Steuerung von autonomen Vehikeln in einer Echtzeit-3D-Umgebung

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Informatiker

an der
Fachhochschule für Technik und Wirtschaft Berlin
Fachbereich Wirtschaftswissenschaften II
Studiengang Angewandte Informatik

1. Betreuer: Prof. Dr. Thomas Jung
2. Betreuer: Philipp Schellbach

Eingereicht von Stephan Ziep
Berlin, 10. 11. 2004

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Erstellung dieser Diplomarbeit unterstützt haben. Mein besonderer Dank für die Betreuung dieser Arbeit gilt Prof. Dr. Thomas Jung, der stets ein offenes Ohr für mich hatte und mir wertvolle Ratschläge gab.

Danken möchte ich auch Eva und meiner Freundin Jana, die viele Stunden mit der Korrektur dieser Arbeit verbracht haben. Weiterhin danke ich Philipp Schellbach, Jörg Wagner und André Dittrich für die vielen kritischen Anmerkungen und Verbesserungsvorschläge.

Inhaltsverzeichnis

1. Vorwort

Diese Diplomarbeit beschäftigt sich mit der Steuerung von autonomen Vehikeln in einer Echtzeit-3D-Umgebung. Dazu werden Steuerungsverhalten eingeführt, die es dem Vehikel erlauben, in natürlich wirkender Weise durch eine 3D Umgebung zu navigieren, Hindernissen selbständig auszuweichen und in kombinierter Form komplexeres Gruppenverhalten zu imitieren. Hauptaugenmerk liegt hierbei auf der Beschreibung und Implementation dieser Steuerungsverhalten sowie die Beschreibung von Schnittstellen zur unterliegenden Fahrphysik und der darüber liegenden Wegplanung bzw. Zielfindung. Dieses Steuerungssystem wird in die Echtzeit 3D-Engine der Firma YAGER Development integriert.

Die vorliegende Arbeit wurde in enger Zusammenarbeit mit der Firma YAGER Development GmbH entwickelt. YAGER Development ist ein junges, innovatives Unternehmen auf dem Gebiet der Entertainment-Software. Es beschäftigt sich vornehmlich mit der Entwicklung und Produktion von Unterhaltungssoftware, insbesondere mit der Entwicklung hochwertiger Technologien und der entsprechenden Inhalte für die Bereiche der Vollpreis- und Budget Computerspiele.
YAGER-Development wurde im Jahr 1999 von den fünf Gesellschaftern Philipp Schellbach, Timo Ullman, Uwe Beneke, Roman Golka und Matthias Wiese gegründet und beschäftigt heute 17 Mitarbeiter.
Das erste Spiel der Firma wurde im vergangenen Jahr in mehreren Ländern veröffentlicht. In Anlehnung an den Firmennamen lautete der Titel „Yager“ und kam für PC und Microsoft XBOX™ in den Handel. Seitdem arbeitet die Firma an einem neuen Projekt, befindet sich derzeit im Gespräch mit unterschiedlichen Publishern und ist mit der Entwicklung von mehreren Prototypen beschäftigt.

2. Einführung

Computerspiele werden immer komplexer und aufwändiger. Es ist heute nicht mehr ungewöhnlich hunderte von Einheiten gleichzeitig auf dem Bildschirm darzustellen (Bsp: Herr der Ringe: Die Schlacht um Mittelerde). Diese Massen manuell zu bewegen, sei es durch vorgefertigte Splines oder vorprogrammierte Bewegungen, wäre unwirtschaftlich, zu statisch und vorhersehbar. Änderungen am Bewegungsablauf wären nur mit einigem Aufwand möglich. Ein Weg den Entwicklungsaufwand zu reduzieren, besteht darin, den Verantwortungsbereich der Bewegung an die einzelnen Einheiten abzugeben. Die Einheiten werden nicht länger bewegt, sie bewegen sich nun selbst. Ein Ziel ist das einzige was sie benötigen. Auf diese Art verliert der Entwickler zwar einen Teil der Kontrolle über die einzelnen Einheiten, gewinnt dadurch aber Dynamik, Flexibilität und Bewegungen, die beim wiederholten Betrachten unterschiedlich ablaufen.
Ein System, das die Ziele der Einheiten, die Art und Weise sowie das Aussehen ihrer Bewegung verwaltet, wird von einem Großteil der Computerspiele benötigt. Ein solches Steuerungssystem bildet die Grundlage, auf der die Künstliche Intelligenz (KI) aufbaut.

Das Einsatzgebiet des Steuerungssystems, das in dieser Arbeit vorgestellt wird, bilden 1st / 3rd Person–Action Spiele. In der Regel beinhalten diese Spiele nicht nur Bewegung zu Fuß, sondern auch wie in neueren Spielen üblich (*Halo*, *Battlefield 1942*), um ein vielfaches schnellere Bewegung mit den unterschiedlichsten Vehikeln zu Land und Luft. Schnelle Bewegung bedeutet auch, dass die Vehikel schneller fahren als wenden können, weshalb die Steuerungsverhalten die zukünftigen Konsequenzen ihres Handelns immer berücksichtigen müssen.

Eine weitere Herausforderung an ein robustes Steuerungssystem ist die Spielphysik. Sie übernimmt eine wichtige Rolle in der Interaktion des Spielers mit der Spielwelt. Dort definiert die Physik Regeln, die denen der Realität ähnlich sind, so dass sich der Spieler schnell in der Welt zurechtfindet, sich bewegen kann und mit dieser Welt in Interaktion treten kann. Der Spieler kann so im Voraus abschätzen, wie sich ein Gegenstand oder Vehikel in der Spielwelt verhält. Um diesen Eindruck von Realismus zu unterstützen, müssen Vehikel über physikalische Eigenschaften wie Masse und Bremskraft verfügen. Somit muss auch das Steuerungssystem in der Lage sein, gewisse Situationen einschätzen zu können und entsprechend zu handeln.

Hinzu kommt, dass bei Computerspielen die zur Verfügung stehenden Ressourcen wie Speicher und Prozessorleistung sehr eingeschränkt sein können. Das trifft vor allem auf Spielkonsolen wie Microsoft XBOX™, Sony Playstation 2™ etc. zu. Aus diesem Grund müssen Algorithmen und Systeme effizient und „billig“ gestaltet sein. Dabei kommt es weniger auf die Exaktheit

von Algorithmen an. Auf dem Gebiet der KI für Computerspiele interessiert vielmehr, ob etwas glaubhaft aussieht und den Spieler davon überzeugt, dass ein Computergegner eine komplexe Einheit mit eigener Intelligenz sei. "*As long as the player has the illusion that a computer-controlled character is doing something intelligent, it doesn't matter what AI (if any) was actually implemented to achieve that illusion*" [1]

Im Allgemeinen werden Steuerungssysteme von den wenigsten Spielern direkt wahrgenommen. Wenn Sie auffallen, dann meist negativ. Einem Spieler, der von einem Monster gejagt wird, wird es mit Sicherheit auffallen, wenn es an einem Hindernis steckenbleibt. Stellt man jedoch einem akkuraten Steuerungsalgorithmus, einen Algorithmus gegenüber, der das Verhalten nur näherungsweise aber effizient simuliert, wird der Spieler den Unterschied höchstwahrscheinlich nicht bemerken. Aus diesem Grund versuchen KI-Programmierer bei Steuerungssystemen Ressourcen zu sparen, um diese den Komponenten der KI zur Verfügung zu stellen, die für den Spieler sichtbarer sind (Taktik, Strategie, Perönlichkeit, etc.).

2.1 Aufgabenstellung

Die Aufgabe ist, ein Steuerungssystem zu planen und implementieren, das die drei eingangs erwähnten Kriterien erfüllt: die Unterstützung von Vehikeln mit sehr unterschiedlichen Geschwindigkeiten, die Einbeziehung der Physik in die Berechnung der Steuerungskräfte sowie der Einsatz Ressourcen schonender Algorithmen. Dieses Steuerungssystem soll in die YAGER-Engine, eine Eigenentwicklung der Firma YAGER Development GmbH, implementiert werden. Eine nähere Beschreibung der Engine findet im Kapitel 4 *Analyse* statt.

Das Steuerungssystem für die YAGER-Engine soll Panzer, Fußtruppen, PKW und LKW gleichermaßen bewegen können. Dabei sollen diese Vehikel selbständig dynamischen Objekten ausweichen und Pfaden folgen. Weiterhin soll sichergestellt werden, dass sich die Vehikel stets innerhalb gewisser Grenzen (Korridore) bewegen, die vom jeweiligen Leveldesigner definiert werden. Vehikel müssen in der Lage sein, sowohl von Computergegnern als auch von Spielern gesteuert zu werden. Autonome Bewegung einzelner Vehikel aber auch koordinierte Gruppenbewegungen, Einheitenformationen, taktische Manöver und dergleichen sollen unterstützt werden. Eine vollständige Anforderungsdefinition liefert das Kapitel 5 *Design*.

1 [LIDEN03] S. 41

2.2 Überblick

Diese Arbeit ist nach den einzelnen Phasen der Systementwicklung gegliedert, wobei anfangs abstrakt gehaltene Aussagen immer weiter konkretisiert werden. Nachdem nun die Aufgabenstellung formuliert worden ist, gibt das folgende Kapitel einen Einblick in die Grundlagen der Vehikelsteuerung. Dort wird der Grundaufbau eines Steuerungssystems für autonome Vehikel vorgestellt. Das Kapitel 4 *Analyse* stellt die YAGER-Engine vor, für die dieses Steuerungssystem entworfen und implementiert werden soll. Hier werden die Gegebenheiten näher beleuchtet, unter denen das zukünftige Steuerungssystem arbeiten soll. Das Kapitel 5 *Design* beschäftigt sich mit der Modellierung der Systems. Hier erfolgt eine Anforderungsdefinition, aus der im weiteren Verlauf die einzelnen Bestandteile des Systems als Klassen und Beziehungen erkannt und modelliert werden. Das Kapitel 6 befasst sich mit Fragen der Implementierung des Steuerungssystems. Die einzelnen zugrundeliegenden Algorithmen werden hier ausführlich erklärt und mit Abbildungen und Quellcodebeispielen verdeutlicht. Im letzten Kapitel erfolgt schließlich eine Auswertung und die Erläuterung der aufgetretenen Probleme.

2.3 Begriffsdefinition

Wichtige Begriffe für diesen Themenkreis werden genau definiert, um Verständnisproblemen vorzubeugen und ein einheitliches Vokabular zu schaffen.

Autonomer Agent, Vehikel, KI, Charakter, NPC

Unter diesen Begriffen versteht man die rechnerinterne Repräsentation eines sich bewegenden, seine Umwelt wahrnehmenden, künstlichen Lebewesens, das seine Umwelt beeinflussen kann und bis zu einem gewissen Grad autonom handelt. Ein *Agent* wird nicht durch eine zentrale Instanz gesteuert.

Der Begriff *Vehikel* bezieht sich im Text auf die „Verkörperung“ des Agenten und wird insbesondere bei der Darstellung von Implementationen gebraucht. Vehikel meint hier in der Regel jede Art von Fortbewegungsmittel, sei es Fahrzeug, ein Schiff, ein Flugzeug oder einen Charakter. Der Begriff *KI* bezieht sich meist auf den „Geist“ bzw. die planerische Komponente eines Agenten.

Autonome Agenten werden häufig in Computerspielen und interaktiven Medien eingesetzt und besitzen die Fähigkeit, ihr Verhalten zu improvisieren. In Form von Lebewesen werden sie auch häufig Non-Player-Characters (NPC) genannt. Dem entgegengesetzt steht ein Charakter in einem Animiationsfilm, dessen Aktionen im Allgemeinen vordefiniert sind.

Frame

Zeitabschnitt der eine komplette Engine Aktualisierung, inklusive Rendering, KI und Physik darstellt. Ein *Frame* entspricht einem Zustand zu einem bestimmten Zeitpunkt. Die Geschwindigkeit von Computerspielen wird in Frames pro Sekunde (fps) gemessen.

2.4 Vereinbarungen

Koordinatensystem

In dieser Arbeit wird ein linkshändiges Koordinatensystem verwendet. Die x-Achse zeigt nach rechts, die y-Achse ist nach oben gerichtet ist und die z-Achse zeigt nach vorn.

Steuerungsverhalten

Die Bezeichnungen der einzelnen Steuerungsverhalten werden nicht übersetzt, da die englischen Begriffe inzwischen Teil eines einheitlichen Vokabulars im Themengebiet Vehikelsteuerung sind.

2.5 Hinweise zum Lesen

Zum besseren Verständnis und zur Strukturierung werden in dieser Arbeit verschiedene Absatzformate und Schriftarten genutzt, deren Bedeutung hier erklärt wird.

Wichtige Begriffe

Wichtige Begriffe werden bei ihrem ersten Vorkommen im Text *kursiv* dargestellt.

Quelltext

Der Verwendete Quelltext in dieser Arbeit ist vereinfacht worden, um sich auf die Beschreibung der Algorithmen zu konzentrieren. Er ist in C++ Syntax gehalten und wird in der Schriftart `Courier` dargestellt. Variablen, Konstanten, Kommentare, Prozedur- und Funktionsbezeichnungen sind durchgängig in Englisch gehalten.

```
void PrintTitle(void)
{
     cout << „Steuerung von autonomen Vehikeln\n“;
     cout << „in einer Echtzeit-3D-Umgebung.\n“;
}
```

3. Grundlagen

In diesem Kapitel werden die Grundlagen der Steuerung von Vehikeln mithilfe von Steuerungsverhalten vorgestellt. Es dient der Vermittlung eines ersten Einblicks in dieses Themengebiet und zeigt dem Leser den Aufbau und die Funktionsweise eines Steuerungssystems für Vehikel.
Zunächst wird die Geschichte der Steuerungsverhalten beschrieben und Entwickler sowie deren Arbeiten vorgestellt (Kapitel 3.1). Anschließend werden die Komponenten eines Steuerungssystems eingeführt (Kapitel 3.2), um in den folgenden Unterkapiteln erläutert zu werden (Kapitel 3.3 - 3.5). Kernstück dieses Kapitels bildet der Abschnitt 3.4 *Steuerungsebene,* in dem die meist verwendeten Steuerungsverhalten vorgestellt werden.

3.1 Entwicklung

Das Konzept der Steuerungsverhalten geht auf das „Boid-Modell“ von Craig Reynolds aus dem Jahre 1987 zurück. Dieses Computermodell basiert auf einzelnen autonomen Agenten, deren Summe an Einzelbewegungen das komplexe Verhalten großer Schwärme nachahmt. Die einzelnen simulierten Kreaturen werden hierbei als „Boid“ bezeichnet. Der Begriff bedeutet Birdoid, was so viel heisst wie: „vogelähnliches Wesen“. Reynolds erkannte, dass sich das komplexe und unvorhersehbare Bewegungsmuster von großen Schwärmen auf drei einzelne Verhaltensweisen reduzieren lässt: Separation (Verteilung), Cohesion (Zusammenhalt) und Alignment (Ausrichtung).
Die Verhaltensweise der Separation bewirkt, dass die einzelnen Agenten immer einen definierten Mindestabstand zueinander halten. Dagegen bewirkt Cohesion, dass der Schwarm zusammenbleibt. Alignment bestimmt die Ausrichtung und Geschwindigkeit der einzelnen Agenten. Die Agenten entscheiden autonom und nur aufgrund lokaler Informationen. Durch Kombination dieser drei Verhaltensweisen gelang es Reynolds, größere Schwärme von Tieren wie Fische und Vögel am Rechner zu simulieren.

Ein erstes Beispiel für den Einsatz von Steuerungsverhalten ist der Kurzfilm „Stanley and Stella in: Breaking the Ice“, der 1987 im Electronic Theater der SIGGRAPH[2] uraufgeführt wurde. Die Ergebnisse dieser Arbeit präsentierte Reynolds in seinen Arbeiten „Flocks, Herds and Schools – A Distributed Behavioural Model“ [REYNOLDS87] und „Not Bumping Into Things“ [REYNOLDS88]. In Letzterem geht er näher auf die Vermeidung von Hindernissen ein.

2 Jährliche Konferenz zum Thema Computergrafik der Special Interest Group on Graphics, Teilgruppe der Association for Computing Machinery (ACM)

Abbildung 1: Szene aus "Stanley and Stella in: Breaking the Ice" 1987

Auf der Games Developer Conference 1999 stellte Reynolds dann eine Erweiterung seiner Arbeit unter dem Titel „Steering Behaviours for Autonomous Characters“ [REYNOLDS99] vor. Hierin werden Verhalten eingeführt, die eine bestimmte Reaktion des Vehikels auf seine Umgebung definieren. Durch Kombination dieser „Bausteine“ lassen sich komplexere Verhaltensweisen erstellen wie z.B. „Fahre von Punkt A zu Punkt B, ohne gegen Hindernisse zu stossen“. Diese als „Steering Behaviours“ bezeichneten Verhaltensweisen bilden die Schnittstelle zwischen der eigentlichen Fortbewegung und der Planungsebene (Strategie) in einem autonomen System. Das folgende Kapitel beschäftigt sich näher damit.

Abbildung 2: Screenshot aus "Dungeon Keeper 2", Bullfrog Productions Ltd.

Eine erste erfolgreiche Implementation in ein Computerspiel stellte Robin Green ein Jahr später auf der SIGGRAPH 2000 vor. Seine Arbeit unter dem Titel „Steering Behaviours“ [GREEN00] präsentiert Codebeispiele der Implementation einzelner Verhaltensweisen in der Sprache C++ und stellt weitere mögliche Probleme sowie deren Lösungen vor. Die Arbeit entstand bei der Entwicklung des Computerspiels „Dungeon Keeper 2“ und baut direkt auf Reynolds Arbeiten auf. Die im Spiel auftretenden Charaktere werden durch Steuerungsverhalten animiert und zeigen, welcher Grad an zusätzlichem Realismus mit dieser Methode erreicht werden kann.

3.2 Aufbau

Craig Reynolds unterteilte den Aufbau des Bewegungsverhaltens eines autonomen Agenten in drei Ebenen: *Planungsebene*, *Steuerungsebene* und *Fortbewegungsebene*. Die Planungsebene repräsentiert hier die Aufgaben oder Ziele des autonomen Agenten. Strategische Entscheidungen, Wegfindung und Zielauswahl finden in dieser Ebene statt. Hier ist die klassische KI angesiedelt.

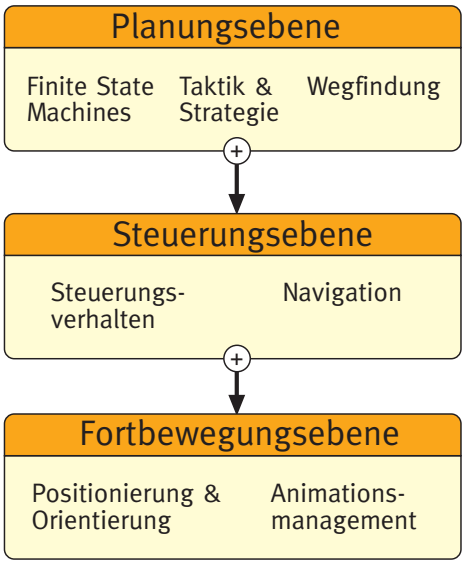


Abbildung 3: Aufbau des Bewegungsverhaltens eines autonomen Agenten

Die Steuerungsebene zerlegt die abstrakte Zielvorgabe in einzelne, konkrete Unterziele. Hierbei ist ein Unterziel mit einem Steuerungsverhalten gleichzusetzen. Die aus den einzelnen Steuerungsverhalten resultierenden Steuerungssignale werden an die Fortbewegungsebene gesendet, wo die Richtungssignale in Bewegung umgesetzt werden. Die Fortbewegungsebene wird durch das Vehikel dargestellt.

Dieser Aufbau lässt sich an einem Beispiel veranschaulichen:
Ein Gruppe Cowboys bewacht eine Kuhherde. Plötzlich entfernt sich ein junges Kalb von der Herde. Der Anführer gibt einem der Cowboys den Befehl, er solle das Kalb wieder einfangen. Der Cowboy reitet mit seinem Pferd zum Kalb, unter Vermeidung etwaiger Zusammenstöße mit stacheligen Kakteen und Steinen und führt das Kalb wieder zurück.

Der Anführer repräsentiert die Planungsebene. Er setzt das Ziel für den Cowboy. Der Cowboy verkörpert die Steuerungsebene. Er zerlegt das Ziel in kleinere Unterziele, wie 'reite zum Kalb', 'vermeide Kakteen', 'hole das Kalb'. Der Cowboy steuert sein Pferd mit Hilfe von Steuerungssignalen (links, rechts, langsamer etc.) zum Ziel. Das Pferd stellt die Fortbewegungsebene dar. Es nimmt die Steuerungssignale des Cowboys und wandelt diese in Bewegung um.
(Vgl: [REYNOLDS99])

Die einzelnen Schichten des Systems sind austauschbar. So kann der Cowboy mit den unterschiedlichsten Motivationen in Bewegung gesetzt werden ('geh zur Bar und bestelle Bier'), die Steuerungsart kann sich ändern (wankend wegen Trunkenheit) und auch die Art der Fortbewegung ist unabhängig (zu Fuss). Ob der Cowboy mit dem Pferd, mit einem Motorrad oder mit

einem Auto unterwegs ist, spielt eine verhältnismäßig geringe Rolle für die oberen Schichten. Allerdings muss das Steuerungsverhalten die unterschiedliche Manövrierfähigkeit und Agilität der einzelnen Vehikel berücksichtigen. Beispielsweise hat ein Pferd einen geringeren Wendekreis als ein PKW.

Im Folgenden wird die Bedeutung und Funktionsweise jeder Ebene betrachtet. Diese Arbeit legt den Schwerpunkt auf die Steuerungsebene und wird die angrenzenden Ebenen sowie deren Schnittstellen beschreiben.

3.3 Planungsebene

Um scheinbar intelligentes Verhalten zu erzielen, wird mehr als intelligente Bewegung benötigt wie sie durch Steuerungsverhalten erreicht wird. Ein autonomer Agent benötigt Ziele und Gründe um sich an einen Ort zu bewegen und einen Weg dorthin zu finden. Eine übergeordnete KI muss aufgrund dieser Ziele entscheiden, welche Orte der Agent aufsuchen bzw. vermeiden und welche Aktionen er dort ausführen soll. (Vgl: [ALT&KING03])
Diese übergeordnete KI ist die Planungsebene. Zu ihren Aufgaben gehört die Entscheidungsfindung, die Routenberechnung durch das Wegenetz, das Umschalten bzw. Anpassen der Steuerungsverhalten an neue Umgebungssituationen, das Wählen von Taktiken und Angriffs- bzw. Verteidungsstrategien sowie das Setzen von Zielobjekten, die bekämpft, verteidigt oder bewacht werden müssen.

Die Planungsebene kann ihre Informationen bzw. Befehle auch von übergeordneten KI-Ebenen empfangen. Beispielsweise von der globalen KI, die Ereignisse auf Weltebene steuert oder einer so genannten *Squad-Level KI* (Gruppen-KI), die das Verhalten von einzelnen Agenten als Gruppe steuert. Eine Squad-Level KI ist für die taktische Positionierung, Manöver und Formation der Gruppenmitglieder verantwortlich. Diese Abstrahierung der KI in unterschiedliche Verantwortungsbereiche kann noch weiter geführt werden, um komplexe Ziele strategischer Art handhabbar zu machen. Die Idee stammt ursprünglich aus dem militärischen Bereich, wo seit jeher Gruppen von Einheiten zu größeren Verbänden zusammengefasst wurden.

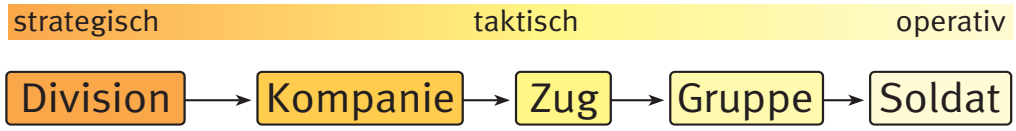


Abbildung 4: Hirarchische Untergliederung der Einheiten beim Militär

Höher gelegene Ebenen arbeiten mit abstrakteren Daten, treffen strategische Entscheidungen und behandeln größere Formationen als eine Einheit. Sie überlassen die praktische Umsetzung den jeweils untergeordneten Ebenen, deren Entscheidungen eher taktischer und operativer Natur sind.

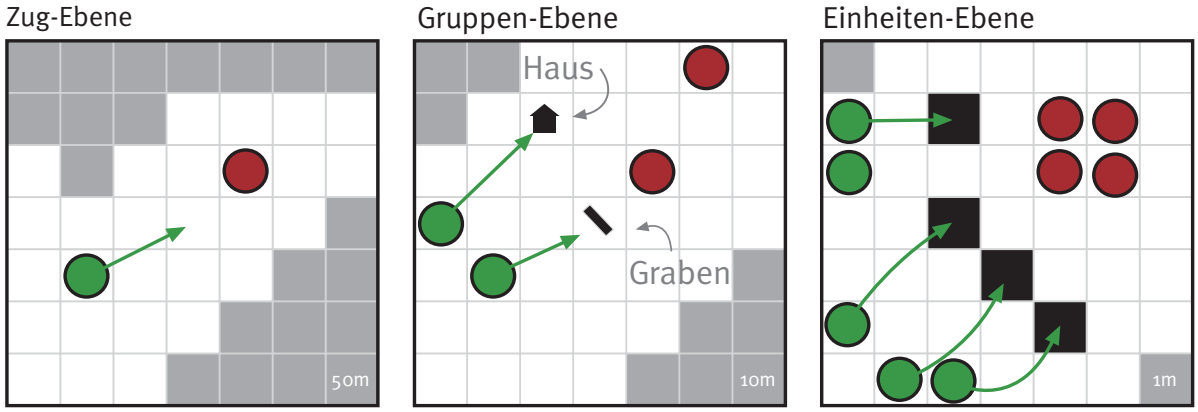


Abbildung 5: Unterschiedliche Sichten und Auflösungen hierarchisch gegliederter KI

Das Beispiel in Abbildung 5 zeigt einen Zug, der den Befehl erhält, sich gegen einen Feind zu bewegen. Auf der Zug-Ebene wird dieser Befehl für die Gruppen-Ebene übersetzt. Hier sollen die beiden Gruppen, aus denen der Zug besteht, im Haus und im Graben Deckung suchen. Die Gruppen-Ebene übersetzt den Befehl für die Einheiten-Ebene in konkrete Einheitenpositionen. Die Einheiten-Ebene ist nun dafür zuständig, dass der einzelne Soldat seine Position erreicht und in Deckung geht. Auf diese Weise ist es gelungen, ein komplexeres Problem in handhabbare Teilprobleme zu zerlegen. Weitere Informationen zum Thema Multi-Tiered KI sind bei [RAMSEY03] und [KENT03] zu finden.

