die Klassen besser einordnen zu können, besitzen die Klassennamen ein gemeinsames Prefix **Acis**. Die geometrischen und topologischen Daten von starren Körpern verändern sich nach dem Einlesen nicht mehr. Deshalb läßt sich der Speicheraufwand optimieren. Wie aus den obigen Abbildungen erkennbar ist, können doppelt gespeicherte Informationen vollständig vermieden werden. Treten bei einem Objekt zwei gleiche Teilstrukturen (Punkte, Kanten, etc.) auf, so werden zwei Referenzen (Verweise, Pointer) auf ein und dieselbe Struktur (**AcisPoint**, **AcisEdge**, etc.) generiert und die Daten selbst nur einmal gespeichert.

Eine Ausnahme bildet das von [Beh00] entwickelte CAD-Programm, in dem Körper geschnitten oder vereinigt werden. Hier werden Teilstrukturen der Objekte geändert und deshalb mit einem Referenzmechanismus verwaltet. Wird die entsprechende Teilstruktur mehr als einmal referenziert, wird eine Kopie angelegt, um die geplanten Änderungen sicher durchführen zu können.

Alle restlichen Klassen besitzen das Prefix **Sl**, um sie als zur SiLVIA-Bibliothek zugehörig zu kennzeichnen. Die Objektklasse benennen wir **SlObject**, die Objekte werden von der Klasse **SlGroup** zu Gruppen zusammengefaßt und mehrere Gruppen bilden eine Welt, durch die Klasse **SlWorld** implementiert. Die Struktur wird in Abbildung 5.2 dargestellt.

Das Modell ist also speichereffizient. Bedeutet dies jedoch eine Ineffizienz in der Zu-

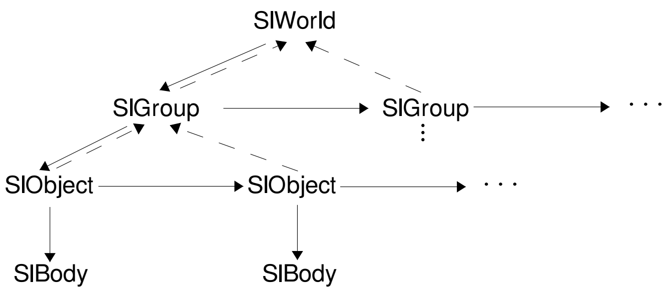


Abbildung 5.2.: Das Datenmodell von SiLVIA

griffszeit? Um an die geometrischen Daten zu gelangen, muß die gesamte Struktur durchmustert werden – angefangen von der Klasse **SlObject**, über **AcisBody** bis hinunter zu **AcisPoint**. Unsere Objekte besitzen jedoch geometrische und physikalische Einschränkungen, so daß jedes Objekt nur aus einer Schale, einem Klumpen und einem Körper besteht. Deshalb müssen lediglich Flächen und Kanten durchlaufen werden, um die Punktkoordinaten zu erlangen. Dies und die Beobachtung, daß das Durchlaufen unserer Struktur mit einfachen Pointern in Linearzeit geht, läßt uns die Zugriffszeit ebenfalls als hinreichend effizient betrachten.

5.1.2. Kollisionsroutinen

Die Kollisionsroutinen wurden im Rahmen einer weiteren Arbeit [War99] implementiert und beinhalten im wesentlichen Kollisionserkennungsroutinen aufgeteilt auf die Klassen **SlWorld**, **SlGroup** und **SlObject**. Objektbewegungen durch den Benutzer werden durch den Aufruf der Funktion **SlWorld::addMoveWish** realisiert, welche den Bewegungswunsch in einer Liste **moveWishList** abspeichert. Die Reaktion auf die berechneten Kollisionen findet durch den Aufruf der Funktion **SlWorld::simulateOneStep**, welche einen Zeitschritt der Simulation ausführt. Die gespeicherten Bewegungswünsche in der Liste **moveWishList** werden nach jedem Simulationsschritt wieder gelöscht.

5.2. Netzwerkbibliothek TransWire

Ursprünglich eine von [BL98] entwickelte Bibliothek zum Datenaustausch zwischen Java- und C-Programmen, wurden die Routinen im Rahmen dieser Arbeit zu einer kompletten Netzwerkbibliothek ausgebaut.

Die notwendigen Daten, um eine Netzwerkverbindung aufzubauen, bestehen, wie wir uns aus Kapitel 3.2 erinnern, aus einem Quintupel *{Protokoll, lokale Adresse, lokaler Prozess, entfernte Adresse, entfernter Prozess}*. Weiter erinnern wir uns, daß Adressen, beziehungsweise Rechner, mittels *IP-Nummer* und Prozesse durch eine *Portnummer* identifiziert werden.

5.2.1. C++ Connector Klassen

Um ein bestimmtes Protokoll zu identifizieren, benötigen wir drei Parameter: die Protokollfamilie, den Protokolltyp und einen Protokollbezeichner. Da wir uns nur für die Protokolle UDP und TCP interessieren, geben wir in Tabelle 5.1 die für UDP und TCP entsprechend auszuwählenden Konstanten für Familie, Typ und Bezeichner an. Die entsprechenden Konstanten sind in der Header-Datei **sys/types.h** definiert.

Protokollfamilie	*Protokolltyp*	*Protokollbezeichner*	*Ausgewähltes Protokoll*
PF_INET	**SOCK_STREAM**	**IPPROTO_TCP**	TCP
PF_INET	**SOCK_DGRAM**	**IPPROTO_UDP**	UDP

Tabelle 5.1.: Konstanten zur Auswahl des Protokolls

Da wir ein Internetprotokoll benutzen, bestehen unsere Socketdaten aus IP-Nummer und Portnummer. Socketdaten werden allgemein in der Struktur

```
struct sockaddr {
    u_short sa_family;    /* Protokollfamilie */
    char    sa_data[14];  /* Protokolldaten   */
}
```

gespeichert. Die Struktur der Internetprotokolle besteht aus

```
struct in_addr {
    u_long s_addr; /* IP-Nummer 32-bit */
}

struct sockaddr_in {
    short          sin_family;  /* PF_INET */
    u_short        sin_port;    /* Portnummer 16-bit */
    struct in_addr sin_addr;    /* IP-Nummer  32-bit */
    char           sin_zero[8]; /* wird nicht benutzt */
}
```

Das Array **sin_zero** dient dem Größenabgleich der beiden Strukturen. Jetzt können an allen Stellen, an denen eine Variable vom Typ **sockaddr** verlangt wird, auch eine vom Typ **sockaddr_in** übergeben werden und umgekehrt.

TransWire kapselt Protokoll, Portnummer und Adresse in der Klasse **Connector**. Somit spezifizieren zwei **Connector**-Instanzen mit demselben Protokoll genau das benötigte

Quintupel einer Netzwerkverbindung. Alle Quellcode Angaben beziehen sich bis auf weiteres auf die Klasse **Connector**.

Die Klasse **Connector** besitzt eine Variable **address** des Typs **struct sockaddr_in**. Die Größe dieser Socketstruktur wird ebenfalls gespeichert in einer Variablen **address_len**. Die Klassen **ServerConnector** und **ClientConnector** sind von der Klasse **Connector** abgeleitet und stellen jeweils Funktionen für Server- und Client-Algorithmen zur Verfügung.

Wir wollen zunächst noch weitere Variablen der Klasse **Connector** betrachten, weiter unten folgt die Beschreibung der anderen in TransWire vorhandenen Klassen.

Jede erzeugte Verbindung (beziehungsweise jedes erzeugte Socketobjekt) besitzt eine Nummer, welche wir in der Variablen **fd** speichern. Weiterhin brauchen wir einen internen Puffer, um gelesene Daten zwischenzuspeichern, denn die Systemaufrufe zum Lesen (**read**) und Schreiben (**write**) von Daten sind bei Unix Systemen nicht gepuffert.

