Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Simulation, Systemoptimierung und Robotik

Effiziente Kollisionserkennung und echtzeitfähige Simulation der Kinematik, Dynamik und Sensorik autonomer Fahrzeuge

Efficient collision detection and realtime simulation of kinematics, dynamics and sensors of autonomous vehicles

Diplomarbeit von Karen Petersen

Aufgabensteller:	Prof. Dr. Oskar von Stryk
Betreuer:	Martin Friedmann
Abgabetermin:	29.08.2007

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 29. August 2007

Karen Petersen

Kurzzusammenfassung

Im Rahmen dieser Arbeit wurde ein effizientes Kollisionserkennungssystem für ein bestehendes Simulatorframework entwickelt. Dies beinhaltet sowohl Primitiv-Tests zwischen allen verfügbaren Grundkörpern (Kugeln, Quader, Zylinder und Ebenen) als auch, zur Effizienzsteigerung, das Erstellen von Hüllkugelhierarchien für zusammengehörende Objektgruppen. Der Aufbau der Hierarchien für ungeordnete Objektmengen erfolgt mit einem top-down-Ansatz. Für Robotermodelle, die bereits in einer baumförmigen Struktur vorliegen, orientiert sich die Hierarchie an der vorhandenen Struktur. Die gesamte Implementierung ist so aufgebaut, dass sowohl die verwendeten Hüllkörper, als auch das Vorgehen beim Aufbau der Hierarchien sowie die einzelnen Primitiv-Tests ohne großen Aufwand ausgetauscht werden können.
Weiterhin wurden verschiedene Fahrzeuge mit Differentialgetriebe und Ackermann-Lenkung modelliert. Für die Sensorik der Fahrzeuge wurde die Simulation von Laserscannern betrachtet, hierfür wurden zwei vollständig verschiedene Ansätze verwendet: das Auslesen des Tiefenpuffers der rasterisierten Szene und das Schneiden einer Menge von Strahlen gegen die Objekte der Szene. Dies beinhaltet insbesondere die Implementierung von Schnitttests zwischen Strahlen und den verschiedenen Primitiven, aus denen die Szene aufgebaut ist. Beide Ansätze wurden implementiert und bezüglich ihrer Effizienz verglichen.

Abstract

In this work an efficient collision detection system for an existing simulator-framework has been developed. It contains primitive-tests between all available bodies (spheres, boxes, cylinders and planes) as well as bounding sphere hierarchies for different compounds to ameliorate the performance. The hierarchies for arbitrary compounds are built with a top-down approach. For robots, which are already organized in a tree-structure, this structure is mainly maintained for the bounding sphere hierarchy. Everything is implemented in a modular way, so that the bounding volumes, the algorithms for building the trees and the primitive-tests can easily be replaced by other algorithms.
Furthermore there were built models of different vehicles with differential drive or Ackermann-steering. For the vehicles' sensor systems there were observed different ways for simulating a laser range finder. One approach is to read the depth buffer of the current scene, the other approach is to create a number of rays and compute the intersection of these rays with all the other objects in the scene. This especially includes the implementation of intersection-tests between rays and the different bodies, of which the scene consists. Both approaches were implemented and compared with respect to their performance.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Kapitel 1

Einleitung

Am Fachgebiet Simulation und Systemoptimierung des Fachbereichs Informatik der Technischen Universität Darmstadt werden verschiedene Arten von Robotersystemen wie z.B. bionische Roboter, fahrende und laufende Roboter entwickelt und untersucht. Ein Forschungsschwerpunkt liegt auf der Kooperation homogener und heterogener Teams mobiler Roboter. Die Zusammenarbeit von Humanoidrobotern und Fahrzeugen wurde im Rahmen einer Dissertation [13] untersucht. Homogene Teams des Fachbereichs kommen auf dem RoboCup in der Vierbeiner-Liga (die Darmstadt Dribbling Dackels als Teil des German Teams [20]) und in der Humanoid-Liga (die Darmstadt Dribblers [9]) zum Einsatz. Um Anwendungen in diesen Gebieten zu testen, ist es von Vorteil, dies möglichst oft mit Hilfe eines Simulators zu tun. Hierfür muss die Hardware nicht zur Verfügung stehen und man vermeidet unnötigen Verschleiß.
Als Simulationsumgebung für verschiedene Anwendungen der mobilen Robotik wird am Fachbereich `MuRoSimF` [10] entwickelt. Dies ist ein Framework, dass die Simulation verschiedener Szenarien mit fahrenden und laufenden Robotern in Echtzeit ermöglicht. Die Software, die sonst auf den echten Robotern läuft, kann so im Simulator getestet werden. Doch die Simulation beschränkt sich bisher auf die Kinematiksimulation und Visualisierung der Roboter sowie die Simulation von Kameras.

Im Rahmen dieser Arbeit soll `MuRoSimF` erweitert werden, damit es noch vielseitiger einsetzbar wird. Ein wichtiger Bestandteil hierfür ist die Implementierung einer schnellen Kollisionserkennung, damit auch Dynamiksimulationen in Echtzeit durchgeführt werden können. Dies beinhaltet sowohl die Implementierung von Funktionen, die zwischen zwei gegebenen Objekten Kollisionen berechnen, als auch die Untersuchung und Umsetzung von Strategien, die schnell zu einem frühen Zeitpunkt Kollisionen ausschließen, wie z.B. Hüllkörperhierarchien oder Raumunterteilungsverfahren. Außerdem soll `MuRoSimF` verstärkt zur Simulation autonomer Fahrzeuge eingesetzt werden. Für diese Fahrzeuge müssen Modelle erstellt und eine Ansteuerung implementiert werden. In autonomen Fahrzeugen gehört ein Abstandssensor zu den wichtigsten Sensoren. Da außer Kameras bisher keine Sensoren zur Verfügung stehen, soll die Simulation von Laserscannern ermöglicht werden.
Um das modulare Konzept von `MuRoSimF` fortzusetzen, sollten alle neuen Funktionen so auf-

gebaut sein, dass sich die verwendeten Algorithmen leicht austauschen lassen.

Diese Arbeit ist wie folgt aufgebaut: In Kapitel 2 wird zunächst ein Überblick darüber gegeben, was es bisher an vergleichbaren Simulatoren gibt und wie diese arbeiten. In Kapitel 3 werden der Aufbau und die Arbeitsweise von `MuRoSimF` genauer beschrieben. Kapitel 4 vermittelt die wichtigsten Grundlagen, die für die Kollisionsberechnung benötigt werden. Kapitel 5 geht zunächst näher darauf ein, welche Möglichkeiten es für die effiziente Umsetzung der Kollisionserkennung gibt, im Anschluss wird die Umsetzung der Kollisionserkennung für `MuRoSimF` erläutert. Die Modellierung und Simulation von Fahrzeugen sowie die Simulation von Laserscannern werden in Kapitel 6 behandelt. In Kapitel 7 werden die erzielten Ergebnisse beschrieben, in Kapitel 8 wird diskutiert, welche Erweiterungen aufbauend auf diese Arbeit möglich sind. In Anhang A werden alle verfügbaren Robotermodelle vorgestellt und in Anhang B wird schließlich beschrieben, welche Möglichkeiten zur Konfiguration der Anwendung vorhanden sind.

Kapitel 2

Stand der Forschung

Es gibt bereits eine Vielzahl von Simulatoren für Mehrkörpersysteme, die alle mehr oder weniger realistisch die Gesetze der Physik simulieren. Die meisten dieser Simulatoren lassen sich über eine Beschreibungssprache oder Ähnliches konfigurieren. Für die Modellierung von Robotern stehen meistens verschiedene Sensoren zur Verfügung.
Der aufwendigste Teil einer solchen Simulation ist das Auffinden von Kollisionen und die Berechnung einer realistisch wirkenden Reaktion der Körper auf diese Kollisionen. Für diese Aufgaben greifen die meisten Simulatoren auf externe Bibliotheken zurück.
In den folgenden Abschnitten werden zunächst solche Bibliotheken (sogenannte *Physik-Engines*) beschrieben. Im Anschluss werden verschiedene Simulationspakete vorgestellt, mit denen unterschiedliche Simulatoren erstellt werden können, und schließlich werden einige spezialisierte Simulatoren beschrieben.

2.1 Physik-Engines

Die *Open Dynamics Engine ODE* [21] ist eine frei verfügbare Open-Source Bibliothek zur Simulation von Starrkörperdynamik. Es stehen verschiedene Gelenktypen zur Verfügung, mit denen einzelne Objekte zu einem beweglichen Mehrkörpersystem verbunden werden können. Jedes Gelenk wird als eine Sammlung geometrischer Zwangsbedingungen modelliert. Als Kollisionsprimitive stehen Kugel, Quader, Ebene, Zylinder, Strahl und Dreiecksnetz zur Verfügung. Es ist möglich, Objekte in Gruppen einzuteilen, für die dann gleichmäßige Gitter oder hierarchische Raumunterteilungen erstellt werden, sodass nicht alle Objekte gegeneinander geschnitten werden müssen. Dies kann man nutzen, um Hüllkörperhierarchien zu erstellen. Dieses integrierte Kollisionserkennungssystem kann durch ein anderes ersetzt werden, solange die benötigten Kontaktmodelle beibehalten werden. ODE arbeitet mit einem harten Kontaktmodell, d.h. zwei Körper können sich zu keinem Zeitpunkt gegenseitig durchdringen. Im Fall einer Kollision werden neue Zwangsbedingungen zwischen den betroffenen Primitiven eingeführt, die wieder aufgelöst werden, sobald der Kontakt zwischen den Objekten nicht mehr besteht. Die benötigte Rechenzeit für

die Vorwärtsdynamik hängt linear von der Anzahl der Objekte und der Gelenke und kubisch von der Anzahl der Zwangsbedingungen ab.

Die *AGEIA PhysX-Technologie* [1] ist eine Physik-Engine, die ihre Berechnungen auf einen externen Physikbeschleuniger (eine *PPU, physics processing unit*) auslagern kann. Ist keine solche externe Karte vorhanden, funktioniert die Simulation trotzdem, allerdings weniger schnell und mit weniger Details. Über die interne Arbeitsweise ist auf [1] nichts ausgeführt, es entsteht der Anschein, dass hauptsächlich die Rechenleistung der Karte ausgenutzt wird (bis zu 530.000.000 Kugel-Kugel-Tests pro Sekunde) anstatt Algorithmen zur Beschleunigung der Physik-Simulation zu optimieren.

Karma [6] ist eine von Epic games entwickelte Physik-Engine. Mehrkörpersysteme können durch das Verbinden von Primitiven erstellt werden. Die Verbindung zwischen zwei Primitiven wird als geometrische Zwangsbedingung modelliert. Für jedes Objekt können verschiedene Parameter wie z.B. Reibungsfaktoren, eingestellt werden, die die Interaktion mit anderen Objekten beeinflussen. Außerdem wird für jedes Objekt entschieden, wie exakt die Bewegungssimulation sein soll. Wird keine exakte Bewegungssimulation benötigt, bekommt das Objekt einen stark vereinfachten Trägheitstensor zugewiesen, andernfalls wird ein genauer Trägheitstensor berechnet. Die Kollisionsberechnung erfolgt nicht mit den Objekten selbst, sondern mit ihren Hüllkörpern. Als Hüllkörper werden k-DOPs (siehe Kapitel 5.1.1.4) verwendet. k kann für jedes Objekt individuell gewählt werden, es stehen 6-DOPs, 10-DOPs, 18-DOPs und 26-DOPs zur Verfügung.

2.2 Simulations-Pakete

SimRobot [16] ist eine Simulationsumgebung für verschiedene mobile Roboter. In [16] wurde die Simulation von vierbeinigen Robotern und Fahrzeugen gezeigt, weitere Roboter können über RoSiML, einer auf XML basierenden Beschreibungssprache, definiert werden. Die Dynamiksimulation basiert auf ODE, zur Visualisierung wird OpenGL benutzt. Es werden verschiedene Sensortypen bereitgestellt, darunter Kameras, Abstandssensoren und taktile Sensoren. Die Abstandssensoren werden durch Auslesen des Tiefenpuffers realisiert. Über die graphische Benutzeroberfläche ist es möglich, mit den Objekten zu interagieren und verschiedene Visualisierungen anzeigen zu lassen.

Webots [17] ist eine kommerziell verfügbare Simulationsumgebung. Es gibt eine umfangreiche Sensorbibliothek, verschiedene Umgebungen und viele Robotermodelle, darunter fahrende, laufende und fliegende Roboter. Über einen Editor können weitere Robotermodelle und Szenarien definiert werden. Die Physik-Simulation basiert auf ODE, die Visualisierung erfolgt mit OpenGL. Wie die Sensordaten simuliert werden, wird nicht genauer beschrieben.

Das *Microsoft Robotics Studio* [18] ist ein Paket für die Entwicklung von Roboter-Anwendungen. Es beinhaltet Werkzeuge, um Programme für reale Roboter zu entwickeln sowie eine Simulationsumgebung, in der die Programme getestet werden können. Die Simulationsumgebung kann entweder über eine graphische Benutzeroberfläche mittels Drag and Drop erstellt oder mit Hilfe

einer Programmiersprache wie C# erstellt werden. Hierfür stehen viele Basiselemente, Motoren und Sensoren zur Verfügung. Deren Umsetzung wird nicht beschrieben. Die Physik-Simulation basiert auf der AGEIA PhysX Technologie.

USARSim [4], [26] hat sich von einem Simulator, der auf fahrende Roboter in Rettungs-Szenarien spezialisiert war, zu einer vielseitigen Simulationsumgebung entwickelt. Die gesamte Simulation basiert auf der *Unreal Engine* von Epic Games [7], dies ist eine vollständige Game-Engine, die außer der Physiksimulation auch Visualisierungs-Werkzeuge und die Einbindung von Beschreibungssprachen bereitstellt. In der Unreal Engine 2 wird Karma eingesetzt, die Unreal Engine 3 benutzt die AGEIA PhysX-Technologie.
Es existieren bereits viele unterschiedliche Umgebungen, zahlreiche Sensoren wie z.B. Kameras, Abstandssensoren und taktile Sensoren sowie zahlreiche verschiedene Robotersysteme, darunter Fahrzeuge, U-Boote, fliegende und laufende Roboter. Über eine Beschreibungssprache können weitere Sensoren, Roboter oder Umgebungen modelliert werden. Die genaue Umsetzung der Sensoren ist nicht beschrieben. Es wird lediglich erwähnt, dass man Zugang zu allen in der Szene vorhandenen physikalischen Eigenschaften hat. Ein auf USARSim basierender Simulator wird in der RoboCup Rescue Simulation League eingesetzt.

Spark [19] ist ein Paket, mit dem unterschiedliche Simulationen erstellt werden können. Über verschiedene Beschreibungssprachen können Roboter und die Umwelt konfiguriert werden. Für die Dynamik-Simulation wird ODE benutzt. Sensoren stehen laut [19] noch keine zur Verfügung. In der RoboCup 3D Simulation League wird seit 2004 ein auf Spark basierender Simulator verwendet.

Gazebo [15] wird im Rahmen des *Player/Stage* [11] Projekts entwickelt. Player dient als Schnittstelle zwischen Software und Roboter bzw. Software und Simulator des Projektes. Stage ist eine 2D-Simulation für viele Roboter, Gazebo ist ein 3D Simulator, dessen Physik-Simulation auf ODE basiert. Es können neue Roboter modelliert werden und andere Szenen definiert werden. Es stehen 3 Sensortypen zur Verfügung: Kameras, Abstandssensoren und Odometrie für Fahrzeuge. Es gibt einen eindimensionalen Abstandssensor, der den Abstand zum nächsten Objekt entlang eines Strahls bestimmt. Mehrdimensionale Abstandssensoren wie Laserscanner und Radar bauen auf diesem Basismodul auf.

2.3 Spezialisierte Simulatoren

UCHILSIM [25] ist ein Simulator, der speziell für die RoboCup four-legged league entwickelt wurde. Daher ist außer dem Sony AIBO kein anderes Robotermodell verfügbar. Die Visualisierung erfolgt mit OpenGL, die Dynamiksimulation greift auf ODE zurück. Für die Kollisionserkennung werden zwei Ansätze verfolgt: Der exaktere Ansatz verwendet für jeden Starrkörper ein vereinfachtes Gittermodell. Der schnellere Ansatz nähert jeden Starrkörper entweder mit einer Kugel oder mit einem Quader an. Über die graphische Benutzeroberfläche können zur Laufzeit Objekte platziert und bewegt werden und es kann zwischen verschiedenen Visualisierungen hin-

und hergeschaltet werden. Außer der Kamera des AIBOs werden in [25] keine weiteren Sensoren erwähnt.

ÜberSim [3] ist ein Simulator, der speziell für die RoboCup Small Size League entwickelt wurde. Es stehen ein differentialgetriebenes und ein omnidirektionales Fahrzeug zur Verfügung. Da in der Small Size League eine zentrale Deckenkamera benutzt wird und die Roboter keine eigenen Sensoren haben, stehen auch in der Simulation keine Sensoren zur Verfügung. Die Dynamik-Simulation beruht auf ODE.

2.4 Zusammenfassung

Alle hier beschriebenen Simulatoren greifen entweder auf ODE oder eine andere Physik-Engine zurück. Dies bringt den Vorteil mit sich, dass man ein vollständiges System für die Physiksimulation nutzen kann, ohne dass man sich um die dahinterliegenden Algorithmen kümmern muss. Doch genau dort liegt auch der Nachteil: Man kann nicht je nach Bedarf verschiedene Algorithmen einsetzen, sondern ist darauf beschränkt, was die jeweilige Bibliothek bietet. Ein Austausch der Algorithmen ist nur mit großem Aufwand möglich.

Die meisten der hier betrachteten Simulatoren stellen auch Abstandssensoren zur Verfügung. Manche lesen hierfür mit OpenGL den Tiefenpuffer aus, andere suchen entlang eines Strahls nach dem nächsten Objekt, einige beschreiben nicht näher, wie die Abstandssensoren umgesetzt werden. Keiner hat beide Ansätze umgesetzt und die Performance miteinander verglichen. Es ist zu erwarten, dass je nach Situation ein Ansatz besser funktioniert als der andere. In dieser Arbeit werden daher die Möglichkeiten der Simulation vorgestellt und miteinander verglichen.

In Tabelle 2.1 werden alle vorgestellten Simulatoren bezüglich der für diese Arbeit wichtigen Eigenschaften miteinander verglichen.

Simulator	Physik-Engine	Umsetzung der Abstandssensoren
SimRobot	ODE	Auslesen des z-Buffers
Webots	ODE	keine Angabe
Microsoft Robotics Studio	AGEIA PhysX	keine Angabe
USARSim	Unreal Engine	keine Angabe
Spark	ODE	nicht vorhanden
Gazebo	ODE	Abstandsberechnung entlang Strahlen
UCHILSIM	ODE	nicht vorhanden
ÜberSim	ODE	nicht vorhanden

Tabelle 2.1: *Vergleich der vorgestellten Simulatoren*

Kapitel 3

Das Simulatorframework MuRoSimF

Das Simulatorframework MuRoSimF (Multi-Robot-Simulation-Framework) [10] wird seit 2005 am Fachgebiet Simulation und Systemoptimierung des Fachbereichs Informatik der TU Darmstadt von Martin Friedman entwickelt. In diesem Kapitel wird zunächst der Stand von MuRoSimF zu Beginn dieser Arbeit vorgestellt, im Anschluss werden die Neuerungen vorgestellt, die parallel zu dieser Diplomarbeit (und darauf aufbauend) entstanden sind.

Die Idee hinter MuRoSimF ist es, ein offenes modulares Framework für verschiedene Arten von Simulationen zu schaffen. Es soll einfach zu erweitern sein und alle verwendeten Algorithmen sollen leicht ausgetauscht werden können. Daher wird für die Dynamiksimulation, im Gegensatz zu allen in Kapitel 2 beschriebenen Simulatoren, nicht auf eine Bibliothek wie ODE oder auf eine andere Physik-Engine zurückgegriffen.

3.1 Ausgangslage von MuRoSimF

Die Welt und die Roboter werden als Sammlungen von Objekten repräsentiert. Dies sind sowohl sichtbare Objekte als auch unsichtbare Hilfsobjekte wie z.B. die Gelenke eines Roboters. Die sichtbaren Teile der Simulation sind aus verschiedenen Primitiven aufgebaut. Es stehen Kugeln, Quader, Zylinder und Ebenen zur Verfügung. Durch den modularen Aufbau von MuRoSimF können bei Bedarf leicht weitere Primitive hinzugefügt werden.

Jedes Objekt besteht aus einer Sammlung von Eigenschaften wie z.B. Position, Geschwindigkeit, Form oder Masse. Diese Liste kann auch nach der Initialisierung noch um weitere Eigenschaften ergänzt werden. Auf diese Art ist es möglich, jedes Objekt mit genau den Eigenschaften auszustatten, die es für die spezielle Simulation auch benötigt. Jeder Algorithmus erhält eine Liste von Objekten, auf denen er arbeitet. Er kann dann die benötigten Eigenschaften, falls sie noch nicht vorhanden sind, an ein Objekt anhängen.
Dieses Konzept hat den Vorteil, dass sehr leicht neue Eigenschaften definiert und benutzt werden

können. Die Objekte selbst müssen dafür nicht verändert werden, sie bekommen die neue Eigenschaft einfach in ihre Liste geschrieben. Algorithmen, die mit dieser neuen Eigenschaft arbeiten, können sie einfach auslesen, alle anderen Teile der Simulation sind von dieser Änderung nicht betroffen.

Die Visualisierung der Szene erfolgt mit OpenGL. Für jede Eigenschaft, wie z.B. Form, Farbe, Material oder Textur, gibt es ein eigenes Render-Modul. Alle Render-Module und alle darzustellenden Objekte werden bei einem zentralen `RenderManager` registriert. Der `RenderManager` ordnet jedem Render-Modul genau diejenigen Objekte zu, die auch die Eigenschaft, die das Modul visualisiert, besitzen (siehe Abbildung 3.1).
Dieses Vorgehen ermöglicht es, sehr einfach Einfluss darauf zu nehmen, wie detailliert eine Szene dargestellt wird. Durch Umkonfigurieren oder Austauschen eines Moduls kann die Visualisierung sofort verändert werden. Für zeitkritische Anwendungen ist es möglich, einzelne Module, die nicht unbedingt benötigt werden, wegzulassen.

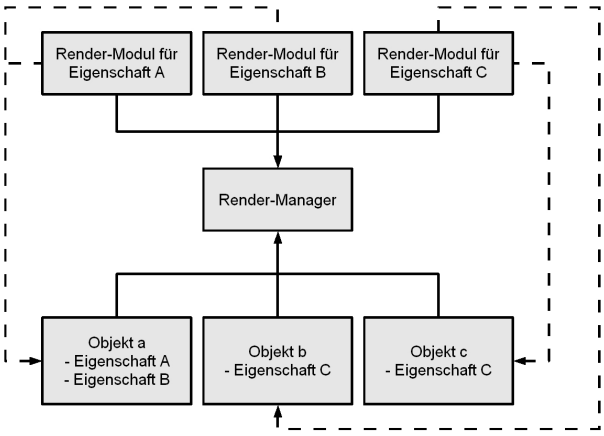


Abbildung 3.1: *Übersicht der Funktionsweise des `RenderManagers`. Alle Render-Module und alle Objekte werden beim `RenderManager` registriert (durchgezogene Linien). Der `RenderManager` ordnet dann jedem Render-Modul alle Objekte mit der passenden Eigenschaft zu (gestrichelte Linien).*

Die Kollisionserkennung besteht aus drei Komponenten:

- Der *`CollisionDetectionTask`* ist das zentrale Modul. Er ist dafür zuständig, die gesamte Aufgabe der Kollisionserkennung zu organisieren. Er kann dafür auf die beiden anderen Komponenten zugreifen. Im `CollisionDetectionTask` werden alle Objekte, die in die Kollisionserkennung einbezogen werden sollen, angemeldet. Die Objekte können entweder einzeln oder als Gruppe registriert werden. Darüber hinaus kann angegeben werden, ob die Objekte beweglich sind oder nicht und ob Kollisionen innerhalb einer Gruppe berechnet werden sollen.

- Die *Kollisionserkenner* führen die eigentliche Berechnung aus, ob zwei Primitive miteinander kollidieren oder nicht. Jeder Kollisionserkenner ist für ein bestimmtes Paar an Primitiv-Typen zuständig und stellt eine Funktion zur Verfügung die prüft, ob ein bestimmtes Objektpaar verarbeitet werden kann.
  Jede Instanz eines Kollisionserkenners kann einen eigenen Kollisionsbehandler haben. Ist eine Kollision berechnet worden, wird sofort eine Funktion zum Behandeln der Kollision aufgerufen.

- Die *Kollisionsbehandlung* berechnet die Kollisionsantwort und setzt diese um. Im aktuell verwendeten Algorithmus werden die aus den Kollisionen resultierenden Kräfte berechnet und auf die Objekte angewendet.

Die einzige Schnittstelle zur Simulation ist das Anmelden der Objekte im `CollisionDetectionTask`. Es ist also ohne großen Aufwand möglich, einzelne Teile der Kollisionserkennung durch andere zu ersetzen. Solange die Schnittstelle beibehalten wird, kann auch der gesamte `CollisionDetectionTask` ausgetauscht werden.

Im vorhandenen `CollisionDetectionTask` werden alle zur Verfügung stehenden Kollisionserkenner registriert. Wird ein neues Objekt angemeldet, werden alle möglichen Kollisionspartner bestimmt: Ist das neue Objekt unbeweglich, kommen nur die bereits vorhandenen beweglichen Objekte in Frage, andernfalls alle bereits vorhandenen Objekte. Ist das Objekt Teil einer Gruppe, innerhalb der Kollisionen auftreten können, ist auch jedes andere Mitglied der Gruppe ein möglicher Kollisionspartner. Für jedes zu schneidende Objektpaar wird, falls vorhanden, ein passender Kollisionserkenner erzeugt. Wird keiner gefunden, können die beiden Objekte keinen Einfluss aufeinander nehmen, sie können sich gegenseitig durchdringen. Jeder neu erzeugte Kollisionserkenner wird mit einem Kollisionsbehandler ausgestattet und in einer Liste abgelegt. In jedem Zeitschritt der Simulation prüft jeder Kollisionserkenner in dieser Liste, ob die ihm zugewiesenen Objekte kollidieren und ruft gegebenenfalls den zugehörigen Kollisionsbehandler zum Auflösen der Kollision auf. Dieses Vorgehen wird in Abbildung 3.2 illustriert.
Zu Beginn dieser Arbeit standen nur zwei Typen von Kollisionserkennern zur Verfügung: einer zur Berechnung von Kollisionen zwischen zwei Kugeln und einer zur Berechnung von Kollisionen zwischen einer Kugel und einer Ebene. Außerdem gab es einen einfachen Dynamikalgorithmus. Die Kollisionserkennung konnte also noch nicht für komplexe Simulationen verwendet werden.