Im Folgenden wird eine der gebräuchlichsten Techniken bei der Umsetzung von Entscheidungsfindungsprozessen in Computerspielen vorgestellt.

Finite State Machines

Eine Finite State Machine (FSM) ist eine nicht-lineare Beschreibung, wie ein Objekt seinen Status, aufgrund von Ereignissen seiner Umgebung oder von sich heraus, mit der Zeit verändert. Sie besteht aus Zuständen und Transitionen zwischen diesen Zuständen. Einen Zustand kann man sich als Verhalten oder Tätigkeit eines Objektes vorstellen. Er beinhaltet meist den Code bzw. die Im-

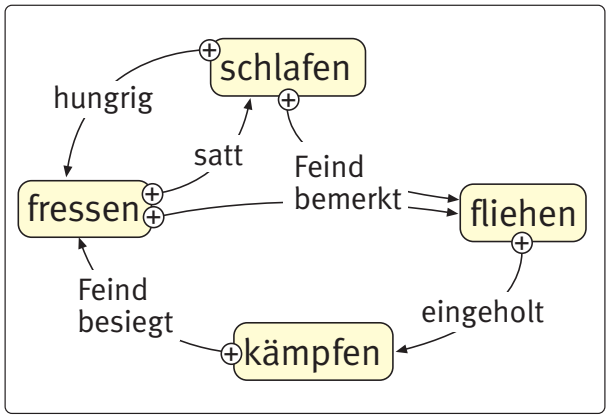


Abbildung 6: vereinfachtes Beispiel einer Finite State Machine

plementation des gewünschten Objekt-Verhaltens. Eine Transition ist eine Bedingung, um von einem Zustand in einen anderen zu wechseln. Aufgrund ihrer intuitiven Erlernbarkeit sind FSMs ein häufig gebrauchtes und mächtiges Werkzeug, um Entscheidungsfindung zu modellieren und zu implementieren.

State Machines ab einer bestimmten Größe tendieren dazu, unübersichtlich zu werden. Um auch bei mehreren Dutzend Zuständen und Transitionen den Überblick zu behalten, ist es sinnvoll die Zustände so zu definieren, dass auch sie wieder komplette unabhängige State Machines sein können. Diese Art FSM wird *Cascaded FSM* genannt. Im Beispiel der Abbildung 6 könnte der Zustand „fressen“ eine eigene FSM darstellen, die wiederum aus den Zuständen „Nahrung suchen“, „Nahrung zubereiten“ und „Nahrung zu sich nehmen“ besteht. Weitere Informationen über FSMs sind bei [FU&HOU03] zu finden.

3.4 Steuerungsebene

Die Steuerungsebene stellt im Bewegungssystem eines autonomen Agenten den Navigator bzw. Lotsen dar. Dabei sind zwei Ansätze grundsätzlich zu unterscheiden. Zum einen gibt es Steuerungsalgorithmen für Langstrecken, beispielsweise den A* Wegfindungs-Algorithmus, der sich durch einen planerischen Ansatz auszeichnet. Zum anderen gibt es Steuerung für Kurzstrecken, deren Merkmal es ist, die Steuerungsimpulse allein durch lokale Informationen zu generieren. Der erste Ansatz ist eher statischer Natur und für die Wegfindung in großen Netzen zuständig, wohingegen die lokale Steuerung auf dynamische Veränderungen der Umwelt reagiert. [TOMLINSON04] Diese Ebene beschäftigt sich ausschliesslich mit Kurzstreckensteuerung.

Die Steuerungsebene empfängt die Zielvorgaben der Planungsebene, nimmt die Umwelt durch verschiedene Sensoren wahr und stellt sicher, dass der Agent sein Ziel erreicht. Die Zielvorgaben beinhalten neben den konkreten Zielpunkten, ob und wie sich der Agent zu seinem Ziel bewegen soll bzw. welches Steuerungsverhalten angewendet werden soll. Durch diese Unterteilung der Aufgabenbereiche wird sichergestellt, dass sich höher gelegene KI-Ebenen nicht mit Fragen der Bewegungsumsetzung auseinander setzen müssen, während die unteren Ebenen nicht das „Warum“ der Bewegung kennen müssen. [ALT&KING03]

Die Steuerungsebene hat mehrere Aufgaben. Zuerst werden die Kräfte aller verwendeten Steuerungsverhalten berechnet. Das Ergebnis sind mehrere Steuerungskräfte, die sich unter Umständen auch widersprechen, d.h. in unterschiedliche Richtungen zeigen können. Diese Kräfte müssen nun zu einer resultierenden Steuerungskraft kombiniert werden. Die errechnete Steuerungskraft wird an die Fortbewegungsebene weitergereicht.

Im Folgenden werden einige Steuerungsverhalten in ihrer Wirkung vorgestellt und unterschiedliche Kombinationsmöglichkeiten der resultierenden Steuerungskräfte diskutiert.

3.4.1 Steuerungsverhalten

Mit Steuerungsverhalten kann man Bewegung erzeugen, die gleichmäßig, natürlich und intelligent wirkt. Sie sind die Bausteine, aus denen sich komplexe Verhaltensweisen erstellen lassen, wie z.B sich zu einem Ziel A zu bewegen, während man immer einen gewissen Mindestabstand zu Ziel B einhält, unter Vermeidung von Zusammenstössen mit Hindernissen. Sie sind rein reaktive Module, besitzen also keine interne Repräsentation der Spielwelt und kommen ohne das Zwischenspeichern von Welt-Zuständen aus. Ihre Reaktionen basieren auf dem Zustand der Spielwelt, wie er durch Ihre Sensorik immer wieder neu überprüft wird.[3]

Steuerungsverhalten können in verschiedene Kategorien unterteilt werden. Zum einen gibt es Verhalten, die nur auf Gruppen von Agenten angewendet werden können wie z.B *Alignment*. Zum anderen gibt es Verhalten wie z.B *Seek,* die nur für Individuen zuständig sind.
Doch es lassen sich auch andere Kategorien anwenden. Verhalten wie „Seek“ und „Pathfollow“ generieren Bewegungen, wohingegen „Obstacle Avoidance“ und „Separation“ vorhandene Bewegungen nur beeinflussen können.

Seek & Flee

Seek und Flee sind Verhalten, die entgegengesetzt wirken. Seek steuert ein Vehikel auf ein Ziel zu, während Flee auf dem kürzesten Weg davon weg steuert. Die Berechnung ist für beide Verhalten gleich, es wird lediglich die berechnete Richtung (Steuerungsrichtung) invertiert. Ein Beispiel für diese Verhalten wäre ein Raubtier, das direkt auf seine Beute zusteuert bzw. die Beute, die vor diesem flieht. Das Ziel kann eine feste Position oder ein sich bewegendes Objekt sein. Ist das Ziel ein bewegliches Objekt, so wird die zukünftige Position dieses Objektes basierend auf der aktuellen Geschwindigkeit und Richtung des Ziels geschätzt und als Zielpunkt verwendet.

Arrival

Das Arrival-Verhalten ist eine Erweiterung des *Seek*-Verhaltens. Es ist ebenfalls in der Lage bewegte und statische Ziele zu verfolgen, die Art des Ankommens unterscheidet sich jedoch stark vom *Seek*-Verhalten. Hier wird das Vehikel kontrolliert abgebremst, sobald es sich innerhalb eines bestimmten Radius um den Zielpunkt bewegt, um genau auf ihm zum Stehen zu kommen.

3 Vgl. Reaktives Verhalten [DARBY03] S.472

Pathfollowing

In diesem Verhalten verbinden sich die beiden Steuerungsansätze: Kurz- und Langstrecke. Hier wird ein durch Langstrecken-Steuerung vorberechneter Pfad eingesetzt, dem das Pathfollowing-Verhalten zu folgen hat. Es wird davon ausgegangen, dass der Pfad ungeglättet vorliegt, d.h. die einzelnen Pfadpunkte sind nicht durch Splines oder Bezierkurven miteinander verbunden, sondern durch normale Linien. Pfadverfolgung ist eines der wichtigsten Verhalten, da es häufig eingesetzt wird und die Wahl des Pfades ein gutes Mittel ist, das Verhalten des Vehikels zu kontrollieren. Das Vehikel muss das Ende des Pfades unter allen Umständen erreichen. Dabei kommt es darauf an, dies möglichst realitätsnah zu tun, und nicht nur stur auf dem Pfad entlang zu fahren. Unnatürliches Verhalten wie abrupte Wendungen oder häufige Orientierungsänderungen gilt es zu vermeiden. Dazu müssen die Bewegungen der Vehikel geglättet werden, was die Aufgabe dieses Verhaltens ist.

Wander

Um Vehikeln die Möglichkeit zu geben, selbständig und ohne vordefiniertes Ziel durch ihre Umgebung zu navigieren oder ihren gerichteten Bewegungen ein gewisses Maß an Zufälligkeit hinzuzufügen, verwendet man das Wander-Verhalten. Die Bewegungen eines Vehikels wirken so wirklichkeitsnäher, da sie an kleinere Schlenker erinnern, die in der Realität beispielsweise durch Wind oder Bodenunebenheiten hervorgerufen werden.

Obstacle Avoidance

Ein Vehikel erhält den Anschein von Eigenintelligenz, sofern sein Verhalten den Erwartungen des Betrachters entspricht, d.h. wenn es sich in natürlicher Art und Weise in seiner Umgebung bewegt. Dazu gehört, dass mögliche Schwierigkeiten vorausschauend erkannt werden, um dagegen zu steuern bevor das Hindernis zum Problem wird. Jedes Lebewesen vermeidet instinktiv Kollisionen, auch wenn diese noch weit entfernt sind. Die Distanz, ab der ein Hindernis als Gefahr eingestuft wird, steigt proportional mit der Geschwindigkeit, mit der sich das Vehikel bewegt. Es werden meist nur Objekte als Kollisionsquellen betrachtet, die im eigenen Weg liegen. Objekte, die sich seitlich des Vehikels oder dahinter befinden, werden nicht berücksichtigt.

Containment

Dieses Verhalten wird verwendet um sicherzustellen, dass ein Vehikel sich stets innerhalb gewisser Grenzen befindet bzw. diese in keinem Fall überschreitet. Dabei wird ständig überprüft, ob das Vehikel einer Begrenzung zu nahe kommt und gegebenenfalls entgegen gesteuert. Dieses Verhalten ist vor allem an Orten wie Brücken und Abgründen sinnvoll, da es die Vorsicht des Fahrers ausdrücken kann. Doch auch auf normalen Wegen wird Containment eingesetzt. Wenn beispielsweise ein Panzer durch ein Dorf fährt, so ist absolut notwendig, dass er innerhalb der Fahrbahn bleibt und nicht durch Häuser hindurch „abkürzt“.

Die nachfolgenden drei Steuerungsverhalten befassen sich mit Vehikelgruppen. Hier bestimmen die einzelnen Verhalten, wie ein Vehikel auf andere Vehikel in seiner unmittelbaren Nachbarschaft reagiert.

Separation

Separation ist ein Verhalten, dass Vehikel daran hindert, ineinander zu stehen. Zudem veranlasst es sie, den zur Verfügung stehenden Platz besser und realistischer auszunutzen. Es zwingt Vehikel immer einen bestimmten Mindestabstand zueinander zu halten und ähnelt dem menschlichen Verhalten, wie es z.B in einer Fußgängerzone beobachtbar ist. Dort ist jeder Einzelne bestrebt, den Personen in seiner unmittelbaren Umgebung nicht zu nahe zu kommen. Sobald dieser Fall eintritt, weicht der Mensch intuitiv aus.
Dabei entscheidet der Sichtwinkel, ob der Fußgänger bzw. das Vehikel auch hinter ihm befindliche Objekte berücksichtigt. In manchen Situationen ist es hilfreich und natürlicher, wenn der Sichtwinkel derart begrenzt wird, dass das Vehikel nur auf Objekte reagiert, die vor ihm liegen. Für den Fußgänger ist das Geschehen hinter seinem Rücken irrelevant, da diese Informationen für seine Bewegungen nicht benötigt werden. Bei einer Schlangenbildung vor einem schmalen Durchgang wäre ein volles Sichtfeld sogar hinderlich. In diesem Fall würden sich alle Vehikel voneinander abstossen und ein Chaos herbeiführen. Ein eingeschränktes Sichtfeld ließe nur die nachfolgenden auf die vorderen Vehikel reagieren, die beispielsweise ihre Bewegung verlangsamen würden.

Cohesion

Cohesion ist das Gegenstück zu Separation und ist für den Zusammenhalt einer Gruppe zuständig. Es sorgt dafür, dass zusammengehörige Vehikel beieinander bleiben und eine erkennbare Gruppe bilden. Cohesion sollte allerdings nur in Verbindung mit Separation eingesetzt werden, da die Vehikel auf einen gemeinsamen Punkt zusteuern und sich sonst überschneiden könnten.

Alignment

Mitglieder einer Gruppe passen sich in Ausrichtung und Geschwindigkeit einander an. Das ist notwendig, um das Risiko von Zusammenstößen und Konflikten zu verringern. In der Natur ist diese Eigenschaft bei großen Herden und Schwärmen sehr gut beobachtbar. Die Gruppe agiert hier scheinbar als eine Einheit, die von einem zentralen Willen gesteuert wird. Die Gruppenbewegung ist jedoch das Resultat der individuellen Bewegungen ihrer Mitglieder. Um dieses Verhalten nachzuahmen, wird Alignment eingesetzt.

3.4.2 Kombination von Steuerungsverhalten

Jedes einzelne Steuerungsverhalten analysiert seine Umgebung nach unterschiedlichen Kriterien. Mit seiner internen Logik generiert es einen Richtungsvorschlag für den Agenten. Dieser Vorschlag ist die Steuerungskraft und besteht aus einem normalisierten 3D-Vektor. Jedes Verhalten besitzt eine Gewichtung, mit der die Steuerungskraft noch einmal gedämpft werden kann.

Es ist nun Aufgabe der Steuerungsebene, alle relevanten Daten zu sammeln und aus ihnen eine einzige Steuerungskraft zu ermitteln. Einige dieser Kräfte sind in diesem Frame essentiell für das Vehikel, einige sind nur kosmetischer Natur, einige sind groß und andere sind winzig. Einige Kräfte versuchen sich gegenseitig auszulöschen (z.B. Separation und Containment), einige bringen das Vehikel in einen ungültigen Zustand (z.B eingeschlossen in ein Hindernis) und wieder andere können für diesen Frame komplett vernachlässigt werden, da sie vom Verhalten momentan nicht benötigt werden (z.B. Flee ohne Gefahr).

Diesen unterschiedliche Kräfte müssen nun kombiniert werden, um eine finale Steuerungskraft zu generieren, die der gewünschten Verhaltensweise des Agenten am nächsten kommt.

Der einfachste Weg alle Steuerungskräfte zu kombinieren, ist die Bildung des Durchschnitts. Durch die bereits enthaltene Gewichtung der Verhalten ist dies ein *gewichteter Durchschnitt*. Der entscheidende Nachteil dieses Verfahrens, liegt jedoch in seiner Unentschiedenheit. In kritischen Situationen wie bevorstehenden Kollisionen, muss sofort reagiert und entgegen gesteuert werden. Wenn allerdings die Steuerungskräfte in entgegengesetzte Richtungen zeigen, löschen sie sich zu einem Großteil gegenseitig aus. Der Agent unternimmt nur geringe Richtungsänderungen und befindet sich weiterhin auf Kollisionskurs.

Ein weiteres Problem bei der Durschnittsbildung ist die bestehende Möglichkeit einer wahllosen Kombination von “Entweder-Oder“-Verhaltensweisen. Während es beispielsweise für ein Reh natürlich ist, *entweder* Gras zu fressen *oder* vor Wölfen zu fliehen, ist es offensichtlich wenig hilfreich, auf der auf der Flucht vor Wölfen etwas Gras zu fressen.

Um diese Probleme zu lösen, muss dem Steuerungsverhalten mitgeteilt werden, welche Priorität es in der Berechnung des Gesamtverhalten besitzt. Das geschieht durch die Reihenfolge, mit der die Verhalten zusammengestellt werden. Die in dieser Arbeit verwendete Methode der Priorisierung wird in Kapitel 6.4.2 *Kombination der Steuerungsverhalten* ausführlich beschrieben. Weitere Methoden für die Kombination von Steuerungsverhalten sind bei [REYNOLDS99] und [GREEN00] zu finden.

3.5 Fortbewegungsebene

Die Fortbewegungsebene ist die Verkörperung des autonomen Agenten. Hier werden die Steuerungssignale der Steuerungsebene in die Bewegung des „Körpers" des Agenten umgewandelt. Die Aufgabe dieser Ebene besteht darin, den Agenten letztendlich zu animieren. Verschiedene situationsbezogene Animationen wie Beinbewegung, Raddrehung etc. werden gestartet und die Position des Agenten wird verändert. Diese Bewegung unterliegt den Beschränkungen des zugrunde liegenden physikalischen Modells.

Reynolds und Greens Arbeiten liegt ein stark vereinfachtes und idealisiertes Fahrzeugmodell zugrunde. Dabei sollte eine Grundlage geschaffen werden, die auf einen Großteil der Fortbewegungsarten anwendbar ist, denn das Modell legt sich auf keine bestimmte Fortbewegung fest, sondern bildet den kleinsten gemeinsamen Nenner. Das Vehikel ist als Punkt mit einer gerichteten Geschwindigkeit definiert. Die Geschwindigkeit ermittelt sich aus der Steuerungskraft der Steuerungsebene, die zur aktuellen Geschwindigkeit addiert wird. Zur Position des Vehikels wird dann die Geschwindigkeit addiert, um die aktualisierte Position zu erhalten. Diese als Euler-Integration bekannte Methode ist die einfachste und ungenaueste Integrationsmethode, reicht jedoch vollkommen aus, um Bewegung zu erzeugen. Das lokale Koordinatensystem wird für jeden Frame neu an den Geschwindigkeitsvektor angepasst, wobei die resultierenden Kräfte geglättet werden, indem der Durchschnitt der letzten drei Steuerungskräfte gebildet wird. Dies dient der Vermeidung von Zittern und abrupten Richtungswechseln des Fahrzeugs bei entgegengesetzten Kräften.

4. Analyse

Dieses Kapitel dient der Beschreibung der Gegebenheiten, unter denen das zukünftige Steuerungssystem zum Einsatz kommt. In Kapitel 4.1 wird die YAGER-Engine kurz vorgestellt und die Arbeitsweise wichtiger Prinzipien und Teilsysteme wie Routen Management und Fahrphysik näher erklärt. Hier werden darüberhinaus der Aufbau und einige Werkzeuge zur Erstellung von Levels in der YAGER-Engine vorgestellt. Das anschliessende Kapitel 4.2 beschreibt die bisherigen Vorgehensweisen wenn es darum ging, Vehikel und andere Künstliche Intelligenzen zu steuern. Im letzten Kapitel 4.3 erfolgt eine Bewertung des bisherigen Steuerungssystems.

4.1 Aufbau der YAGER-Engine

Die Grundlage für die Umsetzung von Spielen und Prototypen ist eine selbstentwickelte Spiele-Engine, die in C++ geschrieben ist – die YAGER-Engine. Spiele-Engines können in zwei Kategorien unterteilt werden: Indoor und Outdoor. Während Indoor-Engines speziell für die Darstellung von Innenräumen ausgelegt sind, sind Outdoor-Engines für große Außenareale optimiert. Die Optimierung erfolgt durch den Einsatz unterschiedlicher Techniken, die das Rendering der Umgebung beschleunigen. Dabei hängt die Art des Rendering sehr stark von der Spielidee ab, die mit der Engine umgesetzt wird. Faktoren wie der Abstand zum Spielgeschehen oder die Übersicht in der Spielwelt (Frosch-/Vogelperspektive) spielen entscheidende Rollen bei der Wahl der eingesetzten Techniken. Treten häufig Verdeckungen der Spielwelt auf, was der Fall bei Indoor-Umgebungen ist, so wird man die Engine daraufhin optimieren, verdeckte Objekte nicht zu rendern (z.B. Occlusionculling). In Outdoor-Umgebungen treten seltener Verdeckungen auf. Dort ist es sinnvoller weit entfernte Objekte vereinfacht darzustellen, um das Rendering zu beschleunigen. Ein solches System wird Level-of-Detail-System (LOD) genannt.
Die YAGER-Engine ist eine Outdoor-Engine, die ihren Fokus bisher auf Flugspiele mit großer Sichtweite hatte. Nach Erscheinen des ersten Titels befindet sich die Engine jedoch in einer Umbau-, bzw. Erweiterungsphase zu einer Outdoor-Engine mit dem Fokus auf Fahrzeuge in 1st bzw. 3rd Person Ansicht. Die Kamera ist nun näher am Boden, daher stehen große Sichtweiten nicht mehr im Vordergrund. Vielmehr muss der Nahbereich detaillierter dargestellt werden.

Die YAGER-Engine besitzt eine Schnittstelle für die Scriptsprache TCL[4]. Damit kann die Engine nicht nur durch Scripts beeinflusst werden, sondern der gesamte Spielinhalt inklusive Objekten und Spiellogik wird durch Scripts festgelegt und gesteuert. Dadurch wird eine Trennung von Spiellogik und Programmcode geschaffen und die Grundlage für die Wiederverwendbarkeit der

4 Nähere Informationen zu TCL sind unter www.tcl.tk zu finden

Engine in anderen Projekten geschaffen. Die Scriptingtechnologie wird im Kapitel 4.1.3 „Scripting“ gesondert behandelt. Im Folgenden werden die Kernkomponenten der YAGER-Engine mit ihren Aufgaben vorgestellt.

4.1.1 Kernkomponenten

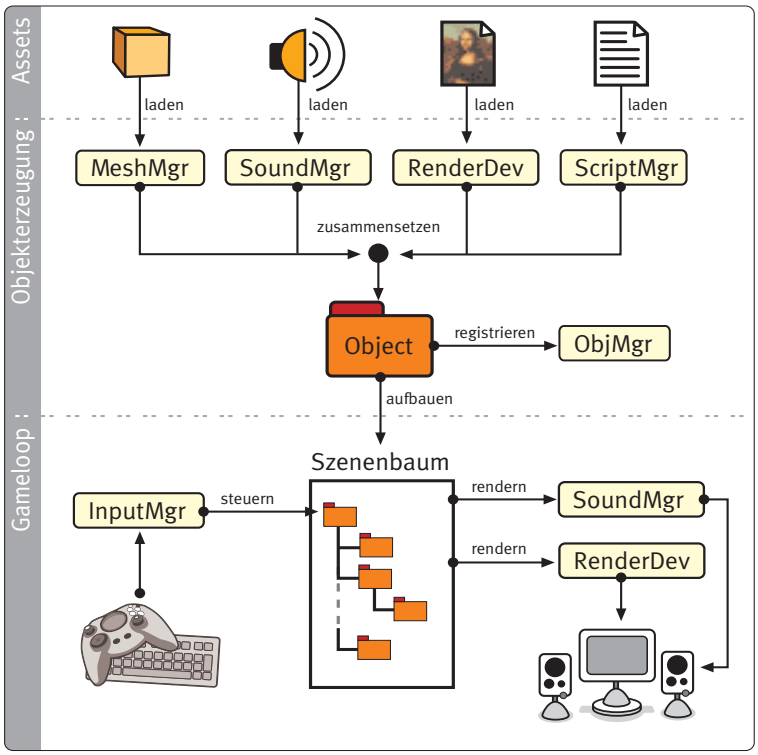


Abbildung 7: Aufbau der YAGER-Engine (stark vereinfacht)

Die YAGER-Engine besteht aus mehreren Komponenten mit unterschiedlichen Aufgabenbereichen – den *Managern*.

Der *MeshManager* ist für das Laden und die Verwaltung von 3D-Objektdaten zuständig. Der *SoundManager* lädt, verwaltet und spielt Sounds und Musik ab. Das *Renderdevice* wird vom MeshManager zum Laden der eingebetteten Objekttexturen verwendet und ist für die Darstellung (Rendering) der kompletten Szene zuständig. Der *ScriptManager* lädt Scripte beim Start der Engine ein und sorgt für die Interpretation dieser Scripte zur Laufzeit. Eingabegeräte wie Maus, Tastatur und XBOX-Controller™ senden ihre Daten an den InputManager, der wiederum ausgewählte Sceneobjekte steuert.

Kern der YAGER-Engine ist der *Szenenbaum*. Die *Assets*[5] werden zu Beginn von den entsprechenden Managern geladen und zu Szeneobjekten hierarchisch zusammengesetzt. Es ensteht der Szenenbaum. Diese Objekte registrieren sich beim *ObjektManager*, der dafür sorgt, dass durch den Szenenbaum navigiert und gesprungen werden kann. Um die Scene darzustellen, wird nun der Szenenbaum traversiert. Alle sichtbaren Objekte werden gerendert und jede hörbare Soundquelle wird abgespielt.

Um zu erklären, wie aus dieser bislang statischen Szene eine dynamische Spielwelt entsteht, muss der Szenenbaum eingehender beschrieben werden..

5 Sounddaten, Texturen, Scripte, 3D-Modelle, Animationen, Effekte etc.

4.1.2 Szenenbaum und Controller

Der Szenenbaum beinhaltet die gesamte Spielwelt mitsamt allen dazugehörigen Daten. Fahrzeuge, Charaktere, Effekte, Gebäude, Landschaft und Himmel sind Teil der Szene, die hierarchisch aufgebaut ist. Alle Szenenobjekte erben von der Klasse `CObj`. Sie stellt die grundlegende Funktionalität für die 3D-Objekte zur Verfügung wie Position, Rotation und Transformationen. Davon abgeleitet sind die unterschiedlichsten Klassen, die für den Aufbau eines Spieles benötigt werden. Angefangen bei Kameras (`CViewer`) mit denen die Scene betrachtet werden kann, über Lichter (`Clight`) und animierbare 3D-Objekte (`CBonesObj`), bis hin zu Effekten und Logikobjekten wie Trigger[6] (`CTrigger`). Alle Objekte, die von der Klasse `CObj` abgeleitet sind, sind beliebig hierarchisch kombinierbar. Ausnahmen bilden lediglich die Klassen `CLevelmap`, `CLevelgrass` und `CSky`, von denen es jeweils nur eine Instanz pro Szene geben kann.