Damit haben wir alle Variablen der Klasse **Connector** zusammen:

```
protected:
    int fd;                         // Socketnummer
    struct sockaddr_in address;     // Socketdaten
    int address_len;                // Größe der Socketdaten
    char buf[BUFSIZ];               // interner Puffer
    // ...
```

Wenden wir uns den Funktionen zu. Es existieren Funktionen zum Versenden und Lesen von Nachrichten, welche aus Basistypen bestehen. Das Versenden von komplexeren Nachrichten kann mit diesen Funktionen aufgebaut werden.

```
public:
    void Write(const char *msg); // schreibt eine Zeichenkette
    void Write(char c);          // schreibt ein Zeichen (8 Bit)
    void Write(const int i);     // schreibt eine ganze Zahl
    void Write(const float f);   // schreibt eine Fließkommazahl
    void Read(int& i);           // liest eine ganze Zahl
    void Read(float& f);         // liest eine Fließkommazahl
    void Read(char& c);          // liest ein Zeichen (8  Bit)
    void Read2(char& c);         // liest ein Zeichen (16 Bit)
    void ReadLine(char** c);     // liest eine NULL-terminierte
                                 // Zeichenkette
```

Alle diese Funktionen verwenden die Hilfsfunktionen **readBytes** und **writeBytes** zum Lesen und Schreiben von einzelnen Bytes.

```
protected:
    // Lies eine Anzahl von Bytes
    void readBytes(unsigned int count);

    // Schreibe eine Anzahl von Bytes
    void writeBytes(const char *msg, unsigned int count);
```

Abgeleitet von der Klasse **Connector** sind die Klassen **ClientConnector** und **ServerConnector**. Client-Algorithmen können eine Verbindung mit der Funktion **Connect** aufbauen, Server müssen diese Verbindung mit **Accept** akzeptieren.

```
class ClientConnector : public Connector {
public:
    // Baue Verbindung zum Rechner host auf Port port auf
    int Connect(long host, int port);

    // Scanne das angegebene Subnetz nach Servern
    slist<long>* scanSubnet(long host, long netmask,
                           long broadcast, int port);
};
```

Die Hilfsfunktion **scanSubnet** sucht nach eventuell vorhandenen Servern, welche einen Dienst auf der angegebenen Portnummer bereitstellen. Es wird eine Liste mit IP-Nummern zurückgegeben, zu denen eine Verbindung hergestellt werden kann.

Was passiert nun mit einem Server, der eine Clientverbindung akzeptiert hat? Damit ist ja die einzig verfügbare Verbindung des Servers belegt, und neue Clienten müssen warten, bis diese wieder frei wird. Deshalb besitzt ein Server normalerweise eine zusätzliche Verbindung, um neue Clienten zu empfangen. Kontaktiert ein Client den Server über diese Verbindung, wird zunächst eine neue Verbindung erzeugt, und der weitere Nachrichtenaustausch zwischen diesem Client und dem Server wird dann über die neue Verbindung abgewickelt. So können beliebig viele Clienten mit dem Server kommunizieren.

Der **ServerConnector** speichert die zusätzliche Verbindung in den Variablen **fd_listen**, **address_listen** und **address_listen_len**. Die Funktion **Accept** nimmt mit der zusätzlichen Verbindung neue Clienten an.

```
class ServerConnector : public Connector {
protected:
    int fd_listen;                     // extra Socketnummer für Server
    struct sockaddr_in address_listen; // extra Socketdaten für Server
    int address_listen_len;            // Größe der Socketdaten

public:
    // akzeptiere eine neue Clientverbindung
    virtual void Accept();
```

Die Portnummer wird einem **ServerConnector** Objekt über den Konstruktoraufruf mitgeteilt:

```
public:
    ServerConnector(int listenport);
```

Die **ServerConnector** Klasse besitzt noch eine weitere hilfreiche Funktion **Init**;

```
class ServerConnector : public Connector {
public:
    static void Init(const char *log_name);
};
```

Sie wird aufgerufen, um überflüssige offene Dateien des Programms zu schließen und eine spezielle Logdatei zu öffnen, um Meldungen eines Servers protokollieren zu können.

Wir besitzen noch keine Möglichkeit, unterschiedliche Protokolle wie TCP und UDP festzulegen. Diese Entscheidung muß bei TransWire zur Compilezeit getroffen werden. Ist die Konstante **USE_UDP** zur Zeit des Übersetzens bekannt, unterstützt TransWire das UDP-Protokoll, ansonsten wird das TCP-Protokoll verwendet.
Weitere Konstanten **debuglevel** und **NET_DUMP** regeln die Ausgabe von Debuginformationen von TransWire.

5.2.2. Java Connector Klassen

Zu jeder C++ Connector Klasse existiert das entsprechende Pendant in der Programmiersprache Java. Allerdings können wir in Java auf die dort vorhandene Socketschnittstelle zurückgreifen.

Die Java Klasse **Connector** speichert eine Verbindung in den Variablen **in** und **out** vom Typ **DataInputStream** und **DataOutputStream**. Die Variable **in** ist dabei schon in der Klasse **FilterInputStream** definiert, so daß wir lediglich **out** noch definieren müssen. Beide Variablen benutzen die Java Socketschnittstelle sowie Java Streams, um Daten zu übertragen. Die Java **Stream** Klassen stellen eine mächtige Schnittstelle zur Ein- und Ausgabe beliebiger Java Objekte zur Verfügung.

```
public class Connector extends FilterInputStream {
    protected DataOutputStream out;
}
```

Gekoppelt mit der Socket-Schnittstelle von Java können wir auf einfache Art und Weise eine Klasse **Connector** in Java implementieren, welche die gleichen Funktionen wie die C++ **Connector** Klasse bietet. Die Funktionen zum Lesen und Schreiben von Basistypen und Objekten werden definiert durch:

```
public int readInt() throws IOException;
public float readFloat() throws IOException;
public void read(Connectable c) throws IOException;
public void write(String s) throws IOException;
public void write(char c) throws IOException;
public void write(int i) throws IOException;
public void write(float f) throws IOException;
public void write(Connectable c) throws IOException;
```

Die Klasse **Connectable** ist eine Schnittstelle, die es ermöglicht, beliebige Objekttypen mittels der Klasse **Connector** zu senden und zu empfangen. Garantiert ein Java-Objekt die Implementierung der Schnittstelle **Connectable**, indem es bei der Klassendeklaration **implements Connectable** angibt, kann es über einen **Connector** als Nachricht versandt werden.

```
public interface Connectable {
    public void writeTo(Connector C) throws IOException;
    public void readFrom(Connector C) throws IOException;
}
```

Betrachten wir als Beispiel die Java Klasse **ConnectableBoolean**, welche ihren Wahrheitswert in der **value** Variable speichert. Um diesen Wahrheitswert über eine Netzverbindung zu übertragen, schreibt es eine Null für *wahr* und eine Eins für *wahr*. Ebenso verfährt man beim Lesen des Wahrheitswertes:

```
public class ConnectableBoolean implements Connectable {
    private boolean value;

    public void writeTo(Connector C) throws IOException {
        C.write((value?1:0));
    }
    public void readFrom(Connector C) throws IOException {
        value = (C.readInt()==1);
    }
}
```

Die **Connector** Klasse gestattet außer dem Versenden und Empfangen von Nachrichten noch das *parsen* von empfangenen Nachrichten. Da wir jedoch diese Fähigkeiten in unserer Client-Server Implementierung nicht verwenden, gehen wir nicht näher auf diese

Funktionen ein. Es sei lediglich erwähnt, daß wir den Strom eingehender Nachrichten in einzelne **Token** unterteilen und deren korrekte Zusammensetzung aus Wörtern und Zahlen überprüfen können.

Parallel zu den C++-Klassen existieren noch die Java-Klassen **ClientConnector** und **ServerConnector**. Sie beinhalten im wesentlichen die gleichen Komponenten wie die entsprechenden C++-Klassen, nutzen jedoch ebenso wie die Java-**Connector**-Klasse die von Java bereitgestellte Socket- und Ein-/Ausgabe-Schnittstelle.