Der Dynamikalgorithmus arbeitet mit weichen Kollisionen, d.h. Objekte können sich kurzfristig gegenseitig durchdringen. Als Kollisionsmodell werden zwei Federn zwischen den betroffenen Objekten gespannt, die die Objekte wieder auseinander drücken. Die Federkonstante errechnet sich für jedes Objekt individuell aus seiner Oberflächenbeschaffenheit.
Dieser Algorithmus kann auf einfache Art und Weise auch auf Mehrkörpersysteme angewendet werden: Es wird nur dem Teil des Mehrkörpersystems eine Masse und ein Trägheitstensor zugewiesen, der den Hauptteil des Systems repräsentiert. Die anderen Teile werden als masselos angenommen.

Roboter werden als Baumstrukturen modelliert. Sie bestehen aus einer Wurzel und einer be-

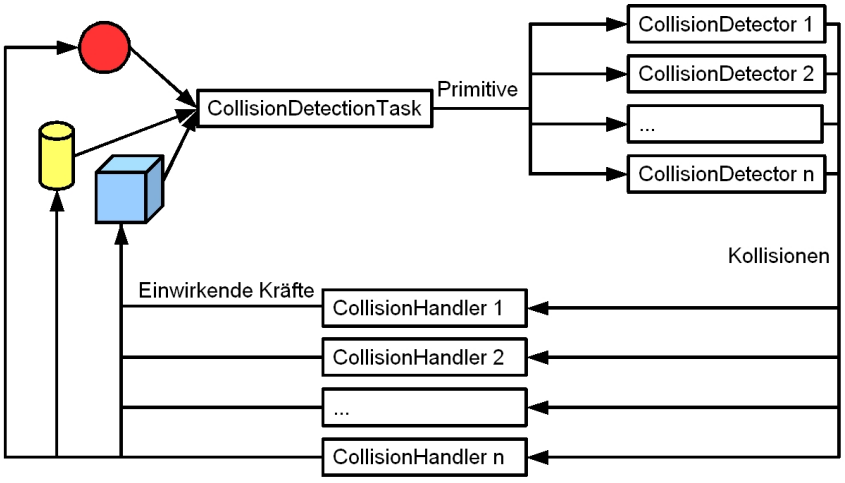


Abbildung 3.2: *Konzept der Kollisionserkennung: Jedes Objektpaar bekommt einen Kollisionserkenner zugewiesen. Die berechnete Kollision wird zur Kollisionsbehandlung weitergereicht. Die dort berechneten Kräfte werden auf die Objekte angewendet.*

liebigen Anzahl an Gabelungen, konstanten und variablen Translationen sowie konstanten und variablen Rotationen. Alle Objekte haben je nach Funktion gewisse feste Eigenschaften, weitere Eigenschaften wie z.B. Form und Farbe können während des Aufbaus des Baumes zusätzlich angehängt werden. Von außen kann man dann entweder auf den gesamten Baum zugreifen oder auf speziellen Untergruppen wie z.B. die Gelenke (alle variablen Translationen und Rotationen) und die Körper (alle Objekte, die eine physikalische Ausdehnung haben).
Es stehen zwei Robotermodelle zur Verfügung: ein Modell des am Fachgebiet für den RoboCup eingesetzten Humanoidroboters HR18 und das Modell eines Pioneer2-DX.
Für den HR18 gibt es eine Kinematiksimulation, mit der der Roboter über ebenes Gelände laufen kann. Es ist dabei nicht möglich, dass der Roboter umfällt. Bei diesem Algorithmus wird davon ausgegangen, dass sich immer mindestens ein Fuß des Roboters auf dem Boden (einer Ebene, die durch $z = 0$ definiert ist) befindet. Ein Fuß wird so lange als Standfuß angenommen, bis dies nicht mehr möglich ist, da sich der andere Fuß weiter unten befindet. Man kann zwar Situationen, in denen keiner der beiden Füße mehr als Standfuß in Frage kommt, konstruieren, doch solche Konfigurationen kommen bei den realen Bewegungen des Roboters nur sehr selten vor, insbesondere bei der Laufbewegung des HR18 treten sie gar nicht auf. Mit diesem Algorithmus lässt sich die Laufbewegung des echten Roboters direkt in der Simulation ausführen und man erhält vergleichbare Ergebnisse.
Auch für den Pioneer steht eine Kinematiksimulation zur Verfügung, mit der er über ebenes Gelände fahren kann. Aus den Drehgeschwindigkeiten der Räder wird über das Kinematikmodell die resultierende Bahn berechnet und auf die Basis des Modells übertragen.
Diese beiden Algorithmen funktionieren in ihrem Einsatzbereich gut, doch sie sind stark auf die jeweiligen Roboterstrukturen angepasst und lassen sich nicht auf andere Strukturen übertragen.

Der Laufalgorithmus kann z.B. nicht für vierbeiniges Laufen erweitert werden. Außerdem können keine äußeren Einflüsse wie z.B. Hindernisse die resultierende Bewegung beeinflussen, der Roboter läuft bzw. fährt einfach durch alles hindurch.

Als einzige Sensoren stehen Kameras zur Verfügung. Die Szene wird auf die gleiche Art wie die Hauptkamera aus dem Blickwinkel der Kamera gerendert. Nachträglich können noch Verzerrungen, wie sie durch unterschiedliche Linsen verursacht werden, auf das Bild angewendet werden.

Ein mit `MuRoSimF` erstellter Simulator wird vom RoboCup-Humanoid Team am Fachgebiet Simulation und Systemoptimierung, den Darmstadt Dribblers [9], zum Testen des Verhaltens der Roboter verwendet. Eine Aufnahme des Simulators ist in Abbildung 3.3 zu sehen. Für diese

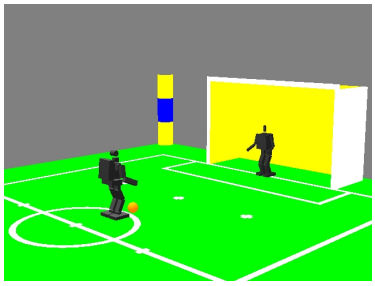


Abbildung 3.3: *Eine Aufnahme aus der RoboCup-Simulation.*

Tests ist die Kinematiksimulation des Roboters sehr gut geeignet, da ein ständiges Umfallen des Roboters nur störend wäre (außer natürlich beim Torwart). Aufgrund der fehlenden Kollisionserkenner war es vor Beginn dieser Arbeit nicht möglich, den Ball zu kicken. Er musste stattdessen immer mit der Hand bewegt werden.

Die Kinematiksimulation des Pioneers und des HR18 wurde in [13] und [14] angewendet, um die Kooperation heterogener Systeme zu zeigen.

3.2 Spätere Erweiterungen

Da im Rahmen dieser Arbeit alle fehlenden Kollisionserkenner entwickelt wurden, kann die gesamte Kollisionserkennung jetzt eingesetzt werden. Das bedeutet z.B. für den auf `MuRoSimF` aufbauenden RoboCup-Simulator, dass der Roboter den Ball jetzt kicken kann und der Torwart sich auf den Boden werfen kann, siehe Abbildung 3.4.

Der Dynamikalgorithmus konnte für viele verschiedene Robotersysteme getestet werden. Da für diesen Algorithmus auch für Mehrkörpersysteme ein konstanter Masseschwerpunkt angenommen wird, konnten vor allem die Bewegungen des HR18 nur schwer umgesetzt werden. Es wurde daher ein weiterer Dynamikalgorithmus entwickelt, bei dem man in einem Mehrkörpersystem

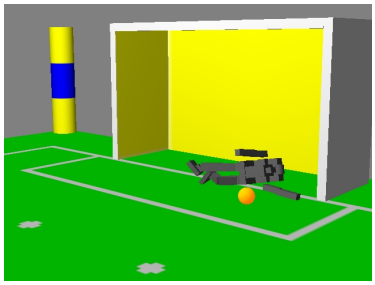


Abbildung 3.4: *Der Torwart, wie er einen Ball hält.*

mit verteilten Massen arbeiten kann. In jedem Zeitschritt wird der neue Masseschwerpunkt berechnet. Dadurch kann jetzt auch das Ausbalancieren einer Bewegung mit den Armen simuliert werden. Für die Kräfteberechnung bei Kollisionen wurde auch für den neuen Algorithmus auf das Federmodell zurückgegriffen.

Durch den modularen Aufbau von MuRoSimF kann man für jeden simulierten Roboter getrennt auswählen, welcher Algorithmus für die Bewegungssimulation verwendet werden soll. Für ein Fahrzeug, dessen Masseschwerpunkt sich nie ändert, reicht der erste Dynamikalgorithmus vollkommen aus, der zweite würde keine Verbesserung bringen. Für die RoboCup-Simulation kann man für den Feldspieler die Kinematiksimulation auswählen und gleichzeitig für den Torwart die Dynamiksimulation. Wenn man die Füße des Feldspielers bei der Kollisionserkennung anmeldet, kann der Ball jetzt auch gekickt werden.

Kapitel 4

Grundlagen für die Kollisionserkennung

In der Kollisionsberechnung wird immer wieder auf die selben Grundelemente zurückgegriffen, die in diesem Kapitel vorgestellt werden.
Es müssen regelmäßig die kürzesten Abstände zwischen den Grundelementen berechnet werden, aus denen die Primitive aufgebaut sind. Dies sind z.B. Punkte, Strecken und Flächen. Daher ist es sehr wichtig, diese Berechnungen schnell und robust durchführen zu können.
Die meisten Algorithmen nutzen die Tatsache aus, dass die verwendeten Primitive konvex sind. Aus den daraus folgenden Eigenschaften können Algorithmen abgeleitet werden, die nur für konvexe Objekte funktionieren, wie z.B. das *Separating Axis Theorem*, das am Ende dieses Kapitels vorgestellt wird.

4.1 Abstandsberechnungen

Für alle Kollisionsberechnungen werden Abstandsberechnungen zwischen grundlegenden Elementen wie Punkten, Geraden oder Strecken benötigt. Die wichtigsten Ideen werden hier vorgestellt. Die exakte Herleitung, weitere Abstandsberechnungen sowie Hinweise zur robusten Implementierung sind in [8] zu finden.

4.1.1 Abstand Punkt-Ebene

Gegeben ist eine Ebene E, definiert durch einen Punkt P auf der Ebene und einen Normalenvektor n. Dann gilt für alle Punkte X auf der Ebene die Gleichung $(X - P) \cdot n = 0$. Ist Q ein beliebiger Punkt im Raum, dann erhält man den dichtesten Punkt R auf der Ebene durch verschieben von Q entlang n, also durch Projektion von Q auf E. Es gilt also $R = Q - tn$ für einen

bestimmten Wert t. Setzt man dies in die Gleichung der Ebene ein, erhält man

$$\begin{aligned} & ((Q - tn) - P) \cdot n = 0 \\ \Leftrightarrow \quad & Q \cdot n - t(n \cdot n) - P \cdot n = 0 \\ \Leftrightarrow \quad & (Q - P) \cdot n = t(n \cdot n) \\ \Leftrightarrow \quad & t = ((Q - P) \cdot n)/(n \cdot n) \end{aligned}$$

Wenn n normiert ist, gilt $n \cdot n = 1$ und man erhält als Abstand von Q zu E

$$t = (Q - P) \cdot n.$$

4.1.2 Abstand Punkt-Gerade

Für die Abstandsberechnung zwischen einem Punkt und einer Geraden wird die geometrische Interpretation des Skalarprodukts benötigt. Für zwei Vektoren u und v kann das Skalarprodukt zwischen u und v als die Projektion von v auf u gesehen werden, man erhält $\|u\|$-mal die Länge von v entlang u (vgl. Abbildung 4.1). Bilden u und v einen stumpfen Winkel, ist dieser Wert negativ.

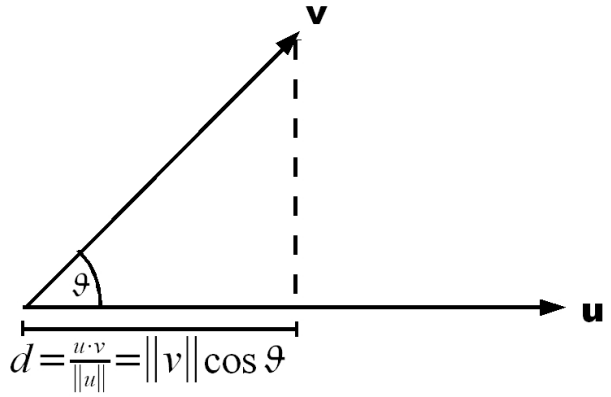


Abbildung 4.1: *Geometrische Interpretation des Skalarprodukts.*

Betrachtet man nun eine Gerade, die durch die beiden Punkte A und B definiert ist, kann jeder Punkt auf dieser Geraden ausgedrückt werden als $P(t) = A + t(B - A)$. Ist Q ein beliebiger Punkt im Raum, dann erhält man den auf der Gerade am nächsten liegenden Punkt R, indem man den Verbindungsvektor $(Q - A)$ auf die Gerade projiziert. Wegen der oben beschriebenen geometrischen Interpretation des Skalarprodukts ist der Abstand von R zu A also $d = (Q - A) \cdot (B - A)/\|B - A\|$. Um dies in die Geradengleichung einsetzen zu können, muss d noch mit $\|B - A\|$ skaliert werden. Man erhält also $R = A + t(B - A)$ mit $t = (Q - A) \cdot (B - A)/\|B - A\|^2$. Der Abstand von Q zur Geraden ist dann der Abstand von Q zu R.

4.1.3 Abstand Punkt-Strecke

Um den Abstand eines Punktes Q zu einer Strecke AB zu bestimmen, wird zunächst die Projektion R von Q auf die Gerade AB berechnet wie in Kapitel 4.1.2 beschrieben. Es werden jetzt drei Fälle unterschieden:

- $t < 0$, dann liegt R außerhalb der Strecke AB auf der Seite von A, der nächste Punkt auf der Strecke zu Q ist also A,
- $0 \leq t \leq 1$, dann liegt R auf der Strecke AB,
- $t > 1$, dann liegt R außerhalb der Strecke AB auf der Seite von B, der nächste Punkt auf der Strecke zu Q ist also B.

Wird t auf den Bereich $[0, 1]$ eingeschränkt, erhält man mit $R = t(B - A)$ den nächsten Punkt auf der Strecke AB zu Q.

4.1.4 Abstand Punkt-Dreieck

Um den nächsten Punkt in einem Dreieck ABC zu einem Punkt P zu bestimmen, wird P zunächst auf die durch ABC definierte Ebene projiziert, man erhält einen Punkt Q. Es muss nun berechnet werden, in welcher Voronoi-Region des Dreiecks Q liegt. Die Voronoi-Regionen erhält man, indem man in jeder Ecke senkrecht zu jeder Kante eine Ebene aufspannt, also z.B. für die Ecke A zwei Ebenen durch A, eine mit Normale $B - A$ und die andere mit Normale $C - A$. Die Voronoi-Region der Ecke ist dann der Durchschnitt der beiden negativen Halbräume. In Abbildung 4.2 liegt der Mond in der Voronoi-Region der Ecke A, die Blume liegt in der Voronoi-Region der Kante BC.
Um zu berechnen, ob Q in der Voronoi-Region einer Ecke von ABC liegt, muss nur getestet werden, ob Q in beiden Halbräumen liegt, die diese Region erzeugen. In diesem Fall ist die zugehörige Ecke der nächste Punkt zu P. Andernfalls muss berechnet werden, ob Q in der Voronoi-Region einer Kante von ABC liegt. Hierzu werden die baryzentrischen Koordinaten u, v, w von Q bezüglich ABC bestimmt. Es gilt $Q = uA + vB + wC$ und $u + v + w = 1$. Damit Q in der Voronoi-Region der Kante AB liegt, muss $w < 0$ sein, außerdem muss Q in den positiven Halbräumen der Ebenen $(X-A)(B-A) = 0$ und $(X-B)(A-B) = 0$ liegen. Analoges gilt für die beiden anderen Kanten. Dann muss nur noch wie in Kapitel 4.1.3 beschrieben der Abstand von P zu der Kante berechnet werden, in deren Voronoi-Region Q liegt. Liegt Q in keiner der Kanten-Regionen, gilt $u > 0, v > 0$ und $w > 0$ und Q liegt im Inneren des Dreiecks und ist der nächste Punkt zu P.

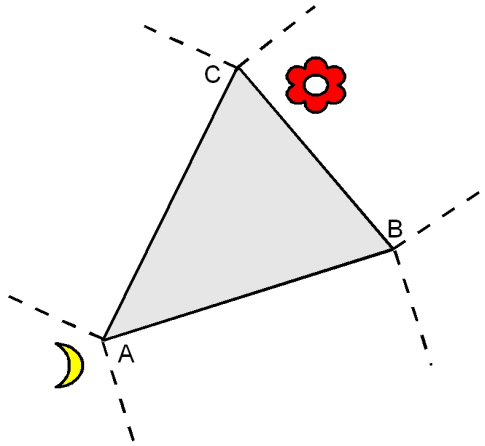


Abbildung 4.2: *Darstellung der Voronoi-Regionen eines Dreiecks.*

4.1.5 Abstand Punkt-Rechteck

Gegeben sind ein Punkt P und ein Rechteck, definiert durch seinen Mittelpunkt M, zwei aufspannende Vektoren e_1 und e_2 und die halbe Ausdehnung in die beiden Richtungen, l_1 und l_2. In dem von e_1, e_2 und $e_1 \times e_2$ aufgespannten Koordinatensystem ist das Rechteck also gegeben durch $[-l_1, l_1] \times [-l_2, l_2] \times [0]$. Stellt man jetzt P in Rechteck-Koordinaten dar, muss man nur die Koordinaten auf diesen Bereich einschränken und man erhält den Punkt des Rechtecks, der am nächsten zu P liegt, vgl. auch Abbildung 4.3.

Dieses Verfahren lässt sich sofort auf Quader verallgemeinern, indem man die z-Koordinate bezüglich des Rechtecks nicht auf 0 setzt, sondern auch hier einen Bereich $[-l_3, l_3]$ zulässt.

4.1.6 Abstand Strecke-Ebene

Man berechnet zunächst den Abstand der beiden Endpunkte der Strecke zur Ebene d_1 und d_2 wie in Kapitel 4.1.1 beschrieben. Liegen beide Punkte auf der gleichen Seite der Ebene (gilt also $d_1 \cdot d_2 > 0$), dann ist der Punkt mit dem kleineren Abstand am nächsten an der Ebene. Andernfalls gibt es einen Schnittpunkt, dieser kann z.B. mit Hilfe der Strahlensätze (siehe [2]) berechnet werden.

4.1.7 Abstand zwischen zwei Geraden

Zwei Geraden im Raum schneiden sich fast nie. Um den geringsten Abstand zwischen zwei Geraden $g_1 = A_1 + s \cdot e_1$ und $g_2 = A_2 + t \cdot e_2$ zu bestimmen, betrachtet man eine Hilfsebene E, die g_2 beinhaltet und parallel zu g_1 ist. E ist also durch den Punkt A_2 und die Richtungen e_1 und

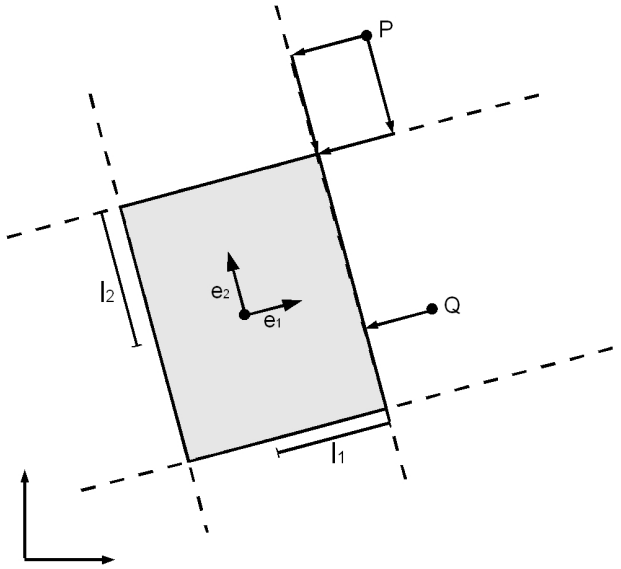


Abbildung 4.3: *Die Koordinaten der Punkte P und Q werden im Rechteck-Koordinatensystem auf den Bereich $[-l_1, l_1] \times [-l_2, l_2]$ eingeschränkt.*

e_2 definiert. Die Normale von E ist dann $e_1 \times e_2$. Da g_1 und E parallel sind, ist der Abstand von g_1 zu E in jedem Punkt von g_1 der gleiche, insbesondere in dem unbekannten Punkt P, der am nächsten an g_2 liegt. Der kürzeste Abstand zwischen g_1 und g_2 stimmt also überein mit dem Abstand eines beliebigen Punktes von g_1 zur Ebene E.
Der Sonderfall, dass die beiden Geraden parallel zueinander sind, muss getrennt behandelt werden, da in diesem Fall die Ebene nicht eindeutig definiert ist. Da der Abstand zwischen zwei parallelen Geraden überall gleich ist, kann man einfach den Abstand eines beliebigen Punktes von g_1 zu g_2 berechnen wie in Kapitel 4.1.2 beschrieben.

Möchte man zusätzlich zum Abstand zwischen den beiden Geraden auch die Punkte bestimmen, die am nächsten an der jeweils anderen Gerade liegen, nutzt man die Tatsache aus, dass der Verbindungsvektor zwischen diesen beiden Punkten senkrecht auf beiden Geraden steht. In diese Bedingung setzt man die Parametergleichungen der Geraden ein und löst nach den Parametern s und t auf. Eine genaue Herleitung der Formel ist in [8] nachzulesen.

4.1.8 Abstand zwischen zwei Strecken

Um den Abstand zwischen zwei Strecken zu bestimmen, muss zunächst der Abstand zwischen den durch die Strecken definierten Geraden berechnet werden. Außerdem werden die Punkte auf den Geraden benötigt, in denen der Abstand minimal wird.
Liegen diese beiden Punkte auch auf den Strecken, muss nichts weiter berechnet werden. Liegt ein Punkt auf der Strecke, der andere aber außerhalb, reicht es nicht, den Punkt einfach bis zum

Endpunkt der Strecke zu verschieben. Es muss dann ausgehend von diesem Punkt der nächste Punkt auf der anderen Strecke bestimmt werden. Liegen beide Punkte nicht auf den Strecken, muss diese Prozedur sogar zweimal ausgeführt werden, wie in Abbildung 4.4 zu sehen ist.

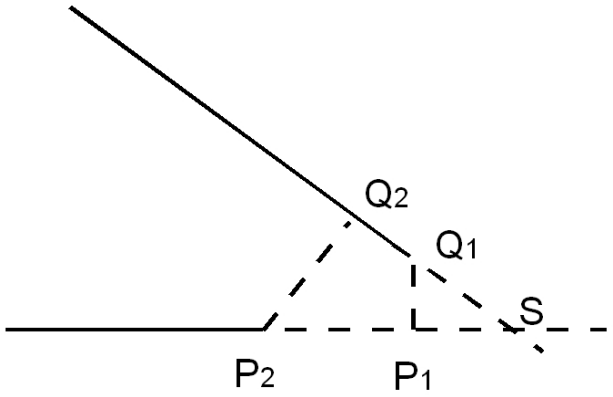


Abbildung 4.4: *S liegt außerhalb beider Strecken. Es muss zweimal der nächste Punkt auf der jeweils anderen Strecke berechnet werden.*

4.2 Konvexe Mengen

Definition: Eine Menge M heißt *konvex*, wenn für alle Punkte $x, y \in M$ und $0 \leq \lambda \leq 1$ gilt

$$(\lambda x + (1 - \lambda)y) \in M$$

Anschaulich bedeutet das, dass für je zwei Punkte in der Menge auch die gesamte Verbindungsstrecke in der Menge liegt, vgl. Abbildung 4.5.

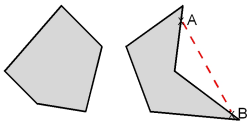


Abbildung 4.5: *Links eine konvexe Menge. Die Menge auf der rechten Seite ist nicht konvex, da z.B. die Verbindungsstrecke von A und B nicht in der Menge liegt.*

Ist eine Menge konvex, stimmt sie mit ihrer konvexen Hülle überein, denn um die konvexe Hülle einer Menge zu erhalten, werden die Verbindungsstrecken zwischen allen möglichen Punktepaaren zur Menge hinzugefügt. Da diese aber in einer konvexen Menge per Definition schon enthalten sind, wird nichts hinzugefügt, die konvexe Hülle stimmt also mit der Menge selbst

überein. Das bedeutet, dass alle Algorithmen, die mit der konvexen Hülle einer Menge arbeiten, ohne Mehraufwand direkt auf der Menge selbst ausgeführt werden können.

Vor allem für die Kollisionsberechnung von großer Bedeutung ist der Trennungssatz. Hierfür wird zunächst die Definition einer Trennungsebene benötigt.

Definition: Trennungsebene:
Gegeben sind zwei konvexe Mengen A und B. Eine Ebene E heißt *Trennungsebene* für A und B, wenn A vollständig in einem und B vollständig im anderen von E erzeugten Halbraum liegt.

Trennungssatz:
Sind zwei konvexe Mengen A und B disjunkt, dann existiert eine Ebene E, die A und B trennt.

Der Beweis ist in [5] nachzulesen.
Der Trennungssatz sagt aus, dass man zwischen zwei disjunkte konvexe Mengen eine Ebene legen kann, sodass jede Menge auf einer anderen Seite der Ebene liegt. Diese Tatsache wird im Separating Axis Theorem (vgl. Kapitel 4.3) verwendet. Für nicht konvexe Mengen kann man nicht immer eine solche Ebene finden, siehe Abbildung 4.6.

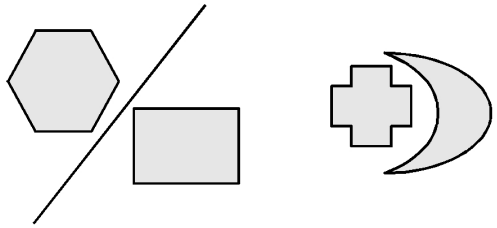


Abbildung 4.6: *Die beiden konvexen Mengen links lassen sich durch eine Gerade trennen. Die beiden Mengen auf der rechten Seite sind zwar disjunkt, es kann aber trotzdem keine Gerade gefunden werden, die sie trennt.*

4.3 Separating Axis Theorem

Der offensichtlichste Ansatz um herauszufinden, ob zwei Polyeder A und B sich schneiden ist, jede Seite von A mit jeder Fläche von B zu schneiden und umgekehrt. Doch dies ist bereits für Polyeder mit wenigen Ecken sehr aufwendig. Möchte man z.B. zwei Quader gegeneinander schneiden, muss man zweimal 12 Kanten gegen 6 Flächen schneiden, es müssen also insgesamt 144 Schnitttests durchgeführt werden. Eine schnellere Variante bietet das *S*eparating Axis Theorem [12].

Definition: separierende Achse:
Gegeben sind zwei konvexe Polyeder A und B. Eine Gerade g heißt separierende Achse, wenn sich die Projektionen von A und B auf g nicht überlappen.