Der Szenenbaum ist mit einer Verzeichnisstruktur vergleichbar, bei der man verschiedene Objekte über Pfade referenzieren kann. Beispielsweise bezeichnet der Pfad „`/gfx/scene/tank01/kette`“ das Objekt „`kette`“, welches ein Kind des Objektes „`tank01`“ ist, der in der Szene hängt. Die Szene selbst ist ein Unterzweig des Szenenbaumes. Es existieren mehrere Unterzweige, von denen allerdings nur „`/gfx/scene`“ die renderbaren Scenedaten beinhaltet. Unter dem Zweig „`/sys`“ befinden sich beispielsweise die Manager, unter „`/ctrl`“ befinden sich noch einmal alle Controller zur leichteren Aktualisierung und unter „`/anim`“ befinden sich alle geladenen Objekt-Animationen. Durch diese Vorgehensweise ist es möglich, die Manager und Daten durch Scripte anzusprechen und zur Laufzeit Operationen auf ihnen durchzuführen.

Szenenobjekte müssen steuerbar sein, um mit ihnen eine interaktive, dynamische Spielwelt gestalten zu können. Zu diesem Zweck wurden die *Controller* erschaffen. Controller werden nicht in den Szenenbaum integriert, stattdessen werden sie direkt an das zu steuernde Objekt gehängt. Die abstrakte Klasse `CcontrollableBase`, von der die Klasse `CObj` abgeleitet ist, stellt dafür die Schnittstelle zur Verfügung. Abbildung 8 auf der nächsten Seite verdeutlicht diese Vererbungshierarchie.

Controller werden dazu verwendet, bestimmte Eigenschaften wie Position, Rotation, Farbe, Textur oder Verhalten des zu steuernden Objektes beliebig zu manipulieren. Die Klasse `CCtrl` bildet die abstrakte Basisklasse für alle weiteren konkreten Controllerimplementationen. Einige Controller sind für das Abspielen von Animationen zuständig (`CBonesAnimCtrl`), andere

6 Objekt, das Ereignisse auslösen kann, sobald bestimmte Objekte mit ihm kollidieren bzw. bestimmte Bedingungen erfüllt sind.

verändern nur die Farbe von bestimmten Effekten (`CGlowCtrl`) oder die Beleuchtungsintensität von Lichtern (`ClightCtrl`). Wiederum andere steuern die Brennweite des aktiven Viewers (`CFovCtrl`). Um Daten der Eingabegeräte erhalten zu können, benötigt ein Objekt einen InputController, der die Daten vom InputManager empfängt. Ein Fahrzeug benötigt einen `CBasephysicsCtrl`, um von der Fahrphysik Gebrauch machen zu können.

Der Controller-Ansatz wurde nach dem bekannten Model-View-Controller Paradigma (MVC) entworfen. Der View wird durch das sichtbare 3D-Mesh dargestellt. Das Modell ist das Szenenobjekt mit all seinen Eigenschaften und der Controller steuert die Modelldaten. Die Änderung der Modelldaten wird nach aussen durch eine Veränderung des Views weitergegeben.

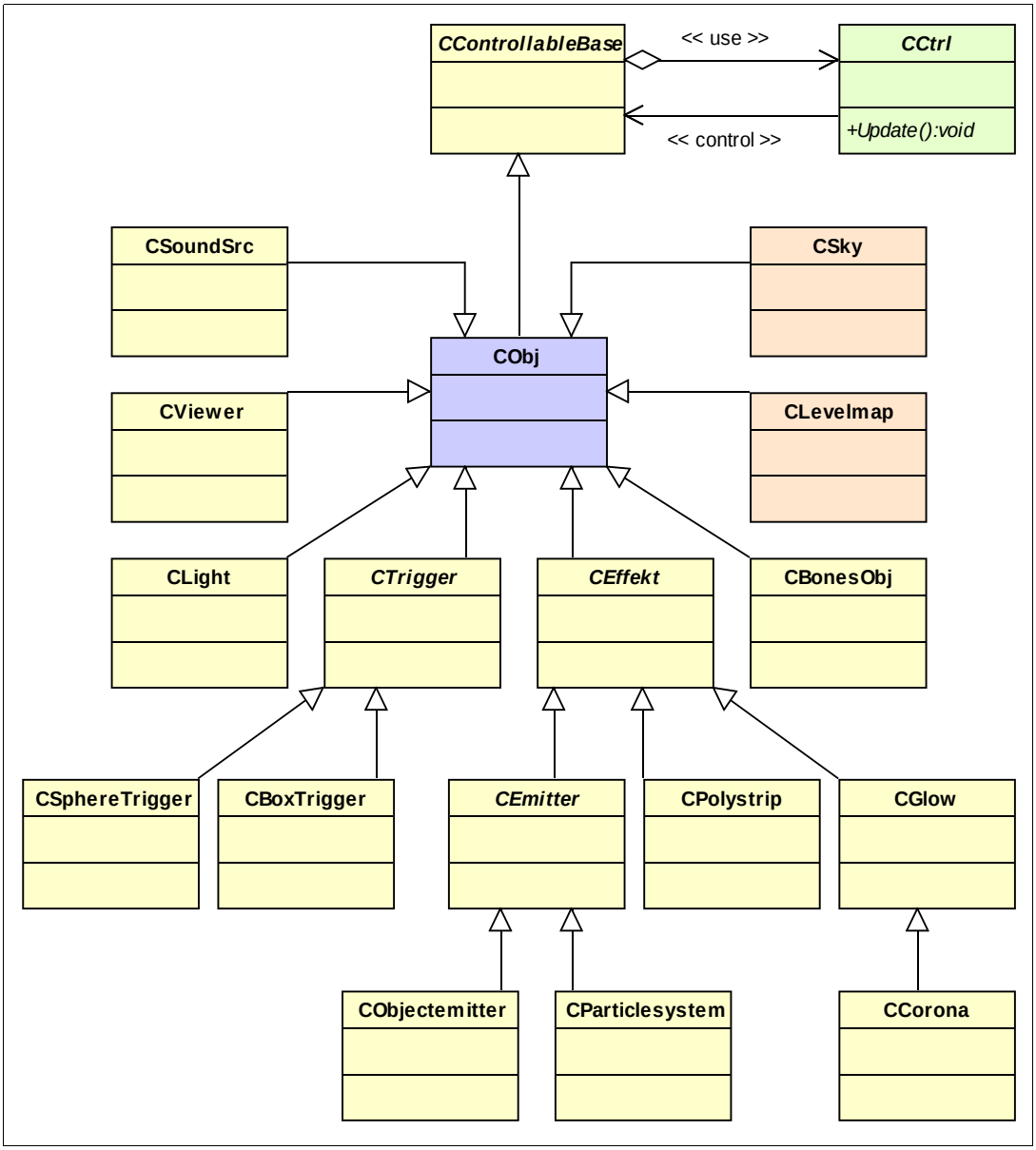


Abbildung 8: UML-Klassendiagramm der Vererbungshierarchie ausgewählter Klassen

4.1.3 Scripting

Mit dem Einsatz von Scriptsprachen wird eine Spieleengine flexibel einsetzbar. Wenn eine Engine durch Scripts gesteuert wird, können verschiedene Spielsituationen mit Hilfe unterschiedlicher Scripts realisiert werden, wobei sich die Engine nicht an die neue Situation anpassen muss. Eine Scriptsprache ist eine leicht erlernbare Highlevel-Sprache, die für Designer und weniger für Programmierer konzipiert ist.

Das Auffinden von Fehlern wird durch den Einsatz einer Scriptsprache deutlich vereinfacht, da sich Testfälle nun gezielt und einfach durch Scripts erzeugen lassen. Daran knüpft das *Rapid Prototyping* an, denn mit Scripts können auch Ideen für Features schnell und unkompliziert ausprobiert werden. Sollte sich eine Idee dann als brauchbar erweisen, kann diese auch als „Hardcode“ in die Engine integriert werden.

Die ersten Scripts kamen in Gestalt der INI-Dateien, in denen Benutzereinstellungen wie Tastenbelegung und Grafikeinstellungen gespeichert wurden, die beim Start der Engine jedesmal neu eingelesen wurden. Heutige Scriptsprachen wie TCL gehen viel weiter. Mit ihnen lassen sich alle Teile der Engine beeinflussen. Sie ermöglichen es, Befehle an die Engine abzusetzen, kennen typisierte Variablen, Kontollstrukturen wie `if..then..else` und Schleifen. Die YAGER-Engine geht sogar so weit, die komplette Spiellogik mit Hilfe der Skriptsprache umzusetzen.

Scripts und YAGER-Engine können sich gegenseitig beeinflussen. Während Scripts Befehle auf einzelnen Engine-Objekten ausführen, kann die Engine auf den Objekten im Szenenbaum sogenannte *Events* auslösen. In den Scripts wiederum kann auf diese Events reagiert werden.

Für nahezu jede Aufgabe existieren in der YAGER-Engine Scripts. Angefangen vom Levellogikscript, das den Ablauf der Spiellogik beschreibt, bis hin zu Controller- und Effektscripts, die bestimmte Eigenschaften der Objekte festlegen.

Klassen, die von der Scriptsprache Gebrauch machen sollen, müssen lediglich definieren, welche Scriptbefehle auf welche Methoden der Klasse verweisen sollen und welcher Art die übergebenen Parameter sind. Jede Klasse ist für seine eigene Scriptanbindung verantwortlich.

4.1.4 Das Routen Management

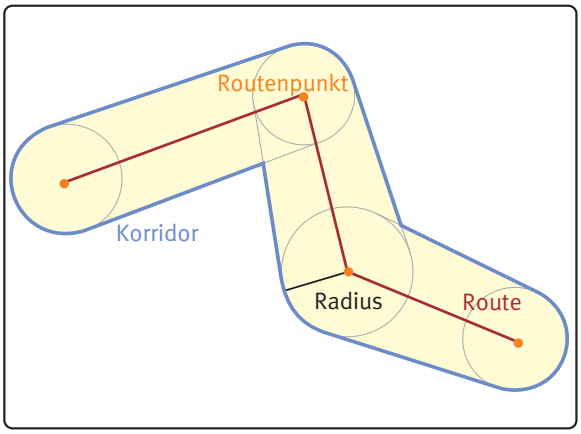


Abbildung 9: Definition einer Route

Das Routenmanagement ist als Bestandteil der YAGER-Engine bereits vor der Planung des Steuerungssystems vorhanden gewesen. Es besteht im Wesentlichen aus drei Komponenten – dem *Pfadnetz*, dem *RouteManager* und der berechneten *Route*. Das Routen Management kann als Service angesehen werden, der der Planungsebene zur Verfügung steht.

Das Pfadnetz ist ein Netz aus untereinander verbundenen Routenpunkten. Soll sich ein Vehikel durch dieses Pfadnetz bewegen, so muss zunächst eine Route durch dieses Netz berechnet werden. Eine Route repräsentiert in der Spielwelt einen sicher begehbaren Bereich. Ein Vehikel kann sich nur mithilfe von Routen durch seine Umgebung bewegen. Ausserhalb der Routen befindet sich in der Regel unbegehbares Terrain wie z.B. Schluchten, Wasserflächen, bebautes Gelände etc. Eine Route besteht aus mindestens zwei Punkten, die durch eine Kante verbunden sind. Zusätzlich besitzt jeder *Routenpunkt* einen *Radius*, mit dem es möglich ist, der Route Breite zu geben und so eine Fläche zu definieren. Diese Fläche wird als *Korridor* bezeichnet und stellt den begehbaren Bereich eines Vehikels dar.

Die Berechnung einer Route durch das Pfadnetz ist die Aufgabe des RouteManagers, der durch die Klasse `CRouteMgr` dargestellt wird und eine Implementation des A* Algorithmus beinhaltet. Dieser Algorithmus ist vollständig, d.h. wenn eine Route zum Zielknoten existiert, so wird diese gefunden. Darüber hinaus ist er auch optimal, d.h. es gibt keine bessere Route zum Zielknoten, als die gefundene. A* ist der meist verwendete Algorithmus zum Berechnen von Routen in großen Netzen bzw. Graphen. Weitere Informationen zu A* und Pfadberechnung sind bei [RABINoo] zu finden.

Dem RouteManager kann eine Anfrage nach einer Route zu einer bestimmten Position oder einem anderen Pfadpunkt gestellt werden. Existiert eine Route zu diesem Punkt, so gibt der RouteManager ein Objekt zurück, das alle benötigten Informationen zu dieser Route, wie Positionen und Radien der einzelnen Pfadpunkte, enthält.

Dieses Objekt ist eine Instanz der Klasse `CPathInfo` (PathInfo) und dient als *Wrapper* des Routenmanagements für die Steuerungsebene. Alle Anfragen und Berechnungen der Steuerungsebene bezüglich des Pfades werden von ihm erledigt. Es existiert keine direkte Verbindung der Steuerungsebene mit dem RouteManager.

4.1.5 Fortbewegungsebene

Das Fahrzeugmodell, das in YAGER-Engine Anwendung findet, ist komplex. Mit diesem Modell sind Vehikel in der Lage, über Bodenunebenheiten zu springen, zu schleudern und sogar umzukippen. Dabei reagiert das Vehikel mit simulierten physikalischen Sekundärbewegungen. Der Vorderteil z.B. senkt sich beim Bremsen ab, richtet sich beim Beschleunigen auf und schwankt bei Kurvenfahrten. Mit diesem Modell sind eine Vielzahl von Vehikeltypen wie z.B. Panzer, PKW, LKW, Motorräder und sogar Charakteren beschreibbar. Allerdings fällt bei Charakteren die physikalische Korrektheit nicht allzu sehr ins Gewicht. Hier kommt es vielmehr auf korrekte Animationen an.

Das Vehikel besitzt eine Vielzahl an physikalischen Eigenschaften. Dazu gehören die Grundgrößen Masse, Schwerpunkt, Trägheitstensor und einige Dämpfungswerte. Hinzu kommen simulierte Größen wie Geschwindigkeit (LinVel) und Drehgeschwindigkeit (AngVel). Sie werden durch das Anlegen von verschiedenen Kräften (AccForce, DecForce), Beschleunigungen und Momenten (Torque) ermittelt.

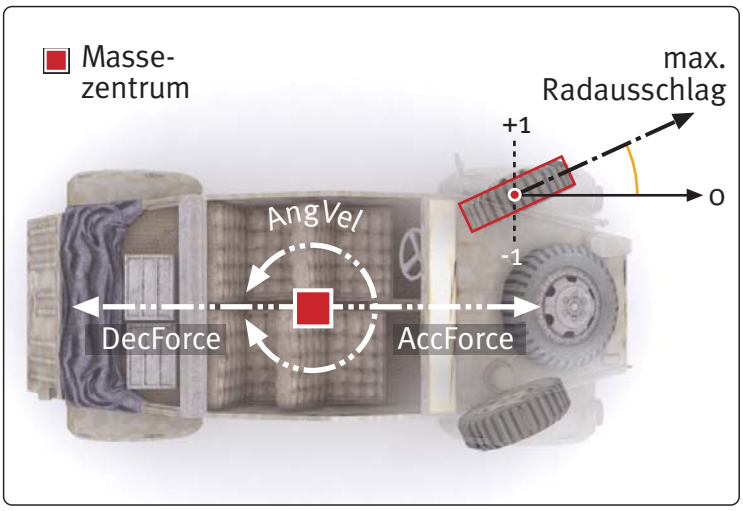


Abbildung 10: Eigenschaften der Fahrphysik am Beispiel des CarControllers

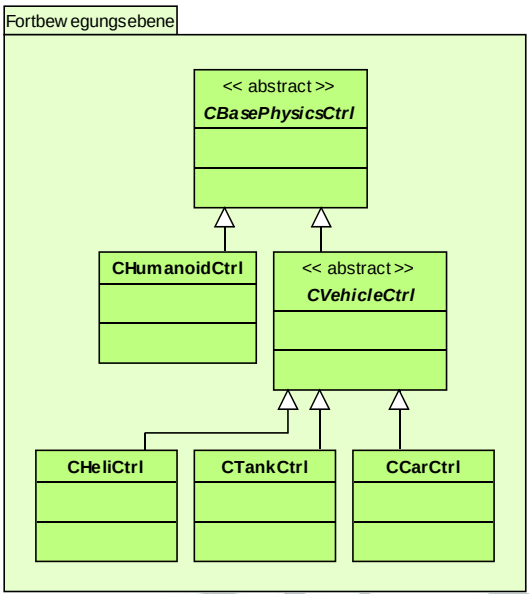


Abbildung 11: Vehikelcontroller

Diese grundlegende Fahrzeugphysik wird von der Klasse `CBasePhysicsCtrl` (*BasePhysicsController*) bereitgestellt. Jedes Vehikel, das auf irgendeine Weise bewegt werden soll, sei es durch die KI eines autonomen Agenten, durch „manuelle Steuerung“ eines Spielers oder durch eine gescriptete Abfolge von Steuerungsbefehlen, benötigt einen Controller, der vom BasePhysicsController erbt. Er stellt die Funktionalität zur Verfügung, mit der sich ein Vehikel fortbewegen kann und liefert die Schnittstelle für die darüber liegende Steuerungsebene. Von dieser Klasse sind konkrete

Implementierungen für Charaktere (`CHumanoidCtrl`) und die einzelnen Vehikelgruppierungen abgeleitet. Beispiele für diese *VehikelController* (`CVehicleCtrl`) sind `CCarCtrl` für Fahrzeuge mit Rädern, `CTankCtrl` für Kettenfahrzeuge und `CHeliCtrl` für Hubschrauber. Diese Controller implementieren die Eigenarten und Besonderheiten der Fortbewegung der jeweiligen Vehikelgruppe. Die folgende Beschreibung des Car-Controllers soll die Funktionsweise der Vehikelcontroller deutlich machen.

Der Car-Controller

Jedes Vehikel, das einen Car-Controller besitzt, kann über eine beliebige Anzahl von Rädern verfügen. Diese Räder sind über ein Federsystem mit dem Vehikel verbunden. Dieses Federsystem sorgt dafür, dass das Vehikel bei Bodenunebenheiten ein sicheres Fahrverhalten behält, da die Kräfte, die auf das Vehikel wirken, gedämpft werden. Ein Nebeneffekt sind die realistischen Bewegungen, die das Fahrzeug z.B. bei Sprüngen, Kurvenfahrten und Vollbremsungen zeigt. Um arbeiten zu können benötigt das Federsystem einige Parameter, die im Folgenden kurz vorgestellt werden.

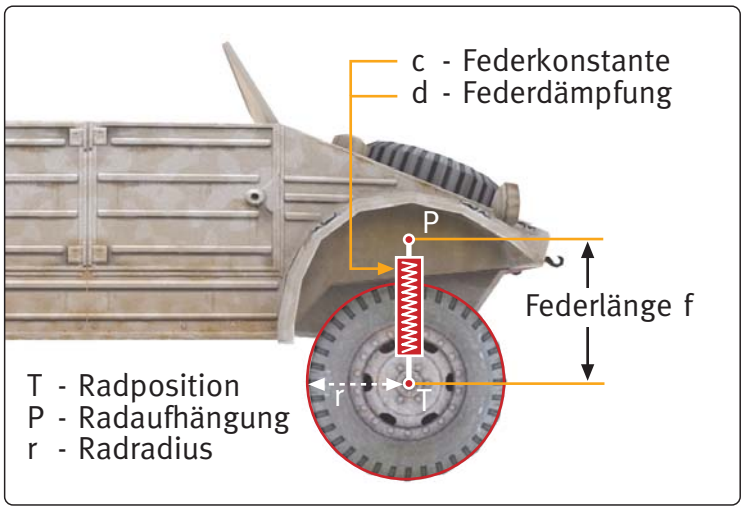


Abbildung 12: Federsystem des CarControllers

An der Position `P` befindet sich die Radaufhängung. An dieser hängt eine Feder mit dem maximalen Federweg `f`, der Federkonstante `c` und der Federdämpfung `d`. Die Federlänge `f` beschreibt die Länge der Feder in Ruhelage, wenn das Vehikel schwebt. Die Feder hat ihre maximale Länge erreicht, wenn das ganze Gewicht des Rades an ihr hängt. Die Federkonstante `c` bestimmt wie hart die Feder ist. Die Federdämpfung `d` definiert das Schwingverhalten der Feder, sie gibt an, wie schnell die Schwingungen abgebaut werden. Durch Änderung dieses Wertes kann zwischen Schwingfall und Kriechfall geregelt werden.

Am anderen Ende der Feder befindet sich ein Rad mit einem bestimmten Radius `r`. Die Position `T` des Rades wird für jeden Frame neu ermittelt. Dabei übt die Federung auf das Rad und die Aufhängung am Vehikel eine gewisse Kraft aus, die dazu führt, dass das Vehikel federt. Die Paramter, die das Federsystem beeinflussen, werden über ein Script an die Engine und damit an den entsprechenden Controller übergeben. Listing 1 zeigt ein solches Controller-Script.

```
#create and link wheels
#--------------------------------------------------------------------------------
setmass             700                 ;# kg
setmasscenter       { 0 30 0 }          ;# Offset vom Mittelpunkt des Meshes
setlinvelmax        22.2                ;# m/s -> 80 km/h
setangvelmax        360.0               ;# Grad/s
setangforcemax      10000               ;# N
setaccforcemax      3500.0              ;# N
setdecforcemax      7000.0              ;# N
setlindamp          0.65                ;# AccForce auf 65% je Sekunde dämpfen
setangdamp          0.5                 ;# AngForce auf 50% je Sekunde dämpfen
setwheelangle       36.0                ;# Grad - maximaler Radausschlag
setinertia          sphere 60 0 0       ;# Trägheitstensor (Kugel)

#Räder erzeugen und anhängen
#--------------------------------------------------------------------------------
# P    -      Radaufhängung       (Vektor)
# f    -      Federweglänge       (float)
# r    -      Radradius           (float)
# s    -      Antrieb             (bool)
# c    -      Federkonstante      (float)
# d    -      Federdämpfung       (float)
#--------------------------------------------------------------------------------
#                P             f      r    s    c       d
addwheel { 20.43 22.2  33.9}  16.5  10.12  1  565.5  3000.0
addwheel {-20.43 22.2  33.9}  16.5  10.12  1  565.5  3000.0
addwheel { 20.43 21.0 -34.9}  16.5  10.12  0  565.5  3000.0
addwheel {-20.43 21.0 -34.9}  16.5  10.12  0  565.5  3000.0
```

Listing 1: Beispielscript eines Car-Controllers

4.1.6 Level und Tools

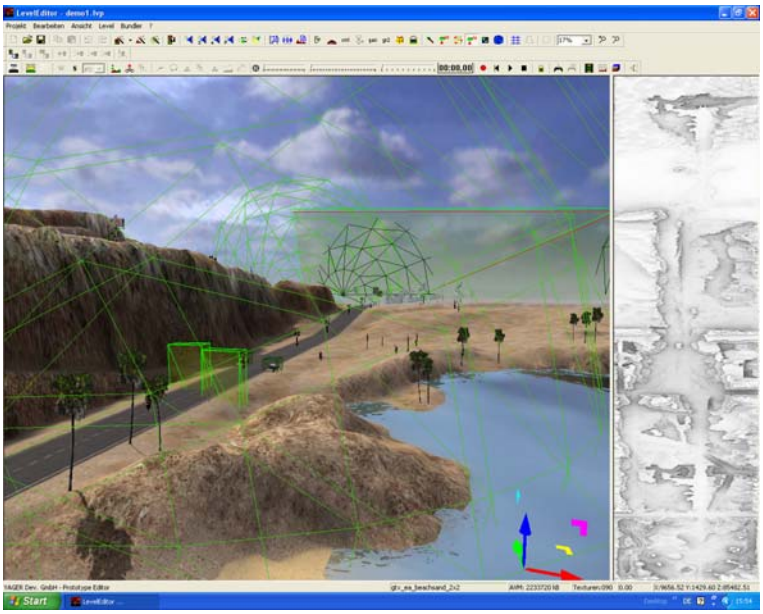


Abbildung 13: YAGER Leveleditor

Der Inhalt des Spiels ist in verschiedene Levels aufgeteilt. Sie beinhalten die einzelnen Assets wie Texturen, 3D-Objekte, Soundeffekte, Musikdaten, Landschafts- und Beleuchtungsdaten. Die Abfolge der unterschiedlichen Spielsituationen und anderer Ereignisse ist im Levellogikscript definiert. In ihm kann auf jeden Bestandteil eines Levels wie Objekte, Effekte, KI, Routendaten etc. Einfluss genommen werden, um das Spielerlebnis zu steuern. Mit Hilfe des hauseigenen Leveleditors werden komplexe und detaillierte Welten erschaffen, die für verschiedene Plattformen kompiliert werden können. Abbildung 13 zeigt den Leveleditor, der eine Scene mit Wasser, Landschaft und Objekten wie Bäumen und Triggern darstellt.

4.2 Künstliche Intelligenz

Die YAGER-Engine ist als „Flug-Action"-Engine konzipiert worden. Alle Gegner bzw. Künstlichen Intelligenzen sollten fliegen und einfache Flugmanöver wie „Ausweichen" durchführen können. Die Steuerung erfolgte durch Finite-State-Machines (FSM1), die in Scripts definiert wurden und die Zielplanung sowie das komplette Verhalten der Gegner bestimmten. In diesen FSMs waren Verhaltensregeln wie „wenn beschossen, weiche nach rechts aus" etc. definiert. Die Flugeigenschaften bestimmte der sogenannte `CAICtrl`, der über Eigenschaften wie Gewicht und Wendigkeit verfügte und das Flugobjekt steuern konnte. Mit diesem relativ einfachen System gelang es, verschiedene Gegner mit unterschiedlichen Schwierigkeitsgraden zu kreieren. Allerdings mit einigem Aufwand, da in jedem Level die State Machines für jeden Gegner angepasst werden mussten.

Nachdem das Spiel „Yager" im Handel erschienen war, erfolgte die Neuorientierung auf „Vehikel-Actionspiele" und eine damit verbundene Umstellung der Engine. Die Art der Vehikelsteuerung zu Luft, bei der sechs Freiheitsgrade existieren, unterscheidet sich sehr stark von

der Vehikelsteuerung an Land, wo die Steuerung lediglich eine Angelegenheit von „links-rechts-schneller-langsamer“ ist. Einerseits mag die Berechnung der Steuerung am Boden einfacher sein, andererseits bietet die Luft mehr Freiheiten. So erscheint das Ausweichen in letzter Sekunde in der Luft wie ein waghalsiges Manöver, am Boden entsteht bei derartigen Aktionen ein vollkommen anderer Eindruck. Sie würden sofort die Frage aufwerfen, ob die drohende Kollision nicht schon früher erkannt werden konnte. Der Eindruck von autonomer Intelligenz, der durch das Vehikel vermittelt werden soll, hängt stark von der Art der Steuerung ab.