5.2.3. Fehlerbehandlung

Wir benutzen Ausnahmen (siehe 4.2.2), um Fehler an aufrufende Funktionen weiterzuleiten. Die Funktion **fail** dient dazu, die Ausnahme **IOException** auszuwerfen und gleichzeitig den Fehler bekanntzugeben:

```
void Connector::Fail(const char *msg, const int info) {
    cerr<<"transwire: cannot continue normal operation:"<<endl<<msg
      <<" ("<<info<<")";
    if (errno!=0) cerr<<": "<<strerror(errno);
    fail();
}

void Connector::Fail(const char *msg, const char *info) {
    cerr<<"transwire: cannot continue normal operation:"<<endl<<msg;
    if (info!=NULL) cerr<<" ("<<info<<")";
    fail();
}

static void fail() {
    cerr<<endl;
    throw Connector::IOException();
}
```

Alle Kommunikationsfunktionen der Klasse **Connector** werfen im Fehlerfall die Ausnahme **IOException** mittels der Funktion **Fail** aus:

```
void Connector::readBytes(unsigned int count) {
    assert(connected());
    if (count > buffer_size)
        Fail(Msg::BufferOverflow);
    int result;
    unsigned int current_buf_count;
    for (current_buf_count = result = 0; current_buf_count<count;
         current_buf_count += result) {
        result = doRecv(fd_read, buf+current_buf_count,
```

```
                        count-current_buf_count, 0);
        if (result<=0) Fail(Msg::CannotRead);
    }
    dump(buf, count, "Reading");
}
```

Die Funktionen, welche TransWire benutzen, entscheiden, wie sie auf diese Ausnahme reagieren sollen. Im Falle unserer Implementierung des Server-Algorithmus, den wir im nächsten Kapitel kennenlernen werden, wird mit dem Trennen der Netzwerkverbindung reagiert:

```
try {
    client->Read(request);
    // ...
}
catch (Connector::IOException) {
    removeConnection(client);
}
```

Ausnahmen können wie normale Klassen voneinander vererbt werden und so eine Klassenhierarchie bilden. Zudem werden ja keine Klassen, sondern Objekte ausgeworfen, und so können zusätzliche Informationen in diesen Ausnahmeobjekten gespeichert und von der Funktion, die diese Ausnahme auffängt, benutzt werden. In Java ist die Ausnahme **IOException** in eine Klassenhierarchie eingebunden, deren Wurzel die Klasse **Exception** ist. Weiterhin kann allen Ausnahmen eine zusätzliche Nachricht in Form einer Zeichenkette **String msg** mitgegeben werden. So ist der Java-Code des Client-Algorithmus in LivingSilvia dementsprechend strukturiert:

```
try {
    while (thisThread==mainThread && request!=Request.Disconnect) {
        request = clientConnector.readInt();
        // ...
    }
}
catch(IOException ioe) {
    ioe.printStackTrace();
}
```

Die Funktion **printStackTrace** gibt dabei die in dem Objekt **ioe** gespeicherten Informationen zur Aufrufreihenfolge aus.

5.3. Client/Server mit LivingSilvia

Das Aufbauen einer Netzwerkverbindung erleichtert uns die TransWire Bibliothek erheblich, und wir können uns bei der Implementierung der Client/Server-Algorithmen auf die wesentlichen Aspekte konzentrieren. Die Benutzung von TransWire wollen wir jetzt anhand der implementierten Architektur von LivingSilvia veranschaulichen.

5.3.1. Nachrichtentypen

Wir haben die in Abschnitt 3.2.2 beschriebenen verschiedenen Nachrichten als Konstanten implementiert. Für C++ Programme stehen diese in der Header-Datei **Request.h**:

```
class Request {
public:
    enum {
        DISCONNECT = 0,
        WORLD_LOAD,
        ROTATION,
        TRANSLATION,
        MAX_OBJECT_NUMBER,
        ROTATION_WITH_CENTER
    };
};
```

Java-Programme entnehmen die Nachrichtentypen ebenfalls einer Klasse mit Namen **Request**:

```
public class Request {
    public final static int Disconnect = 0;
    public final static int LoadWorld = 1;
    public final static int Rotation = 2;
    public final static int Translation = 3;
    public final static int MaxObjectNumber = 4;
    public final static int RotationWithCenter = 5;
}
```

5.3.2. Der C++-Server

Da wir den Verbindungsaufbau den Routinen der TransWire Bibliothek überlassen, verbleibt lediglich das Lesen und Senden von Nachrichten in einer **while**-Schleife. Demzufolge besitzt die Klasse **Server** nur eine einzige öffentliche Funktion **run**:

```
class LSServer {
public:
    void run();
};
```

Die **while**-Schleife der Funktion **run** gliedert sich in die Schritte *lese Anfragen*, *akzeptiere neue Clienten*, *bearbeite Anfragen* und *sende Antworten*.

```
void LSServer::run() {
    fd_set fds;
    do {

        // lese Anfragen
        if (!readRequests(fds))
            continue;

        // akzeptiere einen neuen Client
        if (FD_ISSET(welcome.listenfd(), &fds))
            acceptNewClient();

        // bearbeite Anfragen
        for(iter=connections.begin(); iter!=connections.end(); iter++)
            if (FD_ISSET((*iter).first, &fds)) {
                ServerConnector* client = (*iter).second;
                try {
                    acceptRequest(client);
                }
                catch (Connector::IOException) {
                    removeConnection(client);
                }
            }

        world.simulateOneStep();
        sendDataForMovedObjects();
    } while(1);
}
```

Der Konstruktor lädt das in der Datei **fname** stehende Datenmodell und startet den Netzwerkserver auf dem durch die Konstante **SERVER_PORT** vordefinierten Port.

```
LSServer::LSServer(const char* fname) :
                   welcome(SERVER_PORT), file(fname) {
    // ...
}
```

Wir generieren keinen neuen Prozess für jeden neu angemeldeten Client, wie es bei vielen anderen Netzwerkservern der Fall ist. Wir beschränken uns auf einen Prozess und entledigen uns damit der aufwendigen Kommunikation zwischen einzelnen Prozessen. Stattdessen wird in der Funktion **readRequest** der Status aller Verbindungen mit der **select** Anweisung überprüft:

```
int LSServer::readRequests(fd_set& fds) {
    // ...

    if ((readyfds = select(fd_high, &fds, NULL, NULL, &tv)) < 0) {
        sldebug(1,"Error: selecting from file descriptors\n");
    }
    return readyfds;
}
```

Die Variable **fd_high** enthält das Maximum aller Socketnummern. Die Variable **fds** enthält die Menge aller Socketnummern. Falls eine Verbindung mit der Socketnummer **fd** Daten senden will, gibt die Funktion **FD_ISSET(fd, fds)** einen Wert ungleich Null zurück.

Will die Serververbindung **fd_listen** Daten senden, so meldet sich ein neuer Client an, und die Funktion **acceptNewClient** wird aufgerufen. Diese Funktion akzeptiert einen neuen **client** und trägt ihn in die Liste der vorhandenen Verbindungen ein. Zunächst wird dabei die Verbindung aufgebaut und in der Liste **connections** eingetragen.

```
welcome->Accept();
client = new ServerConnector(0);
welcome->Write((int)(client->listenPort()));
welcome->Disconnect();
client->Accept();
connections.insert(pair<int,ServerConnector*>(client->getFd(),client));
```

Danach erfolgt die Authentifizierung mittels eines Passwortes, definiert durch **SERVER_STRING**.

```
char* clientName;
client->ReadLine(&clientName);
if (strcmp(clientName, SERVER_STRING)) {
    removeConnection(client);
}
```

Anfragen von bestehenden Verbindungen werden durch die Funktion **acceptRequest** beantwortet:

```
void LSServer::acceptRequest(ServerConnector* client) {
    int request;
    client->Read(request);
    switch(request) {
    case Request::WORLD_LOAD:
        client->Write(Request::WORLD_LOAD);
```

```
            client->Write(file.c_str());
            sendDataForAllObjects(client);
            break;
        case Request::ROTATION:
            readRotation(client);
            break;
        case Request::ROTATION_WITH_CENTER:
            readRotation(client, 1);
            break;
        case Request::TRANSLATION:
            readTranslation(client);
            break;
        case Request::DISCONNECT:
            removeConnection(client);
            break;
        case Request::MAX_OBJECT_NUMBER:
            client->Write(SlObject::IdCounter-1);
        default:
            sldebug(1, "Server encountered invalid request "<<request<<endl);
            break;
        }
    }
```