Separating Axis Theorem
Seien A und B konvexe Polyeder. Gibt es eine separierende Achse g, dann sind A und B disjunkt. Sind A und B disjunkt, dann gibt es eine separierende Achse, die entweder

- senkrecht zu einer Fläche von A,
- senkrecht zu einer Fläche von B oder
- senkrecht zu je einer Kante von A und B

ist.

Sind zwei konvexe Polyeder A und B disjunkt, so gibt es zu jeder separierenden Achse g eine Trennungsebene E mit Normale g und umgekehrt.

Wenn man in der Praxis einen Test durchführt, ob eine gegebene Achse zwei Quader trennt, projiziert man nicht die Quader auf die Achse und betrachtet die Lage der Projektion, sondern man summiert für beide Quader die Länge der Projektion der drei Hauptachsen und erhält einen Wert s_1 für den ersten Quader und einen Wert s_2 für den zweiten Quader. Dann berechnet man die Länge s_3 der Projektion des Verbindungsvektors der Mittelpunkte der Quader. Die Achse ist genau dann eine Trennungsachse, wenn $s_1 + s_2 < s_3$ gilt, siehe Abbildung 4.7.

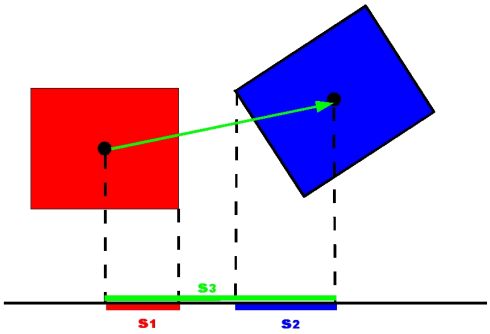


Abbildung 4.7: *Graphische Darstellung des Tests, ob eine Achse Trennungsachse ist.*

Kapitel 5

Kollisionserkennung

Die Aufgabe der Kollisionserkennung ist es herauszufinden, ob sich zwei oder mehr Objekte in einer Szene schneiden. Hierzu muss zunächst die Frage beantwortet werden, *ob* sich die Objekte schneiden, und falls dies der Fall ist, *wo* und *wie* sie sich schneiden.
Kollisionserkennung ist eine sehr komplexe Aufgabe, daher können hier nicht alle Aspekte detailliert beschrieben werden. Für genauere Ausführungen wird auf [8] verwiesen.

Der einfachste Ansatz, jedes Objekt mit jedem anderen zu schneiden, führt bei steigender Anzahl der Objekte wegen des quadratischen Aufwands schnell zu Performance-Problemen. In diesem Kapitel werden zunächst die bekanntesten Ansätze zur Beschleunigung der Kollisionserkennung vorgestellt. Danach werden die einzelnen Kollisionserkenner für Primitive vorgestellt, die im Rahmen dieser Arbeit für `MuRoSimF` entwickelt wurden, im Anschluss wird die Umsetzung der Beschleunigung der Kollisionserkennung für `MuRoSimF` diskutiert.

5.1 Hüllkörperhierarchien

Eine Hüllkörperhierarchie ist eine baumförmige Struktur, der in jedem Knoten ein Hüllkörper zugeordnet ist, der alle im Baum darunter liegenden Objekte umfasst. Um festzustellen, ob ein anderes Objekt mit Objekten des Baumes kollidiert, muss man es nicht gegen jedes einzelne Objekt schneiden, sondern nur gegen die Hüllkörper. Kollidiert ein Objekt nicht mit einem Hüllkörper, kann es auch mit keinem der darunter liegenden Objekten kollidieren. Ein exakter Test zwischen zwei Objekten muss nur dann durchgeführt werden, wenn sich auch auf der tiefsten Stufe noch die Hüllkörper überschneiden.

5.1.1 Hüllkörper

Die Wahl des Hüllkörpers spielt eine große Rolle. Man muss versuchen einen Hüllkörper zu wählen, der sich gut an das Objekt anpasst, der nach Möglichkeit bei Rotation des Objekts nicht neu berechnet werden muss und mit dem sich schnelle Schnitttests durchführen lassen. Außerdem sollte der Speicheraufwand möglichst gering gehalten werden. Der Hüllkörper sollte sich leicht erstellen lassen, um bei der Initialisierung und bei eventuell nötigen Neuberechnungen, z.B. bei Aktualisierung einer Hüllkörperhierarchie, Rechenzeit zu sparen. Je nach Situation sind manche Hüllkörper besser geeignet als andere, doch es gibt keinen Hüllkörper, der in jeder Situation optimal ist.

5.1.1.1 Hüllkugeln

Der einfachste Hüllkörper ist die Kugel. Der Speicheraufwand ist sehr gering, da nur der Mittelpunkt und der Radius gespeichert werden müssen. Für nicht verformbare Objekte muss die Hüllkugel auch bei Rotation und Translation des Objekts nicht neu berechnet werden, es muss lediglich der Mittelpunkt der Kugel verschoben werden. Die Kollisionsberechnung ist sehr schnell: zwei Kugeln schneiden sich nicht, wenn der Abstand ihrer Mittelpunkte größer ist, als die Summe ihrer Radien. Um aus einer gegebenen Punktwolke die kleinste Hüllkugel zu bestimmen, werden in [8] verschiedene Ansätze vorgestellt, der beste ist der Algorithmus von Welzl [24] mit einer durchschnittlichen linearen Laufzeit. Allerdings ist die Hülleffizienz für die meisten Objekte, vor allem für lange schmale Objekte, sehr schlecht, daher ist die Kugel als Hüllkörper in vielen Fällen ungeeignet.

5.1.1.2 Achsparalleler Quader

Ein achsparalleler Quader (auch AABB, Axis Aligned Bounding Box) ist ein Quader, dessen Flächennormalen mit den Koordinatenachsen des Weltkoordinatensystems zusammenfallen. Er lässt sich einfach erstellen, indem man jeweils den größten und kleinsten Wert des Objekts auf den Koordinatenachsen betrachtet. Bei einer Punktwolke mit vielen Punkten steigt allerdings der Aufwand sehr schnell an. In [8] werden weitere Verfahren vorgestellt, um AABBs zu erstellen oder um nach Rotation aus dem alten Quader einen neuen zu berechnen.
Die Kollisionsberechnung zwischen zwei achsparallelen Quadern ist sehr einfach: sie schneiden sich genau dann, wenn auf jeder Koordinatenachse ihre Projektionen überlappen, siehe auch Abbildung 5.1. Man sieht, dass die Projektionen des blauen und des gelben Rechtecks sich auf der x-Achse überlappen, auf der y-Achse allerdings nicht. Die Projektionen des gelben und des roten Rechtecks hingegen überlappen sich auf beiden Achsen. Dieses Vorgehen ist ein Spezialfall des Separating Axis Theorems, das in Kapitel 4.3 vorgestellt wird. Die Hülleffizienz von achsparallelen Quadern ist meist besser als von Kugeln, aber trotzdem noch nicht sehr gut. Außerdem müssen sie bei jeder Rotation des Objekts neu berechnet werden.

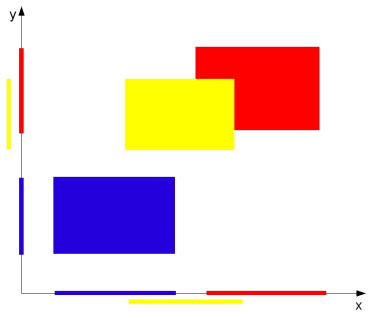


Abbildung 5.1: *Überlappungstest von AABBs*

5.1.1.3 Gerichtete Quader

Gerichtete Quader (auch OBB, Oriented Bounding Box) hüllen die Objekte wesentlich besser ein als achsparallele Quader. Bei Rotation des Objektes kann der Quader einfach mitrotiert werden. Allerdings ist das Erstellen eines guten Quaders nicht trivial. Es gibt Algorithmen [8], die in der Lage sind, die optimale Bounding Box zu erstellen, doch sie sind durch ihre hohe Laufzeit nicht praktikabel. In [8] werden weitere Methoden vorgestellt um Bounding Boxen zu finden, deren Volumen kaum größer ist als das des optimalen Quaders. Der Vorteil des schnelleren Erstellens überwiegt dabei den Nachteil, dass die Ergebnisse nicht den theoretisch möglichen optimalen Quadern entsprechen.
Der Kollisionstest zwischen zwei gerichteten Quadern kann mit dem Separating Axis Theorem (siehe Kapitel 4.3) erfolgen. Für maximal 15 Achsen muss getestet werden, ob sich die Projektionen der beiden Quader auf den Achse überlappen oder nicht. Diese 15 Achsen sind die drei Koordinatenachsen des ersten Quaders, die drei Koordinatenachsen des zweiten Quaders und die 9 Achsen, die zu jeweils einer Achse des ersten und einer Achse des zweiten Quaders senkrecht sind.

5.1.1.4 k-diskrete orientierte Polytope (k-DOPs)

Ein k-DOP ist ein verallgemeinerter Fall des achsparallelen Quaders, den man als 6-DOP betrachten kann. Um eine bessere Hülleffizienz zu erzielen, wird das Objekt mit k Halbebenen mit $k/2$ verschiedenen (aber festen) Normalen angenähert. Je größer k gewählt wird, um so besser passt sich der Hüllkörper an das Objekt an, für große k wird die konvexe Hülle des Objekts immer besser angenähert. Um ein k-DOP zu erstellen, können die Methoden der achsorientierten Quader verallgemeinert werden. Bei Rotation des Objekts muss das k-DOP neu berechnet werden. Für Überlappungstests kann auch hier auf das Separating Axis Theorem (siehe Kapitel 4.3) zurückgegriffen werden. Da die Normalen aber bei allen Objekten übereinstimmen und die Ausdehnung entlang dieser Achsen bekannt ist, ist der Test auf gleiche Art wie bei achsparallelen Quadern durchführbar und ist daher schneller als für orientierte Quader.

Hüllkörper	Hülleffizienz	Kollisionsberechnung	Neuberechnung bei Rotation erforderlich	Erstellung
Kugel	sehr schlecht	sehr einfach	nein	mittel
AABB	schlecht	einfach	ja	einfach
OBB	gut	mittel	nein	schwer
k-DOP	sehr gut	mittel	ja	mittel

Tabelle 5.1: *Vor- und Nachteile verschiedener Hüllkörper*

5.1.1.5 Zusammenfassung

Man sieht (vgl. hierzu auch Abbildung 5.2 und Tabelle 5.1), dass jeder der vorgestellten Hüllkörper sowohl Vor- als auch Nachteile mit sich bringt. Je nach Szenario eignen sich manche Hüllkörper besser als andere. Wenn die Hülleffizienz nicht wichtig ist, da die Objekte meistens weit voneinander entfernt sind, eignen sich Hüllkugeln sehr gut. Vor allem in zweidimensionalen Szenen ist es oft von Vorteil, achsorientierte Quader zu verwenden, da sie hier besonders leicht zu erstellen sind und sich die Algorithmen zur Neuberechnung nach Rotation des Objekts stark vereinfachen. Wenn der Speicheraufwand nicht wichtig ist, aber möglichst viele Kollisionspaare frühzeitig ausgeschlossen werden sollen, eigenen sich orientierte Quader oder k-DOPs sehr gut. Hier kann wieder entschieden werden, was für die jeweilige Szene günstiger ist: ein schneller Überlappungstest zwischen den Hüllkörpern oder der Vorteil, die Hüllkörper nur einmal zur Initialisierung zu berechnen.

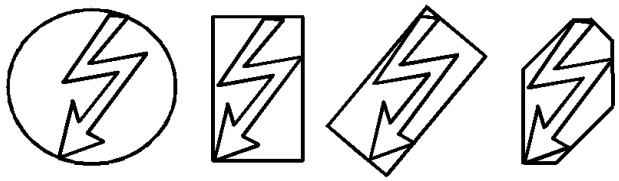


Abbildung 5.2: *Verschiedene Hüllkörper*

Hüllkörperhierarchien können homogen oder heterogen sein, d.h. es wird entweder für jeden Knoten der gleiche Hüllkörpertyp verwendet oder in jedem Knoten ein gut geeigneter. Homogene Hierarchien sind einfacher zu verwalten und die Tests sind schnell durchführbar, heterogene Hierarchien passen sich besser an die Objekte an und schließen somit mehr Kollisionen aus, doch sie verbrauchen auch mehr Rechenzeit.

5.1.2 Aufbau einer Hüllkörperhierarchie

Jedes Objekt in einen Hüllkörper zu verpacken kann zwar auch schon die Performance verbessern, doch es werden immer noch gleichviele paarweise Tests durchgeführt. Der Aufwand wird also nur um einen konstanten Faktor verringert.

Wenn man die Objekte in einer Baumstruktur organisiert, müssen Kindknoten nur dann gegeneinander geschnitten werden, wenn sich ihre Elternknoten schneiden. Hierfür wird jedes Objekt als ein Blatt des Baumes definiert, daraus wird mit verschiedenen Techniken der Baum aufgebaut. Eine Beispielszene und der resultierende Baum sind in Abbildung 5.3 zu sehen.

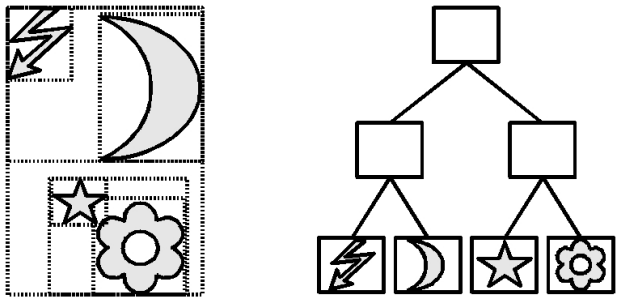


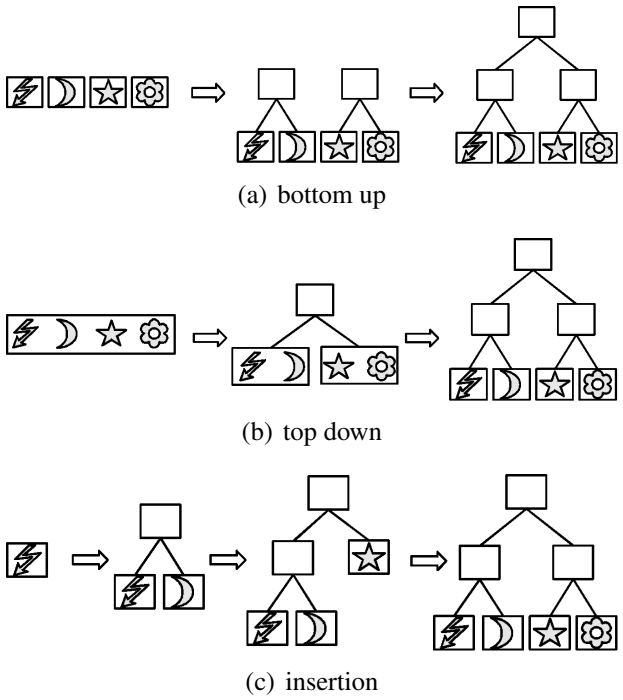
Abbildung 5.3: *Eine Szene mit zugehöriger Hüllkörperhierarchie. Hier wurden achsorientierte Quader als Hüllkörper verwendet.*

Auch für Hüllkörperhierarchien gibt es verschiedene Kriterien, die möglichst gut erfüllt werden sollten um eine gute Performance zu erzielen. Der Baum sollte so aufgebaut werden, dass die Knoten in jedem Teilbaum möglichst nah beieinander liegen. Jeder Knoten sollte einen Hüllkörper mit möglichst geringem Volumen haben, gleichzeitig sollte die Summe der Volumen aller Hüllkörper möglichst gering sein. Hüllkörper auf gleicher Stufe sollten sich möglichst wenig überschneiden um räumliche Redundanzen gering zu halten. Der Baum sollte bezüglich Struktur und Inhalt gut ausbalanciert sein, damit zu einem möglichst frühen Zeitpunkt viele Objekte als Kollisionspartner ausgeschlossen werden können.

Da die Anzahl der möglichen Bäume exponentiell mit der Anzahl der Objekte steigt, ist eine erschöpfende Suche nach dem besten Baum nicht durchführbar. Doch es gibt verschiedene Heuristiken, die mit wesentlich geringerem Aufwand Bäume erzeugen, die der optimalen Lösung sehr nahe kommen. Sie können in die Kategorien *bottom up*, *top down* und *insertion* unterteilt werden. Diese Verfahren werden im Folgenden beschrieben und in Abbildung 5.4 illustriert.

5.1.2.1 bottom up

Beim bottom up Verfahren beginnt man mit der gesamten Menge der Objekte, definiert jedes als ein Blatt des Baumes und legt um jedes einen Hüllkörper. Nun fasst man in jedem Schritt zwei oder mehr Objekte zu einem Knoten zusammen und berechnet einen gemeinsamen Hüllkörper. Dies wird so lange fortgesetzt, bis nur noch ein Knoten übrig ist, der alle Objekte umfasst und die Wurzel des Baumes bildet.
Die Schwierigkeit besteht darin, in möglichst kurzer Zeit zwei Knoten zu bestimmen, die zusammengefasst werden sollen. Sucht man in jedem Schritt das Paar mit dem kleinsten resultierenden Hüllkörpervolumen, beträgt der Aufwand zum Erstellen des gesamten Baums $\mathcal{O}(n^3)$. In [8]



(a) bottom up

(b) top down

(c) insertion

Abbildung 5.4: *Der Aufbau einer Hüllkörperhierarchie, (a) mit bottom up, (b) mit top down und (c) mit insertion.*

werden verschiedene Methoden beschrieben, um dies zu beschleunigen. Dennoch benötigt der Aufbau des Baumes viel Zeit.
Wählt man in jedem Schritt nur ein Knotenpaar zum Zusammenfassen aus, entstehen schnell unbalancierte Bäume. Um dies zu umgehen, können in jedem Schritt mehrere Objektpaare zusammengefasst werden. Wenn in jedem Schritt $n/2$ Paare gebildet werden, ist der Baum besser ausbalanciert und er kann viel schneller aufgebaut werden, allerdings entstehen dadurch wesentlich schlechtere Bäume. Einen guten Kompromiss erhält man, wenn in jedem Schritt k Objektpaare gebildet werden mit einem kleinen Wert k.

5.1.2.2 top down

Beim top-down Ansatz bildet man zunächst einen Hüllkörper, der alle Objekte umfasst. Dann teilt man die Menge in zwei oder mehr Teile auf und bildet wiederum einen Hüllkörper für jede Teilmenge. Dies wird rekursiv weiter ausgeführt, bis entweder in jeder Menge nur noch ein Objekt enthalten ist oder eine zuvor festgelegt maximale Tiefe erreicht wurde. Die Bäume, die hierdurch entstehen, sind zwar meist schlechter als mit bottom-up erstellte Bäume, da sie aber viel schneller und leichter erstellt werden können, werden sie häufiger benutzt.
Einen großen Einfluss auf die Qualität des Baumes hat die Regel, nach der die Objekte aufgeteilt werden. Man sucht dafür eine Achse, auf die man alle Objekte projiziert. Auf dieser Achse wird

dann ein Punkt gewählt, der vorgibt, wo die Menge geteilt wird. Anschaulich entspricht das dem Verfahren, eine Ebene zu suchen, die die Menge in zwei Teile teilt. Sowohl für die Auswahl der Achse als auch für die Auswahl des Punktes gibt es verschiedene Strategien, um jeweils andere der zu Beginn des Kapitel beschriebenen Eigenschaften des Baumes zu verbessern. Sie werden in [8] beschrieben und bewertet.

5.1.2.3 insertion

Bei der insertion Methode (auch: stufenweises Einfügen) beginnt man mit einem leeren Baum und fügt jedes Objekt nach und nach ein. Ein Objekt wird an der Stelle eingefügt, an der es die geringsten Kosten verursacht. Die Kosten werden daran gemessen, wie sehr sich das Volumen der Hüllkörper der bereits vorhandenen Knoten vergrößert. Objekte mit einem großen Hüllkörper werden also meist weiter oben im Baum eingefügt, kleinere Objekte dagegen passen oft in die Hüllkörper bereits vorhandener Objekte mit hinein und werden dadurch weiter unten eingefügt. Objekte, die im Verhältnis zu ihrer Größe weit weg von den anderen liegen, werden ebenfalls weit oben im Baum eingefügt, da beim Zusammenfassen mit anderen Knoten sehr große Hüllkörper entstehen.
Da die Entscheidung, wo ein Objekt eingefügt wird, nur anhand der bisher im Baum vorhandenen Objekte getroffen wird, hängt die Struktur des Baumes stark davon ab, in welcher Reihenfolge die Objekte eingefügt werden. Wenn die Objekte vorher auf irgendeine Art sortiert sind, können daraus stark degenerierte Bäume entstehen. Der beste Ansatz scheint es zu sein, die Objekte in einer zufälligen Reihenfolge einzufügen [8].
Die insertion Strategie ist oft schneller als die top down Methode und produziert oft bessere Bäume. Da nicht alle Objekte zu Beginn vorhanden sein müssen, kann sie vor allem zur Laufzeit eines Programms angewendet werden und eignet sich auch gut zur Aktualisierung dynamischer Bäume.

5.1.2.4 Zusammenfassung

Die Besten Hierarchien erhält man mit der bottom up Methode, allerdings benötigt diese auch die meiste Rechenzeit. Am beliebtesten sind top down Strategien, da sie einfach zu implementieren sind und in akzeptabler Zeit nicht zu schlechte Bäume liefern. Insertion Strategien sind vielversprechend, werden bisher aber kaum eingesetzt und sind daher auch nur wenig erforscht. Die Wahl der Methode hängt auch hier davon ab, was in der jeweiligen Situation als wichtiger gewertet wird: die Geschwindigkeit im Aufbau der Hierarchie oder das frühzeitige Ausschließen möglichst vieler Kollisionen. In jedem Fall muss die durch eine Hierarchie gewonnene Zeit die Zeit überwiegen, die für den Aufbau der Hierarchie benötigt wird.

5.1.3 Schnitt zwischen zwei Hüllkörperhierarchien

Um eine Hüllkörperhierarchie A gegen eine andere Hierarchie B zu schneiden, gibt es verschiedene Ansätze, in welcher der beiden Hierarchien zuerst abgestiegen werden soll. Man kann

- erst vollständig in der Hierarchie mit dem größeren Hüllkörper absteigen, danach in der mit dem kleineren;
- in jedem Schritt auswerten, welches aktuell der größere Hüllkörper ist und in dieser Hierarchie einen Schritt absteigen;
- in jedem Schritt gleichzeitig in A und in B eine Stufe absteigen;
- abwechselnd in A und B absteigen.

Bei den ersten beiden Methoden muss zunächst definiert werden, wann ein Hüllkörper größer ist als der andere (z.B. größeres Volumen oder größere Oberfläche). Die letzten beiden arbeiten immer gleich, ohne auf die Struktur der Objektmengen einzugehen.
Die erste Methode birgt die Gefahr, dass zwar der Hüllkörper der Basis einer Hierarchie größer ist als der der anderen, dafür aber viele kleine Objekte enthalten sind, die alle einmal überprüft werden, bevor in der zweiten Hierarchie eine Stufe abgestiegen wird.
Bei der zweiten Methode muss in jedem Schritt die Größe der Hüllkörper verglichen werden. Vor allem bei Objektsammlungen der gleichen Größenordnung kostet das unnötig viel Zeit.
Die dritte Strategie ist sehr einfach zu implementieren und es ist kein Größenvergleich notwendig. Man kommt wesentlich schneller tiefer in die Bäume hinein und spart einige Überlappungstests. Bei sehr unterschiedlichen Größen von A und B kann diese Methode aber auch wesentlich länger dauern als z.B. die zweite Methode.
Auch die letzte Strategie ist sehr einfach zu implementieren und benötigt, wie die vorletzte, keine Größenvergleiche. Auch hier entsteht bei stark unterschiedlich großen Hüllkörpern das gleiche Problem wie bei der dritten Methode.

Man sieht, dass es auch für Überlappungstests zwischen zwei Hüllkörperhierarchien keine allgemeingültige Lösung gibt. Für verschiedene Szenarien sind jeweils andere Strategien besser geeignet. In [8] wird näher auf die Vor- und Nachteile der verschiedenen Strategien eingegangen.

5.2 Raumunterteilung

Ziel ist es, den Raum in Zellen einzuteilen und für jede Zelle zu speichern, welche Objekte (ganz oder zum Teil) darin liegen. Kollisionen können nur zwischen Objekten auftreten, die in der gleichen Zelle liegen.
Die unterschiedlichen Methoden variieren stark im Aufwand, der zum Erstellen benötigt wird

und dem Nutzen, der daraus gezogen werden kann. Die gebräuchlichsten Strategien werden in den folgenden Abschnitten vorgestellt.

5.2.1 Gleichmäßiges Gitter

Der Raum wird in gleichgroße Quadrate (im zweidimensionalen Raum) bzw. Würfel (im dreidimensionalen Raum) aufgeteilt. Dies ist sehr einfach umzusetzen, da man für jeden Punkt sofort anhand der Weltkoordinaten ablesen kann, zu welcher Zelle er gehört.
Eine schwierigere Aufgabe ist die Wahl der Zellengröße. Das Gitter sollte nicht zu fein sein, da sonst jedes Objekt in sehr vielen Zellen enthalten ist. Das Gitter sollte aber auch nicht zu grob sein, denn dann können sich viele Objekte gleichzeitig in der selben Zelle aufhalten und müssen alle paarweise gegeneinander getestet werden. Sind in einer Szene Objekte von unterschiedlicher Größe vorhanden, kann ein Gitter gleichzeitig zu grob und zu fein sein: die großen Objekte sind dann in vielen Zellen gleichzeitig vorhanden, aber von den kleinen Objekten passen viele gleichzeitig in die selbe Zelle (siehe auch Abbildung 5.5). Daher ist es in vielen Anwendungen schwierig, einen guten Kompromiss für die Größe der Zellen zu finden.

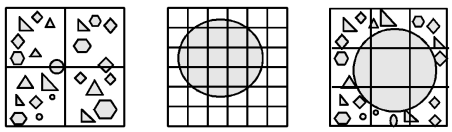


Abbildung 5.5: *Wahl der Zellgröße: (a) zu grobes Gitter, (b) zu feines Gitter und (c) ein Gitter, das sowohl zu fein als auch zu grob ist.*

5.2.2 Octrees

Um einen Octree (im zweidimensionalen: Quadtree) aufzubauen wird zunächst ein achsparalleler Quader gebildet, der die gesamte Welt einschließt. Dieser wird entlang jeder Hauptachse in der Mitte geteilt (siehe Abbildung 5.6), sodass acht Kindknoten entstehen. Jedem Knoten werden die Objekte, die in dem Würfel enthalten sind, zugeordnet. Dieses Vorgehen wird rekursiv fortgesetzt, bis ein Stop-Kriterium erreicht wird. Häufig genutzte Stop-Kriterien sind:

- eine maximale Tiefe des Baumes wurde erreicht,
- eine minimale Größe der Würfel wurde unterschritten oder
- eine minimale Anzahl an enthaltenen Objekten wurde unterschritten.