Zu Beginn der Umstellung wurden zwei Ansätze verfolgt. Zum einen wurde die Fahrphysik, mit der die Fahrzeuge später ausgestattet werden sollten, vorbereitet und implementiert. Getestet wurde sie mit Hilfe von Eingabegeräten wie Maus und Tastatur. Diese Methode sollte die Grundlage für die Steuerung autonomer Agenten schaffen.
Zum anderen sollte schnellstmöglich eine grundlegende Funktionalität für die Fortbewegung und Routenplanung bodenbasierter Vehikel bereitgestellt werden, um damit erste Prototypen realisieren zu können. Die Funktionalität dieser Methode beschränkte sich auf die Berechnung von Routen durch ein Wegenetz und das Anfahren der einzelnen Routenpunkte. Das System war dem von „Yager“ ähnlich. Ein KI-Controller sorgte für die Bewegung, indem nun zwischen Position und Zielpunkt über eine bestimmte Zeit linear interpoliert wurde. Finite State Machines sorgten für die Routenplanung und eine rudimentäre Verhaltenssteuerung.

4.3 Bewertung

Es gibt mehrere Punkte, die eine Neuentwicklung des Steuerungssystems nahelegen. Der offensichtlichste Grund ist, dass die Bewegungsabläufe unnatürlich und steif wirken. Selbst wenn das bisherige Steuerungssystem von den Vorteilen der neuen Fahrphysik profitieren sollte, so ist es trotzdem nur befähigt, Vehikel von Punkt A nach Punkt B fahren zu lassen. Die physikalische Simulation des Vehikels ist dabei nebensächlich.
Die Vehikel wären immer noch nicht in der Lage, Hindernissen auszuweichen oder in irgendeiner Weise dynamisch auf ihre Umwelt zu reagieren. Gruppenverhalten sowie komplexere Verhikelverhaltensweisen werden von diesem System nicht unterstützt und das gesamte Verhaltensrepertoire wird, wie zuvor, in den State Machines definiert werden müssen.

Diese Punkte machen eine Neuentwicklung des Steuerungssystems unumgänglich, wobei schon vorhandene Systeme wie das Routenmanagement, die Fahrphysik und die Planung mittels Finite-State-Machines weiterhin genutzt werden können.

5. Design

Im Folgenden sollen die Anforderungen an ein Steuerungsystem für autonome Agenten definiert werden. Dieses System wird dann mit Hilfe eines objektorientierten Softwaremodells umgesetzt. Kapitel 5.1 liefert eine vollständige Anforderungsdefinition. Hier wird auf benötigte Features eingegangen sowie bestehende Systeme, mit denen das Steuerungssystem zusammenarbeiten soll, vorgestellt. Die Modellierung des Systems findet in Kapitel 5.2 statt. Das Problem wird in einzelne, in sich abgeschlossene Bereiche aufgeteilt, aus denen später Klassen und Objekte entstehen werden. Beziehungen und Abhängigkeiten werden in Form von Assoziationen, Aggregationen und Kompositionen in das theoretische Softwaremodell übertragen. Als Beschreibungssprache wird UML (Unified Modelling Language) verwendet, der im Moment übliche Standard für die Konstruktion, Spezifikation und Visualisierung von Softwaresystemen. Dieses Kapitel konzentriert sich auf die Aufgaben und Kommunikation der einzelnen Systemkomponenten. Konkretere Erläuterungen sind dann im Kapitel 6 „Implementation“ zu finden.
In Kapitel 5.3 werden einige Probleme erläutert, die aus diesem Design entstehen können.

5.1 Anforderungsdefinition

Die Spielebranche ist ein innovatives und kreatives Feld, auf dem neue Ideen und Techniken in rascher Folge erscheinen und entwickelt werden. Die YAGER-Engine befindet sich nach dem Verkauf des ersten Titels in einer Umbau- bzw. Modernisierungsphase. Daher gab es im Verlauf der Entwicklung dieses Systems einige Neuentwicklungen und Änderungen an wichtigen Schnittstellen und Kernkomponenten der Steuerung. Da sich die Firma zur Zeit in einer Prototyping-Phase befindet, musste flexibel auf die Bedürfnisse der Game-Designer eingegangen werden. Dieser Umstand schlug sich auch in der Entwicklung des Systems nieder.

5.1.1 Ziele

Mit dem Steuerungssystem soll der Grundstein für ein komplett neues KI-System gelegt werden. Mit Agenten, die autonom ihren Weg durch eine Umgebung finden, soll den Leveldesignern ein großes Stück an Arbeit abgenommen werden, indem Sie nicht mehr bis ins letzte Detail definieren müssen, wie sich eine Einheit bewegt, sondern nur noch warum und wohin. Mit einem einheitlichen Steuerungssystem für autonome Agenten, das für eine Vielzahl von Genres eingesetzt werden kann, ist es möglich, einmalig einige Verhaltensweisen festzulegen und sie in verschiedenen Leveln und Spiele-Titeln zu verwenden. Mit einem System, das mehrere Verhaltens-Bausteine verwendet, können Synergieeffekte genutzt werden, die aus der Kombination dieser Verhaltensweisen entstehen. Ein interessantes, abwechslungsreiches und weniger vorhersehbares Agentenverhalten ist damit ohne größeren Aufwand möglich.

5.1.2 Anforderungen

Eine wichtige Anforderung ist Flexibilität. Das System muss eine breite Palette von Vehikeln mit unterschiedlicher Fortbewegungsart, Geschwindigkeit und Agilität unterstützen. Dabei sollen diese Vehikel ein realitätsnahes Verhalten zeigen und ein gewisses Maß an Grundintelligenz mitbringen, wenn es darum geht durch ihre Umgebung zu navigieren. Das bedeutet z.B., dass sie selbständig Hindernissen ausweichen und ineinander fahren. Diese Verhaltensweisen sollen leicht erweiterbar, austauschbar und kombinierbar sein.

In der ersten Ausbaustufe ist das Steuerungssystem des Agenten als reaktives System mit begrenzter Planung vorgesehen. Das heißt, der Agent reagiert auf Einflüsse seiner Umgebung und bekommt Ziele vordefiniert, denen er nachgehen soll.

Die zweite Ausbaustufe stellt einen KI-Überbau dar, der es dem Agenten erlaubt strategisch zu planen und seine Ziele selbständig zu definieren. Hier soll auch erstmals Gruppenverhalten implementiert werden, das unter anderem die Bewegung in Formationen unterstützt. Für diese zweite Ausbaustufe soll eine Schnittstelle existieren.

Eine weitere wichtige Anforderung ist, das der Agent trotz seiner Autonomie kontrollierbar bleibt. Er soll von ausserhalb, z.B. durch andere Agenten und State-Machines steuerbar sein. Dazu soll das Steuerungssystem u.a. von der Scripting Technologie der YAGER-Engine Gebrauch machen. Scripts ermöglichen darüber hinaus ein schnelles Prototyping und debuggen.

Speichersparsamkeit und Performanz ist eine weitere Bedingung, um neben dem PC auch auf Spielekonsolen wie die Microsoft XBOX™ lauffähig zu sein. Das Design von zeitkritischen Systemen weist einige Besonderheiten gegenüber herkömmlichen Systemen, wie z.B. betriebswirtschaftlichen Anwendungen auf. Hier stehen das Zeitverhalten, die Synchronistation konkurrierender Prozesse, dynamische Datenstrukturen und eine damit verbundene Speicheranforderung bzw. -bereinigung zur Laufzeit stärker im Vordergrund, als die reine Datenmodellierung.

Die letzte Anforderung ist, dass sich das Steuerungssystem in die YAGER-Engine einfügt und vorhandene Systeme wie Routenmanagement, Statemachines und Fahrphysik nutzt.

5.1.3 Vorhandene Systeme

Nach dem Flugspiel *Yager* wurde ein neues System zur Steuerung von Vehikeln am Boden benötigt. Um Ideen schnell ausprobieren zu können, wurde eine Zwischenlösung entwickelt, die in der Lage war Vehikel am Boden rudimentär zu bewegen und ausreichte, um prototypisch darzustellen wie zukünfitige Titel aussehen werden. Die Bodenfahrzeuge wurden mit den StateMachines des vorherigen Spiels (FSM1) gesteuert und nutzten das Routensystem für die Wegfindung. Zwischen den Positionen wurde linear interpoliert, um das Vehikel zu bewegen.

Da im weiteren Verlauf der Entwicklung klar wurde, dass die bereits bestehenden Planungslösungen der YAGER-Engine wie die FSM1 nicht die Anforderungen an Flexibilität und Arbeitserkeichterung der Leveldesigner erfüllte, wurde das Finite-State-Machine System komplett neu entworfen. Das neue System (FSM3) sollte leichter erweiterbar sein und eng mit dem Steuerungssystem zusammenarbeiten. Das alte FSM-System unterstützte lediglich UND-Verknüpfungen zwischen den Transitionsbedingungen, um den Evaluierungsaufwand möglichst gering zu halten. Die FSM3 sollten flexibler handhabbar sein und zusätzlich ODER-Verknüpfungen, Negationen und Schachtelungen anbieten.

Zum Zeitpunkt der Systemmodellierung waren einige Rahmenbedingungen zu beachten. Es existierte bereits die Fahrphysik, die beliebige Fahrzeuge physikalisch korrekt bewegen konnte. Vehikel mit dieser Fahrphysik wurden „von Hand“ mit diversen Inputgeräten wie Maus, Tastatur und XBOX-Controller™ gesteuert. Dazu wurde eine Schnittstelle definiert, mit der Daten von steuernden Objekten einheitlich übermittelt werden konnten. Die Fahrphysik wurde im Verlauf der Entwicklung stets erweitert und angepasst, die Schnittstelle jedoch blieb konstant.

5.2 Modellierung

In diesem Kapitel erfolgt die Zerlegung des Steuerungssystems in mehrere Untersysteme nach den Methoden der *Objektorientierten Analyse* (OOA). Der Vorteil von OOA gegenüber anderen Modellierungsmethoden wie der Strukturierten Analyse (SA) liegt in der gleichzeitigen Beachtung von Daten und Funktion. OOA verwendet Werkzeuge wie Use-Cases und Klassendiagramme, um Prozesse und Objekte zu dokumentieren und um von der Analyse und Design leichter in die Entwicklung und Implementation überzugehen. Der Focus liegt hier auf Objekten und deren Interaktionen untereinander.

Der erste Schritt bei der Modellierung von Softwaresystemen ist es, die Klassen zu identifizieren, die für die Lösung des Problems eine entscheidende Rolle spielen. Sie ergeben sich in den meisten Fällen direkt aus der Problemstellung und stellen die Kernkomponenten des zukünftigen Softwaresystems dar. Der nächste Schritt besteht in der Einführung der Klassen, die zusätzlich benötigt werden, um die Kommunikation zwischen Objekten zu erleichtern und das Systemdesign zu vereinfachen. Dieser Schritt berührt den Bereich der Entwurfsmuster, die ein effektives Werkzeug in der Objektorientierten Programmierung (OOP) sind. Entwurfsmuster bieten Lösungen für wiederkehrende Probleme im Softwaredesign an und stellen eine Abstraktionsschicht dar, die über einzelnen Klassen, Instanzen und Komponenten liegt[7].

7 Vgl. [COOPER98] S.12

5.2.1 Entwurfsmuster

Im Grunde sind Entwurfsmuster so etwas wie Softwaresysteme. Für eine Vielzahl von Problemen in der OOP wurden Entwurfsmuster erschaffen und dokumentiert. Entwurfsmuster machen objektorientierte Entwürfe flexibler, eleganter und wiederverwendbarer. Ein wichtiger Grund für die Verwendung von Entwurfsmustern ist die Abgrenzung von Klassen untereinander, um sie davor zu bewahren, unnötiges Wissen voneinander zu haben. Die meist verwendeten Muster um das zu erreichen sind Polymorphie, Kapselung, Vererbung und Delegation – die Grundbausteine der OOP.

Abstrakte Klassen und Interfaces sind ein weiteres mächtiges Werkzeug, um saubere und einfache Systeme zu modellieren. Jede Wurzel einer Klassenhierarchie sollte eine abstrakte Klasse zur Basis haben, die keine Methoden implementiert, sondern nur die Methoden der Klasse definiert. In den abgeleiteten Klassen hat man so die Freiheit, die Implementation nach Maß zu gestalten, anstatt nutzlosen Implementationsballast bis hinunter zur letzten Klasse mitzunehmen, wenn von einer konkreten Klasse abgeleitet wird. Der Grundsatz lautet:
„Program to an interface and not to an implementation.“ [8]
Transportiert man diesen Grundsatz, der aus der Java-Welt kommt, in eine C++ Welt, so lautet die Forderung, auf abstrakten Klassen aufzubauen.

Ein weiteres wichtiges Konzept ist die O*bjektkomposition*, d.h. die einfache Konstruktion von Objekten, die andere Objekte (Member) beinhalten. Im Vergleich zur Vererbung ist die Objektkomposition ein flexibles Mittel, um Objekte mit zusätzlicher Funktionalität auszustatten. Die Vererbung ist die engste Beziehung zwischen zwei Klassen, die eine starke, mitunter schwer zu kontrollierende Abhängigkeit schafft.
Vererbung sollte nur in Fällen angewandt werden, in denen folgende Aussage getroffen werden kann: A soll von B erben wenn gilt: A ist ein B. In allen anderen Fällen lautet der Grundsatz :
„Favor object composition over inheritance.“ [9]

Soweit wie möglich wurden die Grundsätze der OOP bei der Entwicklung des Steuerungssystems beachtet und Entwurfsmuster eingesetzt, die zur Vereinfachung des Systementwurfs beitrugen. An den Stellen wo sie Anwendung finden, wird näher auf die Arbeitsweise und Vorteile dieser Muster eingegangen.

8 „Programmiere für eine Schnittdtelle, nicht für eine Implementation“ Vgl. [COOPER98] S.14

9 „Ziehe Objektcomposition der Vererbung vor“ Vgl. [COOPER98] S.15

5.2.2 Systementwurf

Die erste grobe Unterteilung des Steuerungssystems erfolgt analog zum Steuerungssystem, das Reynolds [REYNOLDS99] vorschlägt. Es werden drei Ebenen eingeführt, die voneinander unabhängig arbeiten können und jeweils einen konkreten Aufgabenbereich haben. Das sind die Planungsebene, die Steuerungsebene und die schon vorhandene Fortbewegungsebene. Die Systemarchitektur wurde aufgrund seiner Einfachheit übernommen. Das dreiteilige Konzept, bei dem eine Komponente *plant*, eine zweite Komponente *steuert* und eine dritte Komponente *ausführt*, fühlt sich für diese Problemstellung natürlich an. Es lassen sich zahlreiche Beispiele für ein solches System in der Realität finden. Ein Kapitän (Planung) befiehlt seiner Mannschaft, (Steuerung) mit dem Schiff (Fortbewegung) nach Süden zu segeln. Ein Soldat bekommt den Befehl (Planung), auf eine Zielscheibe zu schiessen. Dazu zielt er und drückt ab (Steuerung). Das Gewehr (Fortbewegung / Ausführender) veranlasst alles weitere, um die Kugel aus dem Lauf zu drücken. Jede der drei Ebenen verlässt sich auf die Durchführung seiner Befehle und muss sich somit nicht mehr um Details kümmern, die im Verantwortungsbereich der nächst unteren Ebene liegen.

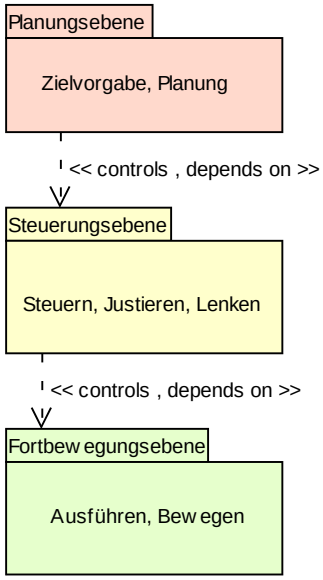


Abbildung 14: Architektur des Steuerungssystems

Die Wahl dieser Architektur liegt jedoch aus einem weiteren Grund nahe. Die YAGER-Engine beinhaltet bereits eine Fortbewegungsebene mit einer Fahrphysik für Vehikel. Die Schnittstelle ist ebenfalls definiert. Zudem sind einige Komponenten einer zukünftigen Planungsebene bereits enthalten, wie z.B das Routenmanagement und FSM1. Daraus folgt, dass zur Umsetzung eines kompletten Steuerungssystems nur noch die mittlere Schicht, d.h. die Steuerungsebene, umgesetzt werden muss.
In Abbildung 21 auf Seite 45 befindet sich das UML-Klassendiagramm des gesamten Steuerungssystems.

Die folgenden Abschnitte befassen sich mit der Fragestellung, welche Klassen an der Problemlösung beteiligt sind und welche Aufgaben die jeweiligen Klassen haben. Das Design der einzelnen Ebenen wird im Detail vorgestellt.

5.2.3 Planungsebene

Diese Ebene soll den planerischen, zielgerichteten Part im Steuerungssystem übernehmen. Hier werden Zielobjekte vorgegeben, Parameter der Steuerung verändert und Routen berechnet. In dieser Ebene wird zusätzlich beeinflusst, wie intelligent der Agent wirkt, wenn er auf Veränderungen seiner Umgebung reagiert. Ist beispielsweise ein Feind in der Nähe, muss abgewogen werden, ob es sich lohnt einen Kampf zu riskieren oder ob Fliehen die bessere Lösung wäre. Der Agent muss ständig seine Umgebung analysieren und Entscheidungen treffen. Finite State Machines sind ein effektives Mittel, um die Entscheidungsfindung zu realisieren.

Bei der Gestaltung des FSM-Systems werden zwei Teilsysteme identifiziert. Zum einen das State-Machine System an sich, mit einem Netz an Zuständen, zwischen denen gewechselt wird. Zum anderen wird ein System benötigt, das es erlaubt komplexe Regeln und Bedingungen für das Wechseln dieser Zustände zu definieren. Das Design des Transitions- und FSM-Systems wird in den folgenden Unterkapiteln erläutert. Die Implementierung wurde dann durch einen Kollegen durchgeführt.

5.2.3.1 Transitionssystem

Transitionen sind Bedingungen für das Wechseln in einen neuen Zustand. Diese Bedingungen liefern stets boolsche Werte und können mitunter recht komplex werden.

Schachtelbare logische Verknüpfungen der Transitionsbedingungen erfordern eine baumartige Struktur. In ihr sind logische Operatoren wie UND, ODER, NOT und die Transitionsbedingungen (Conditions) miteinander verbunden. In Abbildung 15 stellen die orangefarbigen Kästchen die Bedingungen dar. Die Knoten bilden die Operatoren, dessen Kinder entweder Transitionsbedingungen oder Ausdrücke (Expressions) sind. Ausdrücke sind dann wieder logisch verknüpfte Transitionsbedingungen. Die dargestellte Bedingung lautet in Scriptform:

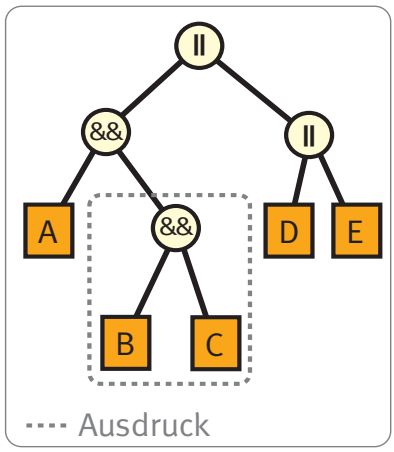


Abbildung 15: Baumstruktur der Transitionsbedingungen

```
Bedingung = ((A AND (B AND C)) OR (D OR E));
```

Aus dieser Beschreibung sind folgende Klassen zur Umsetzung der Transitionsbedingungen zu identifizieren:

CFSM3_T_Node

Diese Klasse (*Node*) ist Grundlage für die Baumstruktur. Von ihr erben `CFSM3_T_TransitionCondition` und `CFSM3_T_Exression`. Die Klasse besitzt die Möglichkeit zur Negation, da sowohl Conditions als auch Expressions negiert werden können. Die Schnittstelle für die Auswertung einer Transitionsbedingung wird hier durch die virtuelle Methode `Evaluate()` definiert.

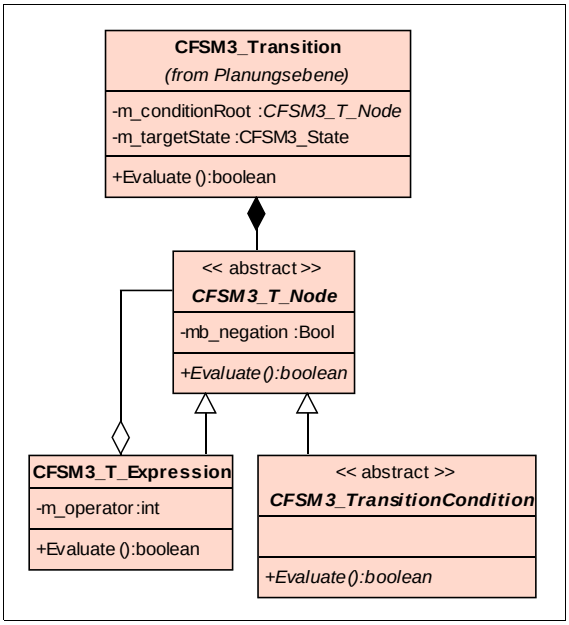


Abbildung 16: UML-Klassendiagramm des Transitionssystems

CFSM3_T_TransitionCondition

Diese Klasse (*Condition*) verkörpert eine Bedingung für das Wechseln eines Zustandes und ist die abstrakte Basisklasse für konkrete Implementierungen. Ein Beispiel für eine Bedingung ist die Abfrage: „`TargetInFiringRange`“, in der geprüft wird, ob sich das aktuell gesetzte Ziel innerhalb des Radius des angeschlossenen Waffensystems des Agenten befindet. Evaluiert diese Bedingung zu `true`, so ist sie erfüllt und der State kann gewechselt werden. Die Condition implementiert die Funktion `Evaluate()`, um das Prüfen der Bedingungen zu realisieren.

CFSM3_T_Exression

Bei dieser Klasse (*Expression*) kommen die logischen Operatoren ins Spiel. Da unäre Operatoren bereits durch die Node-Klasse unterstützt werden, kann man an dieser Stelle davon ausgehen, dass nur noch binäre Operatoren verwendet werden, die zwei Nodes verknüpfen. Das erleichtert das Design, da der Operator als Objekt vollständig eliminiert werden kann und als Member (enum) direkt in die Expression-Klasse integriert werden kann. Die Klasse muss in ihrer Evaluate()-Methode nur noch die Evaluate()-Methoden der beiden Nodes aufrufen, um die Bedingung auszuwerten.

Mit diesen drei Klassen ist es möglich, die geforderten schachtelbaren und komplexen Bedingungen zu definieren. Um die Verbindung zum eigentlichen FSM-System zu schaffen, wird eine weitere Klasse benötigt, die den Übergang zwischen den einzelnen States darstellt.

CFSM3_Transition

Diese Klasse (*Transition*) besitzt eine Transitionsbedingung (`CFSM3_T_Node`) und verweist auf einen Ziel-Zustand (`CFSM3_State`). Bei der Aktualisierung der State-Machine, wird jedes Mal die Bedingung durch Aufruf der Evaluate()-Methode überprüft. Der Condition-Baum wird traversiert und für jede Unterbedingung geprüft. Ergibt die Abfrage `true`, wird zum Ziel-State gewechselt. Die Auswertung dieses Condition-Baumes bzw. der gesamten State-Machine muss nicht in jedem Frame geschehen, da sich das Verhalten eines Agenten normalerweise nicht alle Sekundenbruchteile ändert. Zudem wäre es wäre eine Performance Belastung. Aus diesem Grund wird ein Timer verwendet, der die FSMs nur jede Sekunde aktualisiert.

5.2.3.2 Finite-State-Machine System

Aufbauend auf diesem Transitionssystem erfolgt die Gestaltung des restlichen FSM-Systems, das den Hauptschwerpunkt der Planungsebene in der YAGER-Engine bildet. Das FSM-System besteht im Wesentlichen aus drei Komponenten, die an dieser Stelle kurz vorgestellt werden sollen.

Ein Zustand (*State*) implementiert eine bestimmte Verhaltensweise (*Behaviour*) des Agenten und besitzt beliebig viele ausgehende Transitionen. Mehrere Zustände bilden ein Netz, das Finite-State-Machine (*FSM*) genannt wird.
Diese Komponenten (Abbildung 17) und ihre Aufgaben werden im folgenden detailliert vorgestellt.

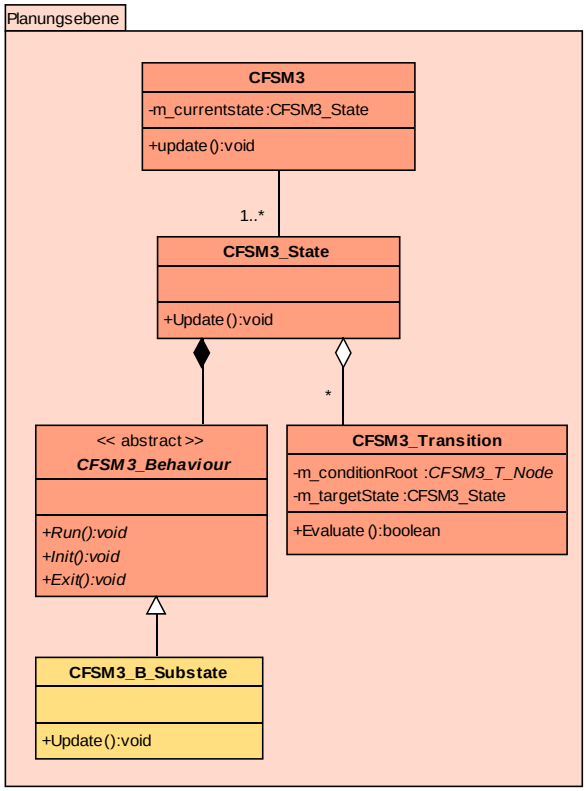


Abbildung 17: UML-Klassendiagramm des FSM-Systems

CFSM3_State

Diese Klasse (*State*) besitzt beliebig viele Transitionen und ein Verhalten (`CFSM3_Behaviour`), dass der gesteuerte Agent annehmen soll. Jeden Update-Zyklus wird das Verhalten ausgeführt und die Transitionen überprüft. Der State ist eine Abstraktionsebene. Er enthält die

Steuerungslogik für die Statewechsel. Ansonsten dient er lediglich als Container für die Objekte, die die eigentliche Arbeit ausführen.
Das „*Befehls-Muster*“ war hier das Vorbild dieser Vorgehensweise. Die eigentliche Arbeit wird durch eine Art *Worker-Objekt,* dem FSM-Behaviour erledigt. Von diesem Worker-Objekt kann es verschiedene Arten geben, ohne das diese dem System bekannt sein müssen. Das erlaubt größtmögliche Flexibilität und Wiederverwendbarkeit, da nahezu jede Aufgabe, auch solche die nicht unbedingt mit Vehikelsteuerung zu tun haben, von den States mit ihren FSM-Behaviours erledigt werden können.