Die verschiedenen Nachrichtentypen des verwendeten Protokolls sind in der Klasse **Request** definiert.
Die Funktionen **readTranslation** und **readRotation** lesen die Daten der unterschiedlichen Anfragen wie sie in Abschnitt 3.2.2 spezifiziert sind und fügt diese Bewegungswünsche der im Server gespeicherten Szene **world** hinzu, wie zum Beispiel bei einer Translation:

```
    void LSServer::readTranslation(ServerConnector* c) {
        c->Read(obj_num);
        readXYZ(c);
        world.moveWishList.add(world.getObject(obj_num), v, SlQuat(),
                              Sl3DVector(), world.deltaT());
    }
```

Zuletzt wird **world.simulateOneStep** aufgerufen, welche einen Zeitschritt des Mehrkörpersystems simuliert und es werden die Daten der bewegten Objekte an die Clienten weitergeleitet. Damit ist die **while**-Schleife des Servers beendet und beginnt wieder von vorne.

5.3.3. Der Java/VRML-Client

Bei der Client-Implementierung werden die Java-**Connector**-Klassen von TransWire, VRML (*Virtual Reality Modelling Language*) sowie die Schnittstelle EAI (*External Autho-*

ring Interface) verwendet. VRML ist eine Beschreibungssprache zur Modellierung von dreidimensionalen Objekten.

Die Klasse **VrmlClient** regelt den Verbindungsaufbau und stellt das Benutzerinterface zur Verfügung. Der Verbindungsaufbau zum Server initialisiert die Variablen **clientConnector**, **browser** und **mover** zum Speichern der Verbindung, der VRML-Daten und zur Bewegungsausführung. Da wir keine Metainformationen über die Objekte des VRML-Modells besitzen, benötigen wir die maximale Objektnummer, um die **for**-Schleife zu terminieren.

Der Übersichtlichkeit halber wurde der Code zur Ausnahmebehandlung weggelassen.

```
    private void connect() {
        clientConnector = new ClientConnector(hostname, port, "LivingSilvia");
        clientConnector.write(Request.MaxObjectNumber);
        int maxObjectNumber = clientConnector.readInt();
        while (browser==null) {
            browser = Browser.getBrowser(this);
            if (browser != null)
                break;
            Thread.sleep (500);
        }
        numberOfObjects=0;
        Node n=null;
        for(int j=1; j<=maxObjectNumber; j++) {
            n = browser.getNode("TS"+j);
            n.getEventOut("isActive").advise(this, "TouchSensor"+j);
            numberOfObjects++;
        }
clientConnector.write(Request.LoadWorld);
        mover.clientConnector = clientConnector;
        connected(true);
((CardLayout)getLayout()).show(this, "navigate");
    }
```

Die Funktion **handleRequest** enthält die beim Client vorhandene **while**-Schleife. Es wird dort zunächst eine Anfrage gelesen und in der Variablen **request** gespeichert. Bewegungsänderungen werden dann an die EAI Schnittstelle zum VRML Modell weitergeleitet, indem die Variable **browser** benutzt wird.

```
    public void handleRequest() throws IOException {
        int request = Request.Rotation;
        while (request!=Request.Disconnect) {
            request = clientConnector.readInt();
            switch(request) {
            case Request.Rotation:
                readRotation.number = clientConnector.readInt();
```

```
                clientConnector.read(readRotation);
                readRotation.applyTo(browser);
                break;
            case Request.Translation:
                readTranslation.number = clientConnector.readInt();
                clientConnector.read(readTranslation);
                readTranslation.applyTo(browser);
                break;
            case Request.LoadWorld:
                clientConnector.readWord("filename");
                clientConnector.readChar("NULL", 0);
                for(int j=0; j<numberOfObjects; j++) {
                    readTranslation.number = clientConnector.readInt();
                    readRotation.number = readTranslation.number;
                    clientConnector.read(readTranslation);
                    readTranslation.applyTo(browser);
                    clientConnector.read(readRotation);
                    readRotation.applyTo(browser);
                }
                break;
           case Request.Disconnect:
                break;
           default:
                System.err.println("Invalid request, ignoring");
                break;
           }
        }
    }
```

In der Variablen **browser** sind alle Objekte gespeichert. Ein Objekt kann mit der Funktion **browser.getNode(**`ObjectN`**)** erhalten werden, wobei *N* die Objektnummer bezeichnet. Speichern wir das Objekt nun in einer Variablen **n**, so können wir mit

```
    n.getEventIn("set_rotation")).setValue(value);
```

beziehungsweise

```
    n.getEventIn("set_translation")).setValue(value);
```

die neue Ausrichtung und Position dieses Objekts setzen. Der Wert **value** beinhaltet im Rotationsfall vier Fließkommawerte $<x_1, x_2, x_3, \rho>$, bei einer Translation drei Werte $<x_1, x_2, x_3>$.

Wie werden nun Bewegungswünsche zum Server gesendet?

Welches Objekt man bewegen will, entscheidet ein Mausklick auf das zu bewegende Objekt. Ist kein Objekt zum Bewegen selektiert, lassen sich die Knöpfe der Bewegungsauswahl nicht bedienen.

Bei einem Knopfdruck fängt das selektierte Objekt an, sich zu bewegen. Der Knopfdruck wird durch das Java-Observer-Modell zur Funktion **mousePressed** geleitet, die die Nummer des bewegten Objekts sowie die geänderten Bewegungsdaten an die Klasse **Mover** weiterleitet. Diese wiederum sendet kontinuierlich einen Bewegungswunsch an den Server.
Die **while**-Schleife der Klasse **Mover** beinhaltet:

```
while(true) {
    Thread.sleep(500l);  // 0.5 Sekunden warten
    movement.writeTo(clientConnector); // Bewegung ausführen
}
```

Es werden also zwei Bewegungswünsche pro Sekunde an den Server geschickt, bis die Schleife durch das Loslassen des Mausknopfes beendet wird:

```
public void mouseReleased(MouseEvent e) {
    moverThread.interrupt();
}
```

Dies wird ermöglicht, indem die Klasse **Mover** als Thread gestartet wird:

```
moverThread = new Thread(mover);
moverThread.start();
```

Zuletzt sei noch gesagt, daß die bestehenden Implementierungen des External Authoring Interface, also der Schnittstelle zwischen VRML und Java, sehr unzureichend sind. Die einzig funktionierende Version ist der VRML Browser von CosmoWorld in der Version 2.1.1. Bezugsquellen der Programme finden sich im Anhang A.2.

5.4. Die Oberfläche slwish

Eine weitere Client Oberfläche vereint alle bisherigen Komponenten. Es werden Visualisierung, Interaktivität und mehrere Benutzer unterstützt. Dazu benutzt **slwish** drei zusätzliche Bibliotheken, die wir im Folgenden vorstellen. Der Name slwish ist zusammengesetzt aus *sl*, dem SiLVIA Prefix und *wish*, dem TCL Skriptprogramm. Abbildung 5.3 veranschaulicht, wie die einzelnen Komponenten zusammenwirken.