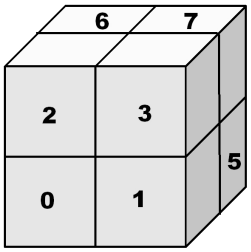
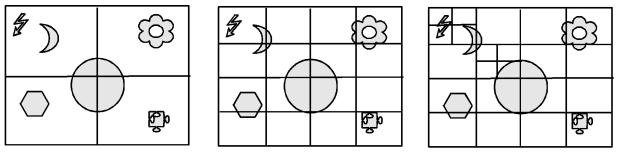


Abbildung 5.6: *Eine Zelle eines Octrees, die wieder in 8 Zellen unterteilt wird.*

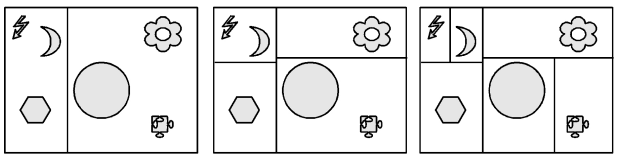
Dadurch können sich Octrees gut an die Größe und die Verteilung der Objekte anpassen. Doch da die Ebenen, an denen die Zellen geteilt werden, vorher festgelegt sind, kann es häufig auftreten, dass Objekte unnötig oft geteilt werden.

5.2.3 k-d-Trees

k-d-Trees können als eine Verallgemeinerung von Octrees betrachtet werden. In jedem Schritt wird nur entlang einer Ebene geteilt, meist wird hierfür immer im Wechsel die xy-, xz- und die yz-Ebene des Weltkoordinatensystems gewählt. Die Trennung muss nicht immer in der Mitte der Zelle erfolgen, in den meisten Fällen können bessere Stellen gefunden werden, an denen weniger Objekte geteilt werden. Ein Schritt im Octree entspricht also drei Schritten im k-d-Tree, bei dem je einmal entlang jeder der drei Ebenen in der Mitte geteilt wurde.
Als Stop-Kriterien können die gleichen Bedingungen verwendet werden wie bei Octrees. Es werden nur binäre Bäume erzeugt, daher ist die Verwaltung im Speicher wesentlich einfacher als bei Octrees. Da die Trennungsebenen dynamisch gewählt werden, passen sich die Zellen besser an die Szene an als bei Octrees, siehe hierzu auch Abbildung 5.7.



(a)



(b)

Abbildung 5.7: *Für die gleiche Szene wird (a) ein Quadtree und (b) ein k-d-Tree konstruiert.*

5.2.4 BSP-Trees

Binary Space-partitioning trees (oder kurz BSP-Trees) stellen die mächtigste Variante der Raumunterteilung dar. Jede Zelle wird, bis zum Erreichen eines Stop-Kriteriums, an einer beliebig orientierten und positionierten Ebene geteilt. Der positive Halbraum vor der Ebene und der negative Halbraum hinter der Ebene bilden dann jeweils neue Zellen.
Es können die gleichen Unterteilungen erstellt werden wie mit gleichmäßigen Gittern, Octrees oder k-d-Trees, umgekehrt ist das aber nicht der Fall. Wählt man z.B. jede Trennungsebene senkrecht zu einer der Achsen des Weltkoordinatensystems, erhält man einen k-d-Tree.
Die Konstruktion von BSP-Trees ist sehr aufwendig, vor allem da das Finden einer guten Trennungsebene nicht trivial ist [8]. Sie werden daher kaum zur Laufzeit aufgebaut oder aktualisiert, sondern eher zum Speichern und Verwalten der statischen Teile einer Szene eingesetzt.

5.3 Vergleich zwischen Hüllkörperhierarchien und Raumunterteilung

Der Hauptunterschied zwischen Hüllkörperhierarchien und Raumunterteilungen ist, dass zwei Hüllkörper der Hierarchie sich überschneiden können, Raumunterteilungen dagegen erzeugen disjunkte Zellen.
Hüllkörperhierarchien werden objektzentrisch gebildet, dadurch entstehen räumliche Redundanzen, d.h. der gleiche Raum wird von mehreren Hüllkörpern überdeckt. Durch die disjunkten Zellen bei Raumunterteilungen entstehen Objektredundanzen, d.h. ein Objekt kann sich über mehrere Zellen erstrecken.

Beide Methoden sind gut dazu geeignet, die Kollisionserkennung zu beschleunigen und können auch für andere Anwendungen wie z.B. Visibilitätsrechnung oder Raytracing verwendet werden. Ist bereits ein Octree, k-d-Tree oder BSP-Tree vorhanden, kann er verwendet werden, um eine Hüllkörperhierarchie aufzubauen.

5.4 Schnitttests zwischen Primitiven

Wenn durch schnelle Tests eine Kollision nicht ausgeschlossen werden konnte, muss ein exakter Schnitttest zwischen den Objekten ausgeführt werden. Dieser Test muss berechnen *ob* eine Kollision auftritt, und im Fall einer Kollision müssen der Angriffspunkt der Kraft, die Normale der Kraft und die Eindringtiefe berechnet werden. In der Literatur werden hier meist die Punkte bestimmt, die das andere Primitiv am stärksten penetrieren. Durch das weiche Kollisionsmodell, das in `MuRoSimF` verwendet wird (vgl. Kapitel 3) traten hierdurch oft zu heftige Reaktionen auf. Deshalb werden auch andere Heuristiken betrachtet, die für diese Situationen entwickelt wurden und die mit dem verwendeten Kollisionsmodell experimentell gute Ergebnisse liefern.

Die in `MuRoSimF` aktuell vorhandenen Primitive sind Kugel, Quader, Zylinder und Ebene. Für Berechnungen mit Zylindern wird häufig auf eine Kapsel zurückgegriffen. Das ist ein Zylinder, der an den Enden nicht abgeschnitten ist, sondern noch jeweils eine Halbkugel aufgesetzt bekommt. Ist r der Radius des Zylinders (und somit auch der Halbkugeln), dann ist jeder Punkt im Raum, dessen Abstand zur Mittelachse der Kapsel kleiner als r ist, Teil der Kapsel. Schnitttests mit Kapseln sind also wesentlich einfacher durchzuführen als mit Zylindern.

5.4.1 Kollisionserkennung zwischen zwei Kugeln

Der Test, ob sich zwei Kugeln schneiden, lässt sich sehr schnell durchführen.
Seien K_1 und K_2 die beiden Kugeln, m_i der Mittelpunkt von K_i und r_i der zugehörige Radius. Dann schneiden sich die Kugeln nicht, falls

$$\|(m_1 - m_2)\| < r_1 + r_2,$$

d.h. falls der Abstand der Mittelpunkte größer ist als die Summe der Radien. Die Kraft wirkt auf K_1 in Richtung

$$n_1 = (m_1 - m2)/\|(m_1 - m_2)\|,$$

auf K_2 wirkt die Kraft in entgegengesetzter Richtung

$$n_2 = -n_1.$$

Angriffspunkt der Kraft ist jeweils

$$p_i = m_i - n_i * r_i,$$

die Eindringtiefe ist der Abstand zwischen p_1 und p_2.

5.4.2 Kollisionserkennung zwischen Kugel und Ebene

Sei m der Mittelpunkt der Kugel, r der Radius, p ein beliebiger Punkt auf der Ebene und n die Normale der Ebene. Die Kugel schneidet die Ebene nicht, falls

$$|(m - p) * n| > r$$

d.h., falls der Abstand von m zur Ebene größer ist als r.

Liegt m über der Ebene, wirkt die Kraft auf die Kugel in Richtung n_k mit

$$n_k = n,$$

andernfalls in Richtung

$$n_k = -n.$$

Die Kraft wirkt auf die Ebene in der entsprechend anderen Richtung n_e, also

$$n_e = -n_k.$$

Der Angriffspunkt p_k der Kraft auf die Kugel liegt bei

$$p_k = m - n_k * r$$

Die Kraft auf die Ebene wirkt im Punkt

$$p_e = m - n_k * |(m - p) * n|$$

5.4.3 Kollisionserkennung zwischen Kugel und Quader

Sei m der Mittelpunkt der Kugel, r der Radius.
Der zu m am nächsten liegende Punkt p_q auf dem Quader kann wie in Kapitel 4.1.5 beschrieben sehr schnell bestimmt werden. Der Sonderfall, dass m im Inneren des Quaders liegt, wird später behandelt.
Gilt jetzt

$$\|p_q - m\| > r,$$

dann tritt keine Kollision auf. Andernfalls ist mit p_q der Angriffspunkt der Kraft auf den Quader gegeben, die Normale n_k für die Kugel ist definiert durch

$$n_k = (p_q - m)/\|p_q - m\|$$

bzw. n_q für den Quader in entgegengesetzter Richtung. Der Angriffspunkt der Kraft auf die Kugel ergibt sich als

$$p_k = m - n_k * r.$$

Liegt m im Inneren des Quaders, wird für p_q der nächstliegende Punkt auf der am nächsten liegenden Seitenfläche gewählt, n_q verläuft dann in Richtung $p_q - m$.

5.4.4 Kollisionserkennung zwischen Kugel und Zylinder

Sei m der Mittelpunkt der Kugel, r_k der Radius der Kugel, z_a und z_b die Punkte, die die Mittelachse des Zylinders definieren, $z_z = (z_a - z_b)/\|z_a - z_b\|$ die normierte Mittelachse des Zylinders und r_z der Radius des Zylinders.
Zunächst wird berechnet, ob sich die Kugel und die den Zylinder einhüllende Kapsel schneiden. Hierzu berechnet man den nächsten Punkt p auf der Strecke $|z_a z_b|$ zu m wie in Kapitel 4.1.3 beschrieben. Sei d der Abstand zwischen m und p. Gilt

$$d > r_k + r_z,$$

dann sind Kugel und Kapsel disjunkt, also können sich auch Kugel und Zylinder nicht schneiden.

Falls sich Kugel und Kapsel schneiden, muss überprüft werden, ob die Kugel auch den Zylinder schneidet. Hierzu teilt man den Raum in 3 Voronoi-Regionen, ausgehend vom Zylinder, ein. Die Regionen werden durch die Ebenen definiert, die die Enden des Zylinders enthalten:

$$E_1 = \{x : (x - z_a) * z_z = 0\}$$
$$E_2 = \{x : (x - z_b) * z_z = 0\}$$

m liegt entweder in der mittleren Region, also vom Zylinder aus betrachtet auf gleicher Höhe, oder m liegt über bzw. unter dem Zylinder.

1) m liegt auf Höhe des Zylinders, es tritt eine Kollision auf. Die Kraft wirkt auf die Kugel in Richtung n_k mit

$$n_k = (p - m)/\|p - m\|$$

und auf den Zylinder in entgegengesetzter Richtung n_z mit

$$n_z = -n_k.$$

Angriffspunkt der Kraft auf die Kugel ist

$$p_k = m - n_k * r_k,$$

Angriffspunkt der Kraft auf den Zylinder ist

$$p_z = p - n_z * r_z.$$

2) m liegt o.B.d.A über E_1. Gilt

$$(m - z_a) * z_z > r_k,$$

ist also der Kugelmittelpunkt weiter von E_1 entfernt als r_k, dann gibt es keine Kollision. Andernfalls muss noch berechnet werden, ob die Kugel und die Kreisscheibe, die den Zylinder abschließt, einen nichtleeren Schnitt haben. Hierzu berechnet man den Punkt p_z, der auf der Kreisscheibe am nächsten zu m liegt. Gilt

$$\|m - p_z\| > r_k,$$

sind Kugel und Zylinder disjunkt. Sonst wirkt die Kraft auf die Kugel im Punkt p_k in Richtung n_k und auf den Zylinder in Richtung n_z mit

$$n_k = (m - p_z)/\|m - p_z\|,$$
$$n_z = -n_k,$$
$$p_k = m - n_k * r_k.$$

5.4.5 Kollisionserkennung zwischen zwei Quadern

Für die Kollisionserkennung zwischen zwei Quadern wird auf das Separating Axis Theorem (siehe Kapitel 4.3) zurückgegriffen. Da jeder Quader 3 linear unabhängige Flächennormalen und 3 linear unabhängige Kantenrichtungen hat, müssen insgesamt höchstens $6 + 3 * 3 = 15$ Tests durchgeführt werden. Überlappen sich die Projektionen der beiden Quader auf einer dieser Achsen nicht, sind die Quader disjunkt.

Überlappen sich die Projektionen der beiden Quader auf allen 15 Achsen, schneiden sich auch die Quader. Als Normale n der Kraft wählt man die Richtung der Achse, auf der die Überlappung der Projektionen am geringsten ist.

Man wählt nun von jedem Quader die Eckpunkte aus, die auf einer Stützebene mit Normale n liegen. Man erhält für jeden Quader einen Streckenzug mit 1-4 Ecken. Diese beiden Streckenzüge projiziert man auf eine gemeinsame Ebene und berechnet das gemeinsame Schnittpolygon. Den Schwerpunkt dieses Schnittpolygons projiziert man auf die Oberflächen der Quader und erhält somit die Angriffspunkte der Kraft.

5.4.6 Kollisionserkennung zwischen Quader und Ebene

Sei n die Normale der Ebene. Um zu berechnen, ob ein Quader eine Ebene schneidet, reicht es, die Lage der Eckpunkte des Quaders relativ zur Ebene zu betrachten. Liegen alle Ecken über oder alle Ecken unter der Ebene, gibt es keinen Schnitt.

Im Fall einer Kollision wurden unterschiedliche Angriffspunkte der Kraft betrachtet. Der größte Teil des Quaders liegt o.B.d.A. über der Ebene.

Tiefster Punkt: Der Punkt, der die Ebene am stärksten penetriert, ist entweder ein eindeutig definierter Eckpunkt des Quaders, oder eine Kante oder Fläche, von denen dann jeweils die Mitte gewählt wird.

Gewichtung der Ecken: Aus allen Punkten, die unter der Ebene liegen, wird ein gewichtetes Mittel gebildet. Als Gewicht wird dabei der Abstand der Ecke zur Ebene verwendet.

Schwerpunkt des Schnittpolygons: Es werden alle Schnittpunkte der Kanten des Quaders mit der Ebene berechnet. Der Schwerpunkt dieses Schnittpolygons wird auf die Oberfläche des Quaders projiziert.

Als Richtung der Kraft wird jeweils für den Quader $n_q = n$ und für die Ebene $n_e = -n$ gewählt. Den Angriffspunkt der Kraft auf die Ebene ist jeweils der Punkt des Quaders zurück auf die Ebene projiziert.

Für die meisten Anwendungen am besten geeignet war das dritte Verfahren, da es nicht sehr anfällig gegenüber Rundungsfehlern ist, der Kollisionspunkt schwankt kaum.

Für die Laufbewegung des Humanoidroboter ist es am besten, den tiefsten Punkt zu berechnen, da er dann wesentlich weniger leicht umfällt.

5.4.7 Kollisionserkennung zwischen Quader und Zylinder

Sei r der Radius des Zylinders, z_a und z_b die Endpunkte der Mittelachse des Zylinders und $z_z = (z_a - z_b)/\|z_a - z_b\|$ die normierte Mittelachse des Zylinders.
Auch hier wird zunächst auf die den Zylinder einhüllende Kapsel zurückgegriffen. Für die Mittelachse und jede der 6 Seiten des Quaders werden jetzt die nächsten Punkte a_i auf der Zylinderachse und b_i auf der Seitenfläche des Quaders bestimmt. Gilt für alle i

$$\|a_i - b_i\| > r,$$

schneidet die Kapsel die Hülle des Quaders nicht, sie liegt also entweder vollständig außerhalb oder komplett innerhalb des Quaders, das Gleiche gilt dann natürlich auf für den Zylinder. Es genügt, einen beliebigen Punkt der Kapsel zu betrachten und auszurechnen, ob er im Inneren des Quaders liegt.

Schneidet die Kapsel eine oder mehrere der Seitenflächen des Quaders muss man noch bestimmen, ob auch der Zylinder diese Seitenflächen schneidet.
Gegeben ist also eine Seite des Quaders, die mit der Kapsel einen nichtleeren Schnitt hat sowie die Punkte a_1 auf der Zylinderachse und b_1 auf der Seitenfläche des Quaders. Ist $b_1 - a_1$ senkrecht zu z_z, dann schneidet auch der Zylinder die Seitenfläche und der Schnittpunkt wird beibehalten. Andernfalls ist a_1 ein Endpunkt der Zylinderachse und es muss die Richtung v bestimmt werden, die in der zu z_z senkrechten Ebene liegt und am stärksten in Richtung Quader zeigt. Schneidet die Strecke zwischen a_1 und $a_1 + v * r$ die Seitenfläche nicht, dann schneidet auch der Zylinder die Seitenfläche nicht.

5.4.8 Kollisionserkennung zwischen zwei Zylindern

Seien a_i und b_i die Punkte, die die Mittelachse von Zylinder i begrenzen und r_i der Radius von Zylinder i, $i = 1, 2$.
Zunächst wird berechnet, ob sich die beiden Kapseln, die die Zylinder einhüllen, schneiden. Hierzu muss nur der Abstand d zwischen den Strecken $|a_1b_1|$ und $|a_2b_2|$ berechnet werden wie in Kapitel 4.1.8 beschrieben. Gilt

$$d > r_1 + r_2,$$

schneiden sich die Kapseln und somit auch die Zylinder nicht.
Andernfalls betrachtet man die Punkte q_1 und q_2, die jeweils auf einer der beiden Strecken und so nah wie möglich an der anderen Strecke liegen. Es wird vorläufig die Normale

$$n = (p_2 - p_1)/\|p_2 - p_1\|$$

angenommen. Die vorläufigen Kollisionspunkte sind

$$p_1 = q_1 + n * r_1$$

und

$$p_2 = q_2 - n * r_2.$$

Falls q_1 und q_2 nicht Endpunkte der Strecken sind, ist n orthogonal zu beiden Mittelachsen und p_1 und p_2 liegen auf dem Zylindermantel und sind somit die richtigen Kollisionspunkte. Andernfalls sei o.B.d.A. q_1 Endpunkt des ersten Zylinders. Es wird nun die Richtung v bestimmt, die orthogonal zur Zylinderachse steht und am stärksten in Richtung des anderen Zylinders zeigt. Auf der Strecke zwischen q_1 und $q_1 + v \cdot r_1$ wird dann nach einem Schnittpunkt mit dem anderen Zylinder gesucht.
Ist auch q_2 Endpunkt des zweiten Zylinders, wird die gleiche Suche nochmals durchgeführt. Wird in beiden Fällen kein Schnittpunkt gefunden, sind die Zylinder disjunkt.

5.4.9 Kollisionserkennung zwischen Zylinder und Ebene

Sei p_e ein beliebiger Punkt auf der Ebene E, n_e die Normale der Ebene, z_a und z_b die Punkte, die die Mittelachse des Zylinders definieren, $z_z = (z_a - z_b)/\|z_a - z_b\|$ die normierte Mittelachse des Zylinders und r_z der Radius des Zylinders.
Der Zylinder wird jetzt auf ein zweidimensionales Rechteck reduziert, indem man nur den Querschnitt entlang einer Hilfsebene E_h betrachtet, die durch die Richtungen e_1 und e_2 definiert wird mit $e_1 = z_z$ und $e_2 = n_e$ oder, falls z_z und n_e parallel sind, e_1 eine beliebige andere Richtung ungleich n_e, z.B. in Richtung der x-Achse des Zylinder-Koordinatensystems. Die Normale von E_h ist gegeben durch $n = e_1 \times e_2$.
Das Rechteck ist durch zwei Richtungen bestimmt: die Richtung z_z der Mittelachse des Zylinders und die Richtung z_e, die in der Ebene E_h liegt und senkrecht zu z_z verläuft, $z_e = n \times z_z$. Die Eckpunkte des Rechtecks sind also gegeben durch

$$a = z_a + z_z * r$$
$$b = z_a - z_z * r$$
$$c = z_b + z_z * r$$
$$d = z_b - z_z * r$$

Liegen a, b, c und d alle auf der selben Seite der Ebene E, schneidet das Rechteck die Ebene nicht und auch Zylinder und Ebene sind disjunkt. Andernfalls wurden zwei verschiedene Ansätze für den Angriffspunkt der Kraft implementiert.

Schwerpunkt der Schnittfläche: Es werden die Schnittpunkte zwischen Rechteck und Ebene berechnet und das arithmetische Mittel gebildet. Dieser Punkt p_1 ist Angriffspunkt der Kraft auf die Ebene. Die Normale n_z der Kraft verläuft in Richtung der Normalen von E, der Angriffspunkt p_2 der Kraft auf den Zylinder ergibt sich also durch Projektion von p_1 entlang n_z auf die Oberfläche des Zylinders.

Tiefster Punkt: Der Punkt, der am tiefsten in die Ebene eindringt, ist entweder einer der vier Punkte a, b, c, d oder die Mitte aus zwei von ihnen, wenn sie den gleichen Abstand von der Ebene haben.

Da die Räder von Fahrzeugen alle zylinderförmig sind, gehört diese Schnittberechnung zu den wichtigsten im Rahmen dieser Arbeit. Da das erste Verfahren weniger anfällig bezüglich Rundungsfehlern ist, eignet es sich zum Fahren wesentlich besser, da der Kollisionspunkt kaum schwankt. Wird mit dem tiefsten Punkt gearbeitet, sieht das Ergebnis ganz anders aus: Liegt der Zylinder auf der Seite (wie es bei Rädern von Fahrzeugen der Fall ist), springt der Kollisionspunkt häufig von einer Seite auf die andere hin und her, da beide Eckpunkte des Rechtecks immer ganz knapp unter oder ganz knapp über der Ebene liegen. Dies führt zu einem sehr holprigen Fahrverhalten.

5.5 Umsetzung der Kollisionserkennung für `MuRoSimF`

Im Rahmen dieser Diplomarbeit wurde ein neuer `CollisionDetectionTask` entwickelt, der den alten (vgl. Kapitel 3) ersetzen kann. Er baut aus den Objektgruppen Hüllkörperhierarchien auf und schneidet diese gegeneinander. Es wird auf Hüllkugeln zurückgegriffen, da sie für starre Objekte zu Beginn der Simulation schnell berechnet werden können und dann nicht mehr aktualisiert werden müssen. Außerdem kann man die Hierarchie für bewegte Systeme wie z.B. einen Roboter schnell aktualisieren.

Da alle Objekte in dieser Arbeit durch Primitive (Kugeln, Quader und Zylinder) gegeben sind, lassen sich alle in Kapitel 5.1.1 vorgestellten Hüllkörper leichter erstellen als für allgemeine Punktwolken. Es stellt sich dann noch das Problem, für zwei dieser Hüllkörper den besten Hüllkörper zu finden, der beide umschließt. Durch den modularen Aufbau von `MuRoSimF` ist es leicht möglich, später noch Hierarchien mit anderen Hüllkörpern zu implementieren.

5.5.1 Aufbau des Baumes

Um in den Algorithmen, die mit den Bäumen arbeiten, viele Sonderfälle zu umgehen, sollen die Bäume so aufgebaut werden, dass sich nur in den Blättern Objekte befinden. Innere Knoten sollen nichts außer einem Hüllkörper für ihre Kindknoten enthalten.

Für den Aufbau der Hüllkörperhierarchie aus Gruppen von Objekten werden zwei Ansätze verfolgt: für beliebige Objektgruppen wird ein top-down-Verfahren eingesetzt, für Gruppen, die bereits in einer Baumstruktur organisiert sind, wie z.B. Roboter, wird diese vorgegebene Struktur weitestgehend beibehalten. Für bewegliche Objektgruppen muss die Hierarchie in jedem Zeitschritt aktualisiert werden, für unbewegliche Objektgruppen muss die Berechnung der Hüllkörperhierarchie nur einmal zur Initialisierung erfolgen.

5.5.1.1 Aufbau des Baumes aus einer Objektmenge

Zunächst werden die Objekte aus der Menge entfernt, die nicht für die Hierarchie geeignet sind. Hierunter fallen alle Ebenen, da für diese kein Hüllkörper gebildet werden kann. Für die Hüllkugelhierarchie werden außerdem große Objekte, die sehr lang und schmal sind entfernt, da das Volumen der Hüllkugeln sehr groß im Verhältnis zum Volumen der Objekte ist und sie somit die gesamte Hierarchie negativ beeinflussen. Ein Beispiel hierfür sind Banden, die um das gesamte Feld verlaufen, siehe Abbildung 5.8. Die aus der Menge entfernten Objekte werden separat gespeichert und später getrennt behandelt.

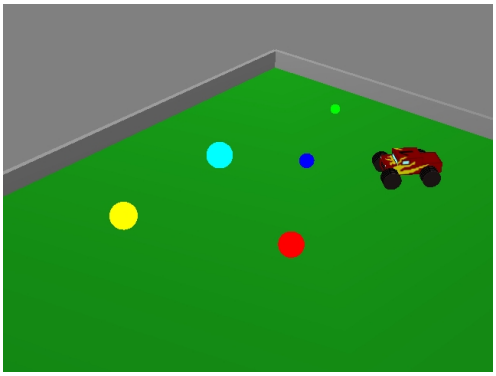


Abbildung 5.8: *Die Hüllkugeln der Banden würden die gesamte Szene einschließen.*

Um aus der übrigen Objektmenge eine Hierarchie aufzubauen, wird ein top-down-Ansatz verfolgt. In jedem Schritt wird die Lage der Objektmittelpunkte auf der x-, y- und z-Achse betrachtet. Die Achse, auf der die Differenz zwischen dem größten und dem kleinsten Wert am größten ist, wird als Trennungsachse ausgewählt. Die Trennung erfolgt am Objektmedian, d.h. bei einer geraden Anzahl von Objekten befinden sich auf beiden Seiten gleich viele Objekte, bei ungerader Anzahl ist auf einer Seite ein Objekt mehr. Dies wird solange weitergeführt, bis in jeder Objektmenge nur noch ein Primitiv enthalten ist.
Dieses Verfahren wurde ausgewählt, um gut ausbalancierte Bäume mit geringer Tiefe zu erhalten. Da die hier verwendeten Objektmengen relativ gleichmäßig verteilt sind, sind die Bäume auch räumlich gut ausbalanciert. Die Trennungsachse und der Trennungspunkt lassen sich einfach und schnell bestimmen. Dennoch sollte dieser Ansatz nur für unbewegliche Objektmengen verwendet werden, bewegliche Objektmengen sind meist in Form von Robotermodellen gegeben, für diese arbeitet das im Folgenden beschriebene Verfahren wesentlich schneller.

5.5.1.2 Aufbau des Baumes aus einem Robotermodell

Da die Robotermodelle bereits in einer Baumstruktur angelegt sind, muss für die Hüllkörperhierarchie kein komplett neuer Baum aufgebaut werden, sondern man kann anhand des Modelles einen geeigneten Baum ableiten.