CFSM3_Behaviour

Diese Klasse (*FSM-Behaviour*) stellt eine Art Modul für die Implementierung von Agentenverhaltensweisen dar. Die FSM-Behaviours sind nicht mit den Steuerungsverhalten (Steering Behaviours) zu verwechseln. Sie sind auf einer höheren Ebene angesiedelt und kontrollieren das Verhalten der Agenten, indem sie die Steuerungsverhalten verwenden, die für die Situation am geeigneten sind. Ein Beispiel sorgt für eine klare Abgrenzung der Begriffe:
Ein Vehikel bekommt den Befehl, ein anderes Vehikel zu verfolgen. Dazu wird in den FSM-State „Moveto“ gewechselt. In dem dazugehörigen FSM-Behaviour erfolgt die Wahl der Mittel, mit der diese Verfolgung geschehen soll. Als erstes wird ein Pfad zum Ziel berechnet und dieser mittels des Steuerungsverhaltens *Pathfollowing* verfolgt. Befindet sich das Ziel in direkter Nähe, so wird innerhalb des FSM-Behaviours „Moveto“ auf das Steuerungsverhalten Arrival umgeschaltet, um auf dem Ziel zum stehen zu kommen.
Die FSM-Behaviours bestimmen das Vehikelverhalten in einer niedrigeren Ebene. Im Gegensatz zu States, in denen das Verhalten bei unterschiedlichen Spielsituationen, z.B. Angriff, Patrouille, Idle, etc. definiert wird, kann man bei FSM-Behaviours das Verhalten in der grundsätzlich gleichen Spielsituation variieren, ohne die State-Machine überladen zu müssen.

CFSM3

Diese Klasse (*FSM3*) stellt die komplette State Machine dar. Hier befindet sich der Update-Zyklus sowie die Schnittstelle zur Scriptsprache, mit der State-Machines durch Scriptaufrufe von aussen erzeugt und beeinflusst werden können.
Die FSMs können vom AI-Controller mithilfe von so genannten *Orders* gesteuert werden. Eine Order ist ein Befehl, der die State Machine dazu veranlasst, in einen Ziel-Zustand zu wechseln sofern die Bedingungen erfüllt sind. Innerhalb der Zustände kann die FSM je nach Bedarf Einfluss auf die Steuerungsebene nehmen.

Durch die FSM wird der Vorgang der Steuerung in Gang gesetzt. Das war schon im alten System der Fall, die FSMs mussten jedoch für jede KI und jeden Level neu erstellt werden. Der Leveldesigner gestaltete das gesamte Verhalten der Vehikel in Scripts. Dies stellte sich als äußerst arbeitsaufwändig heraus, da kein Teil der bereits geleisteten Arbeit wieder verwendet werden konnte.

Das neue Steuerungssystem verfolgt einen anderen Ansatz. Hier ist das Verhalten der Agenten in der YAGER-Engine durch FSM-Behaviours vorgegeben. Um das Verhalten steuern und kontrollieren zu können, werden einfach einige Parameter des Agenten verändert. Dieser Ansatz wird als „*Data-Driven*“ bezeichnet. Das erfordert zwar etwas mehr Arbeit auf Seiten der Programmierung. Diese Arbeit muss jedoch nur einmal geleistet werden und nicht pro Level und Künstliche Intelligenz.

Um die Idee der Wiederverwendbarkeit auch für Scripter und Leveldesigner zur Verfügung zu stellen, wurde das FSM-System erweitert. Durch ein spezielles FSM-Behaviour (Abbildung 17, `FSM3_Substate`) wird die State-Machine zu einer hierarchisch gegliederten FSM (*Cascaded-StateMachine*). Mit Cascaded-FSMs (Abbildung 18) ist man in der Lage seine State-Machines Top-Down zu entwerfen, ohne sich von Anfang an mit zuviel States belasten zu müssen. Wo die FSM Verfeinerung benötigt, kann man den Zustand erweitern, indem als Verhalten des Zustandes eine neue State-Machine gesetzt wird. Durch diese Technik ist es möglich, komplexe State Machines durch einfachere wiederverwendbare State Machines zusammenzustellen.

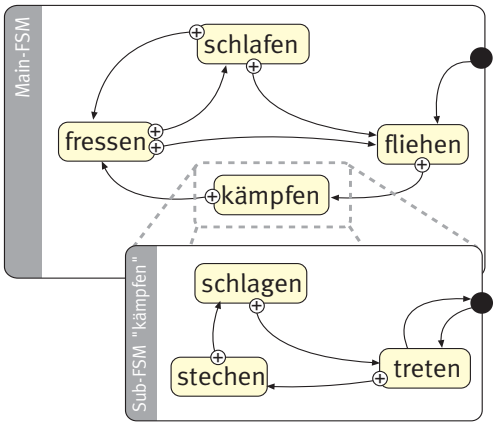


Abbildung 18: Cascaded State Machine

5.2.4 Steuerungsebene

Der Agent hat von der Planungsebene ein Ziel und einen Pfad bekommen um es zu erreichen. Auf der Steuerungsebene wird festgelegt, wie intelligent ein Agent wirkt, wenn er mit seiner Umgebung interagiert. Agenten, die ständig mit Hindernissen kollidieren, unnatürliche Bewegungen sowie nicht nachvollziehbare Manöver durchführen, können den Eindruck von Intelligenz sehr schnell zunichte machen. Es kommt also nicht nur darauf an, dass der Agent intelligent plant. Ebenso wichtig ist es, wie intelligent sich der Agent verhält. Die Aufgabe der Steuerungsebene ist dafür zu sorgen, dass der Agent sein Ziel erreicht und dabei den Eindruck von Intelligenz erzeugt.

Dazu werden Basis-Steuerungsverhalten eingesetzt, die zu komplexeren Verhaltensweisen kombiniert werden können. Diese komplexen Verhaltensweisen definieren das Verhalten des Agenten in einer bestimmten Situation. Ändert sich die Situation, so wird eine Verhaltensweise eingesetzt, die den Agenten angemessener auf die Umgebungsänderung reagieren lässt.
Die Regeln zum Einsatz der verschiedenen Verhaltensweisen in bestimmten Situationen werden auf der Planungsebene durch FSMs festgelegt. Dadurch entstehen Verhaltensmuster, die speziell an verschiedene Agenten und deren Vehikel angepasst werden können.
Jedoch liegt das Hauptaugenmerk in dieser Ebene bei den Steuerungsverhalten, die als atomare Einheiten arbeiten und wie Module zu den komplexeren Verhaltensweisen hinzugefügt werden können. Die folgenden Klassen lassen sich aus dieser Beschreibung identifizieren.

CSteeringBehaviour

Diese Klasse (*Steuerungsverhalten*) bildet die abstrakte Basisklasse für die Steuerungsverhalten. Mit der Methode `Calculate()` definiert sie eine Schnittstelle für die Berechnung der Steuerungskraft. Von ihr werden Klassen abgeleitet, die Steuerungsverhalten wie Seek, Arrival und Obstacle Avoidance implementieren. Zusätzlich besitzt diese Klasse noch eine Gewichtung, die in die Kombination der Steuerungskräfte mit einfließt. Mehr zum Thema Kombination von Steuerungskräften ist in Abschnitt 6.4.2 zu finden. Die Klasse `CSteeringBehaviour` bildet den Grundbaustein des Steuerungssystems. Jedes Steuerungsverhalten liefert eine Steuerungskraft als Rückgabewert zurück.

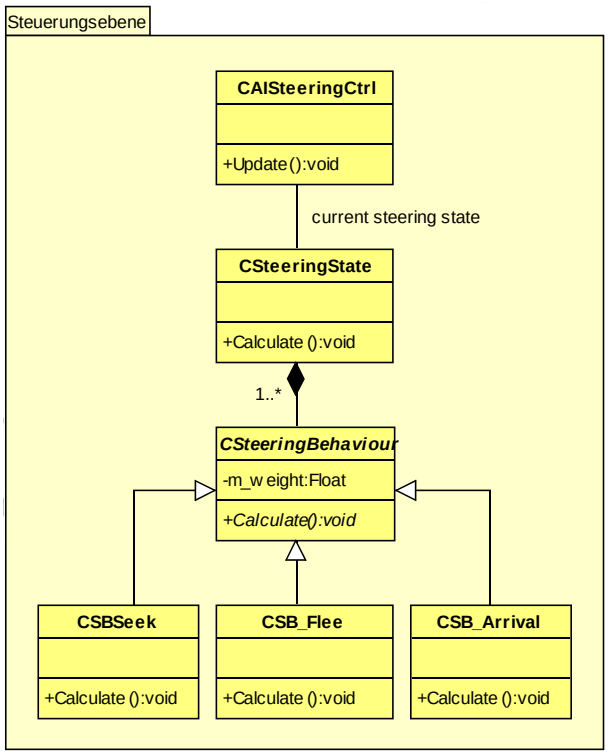


Abbildung 19: UML-Klassendiagramm der Steuerungsebene

CSteeringState

Diese Klasse (SteeringState) stellt die Zusammenfassung mehrerer Steuerungsverhalten zu einer komplexen Verhaltensweise dar. Einen SteeringState kann man sich als Berechnungsvorschrift für das Verhalten eines Agenten vorstellen, bei der die einzelnen Steuerungsverhalten die Funktionen sind. Mit dem SteeringState können beliebige Verhaltensweisen zusammengestellt werden, deren Rückgabewerte zu einer resultierenden Steuerungskraft kombiniert werden. Die Gruppierung der Steuerungsverhalten erfolgt in Containern, in denen die Reihenfolge eine Rolle spielt. Auf diese Weise ist es möglich, einzelne Verhalten gezielt zu priorisieren. Neben den beiden Hauptaufgaben, Gruppierung der Verhalten und anschliessende Kombination ihrer Steuerungskräfte, hat diese Klasse noch eine dritte Aufgabe. Sie ist dafür zuständig, häufig verwendete SteeringStates zu definieren und bei Bedarf automatisch zu erzeugen. Dafür wird das *Factory-Muster* eingesetzt.

Das Factory-Muster ist ein *erzeugendes* Entwurfsmuster, das häufig in der OOP verwendet wird. Das Muster trägt den Namen „Factory“ (Fabrik), weil es die Aufgabe hat Objekte „herzustellen“. Die Factory liefert, basierend auf den übergebenen Parametern, die Instanz einer von mehreren möglichen Klassen zurück. Im Allgemeinen sind alle Klassen, die erzeugt und zurückgegeben werden, für unterschiedliche Daten und Aufgaben spezialisiert und von einer gemeinsamen Basisklasse abgeleitet. (Vgl. [COOPER98] S. 18)
Das Muster wird unter anderem dann eingesetzt, wenn eine Basisklasse vorher nicht weiß, welche konkreten Klassen sie instanziieren soll. Weitere Informationen zu diesem Thema sind bei [GOF96] zu finden.
Bei diesem Projekt wird eine vereinfachte Version der Factory verwendet. So wird keine extra Klasse für die Herstellung von Standard-SteeringStates erzeugt. Die Aufgabe der Factory übernehmen statische Methoden, die aufgrund von Parametern (Strings, Enums) entscheiden, welche SteeringStates zu erzeugen sind. Dabei wird jeder Standard-SteeringState nur einmal erzeugt. Wenn er wiederholt angefordert wird, so wird lediglich die Referenz zu diesem SteeringState zurückgegeben. Diese kleine Optimierung wird vorgenommen, da im Bereich der Computerspiele unnötiger Overhead zu vermeiden ist.
Damit wären die drei Ebenen des Steuerungssystem fast komplett. Die Fortbewegungsebene existiert bereits, die Planungsebene ist entworfen und die Steuerungsebene ist ebenfalls in der Lage Vehikel zu steuern. Was nun noch fehlt, ist eine Verbindung zwischen den drei Ebenen. Es wird etwas benötigt, das die Ziele der Planungsebene entgegen nimmt, die Steuerungskraft von den SteeringStates berechnen lässt und diese an die Fortbewegungsebene weiterleitet. Für diese Aufgabe wird ein Controller benötigt, der für einen einzelnen Agenten die Steuerung übernimmt.

CAISteeringCtrl

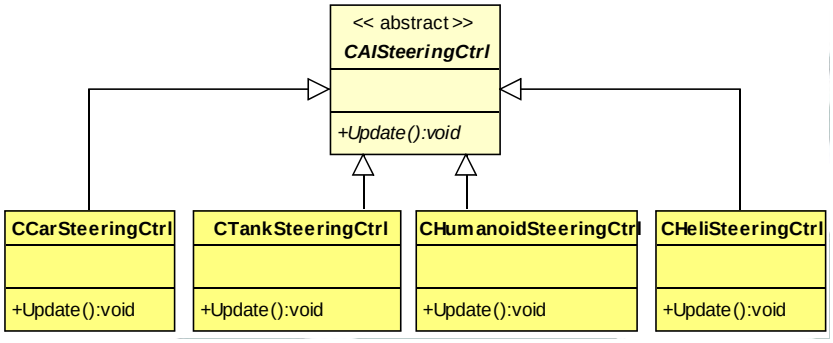


Abbildung 20: UML-Klassendiagramm der Vererbungshirarchie des AISteerringCtrl

Diese Klasse (*AISteerringController*) bildet die Datenbasis und Kommunikationsschnittstelle für die an der Steuerung beteiligten Objekte. Hier können Steuerungsverhalten benötigte Daten wie z.B Zielobjekte und Sensorreichweiten abrufen und speichern. State Machines können die Art der Fortbewegung steuern, indem die aktuellen SteeringStates gesetzt werden. Über den AISteeringController kann auf jeden Bereich des Agenten Einfluss genommen werden. Neben der Gewährleistung der Kommunikation aller Komponenten, hat der AISteeringController noch weitere wichtige Aufgaben.
Die Aktualisierung des gesamten Steuerungssytems liegt in der Verantwortung dieser Klasse. Dies geschieht in der Methode `Update()`, die jeden Frame durch die Engine-Update-Schleife aufgerufen wird. Hier werden die FSMs aktualisiert und die SteeringStates berechnet. Die finale Steuerungskraft, als Ergebnis dieser Berechnungen, wird anschließend bearbeitet und gefiltert, um Zuckungen und Störungen zu beseitigen.
Der AISteeringController ist die abstrakte Basisklasse für konkrete Implementierungen unterschiedlicher Vehikelgruppen. Wie beim BasePhysicsController existieren für jede Vehikelklasse Spezialisierungen des AISteeringControllers. So wird eine Panzer-KI von der Klasse `CTankSteeringAICtrl` bereitgestellt, Fahrzeuge auf Rädern werden durch einen `CCarSteeringCtrl` gesteuert, menschliche Charaktere durch eine `CHumanoidSteeringCtrl` usw.

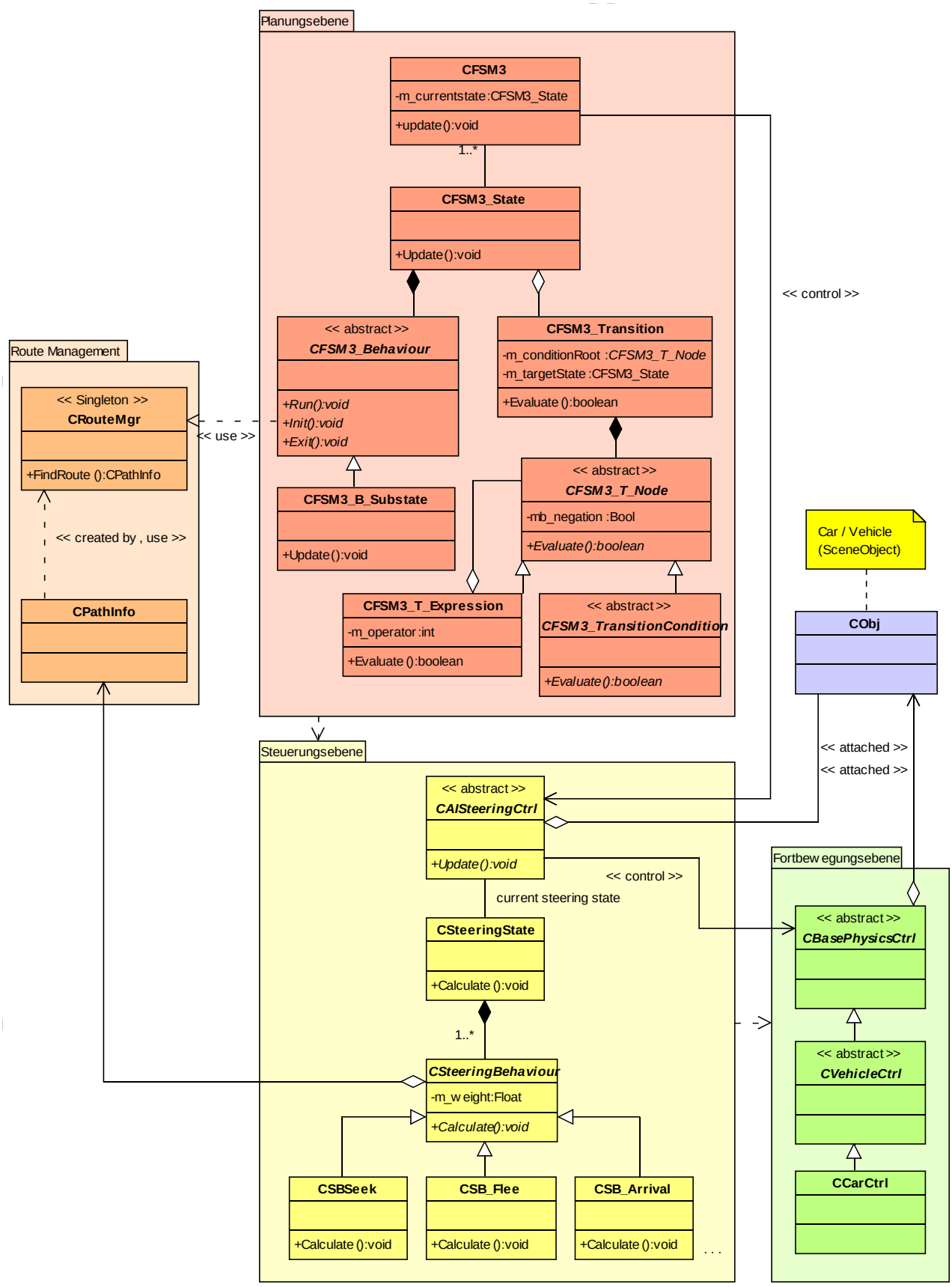


Abbildung 21: UML Klassendiagramm des gesamten Steuerungssystems

5.3 Probleme des Designs

Regelmäßig kommen Wünsche und Erweiterungsvorschläge seitens der Scripter und Leveldesigner zum Steuerungssystem. Einige dieser Wünsche können leicht erfüllt werden, andere sind etwas aufwändiger und wieder andere wiedersprechen dem Systemaufbau und machen deutlich, das es mehrere Herangehensweisen an das Thema der Vehikelsteuerung gibt. Diese Probleme sollen an dieser Stelle kurz dargestellt werden.

Die Systemarchitektur sieht die Übergabe dreier Parameter als einzige Möglichkeit für die Steuerung eines Vehikels vor. Das sind Gas- und Bremsintensität sowie Lenkrichtung. Die alleinige Beschränkung auf diese Schnittstelle führt in einigen Fällen zu Problemen. Steuerungsverhalten, die einige Steuerungswerte offen lassen sollen oder einige Werte nicht berechnen können, sind gezwungen ungültige Werte zurückzugeben. Das birgt die Gefahr unsauberer Hilfskonstruktionen, da sie bei der Kombination gesondert behandelt werden müssen.
Ein Beispiel ist das Steuerungsverhalten „VelocityAlign“, dass nur Einfluss auf die Geschwindigkeit eines Vehikels nehmen soll, jedoch nicht auf seine Steuerungsrichtung. Dieses Problem ist allerdings als eher gering zu bewerten, da die Anzahl solcher Hilfskonstruktionen überschaubar bleibt.

Die Schnittstelle führt darüberhinaus in Spezialfällen wie Zwischensequenzen zu Komplikationen. Hier ist es nahezu unmöglich, dem Vehikel derart gezielte Steuerungsbefehle zu geben, die es dazu veranlassen, vordefinierte Fahrmanöver durchzuführen. Vom Leveldesign wäre ein Spline-basierter Ansatz optimal, mit dem ein Vehikel exakt steuerbar wäre, ohne jedoch auf Features wie die Fahrphysik verzichten zu müssen. Diesen in das System einzufügen erfordert jedoch einen nicht zu unterschätzenden Arbeitsaufwand.
Ein weiteres Problem ist die fehlende Kommunikation zwischen den einzelnen Behaviours. Mitunter ist es notwendig, dass bestimmte Verhalten Daten austauschen. Das ist z.B. der Fall, wenn bei der Pfadverfolgung ein Hindernis den Weg blockiert und das Vehikel gezwungen ist auszuweichen. Das *Obstacle Avoidance*-Verhalten übernimmt ab jetzt die Steuerung und generiert eine Richtung, die das Vehikel auf dem *kürzesten* Weg aus dem Kollisionskurs bringt. Würde *Obstacle Avoidance* zu diesem Zeitpunkt wissen, was das Ziel der Pfadverfolgung war, wäre es zudem in der Lage, den *optimalsten* Weg zu berechnen. Es würde unter Beachtung des Ziels und seines Wendekreises, einen Weg wählen, der es von Hindernis wegbewegt UND seinem Ziel näher bringt. Um das zu realisieren, ist eine weitere Hilfskonstruktion nötig, die die benötigten Daten global für alle Steuerungsverhalten auf dem `CAISteeringCtrl` zur Verfügung stellt.

6. Implementation

In diesem Kapitel wird die Umsetzung des Steuerungssystems in seinen Einzelheiten beleuchtet und einige Details hinzugefügt, die im Design zu kurz kamen. Nach einer kurzen Vorstellung der Entwicklungsumgebung mit deren Hilfe diese Arbeit realisiert wurde, wird in den Kapiteln 6.2 bis 6.3 die Umgebung des Agenten und dessen Sensorik vorgestellt.
Im darauffolgenden Kapitel 6.4 werden Algorithmen, Ideen und Erweiterungsmöglichkeiten der einzelnen Steuerungsverhalten vorgestellt. In diesem Kapitel sind, neben zusätzlichen Abbildungen und Erläuterungen, auch Quellcode-Listings enthalten, die die Implementation deutlicher machen werden. Das letzte Kapitel liefert eine Schnittstellenbeschreibung der Fortbewegungsebene.

6.1 Entwicklungsumgebung

Die YAGER-Engine ist, wie fast alle Echtzeit-3D-Engines, die in Computerspielen Verwendung finden in C++ geschrieben. C++ ist objektorientiert und die Ausführungsgeschwindigkeit ist höher, als bei anderen Hochsprachen wie beispielsweise Java. Ihre Objektorientierung fördert die Modularisierung und Wiederverwendbarkeit von Softwaresystemen. Die Sprache unterstützt erweiterte Sprachkonstrukte wie *Templates* und *Operator overloading*. Mit ihr ist es möglich, für mehrere Plattformen wie die XBOX™ mit nur minimalen Codeänderungen gleichzeitig zu programmieren.
Das Klassenkonzept der Objektorientierten Programmierung spiegelt sich im Prinzip des autonomen Agenten wieder. Durch Kapselung ihrer gesamten Funktionalität, handeln Objekte quasi autonom wenn ihre Eigenschaften geändert werden sollen. Sie verwalten ihre Eigenschaften gewissermaßen selbst, da sie die Änderungen durchführen.
Microsoft Visual Studio .NET 2002™ kam wegen des sehr guten Compilers und Debuggers als Programmierumgebung (IDE) zum Einsatz. Da an der Engine bis zu acht Programmierer gleichzeitig arbeiteten, wurde *Microsoft Visual SourceSafe 6.0* für die Versionskontrolle verwendet.

6.2 Die Umwelt des Agenten

Der Agent nimmt seine Umwelt nur zweidimensional wahr, vergleichbar mit einer Art Draufsicht auf die Welt. Alle für die Steuerung des Vehikels relevanten Objekte wie Routenpunkte, Korridore und Hindernisse werden auf eine Ebene projiziert, auf der sie eine abstraktere Repräsentation ihrer selbst für die Steuerungsebene bilden. Diese Vereinfachung hat mehrere Gründe.