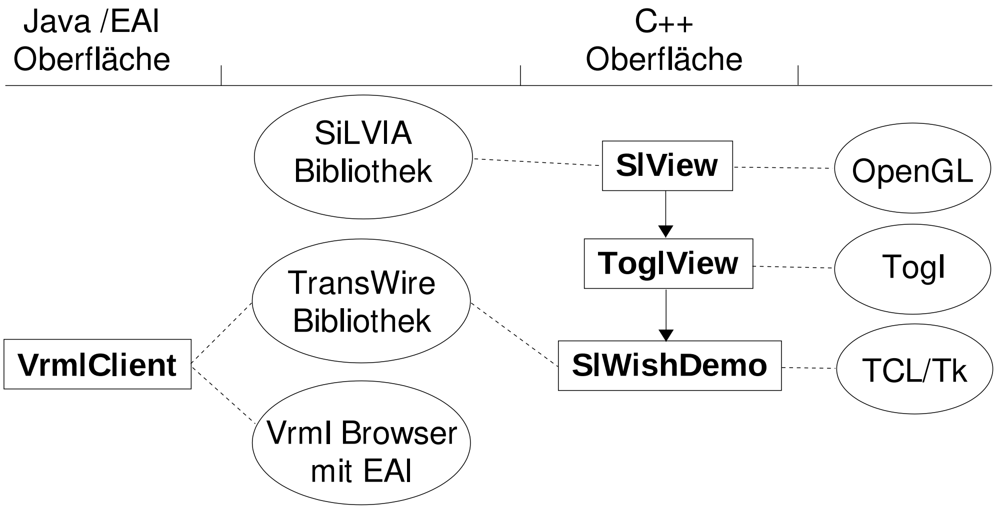


Abbildung 5.3.: Komponenten der Oberfläche

5.4.1. Die generische OpenGL-Schnittstelle

OpenGL [WNDS99] ist eine Grafikbibliothek zur Darstellung von zwei- und dreidimensionalen Objekten. Die Klasse **SlView** bietet eine Schnittstelle für alle Oberflächen, die eine OpenGL-Visualisierung implementieren wollen. Sie beinhaltet Einstellungen wie Lichtfarbe und Intensität der Szene, Materialeigenschaften und Bildschirmkoordinaten der Objekte sowie mehrere Kameras, um die Szene zu beobachten. Weiterhin werden Funktionen zur Navigation der Objekte und Kameras bereitgestellt.

SlView ist ein **Observer** des Modells des Mehrkörpersystems, enthält also eine **update** Funktion, um Nachrichten des Modells zu verarbeiten. Dies sind meist Änderungen von Position oder Ausrichtung eines Objekts oder das Laden einer neuen Szene. In fast allen Fällen wird die Funktion **redraw** aufgerufen, um die Szene mit den geänderten Daten neu zu zeichnen. Der nächste Abschnitt zeigt, wie Tasten- und Mauseingaben von der TCL Schnittstelle empfangen werden. Diese Eingaben werden direkt zu Funktionen der Klasse **SlView** weitergeleitet. **SlView** ist also nicht nur Observer, sondern auch Controller, das heißt für Benutzereingaben zuständig.

Abgeleitet von **SlView** ist zunächst die Klasse **ToglView**, von dieser die Klasse **SlWishDemo**. **ToglView** besitzt einen Verweis auf die Togl Komponente im TCL Skript und verbindet so die OpenGL und Togl Bibliotheken.
SlWishDemo stellt Funktionen bereit zum einfachen Laden und Speichern von Dateien und vor allem die Netzwerkanbindung mittels eines **Connector** Objekts, doch dazu später mehr.

5.4.2. Die TCL/Tk-Schnittstelle

TCL/Tk ist ein zweiteiliges Programmpaket, bestehend aus einer imperativen Skriptsprache TCL (*Tool Command Language*) und der Bibliothek Tk zur Darstellung von Kontrollelementen wie Fenstern, Knöpfen, Menüs und ähnlichem.

Idealerweise, und so auch in unserem Fall, wird mit TCL/Tk eine grafische Oberfläche programmiert, welche keine Codeänderungen an der Bibliothek SiLVIA nötig macht. **slwish** ruft dazu die Hauptfunktion von TCL/Tk auf und überläßt alles weitere dem TCL-Skript und dessen Anbindung an unsere Klasse **SlView**:

```
int main(int argc, char *argv[]) {
    Tk_Main(argc,argv,InitProgram);
    return 0;
}
```

Die Funktionen der Klasse **SlView** lassen sich direkt aus der Skriptumgebung von **TCL** aufrufen, und deren Ergebnisse können in dem Skript verwendet werden. Dazu existiert eine Datei **Callback.h**, in denen Funktionsaufrufe aus TCL entsprechende Aufrufe der Klasse **SlView** verursachen. Wir betrachten die Reihenfolge der Funktionsaufrufe am Beispiel des Ladens einer Datei.

Zunächst ruft das TCL-Skript eine Funktion **TclFileDoOpen** mit dem Namen der zu ladenden Datei als Parameter auf.

```
proc TclFileDoOpen {filename} {
    global silvia
    if [file readable $filename] {
        FileOpen $filename
        set silvia(File) $filename
        set silvia(FileIsNamed) 1
        destroy .cmd.camera
        TclCameraUpdate
        TclResetView
    }
}
```

Nachdem die Existenz der Datei überprüft wurde, wird die C-Funktion **FileOpen** aufgerufen, welche den übergebenen Dateinamen lädt und in **Callback.h** definiert ist. Danach setzt **TclFileDoOpen** globale Parameter des TCL-Skripts auf Standardwerte zurück.
Von TCL übergebene Parameter werden in dem Array **argv** übergeben. **FileOpen** extrahiert den Dateinahmen aus diesem Array und teilt zur besseren Weiterverarbeitung diesen in die Pfad- und Namenskomponente auf. Dann wird, da **demo** eine Instanz der

Klasse **SlWishDemo** ist, die Funktion **SlWishDemo::fileOpen** aufgerufen, welche nun endlich die Datei als neues Mehrkörpersystem lädt.

Wir wollen nicht weiter auf das TCL-Skript eingehen und verweisen an dieser Stelle auf den kompletten Quellcode der Implementierung.

5.4.3. Das Bindeglied Togl

Das Standardpaket TCL/Tk besitzt leider keine Möglichkeit, mit OpenGL generierte graphische Umgebungen in einem Fenster darzustellen. Aus diesem Dilemma befreit uns die Togl-Bibliothek. Sie stellt die Komponente **togl** zur Verfügung, welche ein Fenster erzeugt, in dem OpenGL Befehle zeichnen können. Die aufgerufene Zeichenfunktion wird in **Callback.h** mit der Funktion **Togl_DisplayFunc(Togl_Callback *proc)** spezifiziert, welche in unserem Fall eine Funktion ist, die **SlView::draw()** ausführt, um die dargestellte Szene zu zeichnen.

Die Deklaration eines 500x500 Pixel großen Togl-Fensters geschieht mit

```
togl .gl -width 500 -height 500 -double true -depth true -time 50
```

Der Parameter **double** gestattet, einen neuen Bildschirminhalt erst in einen Puffer zu schreiben und dann darzustellen. Dies vermeidet ein Flackern des Bildes beim Neuzeichnen einer Szene. Der Schalter **depth** generiert einen Tiefenpuffer, um weiter hinten liegende von weiter vorn liegenden Objekten verdecken lassen zu können.

Während des Zeichnens kann Togl zudem noch in regelmäßigen Abständen eine sogenannte Timerfunktion aufrufen. Der Abstand zwischen den Aufrufen wird mit dem **time** Parameter in Millisekunden angegeben. Wir benutzen diese Timerfunktion für verschiedene nützliche Aufgaben:

1. Netzwerkdaten
   Wir prüfen zunächst, ob Netzwerkdaten zur Verarbeitung vom Server geschickt wurden. Die Verarbeitung von Netzwerkdaten werden wir im Abschnitt 5.4.6 etwas näher betrachten.

2. Gravitation
   Falls Gravitation eingeschaltet ist, bewegen wir die der Gravitation unterliegenden Objekte mit einem Simulationsschritt.

3. Simulation
   Falls ein Bewegungswunsch von der Oberfläche anliegt, fügen wir diesen zu den Bewegungswünschen hinzu und führen wiederum einen Simulationsschritt durch.

4. Aufnahme
   Wir können den Ablauf von Simulationen aufnehmen. Ist eine solche Aufnahme gerade aktiv, so speichern wir die aktuelle Konfiguration des Systems, um diese später wieder abspielen zu können.

5. Wiedergabe
   Hier wird überprüft, ob eine solche Wiedergabe gerade läuft. Ist dies der Fall, wird die gespeicherte Konfiguration auf das System übertragen.