Zunächst wird hierfür die Struktur des Robotermodells kopiert. Jeder Knoten, der im Modell eine physische Ausdehnung hatte, bekommt auch im neuen Baum ein Objekt mit den gleichen Eigenschaften angehängt. Alle anderen Knoten (wie z.B. Translationen zum nächsten Knoten) haben kein Objekt.
Dieser Baum ist jetzt für die Kollisionserkennung noch nicht geeignet, da einige Objekte nicht wie gefordert in den Blättern liegen, sondern zu inneren Knoten gehören. Außerdem sind viele Knoten in der Struktur, die für die Kollisionserkennung nicht benötigt werden und den Baum nur unnötig vergrößern. In Abbildung 5.9 ist der kopierte Baum für den Monstertruck zu sehen. Man

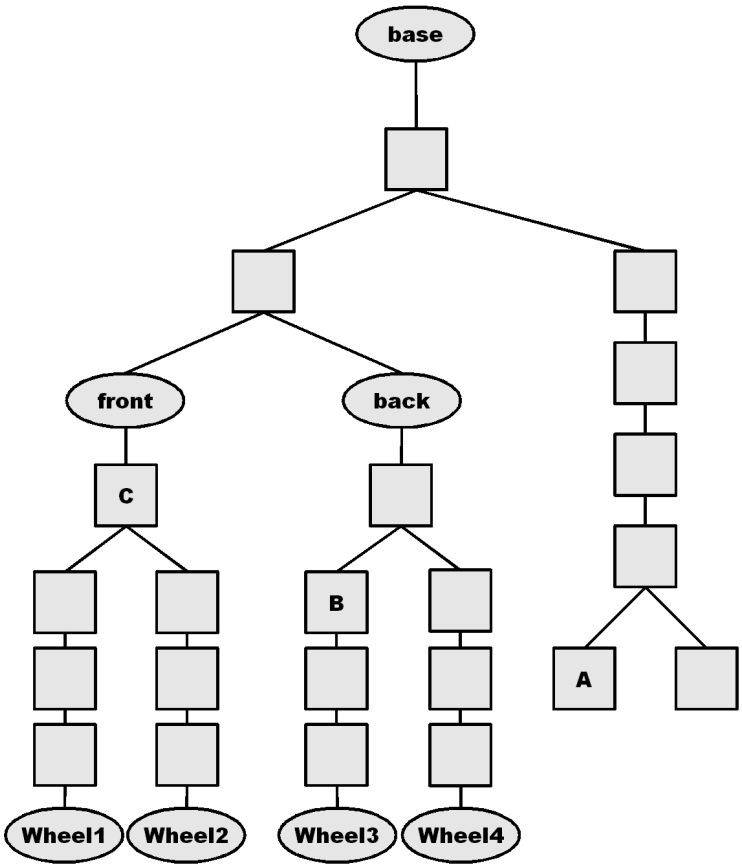


Abbildung 5.9: *Der Baum des Monstertrucks nach Kopieren der Struktur aus dem Robotermodell. Die Quadrate stellen Knoten ohne physikalisches Objekt dar, die Ellipsen Knoten mit physikalischem Objekt.*

sieht, dass der gesamte rechte Zweig (hier werden im Robotermodell die Kamera und der Laserscanner angebracht) für die Kollisionserkennung nicht benötigt wird. Auch im linken Zweig gibt es viele Knoten, die gelöscht werden können.
Der gesamte Baum wird daher, beginnend bei den Blättern, traversiert, dabei werden nicht benötigte Knoten gelöscht. Hierbei wird nach folgenden Kriterien vorgegangen:

- Ein Blatt, das kein Objekt hat, kann gelöscht werden (z.B. der Knoten mit der Beschriftung „A“ in Abbildung 5.9).
- Ein Knoten, der kein Objekt hat, kann gelöscht werden, falls er nur einen Nachfolger hat (z.B. der Knoten mit der Beschriftung „B“ in Abbildung 5.9).
- Ein Knoten, der kein Objekt hat und einziges Kind seines Vorgängers ist, kann gelöscht werden (z.B. der Knoten mit der Beschriftung „C“ in Abbildung 5.9).

Nach Anwenden dieser Regeln verkleinert sich der Baum wie in Abbildung 5.10 zu sehen ist.

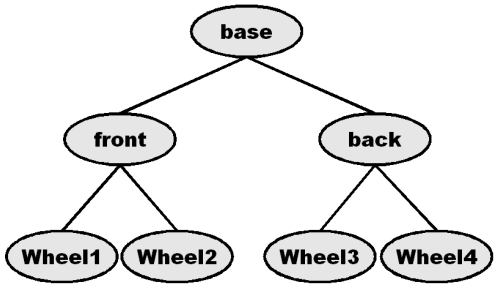


Abbildung 5.10: *Der Baum des Monstertrucks nachdem überflüssige Knoten gelöscht wurden.*

Jetzt besteht noch das Problem, dass sich einige Objekte in inneren Knoten befinden. Für jeden solchen Knoten K_{obj} werden jetzt zwei neue Knoten K_1 und K_2 eingefügt (siehe Abbildung 5.11). K_1 wird an der Stelle eingefügt, wo vorher K_{obj} war, K_{obj} und K_2 werden als Kinder von K_1 definiert und K_2 erhält als Kinder die Knoten, die zuvor Kinder von K_{obj} waren. Der

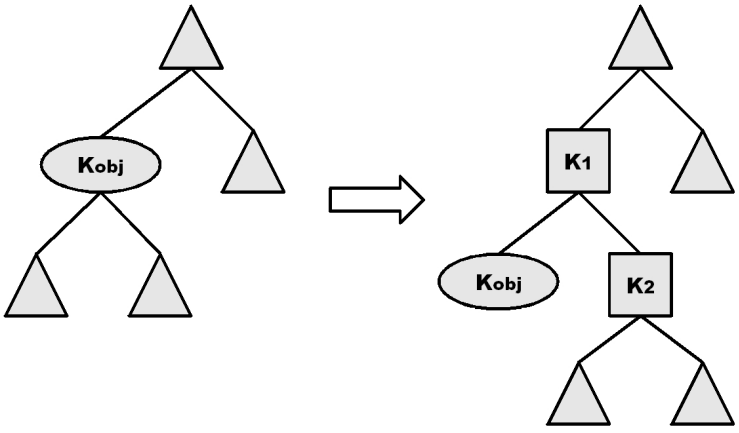


Abbildung 5.11: *Einfügen von zwei neuen Knoten, damit ein Knoten mit physikalischem Objekt kein innerer Knoten mehr ist, sondern ein Blatt. Dreieckige Knoten stehen für beliebige andere Knoten.*

endgültige für den Monstertruck resultierende Baum ist in Abbildung 5.12 zu sehen.
Die Hierarchien für die Robotermodelle müssen nur einmal bei der Initialisierung erstellt werden. Zur Laufzeit müssen nach einer Bewegung des Roboters nur die Hüllkörper neu berechnet werden. Für die Objekte in den Blättern ändert sich die Hüllkugel nicht, daher muss nur in den inneren Knoten eine gemeinsame Hüllkugel für die beiden Kindknoten berechnet werden. Diese Berechnung wird im folgenden Abschnitt beschrieben.

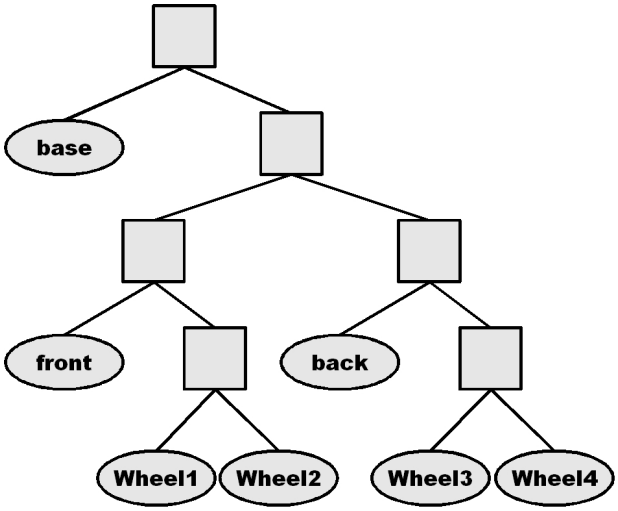


Abbildung 5.12: *Die fertige Hüllkörperhierarchie für den Monstertruck.*

5.5.1.3 Berechnung der minimalen Hüllkugel für zwei Kugeln

Gegeben sind zwei Kugeln K_1 und K_2, definiert durch ihre Mittelpunkte m_1 und m_2 und ihre Radien r_1 und r_2. Es sollen der Mittelpunkt m und der Radius r der minimalen Kugel K berechnet werden, die K_1 und K_2 einschließt. Es ist klar, dass m auf der Verbindungslinie zwischen m_1 und m_2 liegen muss. Liegen m_1 und m_2 im gleichen Punkt, muss auch m hier liegen und r ist das Maximum von r_1 und r_2. K schließt somit K_1 und K_2 ein. Da K mit K_1 oder K_2 übereinstimmt, kann es auch keine kleinere Kugel geben, die beide Kugeln einschließt.
Liegen m_1 und m_2 nicht im gleichen Punkt, beschreibt d den Abstand zwischen den beiden Mittelpunkten und $n = (m_2 - m_1)/\|m_2 - m_1\|$ die normierte Richtung zwischen m_1 und m_2. Man berechnet nun die beiden Punkte der Vereinigung von K_1 und K_2, für die der Abstand am größten ist. Dies sind $p_1 = m_1 - n * \max(r_1, d + r_2)$ und $p_2 = m_2 + n * \max(r_2, d + r_1)$, siehe Abbildung 5.13. m liegt dann genau in der Mitte von p_1 und p_2, also $m = (p_1 + p_2)/2$, r ist der Abstand von m zu den beiden Punkten. K schließt K_1 und K_2 ein, da die beiden Punkte, die voneinander den größten Abstand haben, in K liegen. K ist minimal, da diese beiden Punkte auf dem Rand von K liegen.

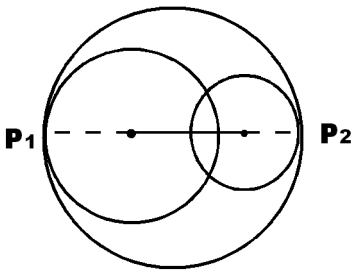


Abbildung 5.13: *Bestimmen einer gemeinsamen Hüllkugel.*

5.5.2 Kollisionstest zwischen zwei Bäumen

Da die Objektgruppen stark bezüglich der Gesamtgröße variieren (der Hüllkörper einer Stadt ist um ein vielfaches größer als der Hüllkörper eines Roboters), wird nicht in beiden Bäumen gleichmäßig abgestiegen. In einer Szene sind oft mehrere Roboter vorhanden, deshalb wird nicht die zunächst größer scheinende Hierarchie komplett traversiert, sondern in jedem Schritt werden die auf dieser Stufe aktuellen Hüllkörper miteinander verglichen und dann in der Hierarchie mit dem größeren Hüllkörper abgestiegen. Da mit Hüllkugeln gearbeitet wird, beschränkt sich dieser Test auf den Vergleich der Radien, der Zeitaufwand ist also nicht sehr groß.

Zusätzlich muss jede Hierarchie gegen die aussortierten Objekte des anderen Baumes getestet werden. Bei Ebenen beschränkt sich dies auf die Berechnung des Abstands des Mittelpunktes der Hüllkugel zur Ebene. Ist dieser kleiner als der Radius der Hüllkugel, müssen auch die Kindknoten betrachtet werden.

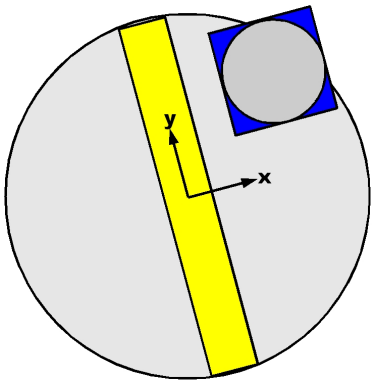


Abbildung 5.14: *Der Schnittest zwischen einem aussortierten Objekt und einer Hüllkugel.*

Für die langen schmalen aussortierten Objekte wird auf einen Test zwischen achsorientierten Quadern zurückgegriffen. Hierzu wird die Hüllkugel H des Baumes in Objekt-Koordinaten des anderen Objekts O umgerechnet. Der Mittelpunkt von O liegt bezüglich dieses Koordinaten-

systems im Ursprung, es müssen also nur die Koordinaten von H betrachtet werden. Ist z.B. die x-Koordinate von H größer als der Radius von H plus die Hälfte der Ausdehnung von O in x-Richtung, kann ein Schnitt ausgeschlossen werden. Das gleiche gilt für die beiden anderen Richtungen. Dies entspricht dem Test, ob die achsorientierten Quader um H und O sich schneiden, siehe auch Abbildung 5.14. Da der Hüllkörper für O jetzt wesentlich enger am Objekt anliegt, können auf diese Art wesentlich mehr Kollisionen ausgeschlossen werden, als mit der Hüllkugel.

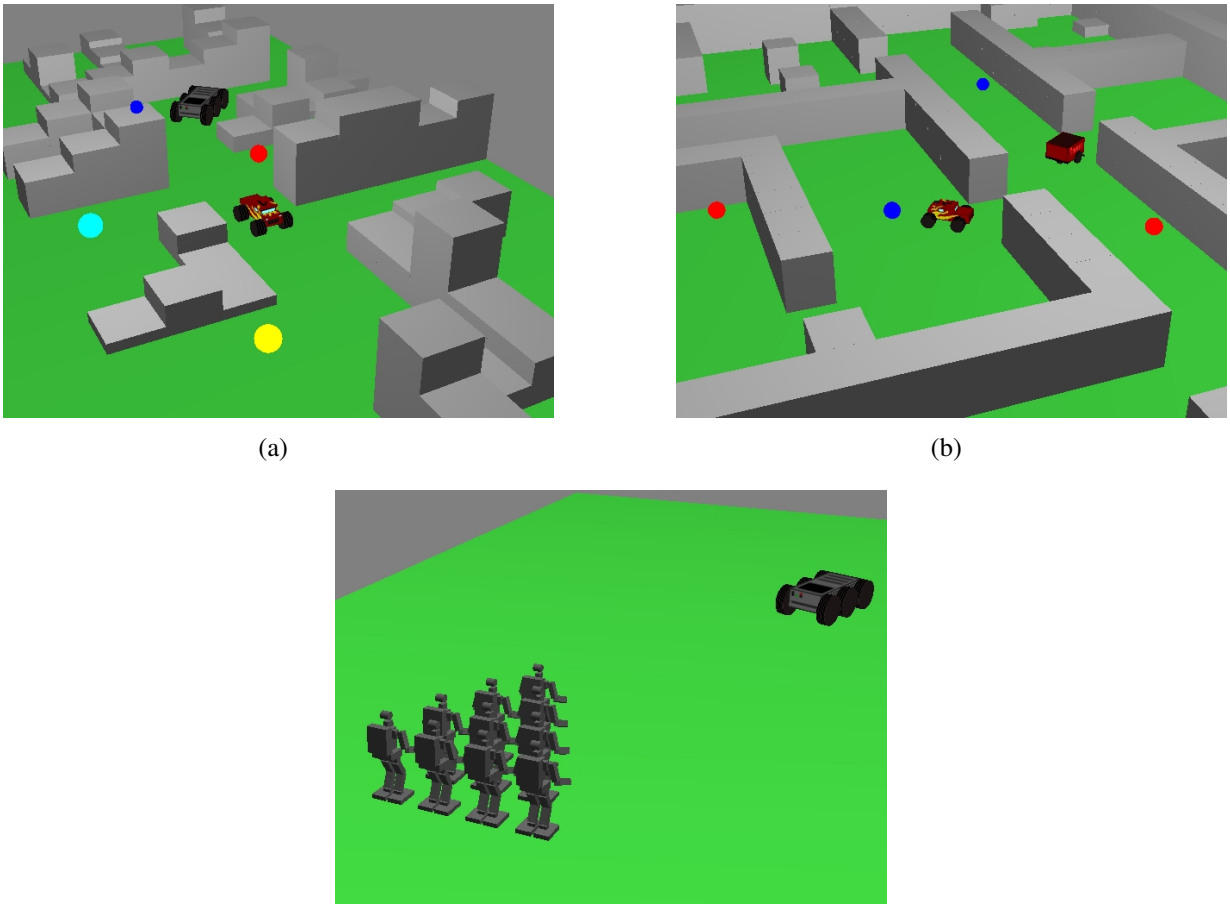
5.5.3 Ergebnisse

Es soll nun betrachtet werden, welche Performance-Steigerungen durch die Hüllkörperhierarchie erzielt werden konnten. Hierzu werden verschiedene Szenen mit diversen festen und beweglichen Objekten betrachtet. Es werden verschiedene Robotertypen eingesetzt, die in Anhang A vorgestellt werden. Sie unterscheiden sich jeweils in der Anzahl ihrer Gelenke und der Anzahl an Objekten, die mit der Umwelt kollidieren können. Einerseits wurde betrachtet, wie viele Roboter im selben Setup gleichzeitig simuliert werden können. Andererseits wurde gezählt, wie viele Schnitttests pro Zeitschritt während der Simulation durchgeführt wurden. Alle Tests wurden auf einem Laptop (Intel Pentium M processor, 1.5 GHz, 512 MB RAM, ATI Mobility Radeon 9700/64MB) durchgeführt.

Ohne die Hüllkörperhierarchie konnten in kleineren Szenarien mit wenigen Robotern und nicht zu vielen anderen Objekten bereits gute Ergebnisse erzielt werden. In einer Umgebung mit 52 Häusern können 2 Monstertrucks mit je einem Laserscanner gleichzeitig simuliert werden. Wird die Hüllkörperhierarchie eingeschaltet, lassen sich in der gleichen Umgebung 8 Monstertrucks simulieren.
Ein weiterer Testfall ist das „Roboterbowling“: Humanoidroboter werden als Pins aufgestellt, ein Volksbot startet in einiger Entfernung und versucht, alle Humanoiden umzufahren. Alle Roboter werden hierbei ohne Sensoren simuliert. Ohne Hüllkörperhierarchie können außer dem Volksbot 2 Humanoidroboter simuliert werden. Mit Hüllkörperhierarchie kann ein vollständiges Bowling-Setup, also ein Volksbot und 10 Humanoidroboter, simuliert werden.

Um die Performance besser vergleichen zu können, wurden die benötigten Schnitttests pro Zeitschritt gezählt. Ohne Verwendung der Hierarchie werden in jedem Zeitschritt gleich viele Tests ausgeführt, da alle Objekte gegen alle möglichen Kollisionspartner geschnitten werden müssen. Unter Verwendung der Hierarchie werden umso weniger Tests benötigt, je weiter die Objekte voneinander entfernt sind. Liegen die Objekte näher aneinander oder schneiden sich, werden mehr Tests zwischen den Hüllkörpern benötigt, da tiefer in den Bäumen abgestiegen werden muss, außerdem steigt auch die Anzahl der Tests zwischen den Objekten, wenn eine Kollision durch Hüllkörpertests nicht ausgeschlossen werden kann. Daher wurden für alle Tests die Objekte innerhalb der Szene bewegt, um einen Überblick zu erhalten, wie viele Tests im schlimmsten und im besten Fall durchgeführt werden und wo der Durchschnitt liegt.

Es wurden unterschiedliche Test-Szenarien erstellt, die im Folgenden beschrieben werden, ein



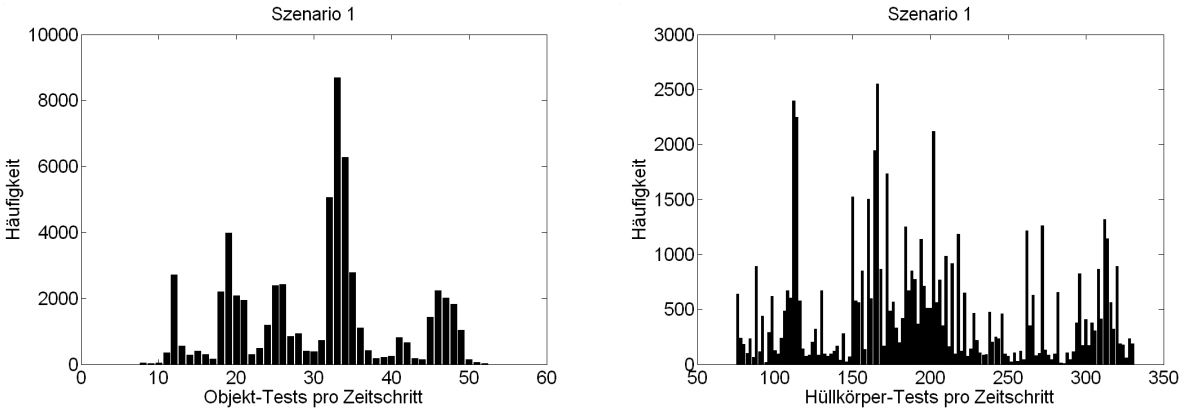
(a) (b)

(c)

Abbildung 5.15: *Die drei Test-Szenarien.*

Screenshot jeder Szene ist in Abbildung 5.15 zu sehen. Eine Übersicht der Testergebnisse ist in Tabelle 5.2 zu finden. Außerdem gibt es zu jedem Szenario Graphiken, die die Verteilung der Tests beschreiben. Auf der x-Achse ist jeweils die Anzahl der Tests angegeben, die in einem Zeitschritt ausgeführt werden, auf der y-Achse ist eingetragen, in wie vielen der Zeitschritte diese Anzahl Tests durchgeführt wurde.

Szenario 1 (Abbildung 5.15(a)): Das erste Szenario stellt eine Szene in einer Stadt dar. Die Stadt besteht aus 52 Häusern, als mobile Objekte gibt es einen Monstertruck, bestehend aus 7 einzelnen Objekten, einen Volksbot, bestehend aus 7 einzelnen Objekten, und 5 Bälle, also insgesamt 9 bewegliche Objekte. Die Szene besteht folglich aus einer unbeweglichen Objektmenge, 2 beweglichen Objektmengen und 5 einzelnen Objekten. Ohne Verwendung der Hüllkörperhierarchie werden in jedem Zeitschritt 1155 Tests zwischen Objekten durchgeführt. Unter Einbeziehung der Hüllkugelhierarchie werden pro Zeitschritt zwischen 8 und 52 Objekt-Tests und zwischen 76 und 330 Hüllkörper-Tests durchgeführt, vergleiche hierzu auch Abbildung 5.16. Durchschnittlich sind das pro Zeitschritt 30 Objekt-Tests und 195 Hüllkörper-Tests.



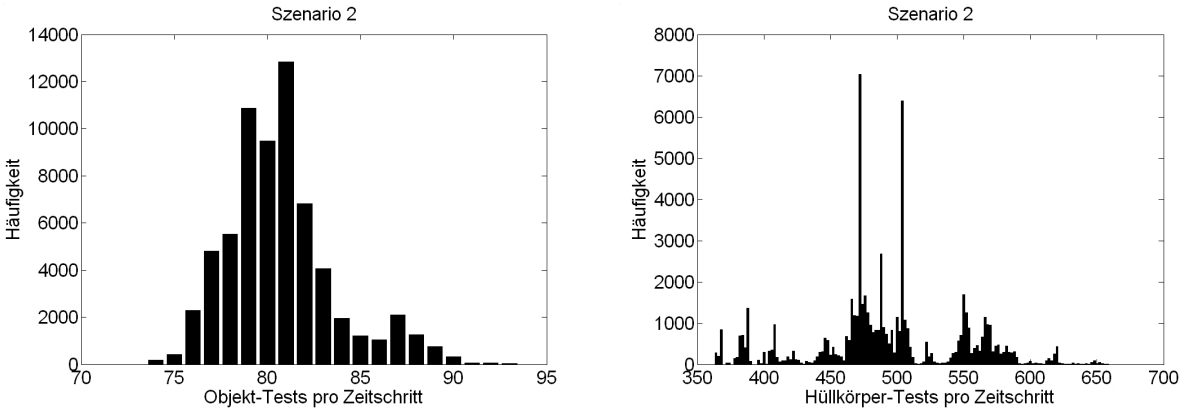
(a) (b)

Abbildung 5.16: *Benötigte Objekt-Tests und Hüllkörper-Tests im ersten Szenario.*

Szenario 2 (Abbildung 5.15(b)): Das zweite Szenario ist eine Indoor-Szene, in der die Wände der Räume aus 186 Blocks aufgebaut sind. Ein Monstertruck (7 Objekte) und ein Pioneer (8 Objekte) suchen nach 8 Bällen, es gibt also insgesamt 23 bewegliche Objekte. Die Szene besteht folglich aus einer unbeweglichen Objektmenge, 2 beweglichen Objektmengen und 8 einzelnen Objekten. Ohne Verwendung der Hüllkörperhierarchie müssten insgesamt 2985 Schnitttests zwischen den Objekten pro Zeitschritt stattfinden, dies ist in Echtzeit nicht mehr möglich. Mit der Hüllkörperhierarchie werden für die gleiche Szene zwischen 74 und 93 Objekt-Tests und zwischen 364 und 658 Hüllkörper-Tests benötigt, durchschnittlich ergeben sich 81 Objekt-Tests und 492 Hüllkörper-Tests, siehe auch Abbildung 5.17

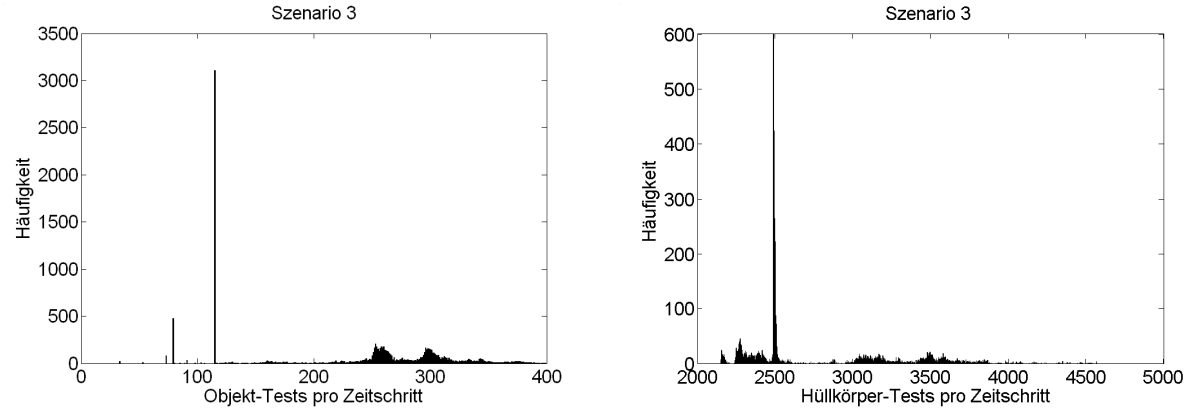
Szenario 3 (Abbildung 5.15(c)): Das dritte Szenario ist das bereits oben erwähnte Roboter-Bowling. Mit 10 Humanoid-Robotern (je 22 Objekte) und einem Volksbot (7 Objekte), also insgesamt 227 bewegliche Objekte auf 11 Objektmengen aufgeteilt, müssten ohne Hüllkörperhierarchie in jedem Zeitschritt 19814 Objekt-Tests durchgeführt werden, es ist also nicht verwunderlich, das dies nicht in Echtzeit ausführbar ist. Mit Hüllkörperhierarchie werden, bevor der Volksbot auf die Humanoid-Roboter trifft, ca. 115 Objekt-Tests und ca. 2500 Hüllkugel-Tests pro Zeitschritt durchgeführt. Beim Zusammenprall steigt dies auf bis zu 400 Objekt-Tests und bis zu 4500 Hüllkörper-Test. Dies ist in Abbildung 5.18 besonders gut zu erkennen.