Zum einen wird das dreidimensionale Terrain, auf dem sich die Vehikel bewegen, aus einer zweidimensionalen Höhenkarte (Heightmap) generiert. Eine Heightmap ist eine Karte mit äquidistanten Stützstellen, die jeweils einen Höhenwert beinhalten und ist mit einer Graustufentextur vergleichbar. Dadurch kommt es nie zu einer Überlappung von Terrain (Höhlenbildung etc.). Zudem ist das, für das Vehikel befahrbare, Terrain relativ stetig und es enthält keine aprupten Höhenänderungen, da die Vehikel andernfalls steckenbleiben würden. Für diesen Zweck kann diese Höhenmap als Ebene angesehen werden, da es eine hinreichend genaue Annäherung darstellt und die dritte Dimension ausser Acht gelassen werden kann.
Zum anderen ist die Steuerung von bodenbasierten Vehikeln eine zweidimensionale Angelegenheit. Es werden nur Signale wie *links, rechts, vor* oder *zurück* verwendet, also Signale die sich auf eine Ebene (X-Z-Ebene) beziehen. Nur spezielle Vehikel wie Hubschrauber und Flugzeuge verlangen eine dreidimensionale Steuerung. Da das Steuerungssystem im Hinblick auf bodenbasierte Vehikel entworfen wurde, kann man sich diese Umstände zunutze machen, um eine optimierte Implementation zu erreichen.
Die Vorteile dieser Vereinfachung liegen auf der Hand. Im Vergleich zu einer dreidimensionalen Implementation ist eine Erhöhung der Performance zu erwarten. Das liegt sowohl an schnelleren als auch an unkomplizierteren Algorithmen, da sich Gleichungen durch den Wegfall einer Dimension teilweise stark vereinfachen.
Jedoch stellt die Vereinfachung der Steuerung als zweidimensionales Problem auch eine Einschränkung dar. Überlappungen im navigierbaren Bereich sollten von vornherein ausgeschlossen werden. Treten diese doch einmal auf z.B. unter Fahrbahnbrücken, so kann es sein, dass zwei Vehikel Kollisionen registrieren, wenn sie sich übereinander befinden. Für diese Fälle müssen spezielle Maßnahmen, z.B. die zusätzliche Betrachtung der Vehikelhöhe bei Kollisionen, getroffen werden. Die Konzentration auf bodenbasierte Vehikel würde für die Steuerung fliegender Vehikel eine Systemerweiterung notwendig machen. Das könnte durch Hinzufügen neuer Steuerungsverhalten geschehen, die nur für fliegende Vehikel zuständig sind und einen dreidimensionalen Steuerungsvektor zurückgeben. Diese Möglichkeit ergibt sich, da der Richtungsparameter der Schnittstelle zur Fortbewegungsebene dreidimensional ist.
Für die aufgeführten Nachteile existieren gangbare Lösungen, daher überwiegt der Vorteil einer erhöhten Performance, die aus einer Beschränkung auf zwei Dimensionen gewonnen wird.

6.2.1 Objekte und Hindernisse

Objekte lassen sich in die Kategorien *statisch* und *dynamisch* unterteilen. Statische Objekte wie Gebäude und Bäume werden zur Laufzeit nicht bewegt und mit ihnen ist keine Interaktion durch den Agenten möglich. Diese Objekte können bei der Steuerung eines Vehikels außer

Acht gelassen werden, da die statische Wegfindung der Planungsebene nur Wege zurückgibt, die ausserhalb des Einflussbereiches statischer Objekte liegen. Dynamische Objekte sind Objekte, die sich bewegen können (Fahrzeuge) bzw. Objekte, mit denen ein Agent interagieren kann (Kisten, Fässer, Trigger etc.).

Um auf Objekte wie Hindernisse, andere Agenten oder den Spieler reagieren zu können, benötigt der Agent im Vorfeld Informationen, welche Objekte dafür relevant sind. Das wird mit Hilfe einer Liste gewährleistet, die zu Beginn erstellt wird und alle Objekte der Spielwelt beinhaltet, die z.B ein Hindernis darstellen sollen. Zur Laufzeit können jederzeit Objekte der Liste hinzugefügt bzw. von der Liste gelöscht werden.
In der YAGER-Engine besteht fast jedes Objekt aus einem sichtbaren 3D-Modell (Mesh)[10] und hat damit äussere Dimensionen. Diese werden durch die sogenannte `WorldBoundingSphere` definiert, eine Kugel die ihren Ursprung im Objektmittelpunkt hat und das komplette Mesh des Objekts umschliesst. Projiziert man diese Kugel auf den Boden, so ergibt sich ein Kreis, der dem Agenten als zweidimensionale Repräsentation eines Hindernisses dient. Das ist der einfachste, schnellste aber auch der ungenaueste Weg, um die Dimensionen eines Objekts zu beschreiben. Bei langgezogenen Körpern, verliert dieser Wert schnell an Aussagekraft, da der Kreis sein Objekt nur noch ungenügend genau umschließt.

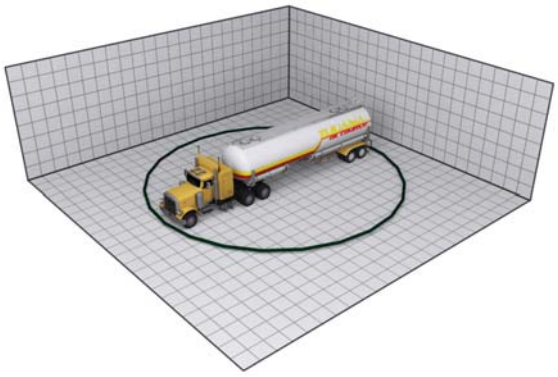


Abbildung 22: Boundingsphere bei langen Objekten

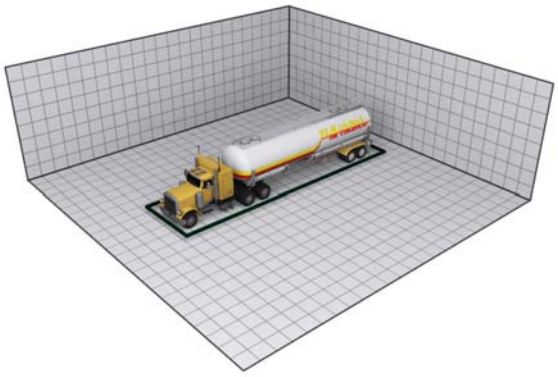


Abbildung 23: Polygone für exakte Passgenauigkeit

Polygone stellen einen präziseren Weg zur Beschreibung der Objektausmaße dar. Mit ihnen lassen sich die relevanten Konturen genau nachzeichnen. Man definiert ein Polygon, das den Umfang des Objekts in der Draufsicht beschreibt. In den meisten Fällen ist eine automatische Generierung dieser Polygone durch Projektion von bestehenden Boundingvolumes auf den Boden möglich. Lediglich bei Objekten, die nach oben hin ausladender werden wie z.B. bei Kränen und Bäumen, ist eine manuelle Erstellung nötig.

10 Ausnahmen bilden funktionale Objekte wie Trigger, Lichter oder Levelgrenzen

6.3 Sensorik

Um natürliches lebensnahes Verhalten simulieren zu können, muss auch der Bereich der Wahrnehmung simuliert werden. Auch reale Akteure reagieren auf Ihre Umwelt und verfolgen ihre Ziele. Daher muss auch der autonome Agent seine Umwelt erkennen, analysieren und seine Bewegung danach ausrichten.

Das Sehen wird in den verschiedenen Steuerungsverhalten unterschiedlich implementiert. Da bei dieser Anwendung die Performance eine wichtige Rolle spielt, werden je nach Situationsanforderung unterschiedlich komplexe Ansätze verfolgt. Im Folgenden werden nun häufig verwendete Sensormodelle vorgestellt.

6.3.1 Sichtbereich

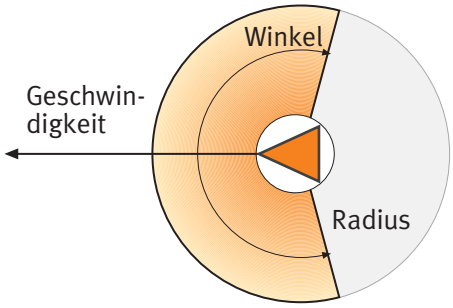


Abbildung 24: Sichtmodell eines autonomen Agenten

Der Sichtbereich eines Agenten ist durch einen Sichtradius, einen Sichtwinkel und eine Sichtachse definiert. Die Größe des Radius bestimmt die Sichtweite des Agenten. Mit dem Sichtwinkel ist es möglich, den Sichtbereich nach hinten einzuschränken bzw. dem Agenten eine volle 360° Rundumsicht zu verleihen. Die Sichtachse kann entweder entlang der Orientierung des Agenten ausgerichtet sein oder von dieser unabhängig sein, um ein Umherschauen mit dem Kopf zu imitieren.

Zur Ermittlung aller Objekte in Sichtweite des Agenten, werden alle relevanten Hindernisse, wie z.B andere Vehikel ermittelt. Bei jedem dieser Hindernisse wird überprüft, ob die Länge des Differenzvektors zwischen Agent und Hindernis kleiner als der Sichtradius ist. Ist dies der Fall, befindet sich das überprüfte Objekt in Sichtweite.

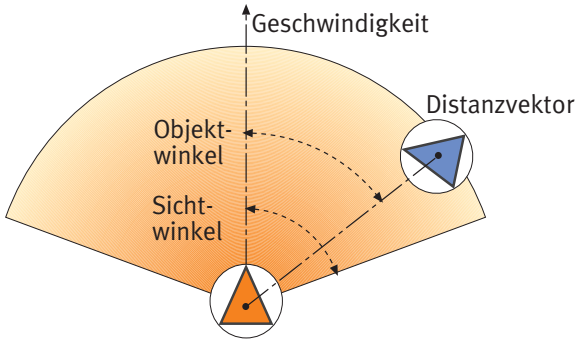


Abbildung 25: Berechnung der Objekte im Sichtbereich

Um zu ermitteln, ob sich das Objekt auch innerhalb des Sichtwinkels befindet, verwendet man das Skalarprodukt. Dabei wird überprüft, ob der halbe Sichtwinkel größer als das Skalarprodukt des Geschwindigkeitsvektors und Distanzvektors ist. Ist dies der Fall, so befindet sich das Objekt im Sichtbereich des Agenten.

6.3.2 Hitboxen und Fühlerstäbe

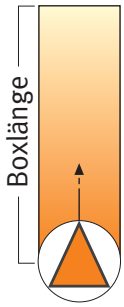


Abbildung 27: Hitbox

Abbildung 26: Fühlerstab

Während der Sichtbereich ein relativ intuitives Modell für die Sensorik eines Agenten darstellt und mit dem Sehen verglichen werden kann, sind Hitboxen und Fühlerstäbe sehr einfache Sensoren und erinnern eher an den Tastsinn.
Als Hitbox wird ein Rechteck bezeichnet, das an ein Vehikel gekoppelt ist. Sie kann unterschiedlich am Fahrzeug ausgerichtet sein und funktioniert wie ein Trigger. Damit ist es möglich, die Wahrnehmung des Agenten in eine bestimmte Richtung wie z.B seine Bewegungsrichtung zu fokussieren.

Um zu ermitteln, ob sich ein Hindernis innerhalb der Hitbox befindet, wird für alle Kanten des Rechtecks berechnet, ob das Objekt links oder rechts neben der Kante liegt. Liegt das Objekt rechts, relativ zu allen Kanten, so befindet sich das Objekt innerhalb der Hitbox. Schlägt ein Kantentest fehl, kann abgebrochen werden, da sich das Objekt mit Sicherheit ausserhalb der Box befindet.

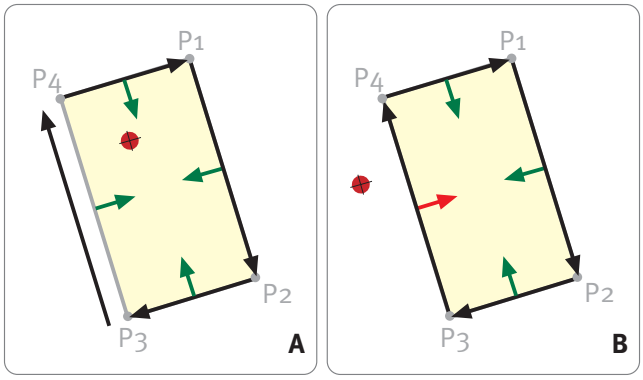


Abbildung 28: Hitbox-Test erfolgreich [A], Test schlägt bei Kante P3-P4 fehl. [B]

Ähnlich wie bei Insekten dienen Fühlerstäbe zur künstlichen Verlängerung des Körpers. Sie stellen die einfachste Form der Wahrnehmung bei autonomen Agenten dar. Hier wird ein Strahl mit einer gewissen Länge vom Vehikel ausgesandt. Damit wird überprüft, ob und wo sich an relevanten Begrenzungen (Hindernissen, Pfad-Korridorgrenzen) Schnittpunkte ergeben. Weiterhin kann mit einem Fühlerstab auch die Penetrationstiefe, sowie der Reflektionswinkel, den Winkel mit dem der Fühler von Hindernis reflektiert wird, ermittelt werden. Mit diesen Informationen ist es dem Agenten möglich, sobald er auf ein Hindernis trifft zu reagieren.

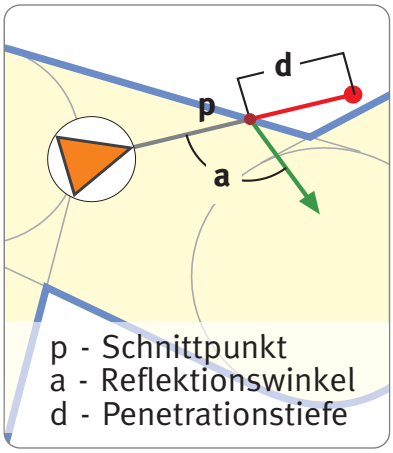


Abbildung 29: Fühlerstab Schnittest

6.4 Steuerungsebene

Das Herzstück der Steuerungsebene eines autonomen Agenten besteht in der YAGER-Engine aus dem `CAISteeringCtrl` (AI-Controller). Jeder Agent besitzt seinen eigenen AI-Controller. Der Controller arbeitet eng mit den State Machines der Planungsebene zusammen. Er stellt unter anderem Methoden und Daten zur Verfügung, die die Arbeitsweise der Steuerungsverhalten beeinflussen wie z.B. das setzen des Zielobjekts, die Größe des Aktivationsradius oder die maximale Vehikelgeschwindigkeit. Dieser „Data-Driven"-Ansatz sorgt dafür, dass Änderungen der Daten des AI-Controllers zu Änderungen im Verhalten des Agenten führen. Auf diese Weise ist es gelungen, die Anzahl der benötigten Schnittstellen zur Beeinflussung und Steuerung des Agenten auf eine zu reduzieren. Der AI-Controller hängt in der Update-Schleife der YAGER-Engine und wird jeden Frame aktualisiert.

Der AI-Controller ist eine abstrakte Klasse. Die konkreten Implementierungen unterscheiden sich für die verschiedenen Vehikelklassen (Tank, Car, Humanoid) nur geringfügig in der Art der Interpretation des finalen Steuerungsvektors. Wenn bei Fahrzeugen die Steuerungskraft nach hinten zeigt, sollen sie sich wieder korrekt ausrichten. Da sie nicht auf der Stelle wenden können, müssen sie ein Wendemanöver durchführen. Panzer und Menschen müssen das nicht tun, da sie in der Lage sind, auf der Stelle zu wenden. Darüber hinaus sollen Panzer vorsichtiger beim Verfolgen eines Pfades vorgehen, in engen Kurven stehenbleiben und auf der Stelle wenden, anstatt aus der Kurve heraus zu driften.

Die eigentliche Steuerungsarbeit wird von den Klassen `CSteeringState` und `CSteeringBehaviour` geleistet. Ein SteeringState steht für die Zusammenfassung mehrerer Steuerungsverhalten zu einem natürlich wirkenden Verhalten. Dafür wird ein *Stack* verwendet, der die Verhalten in ihrer fest gelegten Reihenfolge aufnimmt. Die Steuerungsverhalten, die für die Hauptcharakteristik der Bewegung des Agenten zuständig sind und die Bewegung generieren, werden als unterstes Verhalten auf den Stack gelegt. Verhalten, die diese Bewegung beeinflussen sollen, werden nach ihrer Priorität im Gesamtverhalten des Agenten über diesem Verhalten angelegt.

Die Klasse `CSteeringState` verwendet Factory-Methoden zum Erzeugen der Steuerungsverhalten und häufig verwendeter SteeringStates (Standard-SteeringStates). Um Speicher zu sparen, wird jeder SteeringState nur einmal erzeugt. Beim Aufruf dieser Factory-Methoden wird zuerst nachgeschaut, ob der betreffende SteeringState schon erzeugt worden ist. Ist das der Fall, wird der gespeicherte Pointer zurückgegeben. Wenn nicht, wird der SteeringState neu erstellt inklusive aller benötigter Steuerungsverhalten. Ein Beispiel für einen Standard-SteeringState ist der Arrival-State, der aus den Steuerungsverhalten Obstacle Avoidance, Separation und Arrival besteht (Listing 2).

```
static CSteeringState* CSteeringState::GetState(S_STATE_TYPE type)
{
      ...
      CSteeringState* state = new CSteeringState();
      CSteeringBehaviour* behaviour;
      switch(type)
      {
            case Y_STEERING_STATES::SS_ARRIVAL :
                  behaviour = CreateBehaviour("obstacleavoidance");
                  state->AddBehaviour(b);
                  behaviour = CreateBehaviour("separation");
                  state->AddBehaviour(b);
                  behaviour = CreateBehaviour("arrival");
                  behaviour->SetWeight(1.0f);
                  state->AddBehaviour(b);
                  break;
            ...
      }
}
```

Listing 2: Factory-Methode zum Erzeugen eines SteeringStates

Ein SteeringState ist nicht direkt mit einem Agenten verknüpft. Bei der Berechnung der Steuerungskraft wird jedesmal der Agent mit übergeben, für den die Steuerungskraft errechnet werden soll. Der SteeringState reicht diesen Agenten weiter an die einzelnen Steuerungsverhalten, die somit auf die Daten des Agenten Zugriff haben.

6.4.1 Steuerungsverhalten

Die hier vorzustellenden Steuerungsverhalten basieren auf der abstrakten Klasse `SteeringBehaviour`. Diese Klasse stellt die gemeinsame Funktionalität aller Steuerungsverhalten wie das Setzen der Gewichtung oder das Steuern auf einen Zielpunkt zur Verfügung. Zudem definiert sie die Schnittstelle für die Berechnung der Steuerungskräfte der Verhalten.

```
virtual void Calculate(Vector3D     &force,
                       float        &gas,
                       float        &brake,
                       CAICtrl*     vehicle) = 0;
```

Listing 3: Schnittstelle für die Berechnung der Steuerungsverhalten

Die Steuerungsverhalten liefern drei Werte an den `SteeringState` zurück. Die ersten drei Parameter sind Rückgabewerte der Methode. Der Parameter `force` stellt die Steuerungsrichtung dar, in die das jeweilige Verhalten steuern will; `gas` legt fest, wie stark Beschleunigt und `brake` bestimmt wie stark die Bremse betätigt werden soll. Erhält `gas` einen negativen Wert, so fährt das Vehikel rückwärts. Im Anschluss an die Kombination der der Werte, werden sie der Fortbewegungsebene übergeben.
Der vierte Parameter legt fest, für welches Vehikel die Steuerungskraft ermittelt werden soll. Über diesen Parameter erreicht man an alle für das Steuerungsverhalten relevanten Daten wie Position, Geschwindigkeit, Zielobjekt etc.

Der Großteil der Steuerungsverhalten erhält bei der Erzeugung eine statische Gewichtung, die sich nicht ändert. Das trifft auf alle Verhalten zu, die Bewegung generieren wie z.B Pathfollow und Seek. Verhalten die vorhandene Bewegung beeinflussen wie Obstacle Avoidance und Separation ermitteln dagegen ihre Gewichtung für jeden Frame selbst neu. Diese Selbstgewichtung führt dazu, dass das Verhalten sich den dynamischen Verhältnissen anpasst und je nach Bedarf mehr oder weniger Einfluss auf die finale Steuerungskraft ausübt.

Im Folgenden werden die konkreten Implementierungen der eingesetzten Steuerungsverhalten beschrieben. Im Allgemeinen sind sie entsprechend den Vorschlägen von Green und Reynolds umgesetzt worden. Bei fast allen Implementierungen waren jedoch Abwandlungen und Erweiterungen notwendig.

Das lag vor allem an dem Unterschied der Schnittstellen. Während bei Reynolds und Green nur ein Steuerungsvektor ausreichte, um das Vehikel zu steuern, werden in diesem Steuerungssystem drei Werte für die Steuerung verwendet. Richtung, Bremskraft und Beschleunigung müssen voneinander unabhängig berechnet werden. Mit diesen Werten kann das Vehikel nur indirekt gesteuert werden.
Mit dem Fahrzeugmodell von Green war es möglich, die Geschwindigkeit und Position des Vehikels direkt durch die Steuerungsverhalten zu setzen. Hierbei war die Position direkt von der Vehikelgeschwindigkeit abhängig, die durch die Länge des Steuerungsvektors abgebildet wurde. Da dieses Fahrzeugmodell die Steuerungsalgorithmen vereinfacht, werden sie in der vorliegenden Arbeit angepasst und erweitert, um mit ihnen physikalisch simulierte Fahrzeuge zu steuern. Das *Wander*-Verhalten war das einzige Steuerungsverhalten, dass in der Implementierung von Green übernommen werden konnte.

Für die erste KI-Ausbaustufe sind sechs Steuerungsverhalten notwendig. In dieser Ausbaustufe sollen die Vehikel autonom den Pfaden und Vehikeln folgen. Damit dabei der Eindruck von Intelligenz entsteht, sollen sie zusätzlich Hindernissen ausweichen, untereinander nicht kollidieren und bei Erreichen des Ziels zum Stehen kommen. Im Anschluss werden die Implementationen der dafür benötigten Verhalten (*Seek, Flee, Arrival, Pathfollowing, Obstacle Avoidance, Separation, Wander*) vorgestellt.

6.4.1.1 Seek und Flee

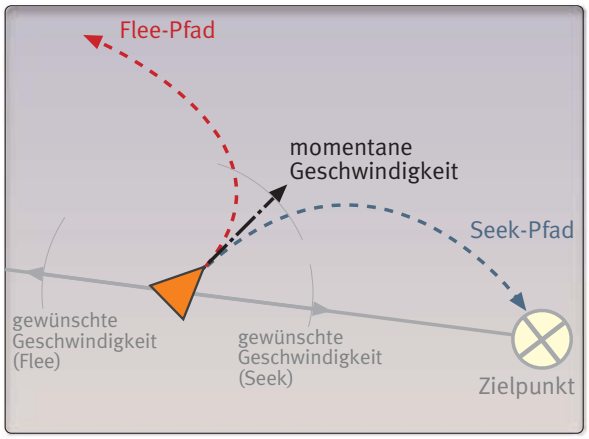


Abbildung 30: Wirkungsweise des Seek-Verhaltens bei statischen Zielpunkten

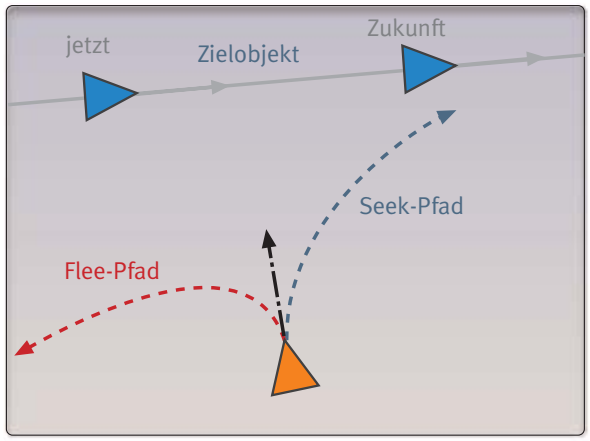


Abbildung 31: Wirkungsweise des Seek-Verhaltens bei dynamischen Zielobjekten

Das Seek-Verhalten steuert auf ein Ziel zu. Hierbei wird zwischen zwei Zielarten unterschieden: statischen und dynamischen. Während bei statischen Zielen direkt auf die Position zugesteuert wird, verwendet man bei dynamischen Zielen die zukünftige Position als Zielpunkt. Um diese zu berechnen, verwendet man die Methode der Koppelnavigation[11]. Hier wird vorausgesetzt, dass das Ziel für eine bestimmte Zeit seine Richtung und Geschwindigkeit beibehält. Die Position $\mathbf{P_{target}}$ und Geschwindigkeit $\mathbf{v_{target}}$ des Ziels müssen bekannt sein. Die zukünftige Position $\mathbf{P_{future}}$ ergibt sich aus folgender Gleichung:

$$\mathbf{P_{future}} = \mathbf{P_{target}} + \mathbf{v_{target}} * \mathbf{time}$$

Wobei $\mathbf{time}$ die Zeit darstellt, die das Vehikel mindestens benötigt um zu seinem Ziel zu gelangen oder genauer gesagt: der Abstand von der Vehikelposition $\mathbf{P_{vehicle}}$ zur Zielposition $\mathbf{P_{target}}$ geteilt durch die momentane Vehikelgeschwindigkeit.

$$\mathbf{time} = \mathbf{length}(\mathbf{P_{target}} - \mathbf{P_{vehicle}}) / \mathbf{v_{vehicle}}$$

Diese Methode stellt nur eine schrittweise Annäherung an die Zielposition zur Verfügung und die zurückgelegte Strecke des Vehikels (Seekt-Pfad siehe Abbildung 31) beschreibt einen Bogen. Es ist mit Sicherheit möglich, den exakten Punkt zu bestimmen, an dem das Ziel abgefangen werden kann. Da die vorgestellte Methode ihren Zweck jedoch erfüllt und zudem noch realistisch wirkt, besteht dazu jedoch keine Veranlassung.

11 Koppelnavigation oder (engl.) Dead Reckoning ist die Ortung eines Vehikels durch Messung von Richtung, Geschwindigkeit und Zeit. Vgl: [LARAMEE03], Grundlage jeder Navigation

Es gibt Fälle, in denen diese errechnete Position nicht geeignet ist, das Vehikel zu verfolgen. Wenn beispielsweise ein Jäger seine Beute vor sich hat und sie auf ihn zukommt. Die errechnete zukünftige Position seiner Beute wird ab einem bestimmten Abstand hinter ihm liegen, obwohl die Beute noch vor ihm ist. In diesem Fall wäre es falsch, die zukünftige Position anstelle der aktuellen Position der Beute zu verfolgen. [GREENoo] Daraus ergibt sich, dass zur Berechnung der zukünftigen Position eines Ziels auch die relativen Ausrichtungen und Positionen von Jäger und Beute zueinander eine wichtige Rolle spielen.

```
//predict target future position
Obj*        target = vehicle.GetTarget();
Vector3D    dist   = (vehicle.pos – target.pos).Length();
float       time   = (dist / vehicle.vel);
Vector3D    fpos   = target.pos + (target.vel * time);
```

Listing 4: Berechnung der zukünftigen Position eines Vehikels

Nachdem die zu verfolgende Position feststeht, wird die Richtung ermittelt, in die das Vehikel steuern soll. Die Steuerungsrichtung ergibt sich aus dem Differenzvektor zwischen der Vehikelposition und der Zielposition. Für das Flee-Verhalten wird dieser Vektor invertiert.
Da die beiden Verhalten stets mit voller Geschwindigkeit einem Ziel entgegen bzw. von diesem weg steuern, wird dabei immer die grösstmögliche Beschleunigung angenommen und nicht abgebremst.

```
    [SEEK]      force       = (vehicle.pos – targetpos);
    [FLEE]      force       = (targetpos – vehicle.pos);
                gas         = 1.0f;
                brake       = 0.0f;
```

Listing 5: Seek Algorithmus

Probleme

Beim Erreichen des Zielpunktes schiesst ein Vehikel, das mit dem Seek-Verhalten angetrieben wird, unweigerlich über das Ziel hinaus. Der Steuerungsvektor wirkt plötzlich in die entgegengesetzte Richtung. Das Vehikel wendet, um wiederum über das Ziel hinaus zu schiessen und bewegt sich schliesslich in elliptischen Bahnen um den Zielpunkt herum, ähnlich einer Motte, die um das Licht kreist.