5.4.4. Speichern und Laden von Szenen

Wie spezifiziert verwenden wir für die Dateieingabe und -ausgabe das Visitor Pattern. Die Menge der Visitor Objekte sind folgende Klassen des Datenmodells: **SlWorld**, **SlGroup**, **SlObject**, **AcisBody**. Die Visitor-Klasse **SlVisitor** bildet die Basisklasse für alle Visitoren dieser Objektmenge. Die Klasse **IOException** dient zur Ausnahmebehandlung im Fehlerfall.

```
class SlVisitor {
public:
    virtual ~SlVisitor() {}
    virtual void visitAcisBody(AcisBody* node)=0;
    virtual void visitSlWorld(SlWorld* node)=0;
    virtual void visitSlGroup(SlGroup* node)=0;
    virtual void visitSlObject(SlObject* node)=0;

    class IOException {};
};
```

Alle Visitor Objekte besitzen eine Funktion **accept**, welche einen Visitor als Parameter erwartet. **accept** ruft dann die entsprechende Visitor Funktion mit sich selber als Parameter auf. Der folgende Code stammt aus **SlWorld.h**:

```
class SlWorld {
public:
    void accept(SlVisitor* v) {
        v->visitSlWorld(this);
    }
};
```

Die implementierten Visitor-Klassen zur Dateiausgabe (Writer) und Dateeingabe (Reader) sind **SlOffReader**, **SlPovrayWriter**, **SlSatWriter**, **SlSatWriter**, **SlVrmlReader** und **SlVrmlWriter**.

Mit der Visitor-Schnittstelle ist es nun einfach, eine Szene beispielsweise im VRML-Format zu speichern. Der Parameter **VRML_INLINE_VRML** gibt an, sowohl die Welt als auch die Objektdaten im VRML Format zu speichern. Die Objektdaten werden dabei in einzelnen Dateien gespeichert, damit mehrere Welten auf die gleichen Objekte beim Laden zugreifen können. Der übergebene Parameter wird als Element in der Klasse **SlVrmlWriter** gespeichert und kann von jeder Elementfunktion dieser Klasse gelesen werden. Das zeigt uns eine weitere Eigenschaft der Visitor Klassen, nämlich die Möglichkeit, Zustände bei der Traversion über die Objektmenge speichern zu können. Dies ist uns beim Lesen und Schreiben von Dateiformaten oft nützlich.

```
SlWorld* world = new Sl3DWorld();
ofstream datei("szene.vrml");
SlVisitor* visitor = new SlVRMLWriter(datei,
                                      SlVrmlWriter::VRML_INLINE_VRML);
```

Ein Visitor kann zudem noch um weitere Klassen erweitert werden. Dies kommt uns bei der Ausgabe von Formaten zugegen, die Darstellungsparameter der OpenGL Schnittstelle verwenden, wie zum Beispiel den definierten Bildausschnitt oder Kameradaten. Deshalb existiert in SiLVIA die Klasse **SlGLVisitor**. Diese Klasse erweitert die Visitor-Objekte um die OpenGL-Schnittstellenklassen **SlView** und **SlCamera**.

```
class SlGLVisitor {
public:
    virtual ~SlGLVisitor() {}
    virtual void visitSlView(SlView* view)=0;
    virtual void visitSlCamera(SlCamera* camera)=0;
};
```

Nun können wir auch die OpenGL-Parameter des Povray-Formats ausgeben, indem wir die Klasse **SlGLPovrayWriter** sowohl von **SlPovrayWriter** als auch von **SlGLVisitor** ableiten und die Funktionen **visitSlView** und **visitSlCamera** implementieren.

```
class SlGLPovrayWriter : public SlPovrayWriter, public SlGLVisitor {
public:
    SlGLPovrayWriter(ostream* o) : SlPovrayWriter(o) {}
    virtual void visitSlView(SlView* view);
    virtual void visitSlCamera(SlCamera* camera);
};
```

Das Visitor-Pattern hat uns die Dateiaus- und eingabe wesentlich erleichtert. Alle Funktionen zur Bearbeitung eines Dateiformats sind in einer Klasse gebündelt. Wir können Zustandsvariablen in dieser Klasse speichern, und es lassen sich leicht sowohl neue Formate als auch, wie wir an den OpenGL Klassen gesehen haben, neue Visitor-Klassen hinzufügen. Wichtig ist die Tatsache, daß das Datenmodell nicht geändert wird, denn in diesem Fall müssen alle existierenden Visitor-Klassen geändert werden. Bei SiLVIA ist eine solche Änderung des Datenmodells so gut wie ausgeschlossen, denn dies würde eine Anpassung der gesamten Bibliothek erfordern.

5.4.5. Anbindung an das Datenmodell

Die Anbindung wird mittels der in Abschnitt 4.1 erläuterten Model-View-Controller Schnittstelle realisiert. Hier ist **slwish** die Sicht und die Klasse **SlWorld** das Modell.

Wie sieht nun die Schnittstelle zur Bewegungsanforderung aus? Wir möchten gerne Bewegungswünsche äußern, deren Auswirkungen sich in Ausrichtung und Position der Objekte widerspiegelt. Dazu existiert in der Klasse **SlWorld** eine Liste **moveWishList**, deren Elemente folgende Datenstruktur besitzen:

```
class SlMoveWish {
public:
    SlObject* object;        // das zu bewegende Objekt
    Sl3DVector translation;  // der Verschiebungsvektor
    SlQuat rotation;         // das Rotationsquaternion
    Sl3DVector center;       // das Rotationszentrum
    // ...
};
```

Wir können nun diese Liste füllen, indem wir die obigen Elemente der Datenstruktur angeben und als neuen Eintrag in die Liste einfügen. Wollen wir ein bestimmtes Objekt o um einen Vektor v verschieben und um das Zentrum c mit der durch das Quaternion q spezifizierten Rotation drehen, rufen wir die Funktion **moveWishList.add(o,v,q,c)** auf.

Die Funktion **SlWorld::executeMoveWishList** arbeitet diese vorher gefüllte Bewegungsliste ab. Die gespeicherten Bewegungswünsche werden dabei so behandelt, als ob sie *gleichzeitig* auftreten, wobei wir Gleichzeitigkeit in 3.3.4 definiert haben.

Wie in Abbildung 4.2 zu sehen, haben wir zwei verschiedene Benutzeroberflächen **slwish** und den VRML-Client LivingSilvia zur Auswahl. Die benutzten Bibliotheken sind die Skriptsprache Tcl sowie deren grafische Komponenten Tk und Togl. Die graphische Ausgabe wird von der OpenGL-Bibliothek unterstützt. Die zweite Benutzeroberfläche basiert auf der 3D-Modellierungssprache VRML und der Programmiersprache Java sowie der Schnittstelle zwischen diesen beiden Bibliotheken, dem External Authoring Interface (EAI). Alles sind weit verbreitete und für mehrere Plattformen zur Verfügung stehende Programmpakete.