Insgesamt sieht man, dass mit der Hüllkörperhierarchie durchschnittlich weniger als 3% der ursprünglichen Objekt-Tests durchgeführt werden, die durchschnittliche Anzahl an Hüllkörper-Tests im Vergleich zu den ursprünglichen Objekt-Tests liegt bei ca. 16%, hierbei muss noch beachtet werden, dass die Hüllkörper-Tests wesentlich schneller durchgeführt werden können, als exakte Objekt-Tests. Selbst im schlimmsten Fall, wenn viele Objekte dicht beieinander liegen oder sich berühren, steigen die benötigten Objekt-Tests nicht über 5% der Tests ohne Hierarchie, die Hüllkörper-Tests bleiben unter 30%.



(a) (b)

Abbildung 5.17: *Benötigte Objekt-Tests und Hüllkörper-Tests im zweiten Szenario.*



(a) (b)

Abbildung 5.18: *Benötigte Objekt-Tests und Hüllkörper-Tests im dritten Szenario.*

	Roboter	statische Objekte	Bälle	ohne Hierarchie	mit Hierarchie
Szenario 1	1 Monstertruck, 1 Volksbot	52 Häuser, keine Banden	5	1155 Tests	durchschnittlich 30 Objekt-Test, 195 Kugel-Tests
Szenario 2	1 Monstertruck, 1 Pioneer	186 Häuser, 4 Banden	8	2985 Tests	durchschnittlich 81 Objekt-Tests, 492 Kugel-Tests
Szenario 3	1 Volksbot, 10 HR18	keine	0	19814	durchschnittlich 227 Objekt-Tests, 2879 Kugel-Tests

Tabelle 5.2: *Übersicht der Performance mit verschiedenen Setups*

Kapitel 6

Modellierung und Simulation von Fahrzeugen

Im Rahmen dieser Arbeit sollten der am Fachgebiet neu erworbene Monstertruck sowie weitere Fahrzeuge simuliert werden. Wie in Kapitel 3 beschrieben, stehen für die Modellierung von Robotern bereits Funktionen zur Verfügung, mit denen eine baumförmige Struktur bestehend aus Gabelungen, konstanten und variablen Translationen und Rotationen aufgebaut werden kann.
Es wurden zwei verschiedene Fahrzeugklassen betrachtet. Sie werden in den folgenden Abschnitten beschrieben.

6.1 Differentialgetriebene Fahrzeuge

Dreirädrige differentialgetriebene Fahrzeuge haben zwei unabhängig voneinander angetriebene Räder und ein passives Stützrad. Wenn beide Räder gleich schnell drehen, fährt das Fahrzeug geradeaus. Das Fahrzeug kann Kurven fahren, indem das kurvenäußere Rad schneller rotiert als das kurveninnere Rad. Es kann eine Drehung auf der Stelle erzeugt werden, wenn beide Räder mit gleicher Geschwindigkeit in entgegengesetzten Richtungen drehen.
Es gibt natürlich auch differentialgetriebene Fahrzeuge mit mehr als drei Rädern, wie z.B. die Volksbots [22]. Es gibt hier ein dreirädriges, aber auch ein vier- und ein sechsrädriges differentialgetriebenes Fahrzeug. Bei mehr als drei Rädern entfällt das Stützrad, dafür sind dann alle Räder angetrieben. Es werden jeweils die Räder auf der linken und der rechten Seite mit der gleichen Geschwindigkeit angetrieben.
Für ein dreirädriges differentialgetriebenes Fahrzeug werden in [23] die Herleitung der Vorwärtskinematik und ein Ansatz für die inverse Kinematik gezeigt. Dies lässt sich für die Volksbots sofort verallgemeinern, da die zusätzlichen Räder mit der gleichen Geschwindigkeit angetrieben werden wie das vordere Rad auf jeder Seite. Die Vorwärtskinematik wurde für den Pioneer2-DX bereits umgesetzt und für die Kinematiksimulation (siehe Kapitel 3) verwendet. Für die Simula-

tion mit Dynamikeinflüssen wird sie nicht benötigt.
Das vorhandene Modell des Pioneers wurde für die Dynamiksimulation angepasst. Außerdem wurde ein sechsrädriger Volksbot modelliert.

6.2 Ackermann-gelenkte Fahrzeuge

Eine weitere Antriebsart ist die Ackermann-Lenkung. Hierunter versteht man die Lenkung, wie sie in vierrädrigen Autos verwendet wird: es gibt zwei lenkbare Vorderräder und zwei Hinterräder. Entweder die Vorderräder, oder die Hinterräder, oder alle vier Räder sind angetrieben. Kurvenfahrten ergeben sich durch das Einschlagen der Vorderräder.
Nach [23] müssen sich für ein reines Rollen alle vier Radnormalen (also die Drehachsen der Räder) in einem gemeinsamen Punkt (bei Fahrt geradeaus im Unendlichen) schneiden. Dieser Punkt wird *momentanes Drehzentrum* (instantaneous center of curvature/rotation, ICC/ICR) genannt. Anschaulich ist klar, dass das kurveninnere Rad stärker eingeschlagen werden muss als das äußere Rad, zudem muss die Rotationsgeschwindigkeit der äußeren Räder größer sein als die der inneren, damit sich das gesamte Fahrzeug gleichmäßig auf der Kreisbahn bewegt.

Um die zueinander passenden Einschlagswinkel auszurechnen, müssen zunächst einige Überlegungen bezüglich der Winkel in Abbildung 6.1 angestellt werden. α_l ist der Einschlagswinkel des linken Vorderrades. β und α_l bilden zusammen einen rechten Winkel, deshalb gilt $\beta = 90° - \alpha_l$. β und γ bilden nach Definition auch einen rechten Winkel, daher gilt $\gamma = 90° - \beta = 90° - (90° - \alpha_l) = \alpha_l$. γ und δ sind Wechselwinkel und sind daher gleich groß [2], es gilt also $\delta = \gamma = \alpha_l$. Die gleichen Überlegungen gelten auch für das rechte Rad und man erhält α_r wie in Abbildung 6.1 eingezeichnet.
α_r muss aus α_l so berechnet werden, dass sich alle vier Radnormalen wie in Abbildung 6.1 in einem gemeinsamen Punkt, dem ICC, schneiden. Hierfür wird zunächst das Dreieck $\{A\ C\ ICC\}$ betrachtet. Es hat in C einen rechten Winkel, daher gilt

$$d_l := Abstand(A, ICC) = l/\sin(\alpha_l).$$

Aus dem Satz des Pythagoras folgt

$$d_l^2 = l^2 + (b + R)^2 \Rightarrow R = \sqrt{d_l^2 - l^2} - b.$$

Damit sind für das rechtwinklige Dreieck $\{B\ D\ ICC\}$ zwei Seiten bestimmt. Die dritte Seite erhält man ebenfalls mit dem Satz des Pythagoras:

$$d_r := Abstand(B, ICC) = \sqrt{l^2 + R^2}.$$

Daraus ergibt sich dann

$$\alpha_r = \arcsin\left(l/d_r\right).$$

Damit sich das Fahrzeug ohne Rutschen fortbewegt, müssen auch die Drehgeschwindigkeiten der Räder aufeinander abgestimmt werden. Bei konstantem Radeinschlag fährt das Fahrzeug eine

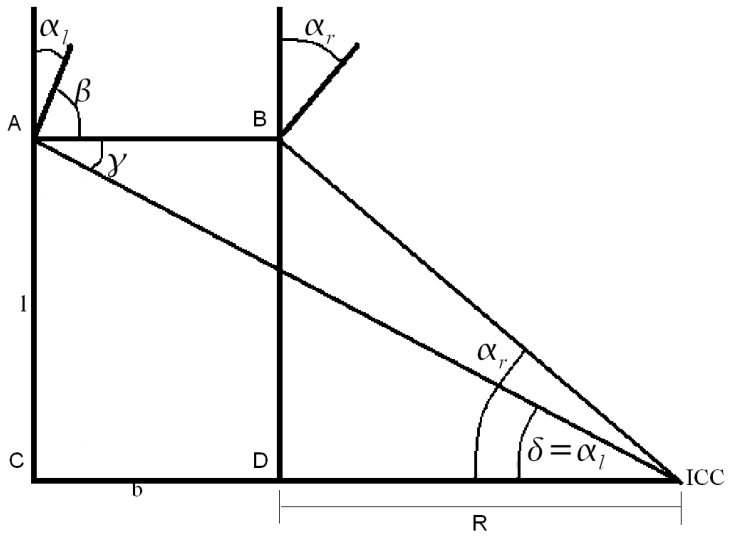


Abbildung 6.1: *Betrachtung der Einschlagswinkel an einem Fahrzeug mit Ackermann-Lenkung.*

Kreisbahn um das ICC. Damit alle Räder in der gleichen Zeit einen vollen Kreis beschreiben, muss das Verhältnis von Geschwindigkeit zum Umfang des Kreises gleich sein. Der Abstand des linken Vorderrades zum ICC ist d_l, der des rechten d_r wie oben berechnet. Wird für das linke Vorderrad eine Geschwindigkeit v_l vorgegeben, muss also gelten

$$\frac{v_l}{U_l} = \frac{v_r}{U_r} \quad \Rightarrow \quad \frac{v_l}{2\pi d_l} = \frac{v_r}{2\pi d_r} \quad \Rightarrow \quad v_r = \frac{v_l d_r}{d_l}$$

Die gleiche Überlegung gilt auch für die Hinterräder mit $d_{l2} = R + b$ und $d_{r2} = R$.

Die Einschlagswinkel der Vorderräder und die Geschwindigkeiten der Räder hängen also nur vom Abstand der Vorderräder zu den Hinterrädern und vom seitlichen Abstand der Räder zueinander ab. Es wurde daher zunächst eine allgemeine Basisklasse für alle Ackermann-gelenkten Fahrzeuge modelliert, die nur anhand dieser Größen ein körperloses Grundgerüst aufbaut. Ein Zweig des Baumes wurde für Erweiterungen in den Kindklassen offen gelassen. In den Kindklassen werden nur noch die geometrischen Abmessungen der einzelnen Fahrzeugteile definiert sowie fahrzeugspezifische Erweiterungen eingefügt wie z.B. Sensoren.
Aufbauend auf dieser Basisklasse wurde ein Modell des Monstertrucks erstellt. Als Erweiterungen befinden sich eine Kamera und ein Laserscanner auf dem Dach. Außerdem wurde, um zu zeigen dass aus dem allgemeinen Basismodell sehr verschiedene Fahrzeuge erzeugt werden können, eine Stretch-Limousine modelliert.
Alle zur Verfügung stehenden Fahrzeuge sind in Abbildung 6.2 zu sehen.

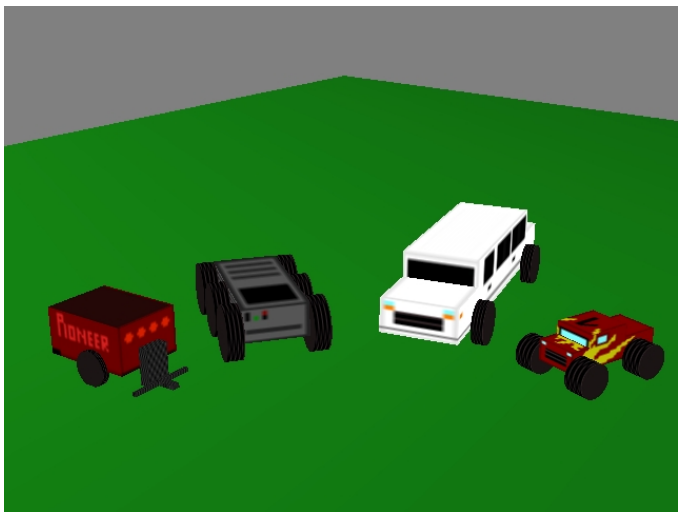


Abbildung 6.2: *Von links nach rechts: Pioneer, Volksbot, Stretch-Limousine und Monstertruck.*

6.3 Fahrverhalten mit Dynamikalgorithmen

Nach der Modellierung der Fahrzeuge musste getestet werden, ob sie auch mit den zur Verfügung stehenden Algorithmen die erwünschten Fahreigenschaften zeigen. Bei ersten Tests mit dem vereinfachten Dynamikalgorithmus zeigte sich das Problem, dass die Haftung der Reifen auf dem Boden so groß war, dass durch das Drehen der Räder keine Bewegung des gesamten Fahrzeugs nach vorne entstand, sondern die Räder blieben auf der Stelle und das Fahrzeug drehte sich um die Räder, vgl. Abbildung 6.3.

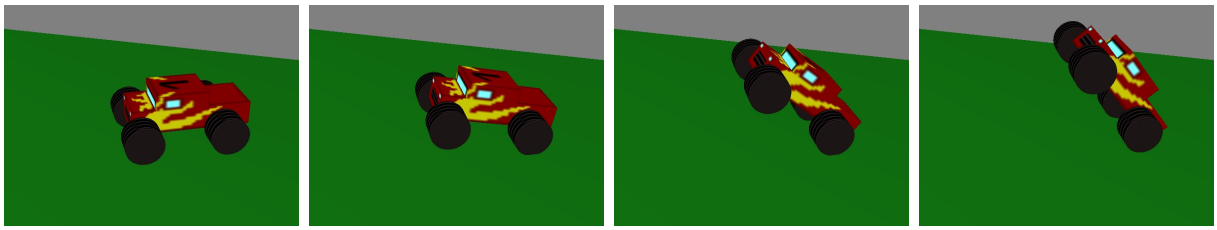


Abbildung 6.3: *Erste Fahrversuche mit dem Monstertruck.*

Dieses Problem konnte schnell behoben werden, indem der Reibungsfaktor der Oberfläche der Räder verringert wurde. Mit diesen Einstellungen fahren alle Fahrzeuge, wie man es erwartet. Fahrzeuge mit großen Rädern können Unebenheiten überwinden, Fahrzeuge mit kleinen Rädern hingegen scheitern bereits an kleinen Hindernissen. Differentialgetriebene Fahrzeuge fahren Kurven, wenn ein Rad schneller dreht als das andere und sie drehen bei entgegengesetzten Rotationsrichtungen auf der Stelle. Fahrzeuge mit Ackermann-Lenkung folgen mit eingeschlagenen Rädern sowohl vorwärts als auch rückwärts der erwarteten Kreisbahn.

6.4 Simulation von Laserscannern

Um einen Laserscanner (oder einen beliebigen anderen Abstandssensor) zu simulieren, gibt es zwei grundlegend verschiedene Ansätze: Man kann entweder auf die OpenGL-Funktionen zurückgreifen und den Tiefenpuffer der rasterisierten Szene auslesen, oder man kann Strahlen gegen die Szene schneiden und daraus den kürzesten Abstand zu einem Objekt bestimmen. Beide Verfahren wurden im Rahmen dieser Arbeit implementiert und werden im Folgenden beschrieben und miteinander verglichen.

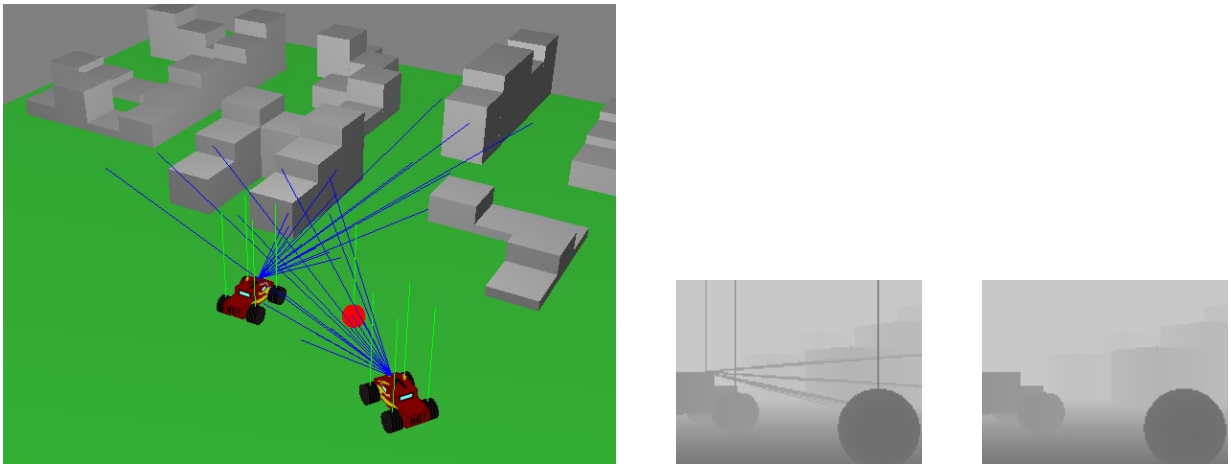
6.4.1 Abstandsbestimmung durch Auslesen des Tiefenpuffers

Die wichtigsten Funktionen, um diesen Ansatz umzusetzen, standen bereits in `MuRoSimF` zur Verfügung. Der `RenderManager` (siehe Kapitel 3) ermöglicht es, eine Szene aus verschiedenen Blickwinkeln zu darzustellen. Bei der Rasterisierung werden pro Pixel sowohl drei Farbwerte für rot, grün und blau gespeichert als auch ein Tiefenwert, der angibt, wie groß der Abstand vom Augpunkt zum gezeichneten Pixel ist. Der Sichtbereich wird durch zwei Ebenen, die *near*- und die *far*-Clipping-Plane, begrenzt. Objekte, die näher am Augpunkt sind, als die near-Clipping-Plane, oder weiter entfernt als die far-Clipping-Plane, werden nicht gezeichnet. Ähnlich wie bei der Simulation der Kameras werden die Bildinformationen ausgehend vom Blickpunkt des Sensors gelesen, für den Laserscanner müssen also die Tiefenwerte ausgelesen werden.
Da im Hauptfenster der Simulation auch Informationen gezeichnet werden können, die vom Laserscanner eigentlich nicht als Hindernisse erkannt werden sollen (z.B. die Anzeige von Kollisionen oder die Visualisierung anderer Laserscanner), wird für die Laserscanner ein eigener `RenderManager` angelegt, bei dem nur die Objekte angemeldet werden, die auch als Hindernisse erkannt werden sollen. Der Unterschied wird in Abbildung 6.4 verdeutlicht. Dieses Vorgehen ermöglicht es auch, bestimmte Objekte gezielt für den Laserscanner unsichtbar zu machen. So hat man die Möglichkeit, „getarnte“ Objekte zu simulieren, oder man kann den Laserscanner durch Wände schauen lassen. Ein weiterer Vorteil der Verwendung eines zweiten `RenderManagers` ist, dass die Reichweite des Laserscanners einfach verändert werden kann, indem man die Position der far-Clipping-Plane des `RenderManagers` ändert. Dadurch müssen die Abstandswerte nach dem Auslesen nicht per Hand auf die Reichweite des Laserscanners gekürzt werden.

6.4.2 Abstandsberechnung durch Schnitt von Strahlen gegen Objekte

Für diese Methode wird eine Menge von Strahlen erzeugt. Für jeden Strahl muss die kürzeste Entfernung zu den Objekten der Szene berechnet werden. Hierfür wird für jeden Primitiv-Typ eine Funktion benötigt, die bestimmt, ob ein Strahl das Primitiv schneidet, und falls dies der Fall ist, wo der erste Berührpunkt vom Strahlursprung aus gesehen liegt. Da kein Laserscanner eine unbegrenzte Reichweite hat, wird nicht mit Strahlen, sondern mit Strecken gearbeitet. Dies ver-

einfacht einige Berechnungen, weiterhin kann die Reichweite des Laserscanners über die Länge der Strecken verändert werden.



(a) (b) (c)

Abbildung 6.4: *(a) zeigt eine Szene mit eingeschalteter Visualisierung von Kollisionen und Laser-Scans. Die beiden kleinen Bilder zeigen Graustufenvisualisierungen der ausgelesenen Tiefenbilder des rechten Fahrzeugs, (b) mit Benutzung des allgemeinen* `RenderManagers` *und (c) mit Benutzung eines eigenen* `RenderManagers`*.*

6.4.2.1 Schnitt zwischen Strecke und Kugel

Die Kugel sei durch ihren Mittelpunkt m und ihren Radius r gegeben, die Strecke durch die Gleichung $S(t) = p+t\cdot d, 0 \leq t \leq t_{max}$. Zunächst wird der Abstand vom Mittelpunkt der Kugel zur Strecke bestimmt, wie in Kapitel 4.1.3 beschrieben. Ist dieser Abstand größer als r, gibt es keinen Schnittpunkt. Andernfalls bestimmt man den ersten Eintritt der Strecke in die Kugel wie in [8] beschrieben. Man setzt die Gleichung der Strecke in die Kugelgleichung $(x-m)\cdot(x-m) = r^2$ ein und erhält

$$(p + t \cdot d - m) \cdot (p + t \cdot d - m) = r^2.$$

Mit $p - m = v$ erhält man

$$\begin{aligned} & (v + t \cdot d) \cdot (v + t \cdot d) = r^2 \\ \Leftrightarrow \quad & (d \cdot d)t^2 + 2(v \cdot d)t + v \cdot v - r^2 = 0 \\ \Leftrightarrow \quad & t^2 + 2(v \cdot d)t + v \cdot v - r^2 = 0 \quad (\|d\| = 1!) \end{aligned}$$

Dies ist eine quadratische Gleichung in t, die keine, eine oder zwei Lösungen haben kann. Da zuvor schon berechnet wurde, dass die Strecke die Kugel schneidet, muss es mindestens eine

Lösung geben. Gibt es zwei Lösungen, wählt man die kleinere, da dies der erste Eintritt der Strecke in die Kugel ist. Ist $t < 0$ bedeutet das, dass der Anfangspunkt der Strecke im Inneren der Kugel liegt.

6.4.2.2 Schnitt zwischen Strecke und Ebene

Diese Berechnung erfolgt ähnlich wie in Kapitel 4.1.6, mit dem Unterschied, dass die Berechnung abgebrochen werden kann, wenn kein Schnitt vorliegt. Ist wieder $S(t) = p + t \cdot d, 0 \leq t \leq t_{max}$ die Gleichung der Strecke und q der Schnittpunkt zwischen Strahl und Ebene, dann gilt $q = S(t_{schnitt})$ mit $t_{schnitt} = \|q - p\|$.

6.4.2.3 Schnitt zwischen Strecke und Quader

Hier wird der in [8] vorgeschlagene Weg verfolgt. Ein Quader wird durch 6 Ebenen definiert, von denen jeweils zwei parallel zueinander sind. Für jedes der drei Ebenen-Paare wird der Raum betrachtet, den sie einschließen. Man berechnet den Bereich der Strecke, der in diesem Raum liegt, indem man die Schnittpunkte der Strecke mit den Ebenen berechnet. Gibt es keinen Schnittpunkt, wird der Punkt gewählt, der am nächsten an der Ebene liegt. Man erhält also für jedes Ebenenpaar zwei Werte t_{min} und t_{max} die beschreiben, welcher Bereich der Strecke im eingeschlossenen Raum liegt. Insgesamt werden 6 Werte $t_{min_x}, t_{max_x}, t_{min_y}, t_{max_y}, t_{min_z}, t_{max_z}$ berechnet. Setzt man $t_{min} = \max(t_{min_x}, t_{min_y}, t_{min_z})$ und $t_{max} = \min(t_{max_x}, t_{max_y}, t_{max_z})$, dann beschreibt t_{min} den Punkt, an dem die Strecke in den letzten der drei Bereiche eintritt, t_{max} beschreibt den Punkt, an dem die Strecke den ersten der Bereiche wieder verlässt. Die Strecke schneidet den Quader also nur, wenn $t_{min} < t_{max}$ gilt. Dieses Vorgehen wird in Abbildung 6.5 verdeutlicht.

6.4.2.4 Schnitt zwischen Strecke und Zylinder

Der Schnittest zwischen Strecke und Zylinder richtet sich auch nach [8]. Zuerst werden die Strecke und der unendliche Zylinder betrachtet, der entsteht, wenn der Zylinder entlang der Mittelachse verlängert wird. Zunächst wird, ähnlich wie beim Schnitt zwischen Strecke und Kugel, die Gleichung der Strecke in die Zylindergleichung eingesetzt. Die Zylindergleichung ist gegeben durch $(v \times d) \cdot (v \times d) = r^2$ mit $v = x - p$, p ein Punkt auf der Zylinderachse, x ein beliebiger Punkt auf der Zylinderoberfläche und d die normierte Richtung der Zylinderachse. Man setzt für x die Gleichung der Strecke ein und erhält wieder eine quadratische Gleichung in t. Hat diese Gleichung keine Lösung, schneidet die durch die Strecke definierte Gerade den unendlichen Zylinder nicht. Gibt es eine Lösung, tangiert die Gerade den Zylinder, bei zwei Lösungen gibt es einen Punkt, an dem die Gerade in den Zylinder eintritt und einen Punkt, an dem sie wieder austritt. Jetzt muss überprüft werden, ob diese Punkte nicht nur auf der Geraden, sondern auch auf der Strecke liegen, also $0 \leq t \leq t_{max}$. Außerdem müssen die Punkte auch auf der Oberfläche des endlichen Zylinders liegen. Sind beide Punkte außerhalb des endlichen Zylinder, gibt es keinen

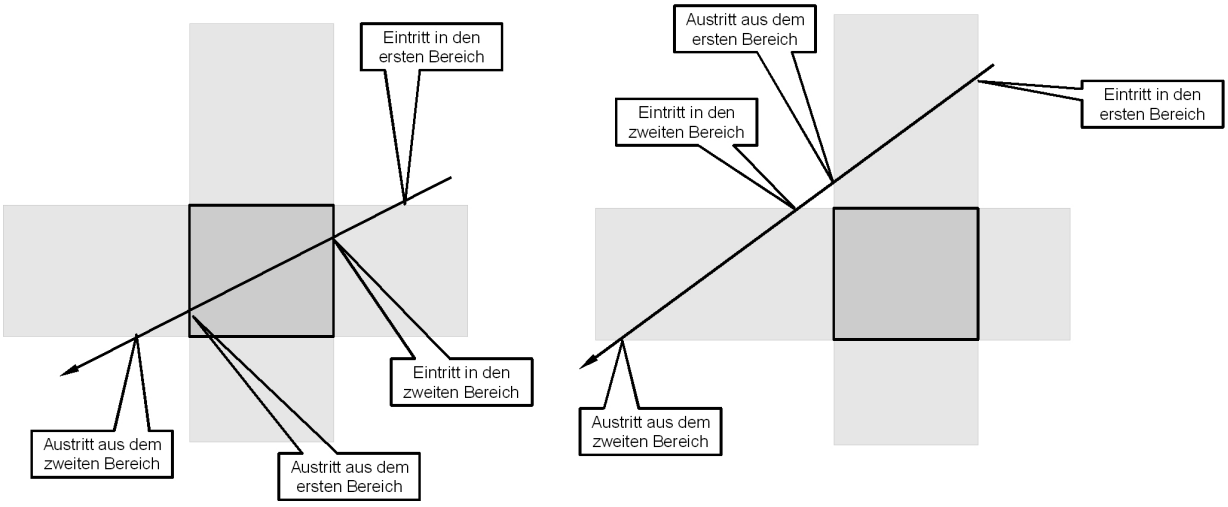


Abbildung 6.5: *Auf dem linken Bild sieht man, dass die Strecke erst in beide Bereiche eintritt, bevor sie aus dem ersten wieder austritt. Auf dem rechten Bild erfolgt der zweite Eintritt erst, nachdem der erste Bereich wieder verlassen wurde, die Strecke schneidet das Rechteck nicht.*

Schnitt. Sind beide Punkte auf der Oberfläche, definiert der nähere den Abstand zum Ursprung der Strecke. Liegt nur ein Punkt außerhalb, muss der Schnittpunkt der Strecke mit den Ebenen berechnet werden, die den Zylinder begrenzen.