6.4.1.2 Arrival

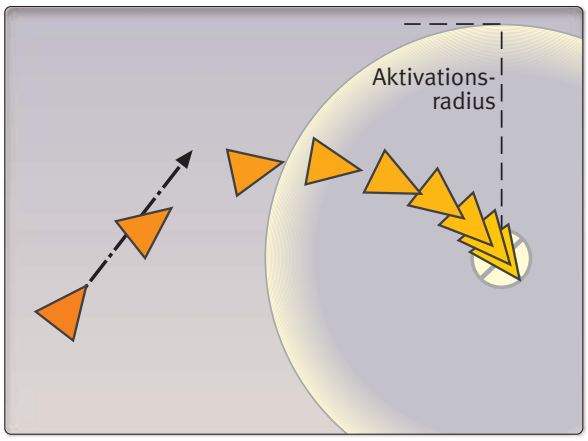


Abbildung 32: Wirkungsweise des Arrival-Verhaltens

Arrival ist eine Erweiterung des Seek-Verhaltens. Die Bestimmung der Steuerungsrichtung erfolgt wie bei Seek. Arrival behebt jedoch das Problem des Überschwungverhaltens, wie man es bei Seek beobachten kann. Hier wird zusätzlich die Bremskraft und die Beschleunigung berechnet, die benötigt wird, um das Vehikel auf dem Zielpunkt zum Stehen zu bringen. Zur Berechnung wird der Aktivationsradius dieses Verhaltens benötigt. Er bestimmt wann abgebremst werden soll. Befindet sich das Vehikel ausserhalb dieses Aktionsradius, so wird analog zum Seek-Verhalten verfahren.

Sobald das Vehikel diesen Radius überquert, wird die benötigte Beschleunigung $\mathbf{f}$, basierend auf den Geschwindigkeiten des Vehikels $\mathbf{v_{vehicle}}$ und des Ziels $\mathbf{v_{target}}$, der Masse des Vehikels $\mathbf{m_{vehicle}}$ und des Abstandes des Vehikels zum Ziel $\mathbf{dist}$ berechnet. Die Formel dafür lautet wie folgt:

$$\mathbf{f = m_{vehicle} * (v_{target}^2 - v_{vehicle}^2) / (2 * dist)}$$

Sobald das Vehikel schneller als sein Ziel ist, wird eine negative Beschleunigung errechnet, die die Bremskraft darstellt. Umgekehrt werden positive Beschleunigungen errechnet, sobald das Ziel eine höhere Geschwindigkeit besitzt als das Vehikel. Zuletzt wird die errechnete Kraft auf die maximale Brems- bzw. Beschleunigungskraft des Vehikels beschnitten. Listing 6 zeigt die vereinfachte Implementierung des Arrival Verhaltens.

```
float dist = (target.pos – vehicle.pos).Length();
if (dist > vehicle.GetArrivalRadius())
{
    //calculate force (acc and brake)
    force = vehicle.mass * ((vehicle.vel * vehicle.vel) –
                            (target.vel * target.vel) / (2 * dist);
    //...clamp force and distribute to gas and brake ...
}
```

Listing 6: Arrival Algorithmus

6.4.1.3 Pathfollowing

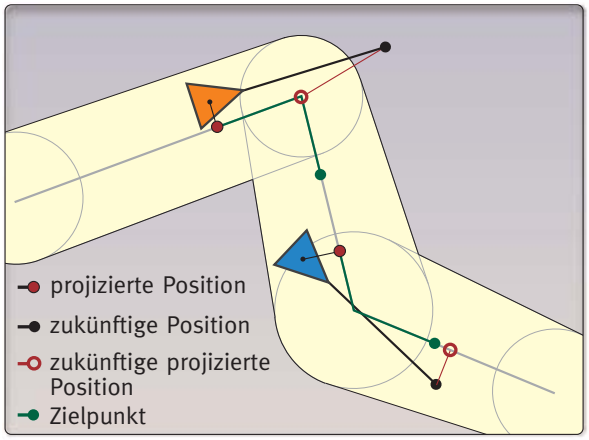


Abbildung 33: Pfadverfolgung mit Rückprojektion

In der Literatur sind einige Lösungsvorschläge für die Implementation eines Verhaltens, das einem Pfad folgt beschrieben. [GREENoo] [REYNOLDS99]. Die *Rückprojektion* (Reprojection) ist die stabilste und zuverlässigste aber auch die aufwändigste Methode. Bei diesem Verfahren wird die Weltposition des Vehikels einer Postion auf der Pfadlinie und damit einer Länge entlang des Pfades zugeordnet. Der umgekehrte Fall, bei dem eine Pfadlänge auf einen Punkt im Weltkoordinatensystem projiziert wird, findet ebenfalls Anwendung. Die Projektionsverfahren werden etwas später beschrieben.

Um zu erreichen, dass das Vehikel die gesamte Fläche des Korridors für seine Fortbewegung nutzt und sich nicht auf der direkten Pfadlinie bewegt, geht man in zwei Schritten vor:

Zunächst wird geprüft, ob die Steuerungsrichtung korrigiert werden muss. Wenn das nicht notwendig ist, wird nichts unternommen. Das Vehikel ist auf Kurs und die bisherige Steuerungsrichtung wird beibehalten. Besteht Steuerungsbedarf, wird ein passender Punkt berechnet, auf den das Vehikel zusteuern soll.

1. Besteht Steuerungsbedarf?

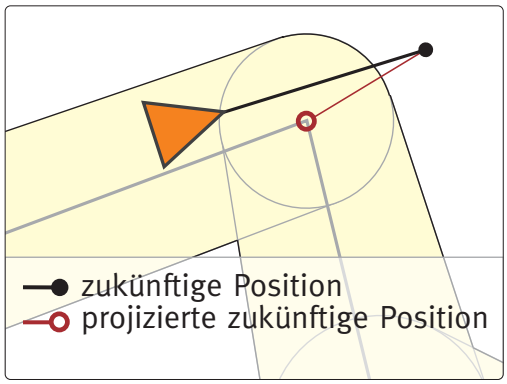


Abbildung 34: Ermittlung des Steuerungsbedarfs bei der Pfadverfolgung

Es besteht Steuerungsbedarf, wenn das Vehikel in naher Zukunft den Pfad verlassen wird. Dazu wird zunächst die *zukünftige Position* Berechnet. Die Zeitspanne, mit der diese Position geschätzt werden soll, ist hier ein Parameter, der das Gesamtverhalten beeinflusst. Ist er zu groß, wird sehr häufig nachkorrigiert. Ist er zu klein, steuert das Vehikel, meist zu spät, erst unmittelbar vor dem Verlassen des Pfades entgegen.

Um zu ermitteln ob die zukünftige Position außerhalb des Pfades liegt, gibt es mehrere Möglichkeiten. Erstens die Kontrolle ob sich der Pfadkorridor und die Strecke zwischen der Vehikelposition und zukünftiger Position schneiden. Existiert ein Schnittpunkt, liegt die zukünftige Position in der Regel ausserhalb des Pfades. Bei

dieser Methode sind jedoch unzählige Spezialfälle zu berücksichtigen, weswegen der folgenden Methode der Vorzug gegeben wurde.

Mittels eines „Nearest-Point-on-Line"-Algorithmus[12] wird die zukünftige Position auf den Pfad projiziert. Anschliessend wird die Länge des Vektors zwischen zukünftiger und *projizierter zukünftiger Position* ermittelt. Ist diese Länge größer als die halbe Korridorbreite, befindet sich die zukünftige Position ausserhalb des Pfades und es besteht Handlungsbedarf.

2. Berechnung der neuen Steuerungsrichtung

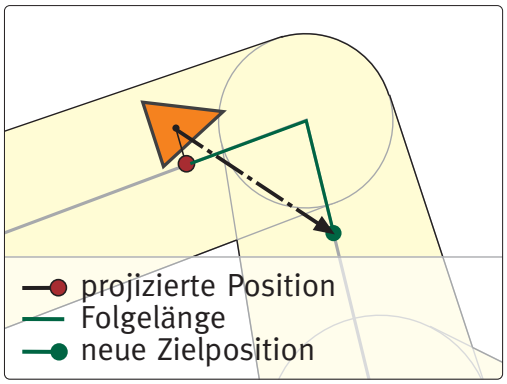


Abbildung 35: Berechnung der neuen Zielposition bei der Pfadverfolgung

Nun muss die neue Steuerungsrichtung das Vehikel auf den Pfad zurückbringen. Dazu wird die Vehikelposition auf den Pfad projiziert. Das geschieht wieder durch den „Nearest-Point-on-Line"-Algorithmus. Dieser *projizierten Position* ordnet man eine Länge auf dem Pfad zu. Zu dieser Länge wird die *Folgelänge* addiert. Die Folgelänge bestimmt, wie stark Kurven geschnitten werden bzw. wie „nachlässig" die Pfadverfolgung wirkt. Ist sie zu kurz, orientiert sich das Vehikel sehr stark an der Pfadmittellinie. Ist sie zu lang, kann es passieren, dass das Vehikel stark abkürzt und den Pfad verlässt. Der Gesamtlänge wird ein Punkt im Weltkoordinatensystem zugeordnet. Er bildet die *neue Zielposition*. Die Berechnung dieses Punktes erfolgt für jeden Frame aufs neue, solange ein Steuerungsbedarf besteht.

Im Folgenden werden die Vorgehensweisen bei der Zuordnung von Punkten zu Pfadlängen und umgekehrt beschrieben.

Zuordnung von Pfadlängen zu Punkten

(`MapPathDistanceToPoint`)

Es wird über alle Segmente[13] des Pfades iteriert, bis das entsprechende Segment S gefunden wurde, in dem die Länge endet. Dabei werden stets die Segmentlängen von der Gesamtlänge subtrahiert. Die verbleibende Restlänge ist Teil der Länge von S und wird einer Zahl (Faktor) zwischen 0.0 und 1.0 zugeordent. Um den gesuchten Punkt auf dem Segment zu errechnen, wird der Vektor von Segmentstart zu Segmentende mit dem Faktor skaliert.

12 Mehr dazu bei D. Eberly, www.magicsoftware.com

13 Ein Pfadsegment ist die Strecke zwischen zwei benachbarten Pfadpunkten

```
Vector3D  MapPathDistanceToPoint(float distance)
{
      for (int i=1; i<path.GetSegmentCount(); i++)
      {
            slength = path.GetSegmentLength(i-1);
            if (slength < distance)
            {
                  distance -= slength;
            }
            else
            {
                  REAL ratio = distance / slength;
                  Vector3D p1 = path.GetPointPos(i-1);
                  Vector3D p2 = path.GetPointPos(i);

                  return (((p2 - p1) * ratio) + p1);
            }
      }
}
```

Listing 7: Zuordnung von Pfadlängen zu Punkten im Weltkoordinatensystem

Zuordnung von Punkten zu Pfadlängen

(`MapPointToPathDistance`)

Der Punkt wird auf jedes Pfadsegment projiziert. Dabei wird die Länge (*length*) des Segments und der Abstand (*dist*) zwischen Punkt und projiziertem Punkt zurückgegeben. *Length* wird solange zur Gesamtlänge addiert bis das Segment gefunden wurde, bei dem *dist* minimal war. Der Gesamtlänge fehlt nun noch die letzte Strecke auf diesem Segment. Bei der Projektion wird eine Ratio berechnet, die auf einer Skala von 0.0 bis 1.0 angibt, wo der projizierte Punkt auf der Strecke von Segmentstart bis Segmentende liegt. Abschließend wird der Vektor zwischen Start und Ende des Segments normalisiert und mit der Ratio skaliert. Die Länge dieses Vektors wird danach zur Gesamtlänge addiert.

```
float  MapPointToPathDistance(Vector3D point, Vector3D proj)
{
      float pathDistance, lengthsum, length = 0.0f;
      float mindist = FLT_MAX;
      for (int i=1; i<path.GetSegmentCount(); i++)
      {
            Vector3D p1 = path.GetPointPos(i-1);
            Vector3D p2 = path.GetPointPos(i);

            dist = ProjectPointOnSegment(point, p1, p2, proj, length);
            lengthsum += length;
            if (dist < mindist)
            {
                  mindist = dist;
                  pathDistance = lengthsum + length;
            }
      }
      return pathdistance;
}
```

Listing 8: Zuordnung von Punkten im Weltkoordinatensystem zu Längen auf dem Pfad

Mit dieser Implementation ist es möglich, eine robuste Pfadverfolgung zu realisieren. Es treten jedoch beim durchqueren der Kurven einige zusätzliche Probleme auf, die in dieser Implementation unberücksichtigt blieben. Da Vehikel normalerweise mit konstanter Geschwindigkeit ihrem Pfad folgen, besteht die Gefahr aus Kurven, an denen der Pfad schnell seine Richtung ändert, heraus getragen zu werden und somit den Pfad zu verlassen. In solchen Fällen muss das Vehikel rechtzeitig abbremsen. Dieses Verhalten wird mit einem Kurvensensor realisiert, dessen *Kurvenfühler* den Pfad vor dem Vehikel abtastet.

Um zu kontrollieren, ab wann und wie stark gebremst werden soll, definiert man zwei Winkel. Winkel a ist nach vorne gerichtet und gibt eine Toleranz an, in der nicht gebremst wird. Winkel b ist nach hinten gerichtet und gibt an, wann die maximale Bremskraft erreicht ist. Der *sensible Bereich* zwischen den Winkeln a und b gibt das Intervall an, in dem zwischen minimaler und maximaler Bremskraft interpoliert wird (Abbildung 36A). Der Ankerpunkt des Kurvenfühlers wird vom Vehikel ausgehend, um einen bestimmten Betrag (*Fühlerlänge*) den Pfad entlang verschoben. Der Winkel, der sich zwischen der Ausrichtung des Vehikels und dem Kurvenfühler bildet wird ständig neu gemessen. Liegt er innerhalb des sensiblen Bereichs, wird das Vehikel abgebremst (Abbildung 36B).

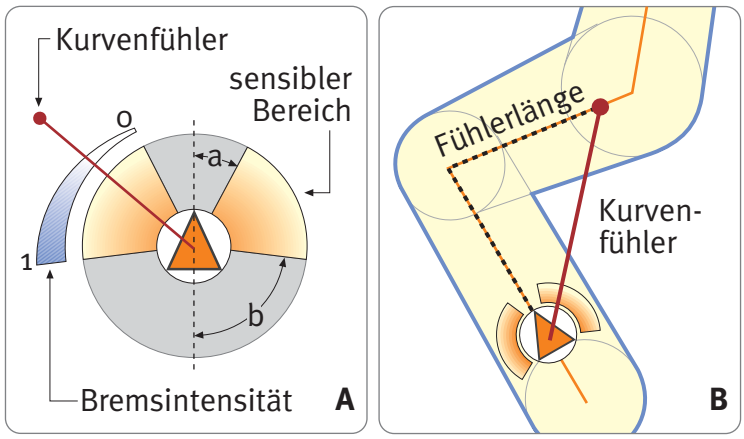


Abbildung 36: Kurvensensor

Damit das Vehikel bei allzu scharfen Kurven durch Bremsen und wegen seines eingeschränkten Wendekreises nicht zum Stehen kommt, muss zusätzlich eine minimale Kurvengeschwindigkeit angegeben werden, die nicht unterschritten werden darf.

Probleme

Probleme können auftreten, wenn sich der Pfad selbst kreuzt (Abbildung 37B) oder der Zielpunkt (`vtarget`) ein unüberwindbares Hindernis durchdringt (Abbildung 37A). Zudem ist darauf zu achten, dass Vehikel zu Beginn einer Pfadverfolgung auch zum Pfad entgegengesetzt ausgerichtet sein können. Das Verhalten muss also immer Zielpunkte berechnen, die das Vehikel auch in die gewünschte Richtung führen. Es ist ebenso wahrscheinlich, dass Vehikel aufgrund äusserer Einflüsse einige Punkte auf dem Pfad nicht erreichen können. Dabei muss trotzdem sichergestellt sein, dass das Vehikel das Pfadende (Zielpunkt) erreicht.

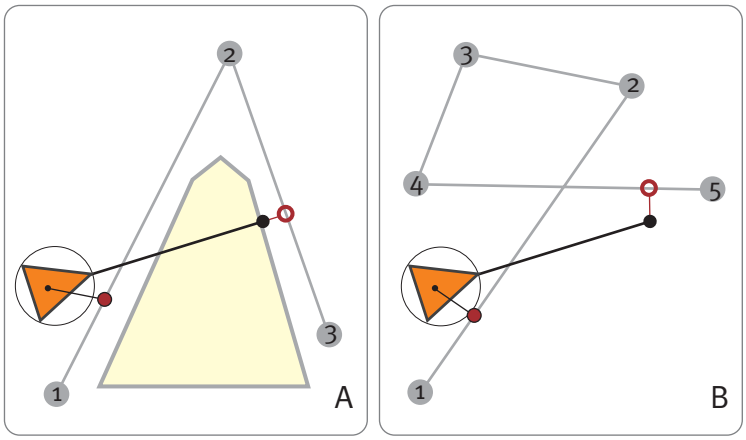


Abbildung 37: Pfadprobleme

6.4.1.4 Obstacle Avoidance

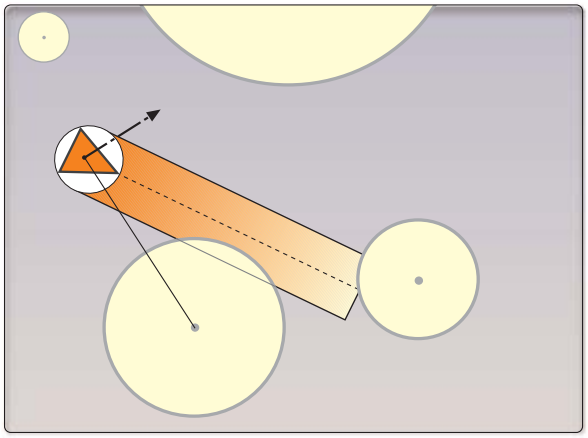


Abbildung 38: Hindernissen ausweichen

Obstacle Avoidance ist das Verhalten, das einem Vehikel ermöglicht dynamischen Hindernissen auszuweichen. Dabei werden nur Objekte als Hindernisse angesehen, die direkt die Bewegungsrichtung des Vehikels kreuzen. Der daraus resultierende Korridor wirkt wie ein Tunnelblick, der in der Implementation dieses Verhaltens als spezielle Hitbox dargestellt wird, das vor dem Vehikel angebracht wird. Diese Hitbox ist so breit wie das Vehikel (`width`) und um ein Vielfaches länger. Die Länge (`length`) steigt proportional mit der Geschwindigkeit. Objekte, die die Hitbox berühren oder schneiden, werden als Hindernis erkannt. Hindernisse sind in dieser Implementation durch einen Punkt mit einem Radius (`obstacle_radius`) abstrahiert. Das Ziel dieses Steuerungsverhaltens ist, die Hitbox stets frei von Hindernissen zu halten. Dazu geht man in zwei Stufen vor. Die erste ermittelt das Hindernis, von dem die größte Kollisionsgefahr ausgeht. In der zweiten Stufe wird eine Kraft ermittelt, die das Vehikel vom Kollisionskurs abbringt.

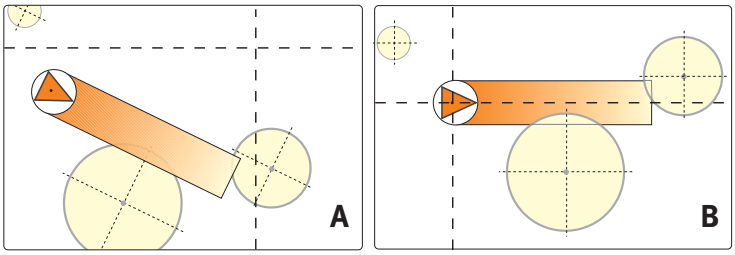


Abbildung 39: Transformation der Hindernisse vom Weltkoordinatensystem [A] zum lokalen Vehikelkoordinatensystem [B]

Um das „gefährlichste“ Hindernis zu ermitteln, werden alle potentiellen Hindernisse in der unmittelbaren Umgebung untersucht und in eine Liste eingetragen. Die Positionen der Listenobjekte werden ins lokale Vehikel-Koordinatensystem transformiert (siehe Listing 9). Damit befindet sich das Vehikel im Ursprung des Koordinatensystems und die Hindernisse sind an den Achsen ausgerichtet, was die weitere Berechnung vereinfacht und beschleunigt. Von der Hindernisposition wird die Vehikelposition subtrahiert und diese neue Position mit der inversen Welt-Rotationsmatrix des Vehikels multipliziert. Dadurch wird das Hindernis gewissermaßen zurückgedreht.

```
Vector3D transformToLocal(Obj* vehicle, Obj* obstacle)
{
      CMatrix3x3  rotMatrix_vehicle = vehicle->GetRotMatrix();
      Vector3D    obstaclePos_local = (obstacle->pos – vehicle->pos);

      rotMatrix_vehicle = rotMatrix_vehicle.Inverse();
      obstaclePos_local = rotMatrix_vehicle * obstaclePos_local;

      return  obstaclePos_local;
}
```

Listing 9: Obstacle Avoidance, Koordinatentransformation

Nachdem die lokale Position berechnet wurde, können jene Objekte ausgesiebt werden, die keine Hindernisse sein werden. Anstatt diese Objekte mit Rechteck-Kreis Schnittpunkttests zu untersuchen, erweitert man den Objektradius um die Rechteckbreite (`extendedradius`), was die Berechnungen vereinfacht. Es genügt ein Linien-Kreis Schnittpunkttest und anhand dreier Fallunterscheidungen kann ermittelt werden, ob ein Objekt sich im Korridor des Vehikels befindet.

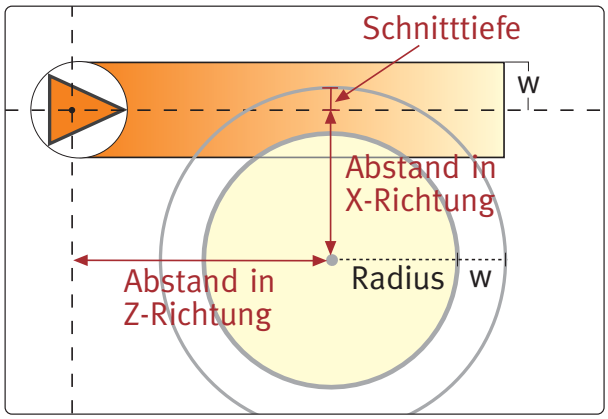


Abbildung 40: Radiuserweiterung

Fall 1: Ein Objekt befindet sich hinter dem Vehikel.

Die x-Komponente seiner Position ist negativ. Es kann allerdings der Fall auftreten, dass ein Objekt hinter dem Vehikel, der Radius jedoch vor dem Vehikel liegt. In diesem Fall befindet sich das Vehikel direkt im Hindernis.

Fall 2: Ein Objekt liegt in weiter Entfernung vor dem Vehikel.

Objekte, die weiter entfernt sind, als die Länge des Rechtecks addiert mit dem Objektradius, können vernachlässigt werden.

Fall 3: Ein Objekt befindet sich zu weit links bzw. rechts.

Objekte, deren erweiterter Radius subtrahiert vom Betrag der y-Position größer als null ist, liegen ausserhalb des Rechtecks.

Sofern nach diesem Test noch mehrere Hindernisse in der Liste verbleiben, wählt man das nächstliegende aus, da es die größte Gefahr darstellt. Listing 10 zeigt die Implementation dieser Vorgehensweise.

```
Container* cnt              = world->GetObstacleContainer();
float      mindist          = FLT_MAX;
Obj*       nearest_obstacle;

for (int i = 0; i < cnt->size; i++)
{
     Obj*        obstacle    = cnt->GetObject(i);
     Vector3D    o_pos       = transformToLocal(vehicle, obstacle);
     float       v_radius    = vehicle->GetRadius()
     float       o_radius    = obstacle->GetRadius();
     float       o_expradius = o_radius + width;

     //test if obstacle is inside the box
     if (( o_pos.Z() < 0.0f ) ||
         ((o_pos.Z() - o_radius) > length ) ||
         ((abs(o_pos.X()) - o_expradius) > 0.0f))
     {
           continue;                                  //outside
     }
     else                                             //inside
     {
           //if closest -> keep the current data
           dist = o_pos.Length();
           if (dist < mindist)
           {
                 mindist          = dist;
                 nearest_obstacle = obstacle;
           }
     }
}
force = Avoid(nearest_obstacle);
```

Listing 10: Obstacle Avoidance, Ermittlung des Hindernisses mit dem geringsten Abstand

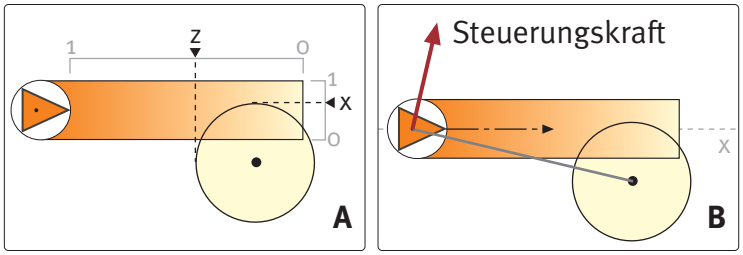


Abbildung 41: Ermittlung der Penetrationstiefe [A], Ermittlung der Steuerungsrichtung [B]

Ist ein Hindernis ermittelt, muss nun die Kraft berechnet werden, die das Vehikel von diesem Hindernis weg bewegt. Der schnellste Weg das zu erreichen besteht darin, im rechten Winkel vom Hindernis weg zu steuern (Abbildung 41B). Demnach würde ein Vehikel das ein Hindernis bemerkt das „Steuer herumreissen“ und abrupt davon weg steuern. Das ist jedoch ein äußerst unnatürliches Verhalten. In der Realität werden Hindernisse mit leichten und frühen Richtungsänderungen umfahren.
Um das Verhalten lebensnaher zu gestalten, benötigt das Vehikel Informationen darüber, wie dringend die Richtungsänderung ist. Die Dringlichkeit ist direkt davon abhängig, wie nah das Hindernis dem Vehikel ist. Ist das Hindernis in weiter Ferne, genügt ein leichtes Ausweichen. Je näher das Hindernis kommt, desto heftiger fällt die Reaktion aus. In der Implementation wird die Penetrationstiefe des Hindernisses mit der Hitbox in X- und Z-Richtung ermittelt (Abbildung 41A). Praxistests haben gezeigt, dass das Produkt der beiden Penetrationstiefen einen brauchbaren Wert für die Dringlichkeit eines Ausweichmanövers ergeben.