Zwischen den Benutzeroberflächen und dem Datenmodell werden nun Nachrichten über Zustandsänderungen ausgetauscht. Das Modell sendet mittels der Funktion **notify** Nachrichten an die Benutzeroberfläche. Mit jeder Nachricht können beliebige zusätzliche Daten versendet werden, beispielsweise ein Zeiger auf das bewegte oder veränderte Objekt.

```
class SlWorld : public SlObservable {
public:
    enum {
        MSG_UNKNOWN=-1,          // unbekannte Nachricht
        MSG_OBJECT_LOADED,       // Ein Objekt wurde geladen
        MSG_OBJECT_DELETED,      // Ein Objekt wurde gelöscht
        MSG_OBJECT_MOVED,        // Ein Objekt wurde bewegt
        MSG_OBJECT_APPROXIMATED, // Ein Objekt wurde trianguliert
        MSG_WORLD_LOADED,        // Eine Welt wurde geladen
        MSG_WORLD_SAVED,         // Die Welt wurde gespeichert
        MSG_WORLD_CLEARED,       // Alle Objekte und Gruppen wurden aus der
                                 // Welt entfernt
        MSG_WORLD_REMOVED,       // Die Welt wurde gelöscht
        MSG_GROUP_DELETED        // Eine Gruppe wurde gelöscht
    };
    // ...
};
```

Änderungen am Modell können mit Funktionsaufrufen der Klassen **SlWorld**, **SlGroup** und **SlObject** vorgenommen werden. Der Einfachheit halber unterteilen wir Bewegungswünsche in *Rotations-* und *Translationswünsche*.

```
class SlWorld : public SlObservable {
public:
    SlWorld* addGroup(SlGroup* group); // Neue Gruppe erzeugen
    void deleteGroup(SlGroup* group);  // Gruppe löschen
    void clear();               // Alle Objekte und Gruppen entfernen
    void executeMoveWishList(); // Ausführen der gesammelten Bewegungswünsche
    // ...
};

class SlGroup {
public:
    SlGroup* addObject(SlObject* object); // Ein neues Objekt erzeugen
    void deleteObject(SlObject* object);  // Objekt löschen
    // ...
};

class SlObject {
public:
    void approximate(double res, int triangulatePoly); // Objekt triangulieren
```

```
    // ...
};
```

5.4.6. Anbindung als Netzwerkclient

Aufgrund der Unterteilung von SiLVIA in Datenmodell und Benutzeroberfläche können wir der Demonstration **slwish** zusätzliche Netzwerkfähigkeiten verleihen.

Die Daten zur Netzverbindung werden in der Variablen **connector** gespeichert, welche vom Typ **Connector**, einer Klasse aus der TransWire-Bibliothek, ist. Der Zustand der Netzverbindung wird in **connected** gespeichert. Ist **connected** auf wahr gesetzt, sind wir mit einem Netzwerkserver verbunden, ansonsten nicht.

Die schon angesprochene Timerfunktion fragt nun in regelmäßigen Abständen ab, ob wir mit einem solchen Server verbunden sind. Ist dies der Fall, werden eventuell an der Verbindung anliegende Bewegungsdaten bearbeitet. Die Benutzersteuerung ändert sich ebenfalls geringfügig. So werden Bewegungswünsche bei einer bestehenden Verbindung nicht mehr an das lokale Modell geschickt, sondern dem Netzwerkserver mitgeteilt.

6. Ergebnisse

Bei der Anwendung der besprochenen Beispielprogramme werden die Erwartungen an die Echtzeitfähigkeit nicht erfüllt. Eine Laufzeituntersuchung zeigt, welche Teile des Programms die Verzögerungen verursachen.

Die Laufzeiten wurden auf einem Rechner mit einem Celeron2 Prozessor mit 600MHz Taktfrequenz und 128MB Hauptspeicher gemessen. Der Server als auch der Client liefen auf demselben Rechner. Die geladene Szene *spiel.wrl* beinhaltet 114 viereckige Flächen. Mit dem **ping** Programm kann die Verzögerung einzelner Datenpakete gemessen werden. Diese beträgt bei einer lokalen Netzwerkverbindung nicht mehr als eine Millisekunde. Das Testprogramm **slwish** verschickt die für unser Protokoll spezifischen Bewegungsanforderungen. Die Verzögerungen werden mittels des Systemaufrufs **gettimeofday** gemessen.

Die Verzögerung durch die Kollisionsberechnung betrug bei dieser Testanordnung im Durschnitt 600 Millisekunden, bei mehreren Kontakten allerdings auch bis zu 3 Sekunden. Schaltet man die Kollisionserkennung aus, verringert sich die Verzögerung durch die Netzwerkverbindung zu einem Rechner im lokalen Netzwerk auf durchschnittlich 15 Millisekunden.

Die Verzögerungen werden also hauptsächlich durch die Kollisionsroutinen verursacht, welches eine Laufzeitanalyse mit dem Programm **gprof** bestätigt, denn allein die Funktion **SlObject::new_collision** verbrauchte beim Test fast die Hälfte der gesamten Rechenzeit.

6.1. Optimierungen

Obwohl die Netzwerkverzögerung keinen nennenswerten Einfluß auf die Gesamtlaufzeit hat, wird dies in Zukunft vielleicht nicht immer der Fall sein, wenn etwa die Kollisionsroutinen entsprechend optimiert werden.

Aus diesem Grund schließt die Arbeit mit einem Ausblick auf mögliche Laufzeitverbesserungen im Netzwerkbereich ab.

6.1.1. Protokoll

Wir haben in unseren Demonstrationen ausschließlich das TCP-Protokoll verwendet, obwohl die TransWire-Bibliothek auch das UDP-Protokoll unterstützt. Wie schon erwähnt, kommt das UDP-Protokoll ohne Fehlerkorrektur aus und hat dementsprechend weniger Ballast, wie etwa Prüfsummen und Sequenznummern, mitzuschleppen. Das UDP-Protokoll eignet sich für Anwendungen, in denen die Netzwerkgeschwindigkeit von großer Bedeutung ist, also für Programme mit kompetitiver Strategie. Ein weiterer Anwendungsbereich ist das Suchen nach einer Verbindung unter sehr vielen (normalerweise hunderten) Rechnern. Der Verbindungsaufbau erfolgt beim UDP-Protokoll bis zu 10 mal schneller als mit dem TCP Protokoll, so daß das Durchsuchen der Netzwerkstruktur schneller geschieht.

Trifft eine der beiden obigen Bedingungen zu, so ist bei der Implementierung das UDP-Protokoll dem TCP-Protokoll vorzuziehen. Zu berücksichtigen ist dabei, daß die schnellere Netzwerkverbindung gegen geringere Zuverlässigkeit eingetauscht wird. Der Programmierer muß sich deshalb Gedanken über die Behandlung von fehlerhaften, duplizierten, verlorengegangenen und doppelt versandten Datenpaketen machen.

6.1.2. Server

Wir wollen im Gegensatz zur Implementierung ein dezentrales System ohne Server betrachten. Ein solches System hat die Eigenschaft, daß das Wissen über das Mehrkörpersystem auf mehrere Prozesse verteilt ist.
Wie kann nun eine solche Verteilung aussehen? Ein intuitiver Ansatz ist die räumliche Aufteilung der Szene auf mehrere Prozesse. Bewegt sich eines der Objekte in einen anderen Teil des Raumes, erfordert dies die Kommunikation der an diesen Raumabschnitten beteiligten Prozesse.

Eine solche Kommunikation muß vor allem Konflikte auflösen. Stellen wir uns vor, ein Objekt befinde sich auf der Grenze zwischen zwei Raumabschnitten und kollidiere mit zwei Objekten wie in Abbildung 6.1 dargestellt. In diesem Fall muß die Kollisionsberechnung die Kontaktkräfte beider Kollisionen berücksichtigen, es existiert in beiden beteiligten Prozessen aber lediglich das Wissen über eine der beiden Kräfte. In diesem Fall müssen beide Prozesse miteinander kommunizieren und Nachrichten über die benötigten Kollisionskräfte austauschen. Demnach kann in dieser Situation der bestehende Algorithmus nicht korrekt arbeiten.

Ein Anpassen der kompletten Kollisionsalgorithmen auf unvollständiges Wissen über den Zustand des Mehrkörpersystems und die in diesem Fall benötigte Kommunikation würde den Rahmen dieser Arbeit sprengen und wird deshalb nicht berücksichtigt.

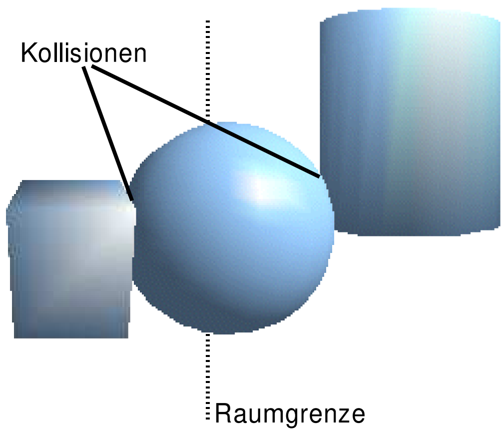


Abbildung 6.1.: Räumliche Aufteilung bei dezentraler Struktur

6.1.3. Oberflächen

Die bisher verwendeten Oberflächen lassen noch einige Erweiterungen zu. So ist etwa die Vewendung der Schnittstellen TCL/Tk, Togl und OpenGL nicht optimal. Das Togl Softwarepaket wird nicht mehr weiterentwickelt und gepflegt und kann mit neueren Versionen von TCL/Tk nicht verwendet werden.