6.4.2.5 Arbeitsweise des Laserscanners

Es wäre viel zu aufwendig, jede erzeugte Strecke gegen jedes Objekt der Szene zu schneiden. Daher wird auf die in Kapitel 5.5 beschriebenen Hüllkörperhierarchien zurückgegriffen. Nur wenn eine Strecke den Hüllkörper eines Knotens schneidet, und der früheste Schnittpunkt näher liegt als der bisher nächste gefundene, muss die Strecke gegen die Kindknoten geschnitten werden.
Da die Stecken alle sehr dicht beieinander liegen, wird jede Strecke zuerst gegen das Objekt geschnitten, das am nächsten an der zuvor bearbeiteten Strecke liegt. Durch Ausnutzung dieser lokalen Kohärenz kann in vielen Fällen gleich zu Beginn eine gute Grenze für die Entfernung bestimmt werden. Es müssen zwar trotzdem noch alle anderen Hierarchien betrachtet werden, aber die meisten können dann schon frühzeitig ausgeschlossen werden.

6.4.3 Ergebnisse

Beide Ansätze wurden erfolgreich implementiert, allerdings ist die Performance sehr unterschiedlich. Beide Ansätze benötigen mehr Zeit, je mehr Objekte in der Szene enthalten sind. Zum Auslesen des Tiefenpuffers müssen dann mehr Objekte rasterisiert werden, für die Abstandsbe-

rechnung müssen mehr Schnitttests durchgeführt werden. Die im Folgenden beschriebenen Tests wurden auf einem Laptop (Intel Pentium M processor, 1.5 GHz, 512 MB RAM, ATI Mobility Radeon 9700/64MB) durchgeführt.
Mit der ersten Methode kann man sehr gut 10 Fahrzeuge mit Laserscannern gleichzeitig simulieren. Bei gleicher Auflösung des Laserscans (160×120 Pixel) würden bei der zweiten Methode insgesamt 19200 Strecken entstehen, für die das nächste Objekt bestimmt werden muss. Dies führt schon bei der Simulation eines einzigen Laserscanners zu Performance-Problemen. Es gibt daher einen Faktor, der über eine Konfigurationsdatei gesetzt werden kann, durch den die Auflösung dividiert wird. Bei einem Faktor von 2 erhält man also eine Auflösung von 80×60 Pixeln, damit ist es auch möglich, ein Fahrzeug mit Laserscanner zu simulieren. Setzt man den Faktor auf 5, kann man gleichzeitig 7 Fahrzeuge mit Laserscannern simulieren, allerdings beträgt die Auflösung dann nur noch 32×24 Pixel, mit Faktor 6 (und damit einer Auflösung von 26×20) können 10 Fahrzeuge mit Laserscannern simuliert werden.

Man sieht also, dass die erste Methode ganz klar effizienter ist als die zweite. Allerdings arbeitet die Hüllkörperhierarchie mit Hüllkugeln, die eine schlechte Hülleffizienz haben. Jede Verbesserung in der Hüllkörperhierarchie verbessert also auch die Performance der Laserscanner, die nach der zweiten Methode arbeiten. Für die erste Methode müssen immer alle Objekte im Sichtbereich rasterisiert werden. Wird nur eine geringe Auflösung benötigt (z.B. ein eindimensionaler Abstandssensor, hierfür muss in jedem Zeitschritt nur eine Strecke gegen die Szene geschnitten werden), fällt der Performance-Unterschied nicht ins Gewicht, da auch der OpenGL-Zugriff eine gewisse Zeit benötigt. In solchen Fällen ist es sinnvoll, auf die zweite Methode zurückzugreifen. Wo genau die Grenze liegt, wann auf welche Methode zurückgegriffen werden sollte, ist schwer zu sagen, da bei sehr geringer Auflösung (unter 20×15) der limitierende Faktor nicht mehr der Sensor ist, sondern die Kollisionserkennung zwischen den Fahrzeugen.

Der Nachteil an Laserscannern, die den Tiefenpuffer auslesen, ist, dass sie mit steigender Reichweite weniger exakt arbeiten, da der Tiefenpuffer nur eine begrenzte Auflösung bietet und besonders bei weit entfernten Objekten ungenau wird. Der Schnitttest zwischen Strecken und Objekten ist unabhängig von der Entfernung gleichmäßig genau, er wird nur durch Rundungsfehler beeinflusst.

Die Funktionen, die für die zweite Methode entwickelt wurden, können leicht für andere Anwendungen weiterverwendet werden. Soll z.B. für Offline-Simulationen ein Raytracer implementiert werden, kann auf die Funktionen zum Schneiden von Strecken und Primitiven zurückgegriffen werden.

Kapitel 7

Zusammenfassung

Im Rahmen dieser Arbeit wurde eine vollständige Kollisionserkennung für `MuRoSimF` entwickelt. Sie setzt sich zusammen aus der Abstandsbestimmung zwischen Basiselementen, den Primitivtests und der Hüllkörperhierarchie.
Abstandsbestimmung zwischen Basiselementen: Es wurden Funktionen implementiert, die schnell und robust den kürzesten Abstand zwischen den Basiselementen Punkt, Gerade, Strecke und Ebene bestimmen. Mit Hilfe der vorgestellten Ideen können leicht Abstandsberechnungen für andere Grundelemente abgeleitet werden.
Primitivtests: Zwischen je zwei der vorgegebenen Primitive (Kugel, Ebene, Quader und Zylinder) kann mit den vorgestellten Tests berechnet werden, ob eine Kollision auftritt, außerdem werden der Punkt, in dem die Kraft angreift und die Richtung der Kraft berechnet. Nur mit diesen Funktionen ist es bereits möglich, kleine Szenen mit Dynamikeinfluss zu simulieren. Die Komplexität der Szenen ist allerdings begrenzt, da der Aufwand für die Kollisionserkennung quadratisch in der Anzahl der verwendeten Objekte ist.
Hüllkörperhierarchie: Aus gegebenen Objektmengen werden Hüllkugelhierarchien aufgebaut. Bei Robotern richtet sich die Struktur nach der bereits vorhandenen Baumstruktur, für ungeordnete Objektmengen wurde ein top-down-Verfahren implementiert, das balancierte Bäume minimaler Tiefe erzeugt. Für die Hüllkörper wurde auf Kugeln zurückgegriffen. Alle verwendeten Algorithmen können durch den modularen Aufbau leicht gegen andere ausgetauscht werden.
Durch den Einsatz der Hüllkörperhierarchie konnte die Anzahl der benötigten Schnitttests gegenüber vorher um durchschnittlich 80% gesenkt werden. In den meisten Anwendungen können zwei- bis drei-mal so viele Roboter wie vorher gleichzeitig simuliert werden.
Die Hüllkörperhierarchien können nicht nur von der Kollisionserkennung genutzt werden, auch andere Algorithmen wie z.B. die Simulation der Laserscanner können darauf zurückgreifen.

Weiterhin wurden verschieden Fahrzeuge mit Differential-Getriebe oder Ackermann-Lenkung modelliert. Für die Ackermann-gelenkten Fahrzeuge wurden Formeln hergeleitet, mit denen der Lenkeinschlag der Räder und die Rotationsgeschwindigkeit der Räder berechnet werden können, so dass sich das Fahrzeug ohne Rutschen, mit reinem Rollen, fortbewegt.
Für die Simulation von Laserscannern wurden zwei verschieden Ansätze umgesetzt. Die erste

Version arbeitet ähnlich wie die simulierten Kameras: mit Hilfe der OpenGL-Funktionen werden die Tiefenwerte des Bildes vom Blickpunkt des Sensors ausgelesen. Die zweite Version erzeugt eine Menge von Strahlen, die gegen die Szene geschnitten werden. Damit nicht jeder Strahl gegen jedes Objekt geschnitten werden muss, wird auf die Hüllkörperhierarchien zurückgegriffen.

Insgesamt wurde `MuRoSimF` um viele Funktionalitäten erweitert, die es ermöglichen, wesentlich vielseitigere Simulationen zu erstellen. Dadurch ist der mögliche Einsatzbereich von `MuRoSimF` stark vergrößert worden. Es ist eine Applikation entstanden, die sehr unterschiedlich konfiguriert werden kann (siehe Anhang B), verschiedene Einsatzmöglichkeiten sind in Abbildung 5.15 zu sehen.

Kapitel 8

Ausblick

Das Simulatorframework `MuRoSimF` wird ständig weiterentwickelt. Aus dieser Arbeit ergeben sich eine Vielzahl an Erweiterungsmöglichkeiten, die hier kurz vorgestellt werden.

8.1 Kollisionserkennung

Die gesamte Kollisionserkennung und die verwendeten Hüllkörperhierarchien wurden so aufgebaut, dass sehr einfach einzelne Teile ausgetauscht oder erweitert werden können.

8.1.1 Verwendung anderer Hüllkörper

Sollen anstatt der Kugeln andere Hüllkörper mit besserer Hülleffizienz verwendet werden, muss jedes Primitiv einen solchen Hüllkörper angehängt bekommen. Zusätzlich muss eine Funktion zur Verfügung gestellt werden, die für zwei dieser Hüllkörper einen gemeinsamen, beide umschließenden Hüllkörper berechnet. Außerdem wird eine Funktion benötigt, die möglichst schnell bestimmt, ob sich zwei Hüllkörper schneiden oder nicht. Alle anderen Teile der Hierarchie, wie z.B. der Aufbau des Baumes oder die Traversierung, sind unabhängig von den Hüllkörpern.

8.1.2 Einbindung anderer Verfahren zum Aufbau des Baumes

Man kann sehr einfach im verwendeten top-down-Verfahren eine andere Strategie zum Aufteilen der Objekte einbinden, hierfür muss nur an einer Stelle eine Funktion ausgetauscht werden. Genauso einfach lässt sich das gesamte Verfahren austauschen, stattdessen kann ein bottom-up- oder insertion-Verfahren verwendet werden. Dies sollte so umgesetzt werden, dass das verwen-

dete Verfahren in der Konfigurationsdatei festgelegt wird. Dadurch ist es möglich, für jede Simulation individuell die beste Aufbau-Strategie auszuwählen.

8.1.3 Zusätzliche Verwendung von Raumunterteilung

Für größere Szenarien ist zu erwarten, dass die reine Verwendung von Hüllkörperhierarchien nicht mehr ausreicht. In einem großen Outdoor-Szenario mit vielen Objekten sollten sehr weit voneinander entfernte Objekte sofort als Kollisionspartner ausgeschlossen werden. Wenn die Szene aber so modelliert wird, dass aus der gesamten statischen Umwelt eine einzige Hüllkörperhierarchie aufgebaut wird, sind mindestens zwei Überlappungstests notwendig um herauszufinden, dass ein Auto im Süden der Stadt nicht mit den Häusern im Norden kollidieren kann. Dies kann umgangen werden, indem auf Raumunterteilungsverfahren zurückgegriffen wird und diese mit den Hüllkörperhierarchien kombiniert werden.
Ein einfacher Ansatz, der aber vermutlich eine starke Performance-Steigerung bewirkt, ist es, ein grobes zweidimensionales Gitter über die Szene zu legen und die statischen Elemente der Szene fest den einzelnen Zellen zuzuordnen. Dadurch ist es nicht mehr nötig, für diese Objekte eine Hüllkörperhierarchie aufzubauen. Für die einzelnen beweglichen Hüllkörperhierarchien muss in jedem Zeitschritt nur berechnet werden, zu welchen Zellen sie gehören. Dies kann man bei einem gleichmäßigen Gitter sehr einfach an den Weltkoordinaten ablesen. Es müssen dann nur noch die Hierarchien und Objekte gegeneinander geschnitten werden, die auch in der gleichen Zelle liegen.

8.2 Sensoren

Die Funktionen für die Simulation des Laserscanners und die Kollisionserkennung ermöglichen es, ohne großen Aufwand weitere Sensoren zu simulieren.

8.2.1 Abstandssensoren

Aufbauend auf die Methoden, die für die Laserscanner entwickelt wurden, können alle möglichen anderen Abstandssensoren simuliert werden wie z.B. Ultraschallsensoren oder Tiefenbildkameras.
Eine spezielle Anwendung, die man hieraus gewinnen kann, ist eine simulierte Lichtschranke. Man verwendet hierfür einen eindimensionalen Laserscanner mit begrenzter Reichweite. Sobald ein Objekt im Bereich des Strahls entdeckt wird, löst die Lichtschranke aus. Es ist dabei nicht nötig, den kleinsten Abstand zum Sensor zu berechnen. Sobald ein Objekt in der Reichweite der Lichtschranke liegt, ist es nicht mehr wichtig, ob ein anderes Objekt noch näher ist, da nur entschieden werden muss, ob der Sensor anspricht oder nicht. Man kann also den Test abbrechen,

sobald ein Objekt in Reichweite gefunden wurde und muss den Strahl nicht gegen alle anderen Objekte oder Hüllkörperhierarchien schneiden.

8.2.2 Kontaktsensoren

Mit Hilfe der zur Verfügung stehenden Kollisionserkenner lassen sich sehr leicht Kontaktsensoren simulieren. Ein Objekt, das auf Kontakte reagieren soll, muss lediglich alle Kollisionen mit anderen Objekten an eine Routine zur Verarbeitung der Sensordaten weiterreichen. Da die Kollisionen sowieso immer berechnet werden, bedeutet dies keinen zusätzlichen Rechenaufwand für die Simulation.
Dieses Vorgehen hat allerdings den Nachteil, dass das gesamte Objekt als Sensor fungiert. Soll z.B. auf einer Fläche eines Quaders ein Kontaktsensor angebracht sein, sollte er nicht darauf reagieren, wenn an einer anderen Stelle des Quaders Kollisionen auftreten.
Dies kann auf zwei Arten umgangen werden:

- Es wird bestimmt, ob der Kollisionspunkt in dem Bereich liegt, den der Sensor abdecken soll. Hier können Probleme auftreten, da bei Kollisionen nicht die gesamte Schnittfläche zwischen zwei Primitiven berechnet wird, sondern nur ein Punkt, in dem die Kraft angreift. Es kann passieren, dass dieser Punkt außerhalb des Sensorbereichs liegt, obwohl die Schnittfläche den Sensorbereich schneidet.

- Alternativ kann man ein kleineres Objekt als Sensor in das größere hineinlegen. Es muss so platziert werden, dass es an der Stelle, wo der Sensor liegen soll, die Oberfläche des größeren Objekts schneidet, an allen anderen Stellen aber einen möglichst großen Abstand zur Oberfläche hat. Soll der Sensor klein sein, kann man eine Kugel mit sehr kleinem Radius an der Oberfläche platzieren. Soll eine größerer Teil einer Seitenfläche eines Quaders abgedeckt werden, kann man einen Quader mit sehr kleiner Höhe verwenden.
  Da das neu eingefügte Objekt vollständig in einem anderen enthalten ist, ist es auch in dessen Hüllkörper enthalten. Nur wenn das umschließende Objekt mit einem anderen Objekt kollidiert, müssen Kollisionen mit dem Sensor berechnet werden. Auf diese Art kostet auch diese Variante kaum Rechenzeit.

8.3 Höhenfelder

Vor Beginn dieser Arbeit stand nur ein Modell eines Indoor-Fahrzeugs, des Pioneers, zur Verfügung. Die neu modellierten Fahrzeuge sind Outdoor-Fahrzeuge. Da der Boden im Freien nur selten ganz eben und glatt ist, ist es wünschenswert, dies auch in der Simulation darstellen zu können.
Um unebenes Gelände darzustellen, bekommt der Boden an diskreten Stellen Werte für die Höhe,

die das Gelände an dieser Stelle haben soll, zugewiesen. Außerdem muss für jede der Stützstellen eine Normale berechnet werden. Um die Kollisionsberechnungen zu vereinfachen, sollten nur Höhenwerte zugelassen werden, die größer oder gleich 0 sind. Für die Visualisierung muss zwischen den diskreten Stützstellen interpoliert werden, um eine gleichmäßige Oberfläche zu erhalten. Wenn das Gitter nicht zu gob gewählt wird, sollte es für die Kollisionserkennung ausreichen, nur an den Stützstellen Schnitttests durchzuführen.
Setzt man voraus, dass das Weltkoordinatensystem immer so ausgerichtet ist, dass die z-Achse nach oben zeigt, kann man über einen einfachen Koordinatenvergleich der Objekte einen kleinen Bereich des Bodens eingrenzen, mit dem das Objekt kollidieren kann. Bestimmt man zuerst den Abstand des Objekts zur Grundebene durch $z = 0$, kann man alle Stützstellen ausschließen, deren Höhenwert unter dem minimalen Abstand des Objekts zur Ebene liegt. Für alle übrigen Stützstellen muss jetzt berechnet werden, ob das Objekt geschnitten wird. Man kann sich jede Stützstelle als Strecke vorstellen, die auf der Grundebene startet und in Richtung der z-Achse verläuft. Man kann daher für die Schnitttests auf die in Kapitel 6.4.1 beschriebenen Funktionen zurückgreifen. Findet eine Kollision statt, verläuft die Normale in Richtung der für die Stützstelle zu Beginn berechneten Normale.

Anhang A

Robotermodelle

In diesem Kapitel werden alle verfügbaren Robotermodelle vorgestellt. Die Struktur aller Modelle ist baumförmig aufgebaut wie in den Abbildungen A.2 - A.6 zu sehen ist. Hier kann man ablesen, aus welchen Teilen die Modelle aufgebaut sind. Ovale Knoten stehen hierbei für Objekte mit physikalischer Ausdehnung, die mit der Umwelt interagieren können, rechteckige Knoten haben nur eine „unsichtbare“ Funktion wie z.B. die Translation zum nächsten Knoten. Da für die Performance der Simulation besonders die Anzahl der physikalischen Objekte und der Gelenke von Interesse ist, werden diese in Tabelle A.1 für alle Modelle zusammengefasst.

Monstertruck
Der Monstertruck (siehe Abbildung A.1(a)) basiert auf der allgemeinen Klasse der Ackermanngelenkten Fahrzeuge. Dies sind Fahrzeuge, die wie ein normales Auto gesteuert werden: die Vorderräder sind lenkbar, alle vier Räder sind angetrieben. Das Grundmodell wurde um eine Kamera und einen Laserscanner erweitert. Insgesamt umfasst das Modell 7 physikalische Objekte, die mit der Umwelt interagieren können und 10 Gelenke, siehe auch Abbildung A.2.

Stretch-Limousine
Die Stretch-Limousine (siehe Abbildung A.1(b)) ist eine mögliche Visualisierung des Grundmodells der Ackermann-gelenkten Fahrzeuge, die Ansteuerung erfolgt also auch wie bei einem normalen Auto. Da es keine Sensoren oder andere Ergänzungen gibt, sieht man im Baum (vgl. Abbildung A.3) auf der rechten Seite einen leeren Knoten, an dem es möglich ist, Erweiterungen anzubringen (wie z.B. beim Monstertruck die Kamera und den Laserscanner). Das Modell enthält 7 physikalische Objekte und 8 Gelenke.

Pioneer
Der Pioneer (siehe Abbildung A.1(c)) gehört zur Klasse der differentialgetriebenen Fahrzeuge, das bedeutet, er hat zwei angetriebene Räder und ein passives Stützrad. Die Räder sind nicht lenkbar, Kurvenfahrten entstehen, indem ein Rad schneller rotiert als das andere. Er hat außer dem Grundkörper und den 3 Rädern an der Vorderseite einen Greifer und eine Kamera, wie in Abbildung A.4 in der rechten Hälfte des Baumes zu sehen ist. Insgesamt hat der Pioneer 8 physikalische Objekte und 8 Gelenke.

Volksbot
Der Volksbot hat auf jeder Seite 3 Räder, die jeweils gleich schnell rotieren. Die Ansteuerung funktioniert also ähnlich wie beim Pioneer. Das Modell des Volksbots (siehe Abbildung A.1(d)) besteht nur aus dem Grundkörper und 6 Rädern, insgesamt sind das also 7 physikalische Objekte und 6 Gelenke, siehe auch Abbildung A.5.

Humanoidroboter
Das Modell des Humanoidroboters HR18 (siehe Abbildung A.1(e))ist das komplizierteste. In Abbildung A.6 sieht man auf der linken Seite zunächst die Beschreibung der Brustkamera, in der Mitte die Beine und Arme und auf der rechten Seite den Kopf mit Kopfkamera. Alles zusammen sind dies 22 physikalische Objekte und 24 Gelenke.

Roboter	Objekte mit physikalischer Ausdehnung	Gelenke
Monstertruck	7	10
Stretch-Limousine	7	8
Pioneer	8	8
Volksbot	7	6
HR18	22	24

Tabelle A.1: *Übersicht: Anzahl physikalische Objekte und Gelenke der Robotermodelle*