Zudem wünscht man sich eine Oberflächenbibliothek, in welcher nur eine einzelne Programmiersprache zum Einsatz kommt und nicht Skriptsprache, C und C++ gleichermaßen. Plattformunabhängige Bibliotheken wie das Fox Toolkit [vdZ00] oder Java3D erfüllen diese Voraussetzungen.

A. Programme

A.1. Quellcode der Implementierung

Die Implementierung, alle genannten Programme sowie Spezifikationen der verschiedenen Datei- und Darstellungsformate befinden sich auf der dieser Arbeit beiliegenden CD.

A.2. Bezugsquellen verwendeter Software

Die Internetadressen der oben genannten Softwarepakete befinden sich im Literaturverzeichnis. Aufgrund der Dynamik der Softwarebranche können Internetadressen bereits mit dem Erscheinen dieser Arbeit ungültig sein.

Aus diesem Grund liegt dieser Arbeit eine CD bei, auf der alle aufgeführten Programme gespeichert sind.

B. Screenshots

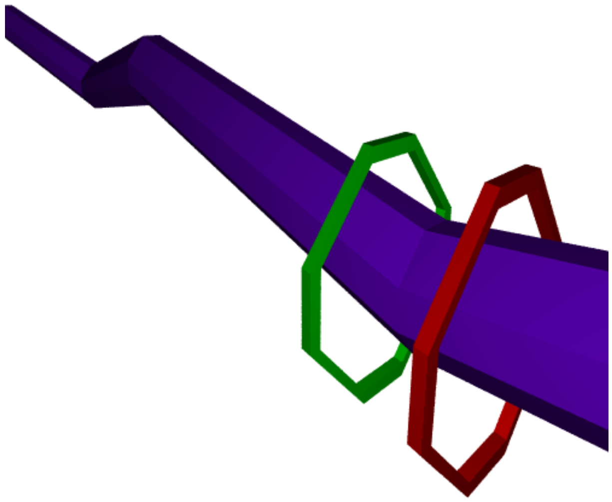


Abbildung B.1.: Die OpenGL Oberfläche. Zwei ringförmige Achtecke befinden sich auf einem gewundenen Stab. Die beiden Teilnehmer müssen ihren Ring von diesem Stab heruntermanövrieren

Abbildung B.2.: Die gleiche Szene aus der Sicht der Java/VRML Oberfläche

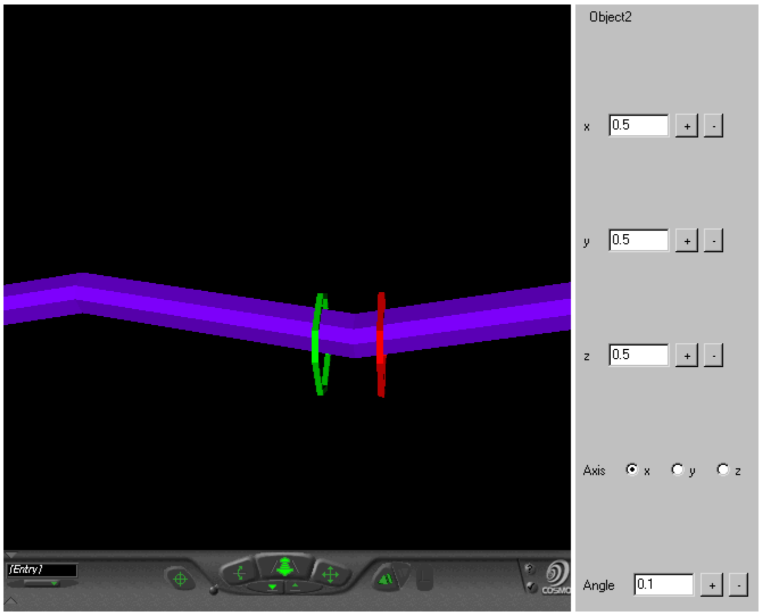


Abbildung B.3.: Die Java/VRML Steuerung

Abbildungsverzeichnis

Literaturverzeichnis

[Ado99] ADOBE SYSTEMS INCORPORATED: *PostScript Language Reference*. Addison-Wesley, dritte Auflage, 1999. `http://partners.adobe.com/asn/developer/PDFS/TN/PLRM.pdf`.

[BCM96] BELL, GAVIN, RIKK CAREY und CHRIS MARRIN: *The Virtual Reality Modeling Language Specification, Version 2.0*, 1996. `http://www.web3d.org/VRML2.0/FINAL/`.

[Beh00] BEHRENS, C.: *Solid Modeling mittels Boolscher Operationen*. Diplomarbeit, Fachrichtung 6.2, Informatik, Universität des Saarlandes, 2000.

[BL98] BAST, HANNAH und STEFFEN LOHSE: *Die TransWire Bibliothek*, 1998.

[CM97] CHRIS MARRIN, SILICON GRAPHICS INC.: *External Authoring Interface Reference*, 1997. `http://www.marrin.com/vrml/eaiwg/ExternalInterface.html`.

[FGM+97] FIELDING, R., J. GETTYS, J. MOGUL, H. FRYSTYK und T. BERNERS-LEE: *Hypertext Transfer Protocol – HTTP/1.1*, January 1997.

[FvDFH96] FOLEY, JAMES D., ANDRIES VAN DAM, STEVEN K. FEINER und JOHN F. HUGHES: *Computer Graphics: Principles and Practice*. Addison-Wesley, Singapore;Tokyo;Madrid, zweite Auflage, 1996.

[GHJV95] GAMMA, HELM, JOHNSON und VLISSIDES: *Design Patterns*. Addison-Wesley, 1995.

[Hal00] HALLAM OAKS PTY. LTD.: *Persistence Of Vision Raytracer*, 2000. `http://www.povray.org`.

[Las99] LASSETER, TOM: *Toy Story 2, The Movie*, 1999. `http://uk.imdb.com/Title?0120363`.

[Len99] LENNERZ, C.: *Impulsbasierte Dynamiksimulation starrer Körper unter Verwendung von Hüllkörperhierarchien*. Diplomarbeit, Fachrichtung 6.2, Informatik, Universität des Saarlandes, 1999.

[Ros89] ROST, RANDI J.: *OFF–A 3D Object File Format*, 1989.

[Sch99] SCHMID, R.: *Approximation verallgemeinerter Polyeder*. Diplomarbeit, Fachrichtung 6.2, Informatik, Universität des Saarlandes, 1999.

[Spa99] SPATIAL TECHNOLOGY: *ACIS-3D-Toolkit SAT Format, Version 6.0*, 1999. `http://www.spatial.com/Support/Publications/sat60/satfile.htm`.

[Sta98] STALLINGS, WILLIAM: *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 1998.

[Str92] STROUSTRUP, BJARNE: *Die C++ Progammiersprache*. Addison-Wesley, zweite Auflage, 1992.

[vdZ00] ZIJP, JEROEN VAN DER: *Fox Toolkit*, 2000. `http://www.cfdrc.com/FOX/fox.html`.

[Wal91] WALLACE, GREGORY K.: *The JPEG Still Picture Compression Standard*. Communications of the ACM, 34/4:30–44, April 1991.

[War99] WARKEN, T.: *Berechnung von Kontaktkräften für eine zwangsbasierte Dynamiksimulation*. Diplomarbeit, Fachrichtung 6.2, Informatik, Universität des Saarlandes, 1999.

[WNDS99] WOO, NEIDER, DAVIS und SHREINER: *OpenGL Programming Guide*. Addison-Wesley, dritte Auflage, 1999.