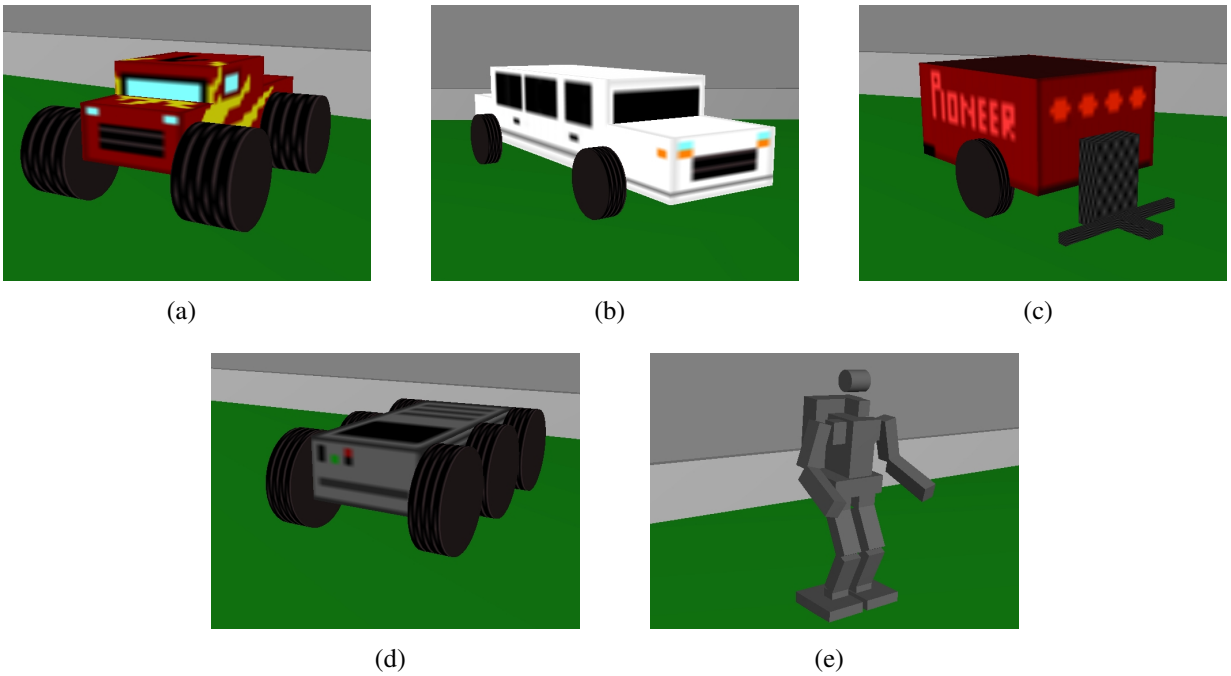


Abbildung A.1: *Die Visualisierung der Robotermodelle.*

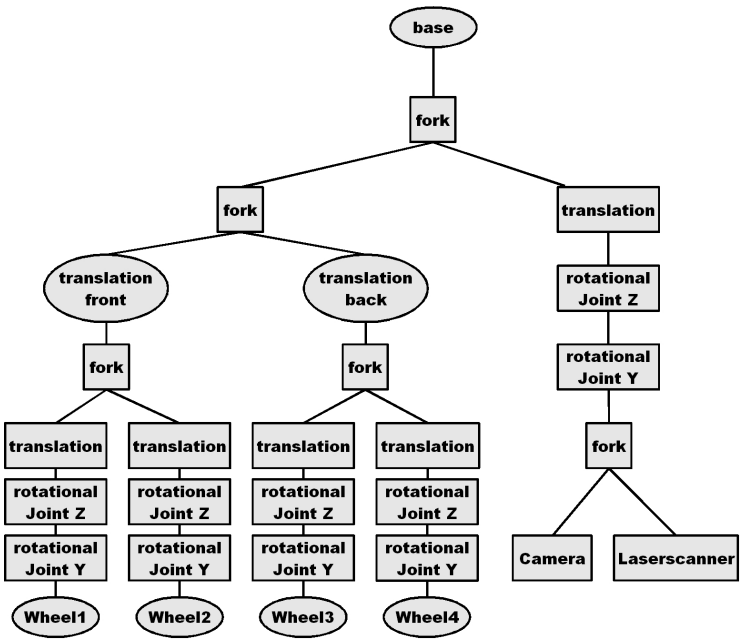


Abbildung A.2: *Modell des Monstertrucks.*

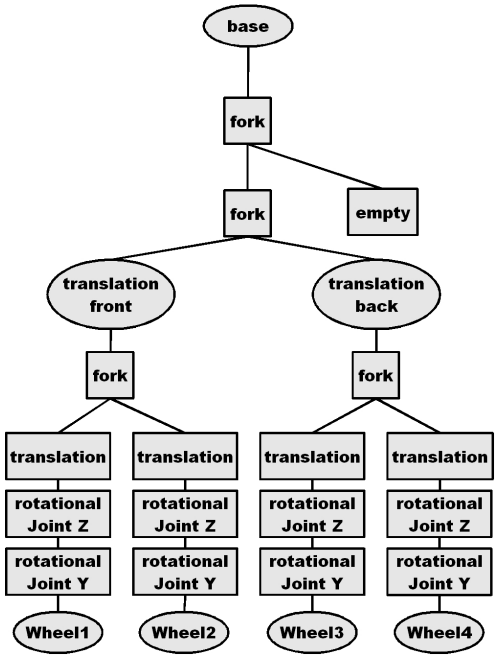


Abbildung A.3: *Modell der Stretch-Limousine.*

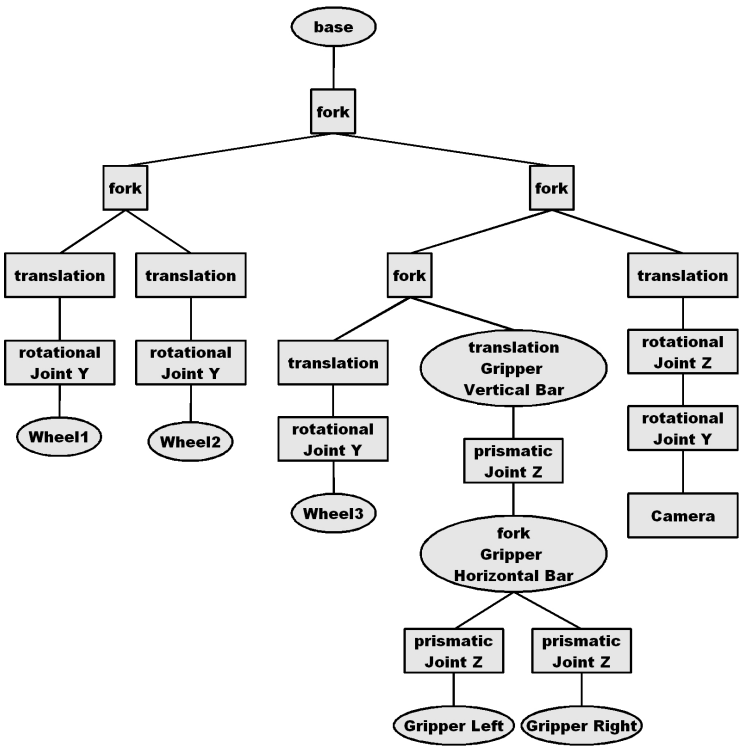


Abbildung A.4: *Modell des Pioneers.*

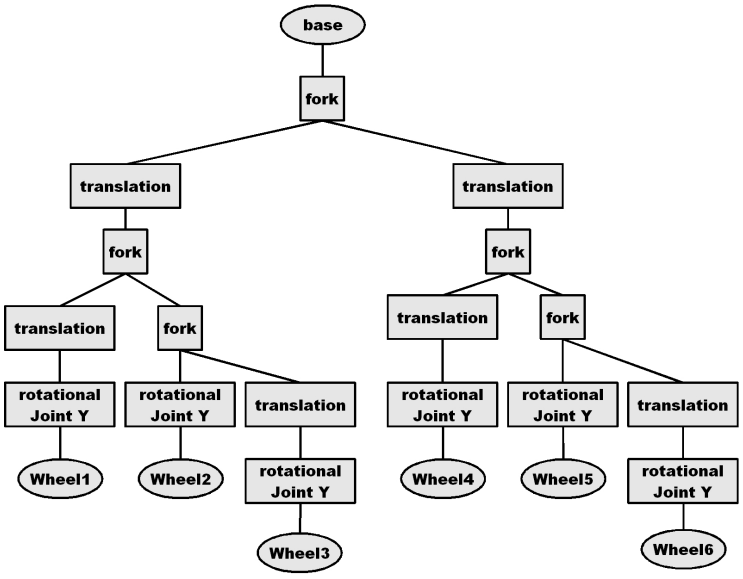


Abbildung A.5: *Modell des Volksbots.*

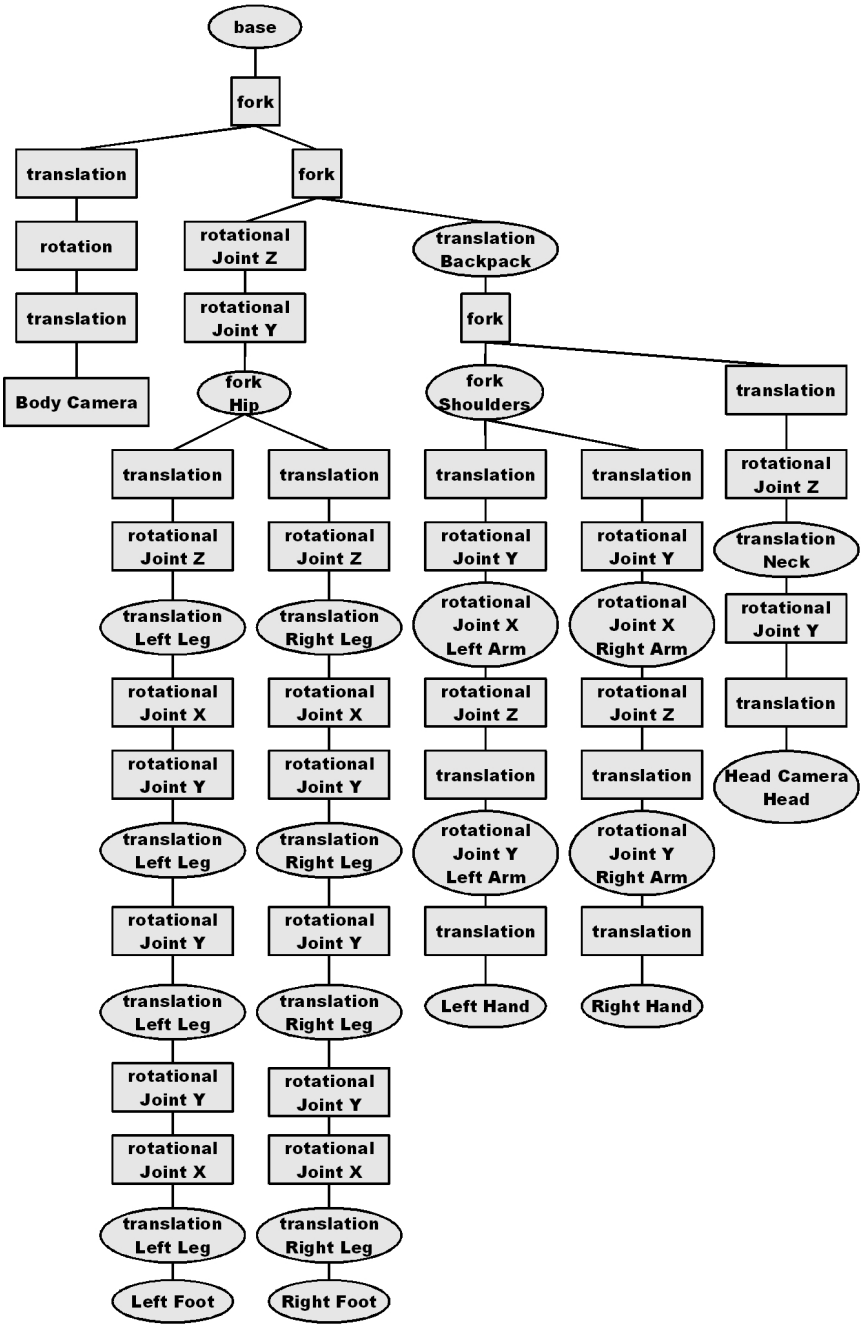


Abbildung A.6: *Modell des HR18.*

Anhang B

Konfigurieren der Applikation

Die Applikation `app_MonsterTruckSim.exe` kann über Dateien konfiguriert werden. Die meisten Einstellungen werden in der Datei `MonsterTruck.cfg` definiert. Hat diese Datei einen anderen Namen, muss dieser beim Starten der Anwendung übergeben werden. Ein Beispiel für eine Konfigurationsdatei ist in Listing B.1 zu sehen.

B.1 Die Konfigurationsdatei

Die Konfigurationsdatei ist in mehrere Bereiche gegliedert. Am Anfang jedes Bereichs steht der zugehörige Name in eckigen Klammern. Danach folgen in beliebiger Reihenfolge die zu diesem Bereich gehörenden Attribute. Alle Attribute sind optional, wenn sie nicht vorhanden sind, werden vordefinierte Standardwerte verwendet. Auf diese Art ist sichergestellt, dass auch alte Konfigurationsdateien weiterhin benutzt werden können, auch wenn neue Attribute hinzugekommen sind. Außerdem muss der Anwender die Attribute, bei denen er die Standardwerte verwenden möchte, nicht explizit eintragen. Kommentarzeilen werden mit einem Semikolon eingeleitet.
Alle Längenangaben sind in Millimetern, alle Winkelangaben in Grad.

Der Abschnitt `[general]`

monsterTrucks Gibt an, wie viele Monstertrucks simuliert werden sollen. Standardmäßig wird ein Monstertruck simuliert.

pioneers Gibt an, wie viele Pioneers simuliert werden sollen. Der Standardwert ist 0.

volksbots Gibt an, wie viele Volksbots simuliert werden sollen. Der Standardwert ist 0.

stretch Gibt an, wie viele Stretch-Limousinen simuliert werden sollen. Der Standardwert ist 0.

humanoids Gibt an, wie viele Humanoidroboter simuliert werden sollen. Der Standardwert ist auch hier 0.

showCollisions Definiert, ob erkannte Kollisionen eingezeichnet werden sollen (showCollisions $= 1$) oder nicht (showCollisions $= 0$). Der Standardwert ist 0.

fullCollision Legt fest, ob alle Teile der Roboter auf Kollisionen überprüft werden sollen (fullCollision $= 1$), oder nur die Teile, die Kontakt mit dem Boden haben, also die Räder von Fahrzeugen bzw. die Füße der Humanoidroboter. Dieser Wert wird für jeden Roboter benutzt, der diesen Wert nicht explizit im entsprechenden Abschnitt definiert bekommt. Der Standardwert ist 0.

lsReach Definiert die Reichweite aller in der Szene verwendeten Laserscanner. Der Standardwert ist 20000.

hierarchy Wählt aus, welcher `CollisionDetectionTask` verwendet wird (hierarchy $= 0$ wählt den alten, hierarchy $= 1$ die Hüllkugelhierarchie aus). Der Standardwert ist 0.

fpsCam Definiert einen Standardwert, wie viele Bilder pro Sekunde die in der Szene verwendeten Kameras liefern soll. fpsCam $= 0$ schaltet die Kameras aus. Dieser Wert kann auch für jeden Roboter individuell gesetzt werden. Der Standardwert ist 5.

fpsLaser Definiert einen Standardwert, wie viele Bilder pro Sekunde die verwendeten Laserscanner liefern sollen. fpsLaser $= 0$ schaltet die Laserscanner aus. Dieser Wert kann auch für jeden Roboter individuell gesetzt werden. Der Standardwert ist 5.

showLaserScans Legt fest, ob es möglich sein soll, eine Visualisierung der Laserscans zu zeichnen. Der Standardwert ist 0.

Der Abschnitt `[dynamics]`

gx, gy, gz Gibt an, in welche Richtung die Schwerkraft wirken soll und wie stark. Die „normale“ Schwerkraft wird angegeben als gx = 0, gy = 0, gz = -9810, dies ist auch der voreingestellte Wert.

rate Setzt die Framerate, mit der die Dynamiksimulation laufen soll. Der Standardwert ist 1000.

Der Abschnitt `[camera]`

posX, posY, posZ Definiert die Position der Kamera. Standardmäßig ist posX $= -10000$, posY $= -4000$, posZ $= 5000$.

yaw Dreht die Kamera um die z-Achse. Der Standardwert ist $30°$.

pitch Neigt die Kamera um die y-Achse. Der Standardwert ist $30°$.

near, far Setzt die near- und far-clipping-Ebene der Kamera. Diese beiden Werte sollten für eine gute Performance möglichst dicht beieinander gewählt werden. Der Standardwert für die near-Plane ist 10, für die far-Plane 20000.

Der Abschnitt `[light]`

posX, posY, posZ Setzt die Position (in mm) der Lichtquelle. Standard ist posX $= 0$, posY $= 0$, posZ $= 2000$.

Die Abschnitte `[<Roboter><Zahl>]`
Für jeden Roboter, der simuliert werden soll, kann ein eigener Abschnitt angelegt werden. <Roboter> muss durch den jeweiligen Robotertypen (monsterTruck, pioneer, volksbot, stretch oder hr18) ersetzt werden, <Zahl> gibt an, der wievielte Roboter definiert wird. Für jeden Typ wird erneut bei 1 angefangen zu zählen. Sollen also 2 Monstertrucks und eine Stretch-Limousine simuliert werden, legt man die Abschnitte [monsterTruck1], [monsterTruck2] und [stretch1] an.

name Gibt dem Roboter einen Namen.

posX, posY Legt die Position des Roboters fest. Standard ist posX $= 0$ und posY $= 0$.

dir Dreht den Roboter um die z-Achse. Der Standardwert ist $0°$.

textureOn (Nicht für HR18) Legt fest, ob eine Textur verwendet werden soll oder nicht. Der Standardwert ist 0.

comport Legt den Comport für die serielle Kommunikation fest. Hierfür gibt es keinen Standardwert.

fullCollision Gibt an, ob der ganze Roboter in der Lage sein soll, mit seiner Umwelt zu kollidieren, oder nur die für den Bodenkontakt vorgesehenen Teile (also die Räder bzw. Füße). Der Standardwert ist der im Abschnitt `[general]` definierte Wert.

Für Fahrzeuge mit einer Kamera (Monstertruck und Pioneer) kommen noch weitere Attribute hinzu.

fpsCam Gibt die Framerate der Kamera an. fpsCam $= 0$ schaltet die Kamera aus. Der Standardwert ist der im Abschnitt `[general]` festgelegte Wert für Kameras.

camImageFile Gibt die Datei an, in der die Kamerabilder abgelegt werden. Der Standardwert ist eine leere Zeichenkette, sodass keine Bilder gespeichert werden.

Fahrzeuge mit Laserscanner (aktuell nur der Monstertruck) stellen weitere Attribute zur Verfügung.

fpsLS Gibt die Framerate des Laserscanners an. fpsLS $= 0$ schaltet den Laserscanner aus. Der Standardwert ist der im Abschnitt `[general]` festgelegte Wert für Laserscanner.

lsImageFile Gibt die Datei an, in der die gemessenen Werte des Scans abgelegt werden. Der Standardwert ist eine leere Zeichenkette, sodass nichts gespeichert wird.

lsType Legt den Typ des Laserscanners fest. lsType $= 1$ liest den z-Buffer aus, lsType $= 2$ schneidet Strahlen gegen die Objekte der Szene. Der Standardwert ist 1.

raysDraw Gibt an, wie viele Strahlen pro Reihe in der Visualisierung des Laserscans gezeichnet werden sollen. Dieser Wert hat keine Auswirkung, wenn diese Funktionalität im Abschnitt [general] deaktiviert wurde. Der Standardwert ist 5.

lsWriteBack (Nur für lsType $= 1$) Legt fest, ob eine Graustufen-Visualisierung des Laser-Scans angezeigt werden soll (lsWriteBack $= 1$) oder nicht (lsWriteBack $= 0$). Dies ist standardmäßig ausgeschaltet.

lsResFactor (Nur für lsType $= 2$) Gibt an, durch welche Zahl die Höhe und Breite der Auflösung des Laserscanners dividiert werden soll. Je größer dieser Wert ist, umso besser ist die Performance, allerdings wird auch die Auflösung des Scans immer schlechter. Der Standardwert ist 5.

Die Humanoidroboter haben zwei Kameras, die daher anders eingestellt werden. Außerdem kommen noch weitere Roboter-spezifischen Einstellungen hinzu.

fpsHeadCam Gibt die Framerate der Kopfkamera an. fpsHeadCam $= 0$ schaltet die Kamera aus. Der Standardwert ist der im Abschnitt `[general]` festgelegte Wert für Kameras.

headImageFile Gibt die Datei an, in der die Bilder der Kopfkamera abgelegt werden. Der Standardwert ist eine leere Zeichenkette, sodass keine Bilder gespeichert werden.

enableHeadCamBlur Schaltet Unschärfe ein ($= 1$) oder aus ($= 0$). Der Standardwert ist 1.

fpsBodyCam Gibt die Framerate der Bauchkamera an. fpsBodyCam $= 0$ schaltet die Kamera aus. Der Standardwert ist der im Abschnitt `[general]` festgelegte Wert für Kameras.

bodyImageFile Gibt die Datei an, in der die Bilder der Bauchkamera abgelegt werden. Der Standardwert ist eine leere Zeichenkette, sodass keine Bilder gespeichert werden.

enableBodyCamBlur Schaltet Unschärfe ein oder aus. Der Standardwert ist 0.

enableBodyCamDistortion Schaltet die Verzerrung ein oder aus. Der Standardwert ist 0.

type Wählt aus, welche Version des HR18 simuliert werden soll (im Moment gibt es die Typen 1 bis 5). Der Standardwert ist 4.

dynamicsOn Legt fest, ob ein Dynamikalgorithmus verwendet werden soll ($= 1$), oder ob die Kinematiksimulation verwendet werden soll ($= 0$). Der Standardwert ist 1.

dynamicsType (Nur mit dynamicsOn $= 1$) Legt fest, welcher Dynamikalgorithmus (1 wählt den neueren, 0 den älteren) für die Bewegungssimulation verwendet werden soll. Der Standardwert ist 1.

rff (Nur mit dynamicsOn $= 1$) Der Rotational Friction Factor ändert die Reibung der Füße auf dem Boden. Wird rff klein gewählt, fällt der Roboter schneller um. Der Standardwert ist 50.

Der Abschnitt `[field]`

length, width Gibt die Größe des Feldes an. Der Standardwert ist $3000 \times 6000mm$.

city Gibt die Datei an, aus der die Stadt ausgelesen werden soll. Standardmäßig wird keine Stadt geladen.

balls Gibt die Datei an, in der einzelne Bälle für die Szene definiert werden. Standardmäßig werden keine Bälle eingefügt.

boards Legt fest, ob um das Feld Banden sein sollen (boards $= 1$) oder nicht (boards $= 0$). Der Standardwert ist 0.

middle Legt fest, ob zusätzlich zu den äußeren Banden noch eine weitere Trennung mitten durch das Feld laufen soll. Der Standardwert ist 0.

Im Abschnitt `[field]` können zwei weitere Dateien angegeben werden, in denen die Umwelt definiert wird. In der city-Datei wird eine Stadt, bestehend aus unbeweglichen Quadern, definiert, in der balls-Datei können bewegliche Kugeln in der Szene platziert werden. Diese beiden Dateien werden in den folgenden Absätzen beschrieben.

B.2 Die city-Datei

In der city-Datei werden statische Quader definiert, die eine Stadt bilden. Ein Beispiel für eine solche Datei ist in Listing B.2 zu sehen.
Mit den ersten 5 Zahlen der Datei wird die Größe der Stadt und die Größe der Häuser festgelegt. Zunächst wird definiert, über welchen Bereich des Feldes sich die Stadt erstrecken soll. Hierfür wird jeweils in x- und y-Richtung ein minimaler und ein maximaler Wert angegeben. Dann wird für die Grundfläche der Häuser die Kantenlänge definiert. Die Grundfläche der Häuser ist immer quadratisch.
Alle weiteren Zahlen definieren Häuser. Über den vorgegebenen Bereich der Stadt wird ein Gitter gelegt, jede Zelle ist so groß wie die Grundfläche eines Hauses. Die Zellen werden zeilenweise durchlaufen, für jede Zelle wird eine Zahl ausgelesen. Der Wert der Zahl bestimmt hierbei die Anzahl der Stockwerke des Hauses. Ein Stockwerk ist immer $0,1m$ hoch. Wird eine 0 ausgelesen, bleibt die Zelle leer.

B.3 Die balls-Datei

In der balls-Datei kann eine beliebige Anzahl beweglicher Kugeln in der Szene platziert werden. Ein Beispiel für eine solche Datei ist in Listing B.3 zu sehen.
Die erste Zahl legt fest, wie viele Kugeln eingefügt werden sollen. Danach muss jede Kugel mit 8 Zahlen beschrieben werden. Die ersten 3 Zahlen geben die Position der Kugel in Weltkoordinaten in mm an. Als nächstes folgt der Radius der Kugel. Die nächsten 3 Werte bestimmen die Farbe der Kugel (RGB), zum Schluss folgt die Masse der Kugel.

B.4 Beispiel-Dateien zum konfigurieren der Szene

```
[general]
;setup some robots
monsterTrucks = 2
pioneers = 1
volksbots = 1
stretch = 1
;draw collisions or not
showCollisions = 0
;enable full collision of all robots
fullCollision = 1
;reach of the laserscanners
lsReach = 4000
;use hierarchy for collision detection or the older stuff
hierarchy = 1
;define default-values for cameras and laser range finders
fpsCam = 5
fpsLaser = 4
;enable a visualisation of laserscans
showLaserScans = 1

[dynamics]
;setup dynamics
gz = -9000
rate = 500

[camera]
;define position and rotation of the interactive camera
posX = -10400
posY = -4400
posZ = 5000
yaw = 30
pitch = 30

[light]
;set the position of the light
```

```
posX = -2000
posY = 4000
posZ = 3000

[monsterTruck1]
;setup the first monsterTruck
name = "MonsterTruck1"
posX = -2000
posY = 2000
dir = 290
comport = Com4:
;show textures or not
textureOn = 1
;setup the sensors
fpsCam = 0
;select the type of the laserscanner
lsType = 1
lsImageFile = "LaserscannerMT1"
fpsLS = 5
;setup visualisation of the laserscan
lsWriteBack = 1

[monsterTruck2]
;setup the second monsterTruck
name = "MonsterTruck2"
posX = -500
posY = 3000
;show textures or not
textureOn = 0
;setup the sensors
fpsCam = 5
camImageFile = "CameraMT2"
;select the type of the laserscanner
lsType = 2
fpsLS = 5
;setup visualisation of the laserscan
raysDraw = 0

[pioneer1]
;setup a pioneer
name = "Pioneer1"
posX = -2000
posY = 1000
dir = 80
;show textures or not
textureOn = 0

[volksbot1]
;setup a volksbot
name = "Volksbot"
posX = -1000
posY = -3000
```

```
dir = 30
comport = Com11:

[stretch1]
;setup a stretched limousine
name = "StretchLimo"
posX = -3000
posY = -3000
textureOn = 1

[field]
;size of the field
length = 10000
width = 10000
;define files with city and balls
city = "city.txt"
balls = "balls.txt"
;should there be boards around the field?
boards = 0
```

Listing B.1: *MonsterTruck.cfg*

```
0 5000 -5000 5000 500
0 0 0 5 0 0 0 0 0 0
0 0 0 8 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
1 3 1 0 0 3 4 5 0 0
0 0 1 0 0 0 0 0 0 0
0 0 3 0 0 0 0 0 0 0
0 0 0 0 0 4 6 6 3 5
0 0 2 0 0 0 0 0 0 0
0 0 2 0 0 0 0 0 0 0
0 0 2 0 0 2 5 0 0 0
0 0 0 0 0 0 3 0 0 0
0 0 0 0 0 0 0 0 4 2
0 0 0 0 0 0 0 0 0 0
0 0 3 4 6 0 0 0 0 0
0 0 0 0 0 0 0 3 0 0
0 0 0 0 0 0 0 5 0 0
```

Listing B.2: *city.txt*

```
3
-1000 -1200 1200 100 1 0 0 0.07
-3100 4000 500 50 0 1 0 0.08
-2800 2700 700 68 0 0 1 0.06
-2500 -2800 800 112 1 1 0 0.04
-3900 2000 1500 120 0 1 1 0.05
```

Listing B.3: *balls.txt*

Literaturverzeichnis

[1] AGEIA PhysX Engine. www.ageia.com/, 2007.

[2] I. N. Bronstein, K. A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. Harri Deutsch, Thun und Frankfurt am Main, 2001.

[3] B. Browning and E. Tryzelaar. Übersim: a multi-robot simulator for robot soccer. In *Proceedings of the International Foundation for Autonomous Agents and Multiagent Systems (AAMAS)*, pages 948–949, Melbourne, Australia, 2003. ACM.

[4] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper. USARSim: a robot simulator for research and education. In *Proceedings of the IEEE 2007 International Conference on Robotics and Automation*, pages 1400–1405, 2007.

[5] M. Dür. *Skriptum zur Vorlesung Einführung in die Optimierung, gehalten im WS 2004/05 an der TU Darmstadt*. 2004.

[6] Epic games. Karma, http://udn.epicgames.com/two/karmareference.html, 2007.

[7] Epic games. Unreal engine, www.epicgames.com, 2007.

[8] C. Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[9] M. Friedmann, J. Kiener, S. Petters, D. Thomas, and O. von Stryk. Darmstadt Dribblers: Team Description for Humanoid KidSize League of RoboCup 2007. Technical report, Technisch Universität Darmstadt, 2007. 11 pages.

[10] M. Friedmann, K. Petersen, and O. von Stryk. Tailored real-time simulation for teams of humanoid robots. In *RoboCup Symposium 2007*, page to appear, Atlanta, GA, USA, July 9-10 2007. Springer-Verlag.

[11] B. P. Gerkey, R. T. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, pages 317–323, Coimbra, Portugal, June 2003.

[12] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996.

[13] J. Kiener. *Heterogene Teams kooperierender autonomer Roboter (Heterogeneous Teams of Cooperating Robots)*. PhD thesis, Technische Universität Darmstadt, December 15 2006.

[14] J. Kiener and O. von Stryk. Cooperation of heterogeneous, autonomous robots: A case study of humanoid and wheeled robots. In *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, page to appear, San Diega, CA, USA, Oct. 29 - Nov. 2 2007.

[15] N. Koenig and A. Howard. Design and use paradigms for gazebo, and open-source multi-robot simulator. In *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2149–2154, Sendai, Japan, September 2004.

[16] T. Laue, K. Spiess, and T. Röfer. Simrobot - a general physical robot simulator and its application in robocup. In A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi, editors, *RoboCup 2005: Robot Soccer World Cup IX*, number 4020 in Lecture Notes in Artificial Intelligence, pages 173–183. Springer, 2006.

[17] O. Michel. Webots: Professional mobile robot simulation. *Journal of Advanced Robotics Systems*, 1(1):39–42, 2004.

[18] Microsoft Robotics Studio. msdn.microsoft.com/robotics/, 2007.

[19] O. Obst and M. Rollmann. Spark – A Generic Simulator for Physical Multiagent Simulations. *Computer Systems Science and Engineering*, 20(5), Sept. 2005.

[20] T. Röfer, J. Brose, D. Göhring, M. Jüngel, T. Laue, and M. Risler. Germin Team 2007: The Germin Natonal RoboCup Team. Technical report, 2007. 10 pages.

[21] R. Smith. Ode - open dynamics engine, www.ode.org, 2007.

[22] Volksbots. www.volksbot.de, 2007.

[23] O. von Stryk. *Skriptum zur Vorlesung Robotik 1, gehalten im WS 2006/07 an der TU Darmstadt*. 2006.

[24] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes in Computer Science*, pages 359–370. Springer, 1991.

[25] J. C. Zagal and J. R. del Solar. UCHILSIM: A dynamically and visually realistic simulator for the robocup four legged league. In *RoboCup 2004: Robot Soccer World Cup VIII*, pages 34–45. Springer, 2004.

[26] M. Zaratti, M. Fratarcangeli, and L. Iocchi. A 3d simulator of multiple legged robots based on USARSim. In G. Lakemeyer, E. Sklar, D. G. Sorrenti, and T. Takahashi, editors, *RoboCup 2006: Robot Soccer World Cup X*, number 4434 in Lecture Notes in Artificial Intelligence. Springer, 2007.