Um zu erreichen, dass das Vehikel sanft seine Richtung ändert, setzt man die Verhaltensgewichtung der ermittelten Dringlichkeit gleich. Somit verfolgt das Vehikel weiterhin seinem Pfad, während das 90 Grad-Ausweichmanöver nur zu einem Bruchteil in die kombinierte Steuerungskraft aller Steuerungsverhalten eingeht. Obstacle Avoidance ist ein Verhalten, das seine eigene Wichtigkeit im Gesamtverhalten des Vehikels selbst ermittelt.

Probleme

Probleme können sich ergeben, wenn Obstacle Avoidance mit weiteren, die Bewegung beeinflussenden Verhalten wie Separation und Containment, kombiniert wird. Aufgrund der vielen gegensätzlichen Kräfte kann die resultierende Bewegung unruhig sein.

Listing 11 zeigt die Berechnung der Steuerungsrichtung (`force`), die Ermittlung der Penetrationstiefen (`pen_x`, `pen_z`) und die anschliessende Berechnung der Dringlichkeit (`urgency`).

```
Vector3D Avoid(CObj* obstacle)
{
     //work out the direction the normal is facing
     float angle = -sign(obstacle.pos.x) * 90.0f;

     //build the normal vektor
     Vector3D dir = (obstacle.pos – vehicle.pos);
     Matrix2x2 rotMatrix(angle);

     //set force direction
     force       = rotMatrix * dir;

     //work out penetration depth in x and z-direction
     float pen_x = (abs(obstacle.pos.x)–obstacle.radius)/(width * 0.5f);
     float pen_z = (abs(obstacle.pos.z)-obstacle.radius–vehicle.radius)/
                   length;

     //clamp values between 0.0f and 1.0f
     clamp(pen_x);
     clamp(pen_z);

     //calculate urgency
     float urgency = pen_x * pen_z;

     SetWeight(urgency);
     return force;
}
```

Listing 11: Obstacle Avoidance, Berechnung der Gewichtung und Steuerungsrichtung

6.4.1.5 Wander

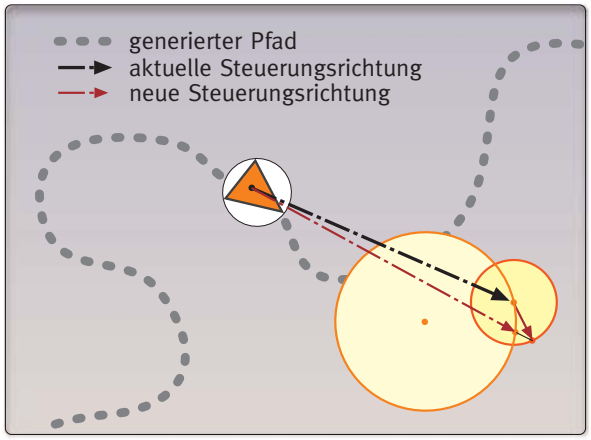


Abbildung 42: Funktionsweise des Wander-Verhaltens

Wenn ein Vehikel mit Wander angetrieben ist, wird der Vorwärtsbewegung ein gewisses Maß an Zufälligkeit hinzugefügt. Es gibt mehrere Möglichkeiten dies zu erreichen. Die einfachste wäre, in jedem Frame einen Zufallsvektor zur aktuellen Geschwindigkeit zu addieren. Die resultierende Bewegung ist allerdings alles andere als natürlich, da sie nicht an ziellos Umherwandern erinnert, sondern an unkontrolliertes Zittern. Stetige Richtungsänderungen sind damit nicht möglich. Die Methode, die sich für diese Aufgabe als robust, schnell und kontrollierbar erwiesen hat, funktioniert wie folgt:

Vor dem Vehikel wird ein Kreis (`circle`) erstellt. Zur Initialisierung sucht man sich einen Zufallsvektor, dessen x- und z-Komponenten im Bereich von +1.0 bis -1.0 liegen. Diesen projiziert man folgendermaßen auf den Kreis: Der Vektor wird normalisiert und mit dem Radius des Kreises multipliziert. Als Zielpunkt des Verhaltens wird der ermittelte Punkt auf dem Kreis genommen. In der nächsten Iteration addiert man einen kleinen Zufallsvektor zu diesem Punkt, projiziert ihn wieder auf den Kreis und wählt den ermittelten Punkt als Zielpunkt aus und steuert auf ihn zu.

Dieses Verhalten ist leicht kontrollier- und parametrisierbar. Die Größe des Kreises (`big_radius`) beeinflusst die Spanne der möglichen Richtungswinkel. Ein kleiner Kreis bedingt eine verhältnismäßig gerichtete Bewegung (B). Ein großer Kreis bewirkt große Richtungsänderungen (D). Die relative Position des Kreises

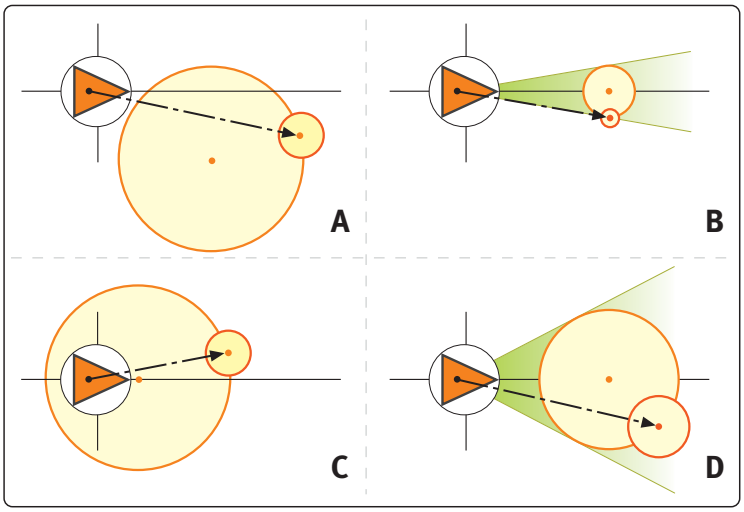


Abbildung 43: Beispiele für Wander-Verhalten

zum Vehikel bestimmt die Ausrichtung der Bewegung: ob rückwärts gerichtete Steuerungsvektoren generiert werden (C) oder ob das Vehikel immer in eine bestimmte Richtung zieht (A). Die Größe des Zufallsvektors (`small_radius`) bestimmt den Grad der Richtungsänderung. Listing 12 zeigt die Implementation des Wander-Verhaltens.

```
//build random vector
float rx = ((rand() / RAND_MAX) * 2.0f) - 1;
float ry = ((rand() / RAND_MAX) * 2.0f) - 1;

Vector3D rand_vector;
rand_vector = Vector3D( rx, 0.0f, ry );
rand_vector.Normalize();
Vector3D circle_pos =  vehicle->GetWanderDirection();

//initialize – circle_pos never set
if (circle_pos == Vector3D::ZERO)
{
      //projection at circle
      rand_vector *= big_radius;
      vehicle->SetWanderDirection(rand_vector);
}
else
{
      //add small offsets to circle_pos
      rand_vector *= small_radius;
      circle _pos += rand_vector;
      circle _pos.Normalize();

      //projection at circle
      circle _pos *= big_radius;
      vehicle->SetWanderDirection(circle_pos);
}

//Convert to World Coordinates and apply force
circle _pos = Convert2WorldCoords(circle _pos);
force = this->SteerForSeek(o_pos, circle _pos);
```

Listing 12: Wander Algorithmus

6.4.1.6 Separation

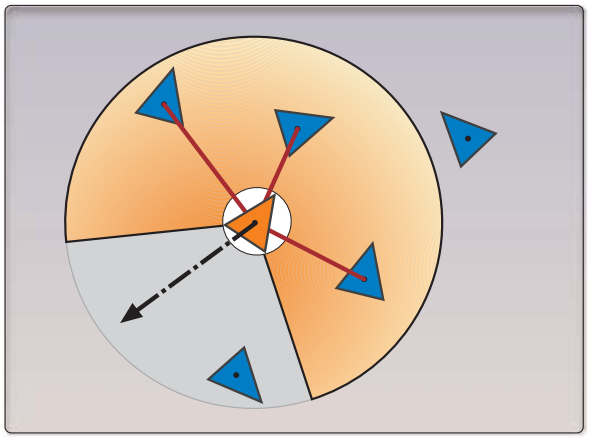


Abbildung 44: Funktionsweise des Wander-Verhaltens

Separation bewirkt, dass Objekte stets einen gewissen Mindestabstand zueinander wahren und den zur Verfügung stehenden Platz besser ausnutzen. Mit Separation ist es möglich, mehreren Vehikeln dasselbe Ziel zu geben, ohne das diese ineinander stecken, wenn sie ankommen. Das Halten des Abstandes wird mit Hilfe von Abstoßungskräften realisiert, die zwischen den Vehikeln wirken. Das funktioniert ähnlich der elektrostatischen Abstoßung von Teilchen. Die Abstoßungskräfte haben die Form einer $1/x^2$ Funktion. Damit ist gewährleistet, dass sich Objekte nur leicht abstossen, wenn sie sich berühren, aber mit Macht auseinander springen, wenn sie ineinander stecken. Zur Berechnung der Abstoßungskraft geht man wie folgt vor:

Zuerst werden alle Objekte (Vehikel und Hindernisse) in der Nachbarschaft des Vehikels ermittelt. Für diese Objekte (`target`) wird jeweils die Richtung (`repeldir`) und die Intensität (`strength`) der Abstoßungskraft berechnet. Die Abstoßungsrichtung bildet der Vektor vom Hindernis zum Vehikel. Sie zeigt also vom Hindernis zum Vehikel. Diese Abstoßungsrichtungen werden für alle Hindernisse zusammenaddiert. Das bewirkt eine resultierende Abstoßungsrichtung, die von allen Hindernissen wegzeigt. Die Intensität der Abstoßungskraft bildet sich aus der maximalen Abstoßungskraft, die von diesen Hindernissen ausgeht. Dazu wird die Kraft für jedes Hindernis berechnet und die größte Kraft als Abstoßungsintensität des Verhaltens verwendet. Dazu wird zunächst die Distanz x ermittelt, mit der das Objekt im Hindernis steckt. Der Abstand der beiden Objekte zueinander, geteilt durch die Summe ihrer Radien, ergibt diese Distanz, die in die Formel $1/x^2$ eingesetzt, die Abstoßungskraft ergibt.

Nachdem Richtung und Intensität ermittelt wurden, wird die Richtung als Steuerungskraft, und die Intensität als Verhaltensgewichtung verwendet. Wie *Obstacle Avoidance* ist *Separation* ein Steuerungsverhalten, das seine Gewichtung selbst berechnet. Im Endeffekt verwendet das Vehikel die Steuerungskraft nur zu einem Bruchteil, wenn die Hindernisse weiter entfernt sind. Das Resultat sind sanfte, realistisch wirkende Ausweichmanöver. Listing 13 zeigt die Implementation dieses Steuerungsverhaltens.

```
CContainer* cnt = world->GetObstacleContainer();
float maxstrength = 0.0f;
for (int i = 0; i < cnt->size; i++)
{
      Obj* target  = cnt->GetObject(i);
      float  radii  = (target->radius + vehicle->radius);

      //neighborhood test
      if (IsObjectInView(target, radii))
      {
            //get repelling direction
            float repeldir   = (vehicle->pos - target->pos);
            force           += repeldir;

            //get maximum repelling strength
            float dist       = (target->pos - vehicle->pos).Length();
            float distance   = dist / radii;
            float strength   = (1.0f / (distance * distance));
            strength         = strength / MAX_REPELLING_STRENGTH;

            if (strength > maxstrength) maxstrength = strength;
      }
}
SetWeight(maxstrength);
```

Listing 13: Separation Algorithmus

Probleme

Separation ist bei gleichförmigen bzw. gleichgerichteten Bewegungen einer Gruppe gut geeignet, Kollisionen zu vermeiden und Vehikel über den zur Verfügung stehenden Platz zu verteilen. Bei ungerichteten Bewegungen wie z.B an Kreuzungen, zeigt das Verhalten allerdings Schwächen, da für eine sichere Kollisionsvermeidung die Berechnung der zukünftigen Vehikel-positionen fehlt. Ein weiteres Problem stellt die numerische Stabilität dar. Wenn es keine Obergrenze an abstossenden Objekten gibt, die in das Verhalten mit einbezogen werden, ist es ziemlich leicht, ein Vehikel an ungültige Positionen der Spielwelt zu drücken (Wände, Hindernisse). Die generierten Abstoßungskräfte können äusserst stark sein (+INF). Darum müssen die auftretenden Kräfte beschränkt werden.

6.4.2 Kombination der Steuerungsverhalten

Die hier verwendete Methode, die einzelnen Steuerungskräfte zu einer finalen Steuerungskraft zu kombinieren, basiert auf einem Energie-Budget. Es stellt die maximal verwendbare Menge an Steuerkraft dar, die dem `SteeringState` zur Verfügung steht. Die einzelnen Steuerungsverhalten sind nach Priorität geordnet, d.h Verhalten mit höherer Priorität werden bevorzugt berechnet. Jedes Verhalten nimmt sich den Teil vom Budget den es benötigt (Selbstgewichtung) bzw. zugewiesen bekommt. Listing 14 zeigt die Implementation dieser Methode.

```
void CSteeringState::Calculate(Vector3D &force, float &gas,
                               float &brake, CAICtrl* vehicle)
{
      Vector3D    sum_force = Vector3D::ZERO;
      float       sum_gas, sum_brake, sum_weight, weight = 0.0f;

      STB_VECTOR_IT vIT = m_behaviours.begin();
      while(vIT != m_behaviours.end())
      {
            if ( sum_weight < 1.0)
            {
                  (*vIT)->Calculate(dt, force, gas, brake, vehicle);
                  weight= min((*vIT)->GetWeight(), (1.0f - sum_weight));
                  sum_weight  += weight;
                  sum_force   += force * weight;
                  sum_gas     += gas   * weight;
                  sum_brake   += brake * weight;
            }
            else
            {
                  break;
            }
            vIT++;
      }
      force = sum_force;
      gas   = sum_gas;
      brake = sum_brake;
}
```

Listing 14: Kombination der einzelnen Steuerungsverhalten eines SteeringStates

In einer kritischen Situation wie dem Ausweichen eines Hindernisses, verbrauchen die wichtigsten Verhalten einen Teil der zur Verfügung stehenden Steuerungskraft zuerst. Ist das Budget aufgebraucht, so werden die verbleibenden, weniger wichtigen Verhalten bei der weiteren Berechnung der finalen Steuerungskraft kurzfristig nicht mehr berücksichtigt. So wird zum Beispiel das Pathfollow-Verhalten für kurze Zeit verworfen, weil Obstacle-Avoidance das gesamte Steuerungskraft-Budget benötigt, um einem Schlagloch auf der Strasse auszuweichen.
Ist die Situation „normal“ und muss keinem Hindernis ausgewichen werden, so werden die oberen Verhalten nicht aktiv, nehmen sich nichts vom Budget und die niedriger priorisierten Verhalten können das Budget verwenden.

6.5 Fortbewegungsebene

Der `CBasePhysicsCtrl` benötigt drei Werte, um das Vehikel zu bewegen: die Richtung (`forcedir`), die Beschleunigung (`accforce`) und die Bremsstärke (`decforce`). `forcedir` stellt hier einen 3D-Vektor dar, wobei `accforce` und `decforce` skalare Werte vom Typ float sind. Der Wertebereich von `decforce` liegt zwischen 0.0 (keine Bremskraft) und 1.0 (maximale Bremskraft). Der Wertebereich von `accforce` umfasst zusätzlich die negativen reellen Zahlen bis -1.0. Damit ist es möglich, das Vehikel rückwärts fahren zu lassen. Diese Werte werden mit den folgenden Funktionen übergeben.

```
CBasePhysicsCtrl::SetForceDir(Vector3D forcedir);
CBasePhysicsCtrl::SetAccForce(float accforce);
CBasePhysicsCtrl::SetDecForce(float decforce);
```

Listing 15: Schnittstelle zwischen Steuerungsebene und Fortbewegungsebene

7. Fazit und Ausblick

Im letzten Kapitel dieser Arbeit erfolgt die Auswertung des Steuerungssystems hinsichtlich der Aufgabenstellung aus Abschnitt 5.1. Das Vehikelverhalten wird anhand zweier Standardsituationen exemplarisch dargestellt. Anschließend wird die beigelegte Testumgebung vorgestellt und die Bedienung erklärt. In der sich anschließenden Problembehandlung werden Schwierigkeiten diskutiert, die in den einzelnen Stadien der Entwicklung dieses Systems, auftraten. Zuletzt wird ein Ausblick gegeben, der mögliche Verbesserungen sowie zukünftige Entwicklungen und Verwendungen darstellt.

7.1 Anwendungsbeispiele

Die nachfolgenden Abschnitte beschäftigen sich mit der Darstellung eines Ausweich- bzw. Überholvorganges. Diese Situationen treten häufig bei der Interaktion zwischen unterschiedlichen Vehikeln auf und gehören zu den Standardsituationen. Es werden charakteristische Momentaufnahmen der Vorgänge beschrieben und das Zusammenwirken der einzelnen Steuerungsverhalten dargestellt. Der Einfluss der Verhalten wird für jedes Vehikel mit Hilfe eines Tortendiagramms (Abbildung 45) verdeutlicht.

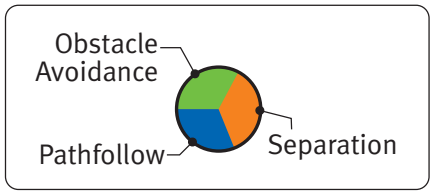


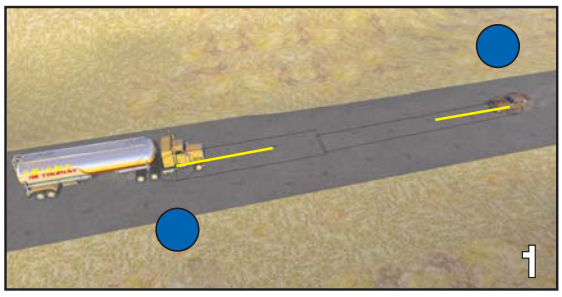
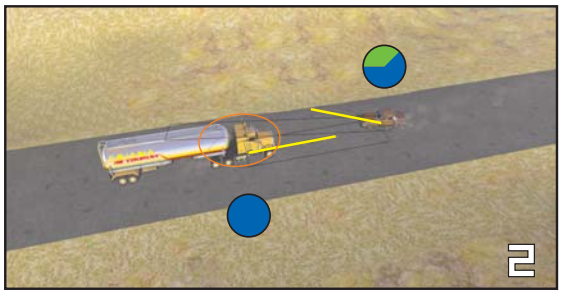
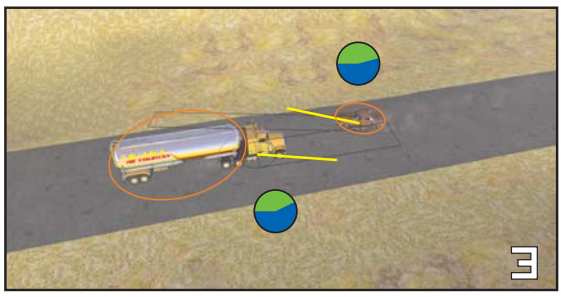
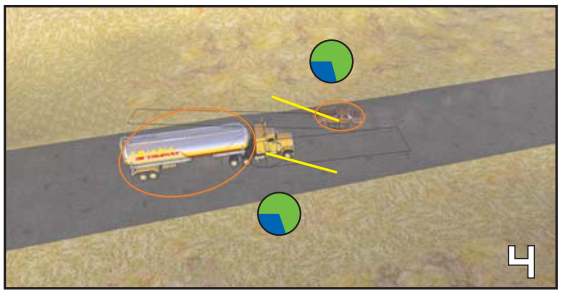
Abbildung 45: Farben der Steuerungsverhalten

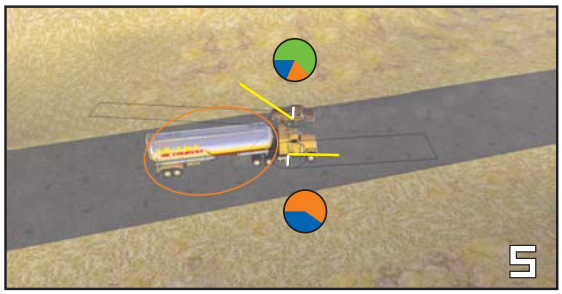
Die aus dem Zusammenspiel der Steuerungsverhalten resultierende Steuerungskraft wird in den Bildern als gelbe Linie dargestellt. Sie stellt die gewünschte Steuerungsrichtung dar. Weicht die Linie stark von der momentanen Fahrzeugausrichtung ab, ist der Steuerungswunsch des Vehikels groß.

Das schwarze Rechteck vor dem Vehikel ist der Hindernissensor. Befindet sich eine Hindernis in ihm, wird es durch einen orangen Kreis markiert. Je näher sich ein Hindernis am Vehikel befindet, desto größer wird die Ausweichdringlichkeit und desto größer der Einfluss auf das Gesamtverhalten.

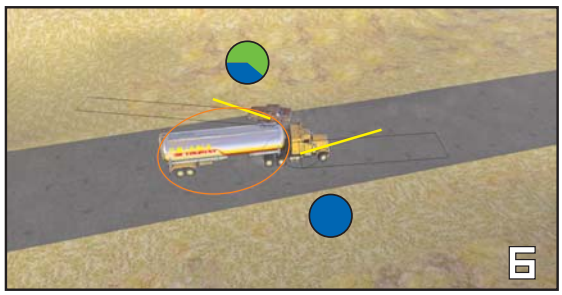
Kommen sich zwei Fahrzeuge zu nahe, stoßen sie sich gegenseitig ab. Dafür sorgt das Separation Verhalten. Die generierte Abstoßungssrichtung wird in den Abbildungen als weiße Linie dargestellt.

7.1.1 Ausweichen

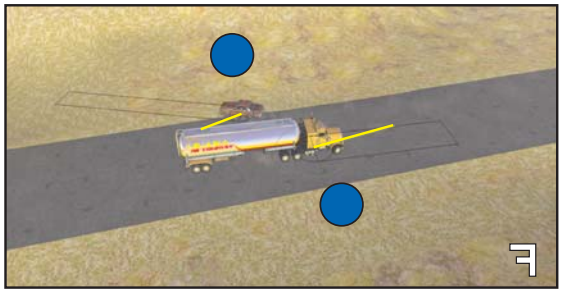
Bild	Beschreibung
	Start der Sequenz Keines der Vehikel befindet sich in den Sensoren des anderen.
	Die Zugmaschine des LKW wurde durch die Hindernissensoren des PKW erfasst. Der PKW startet ein Ausweichmanöver, indem er nach rechts steuert. Die gewünschte Steuerungsrichtung (gelbe Linie) zeigt die Stärke der Korrektur.
	Nun hat auch der LKW den PKW mit seinem Hindernissensor erfasst und startet ebenfalls einen Ausweichvorgang nach rechts. Der PKW ist inzwischen soweit ausgewichen, das sich nicht die Zugmaschine des LKW, sondern der Anhänger im Sensor befindet.
	Die Situation ist unverändert. Da sich die beiden Fahrzeuge jedoch einander näher gekommen sind, verstärkt sich die Ausweichdringlichkeit. Das ist an den gelben Linien beobachtbar, die hier stärker auseinander gehen.



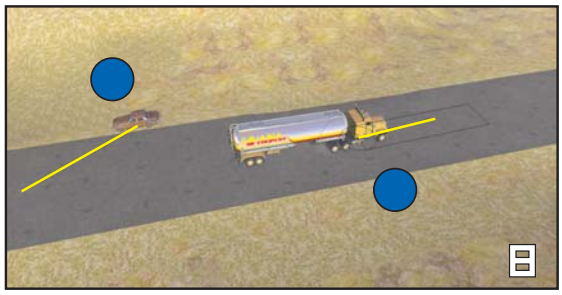
Der PKW befindet sich nun ausserhalb der Hindernissensoren des LKW. Die beiden Fahrzeuge sind einander so nah, dass sie sich gegenseitig mittels des *Separation* Verhaltens abstoßen. Zudem befindet sich der LKW-Anhänger in den Hindernissensoren des PKW, was den PKW zu weiterem Ausweichen veranlasst.



Der LKW steuert wieder auf seinem alten Kurs, da er kein Hindernis mehr vor sich hat. Der PKW weicht weiterhin dem Anhänger aus.



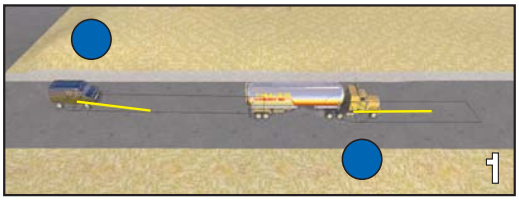
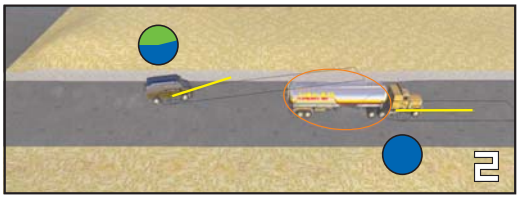
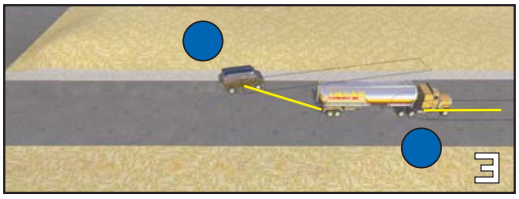
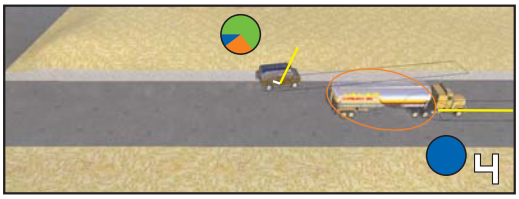
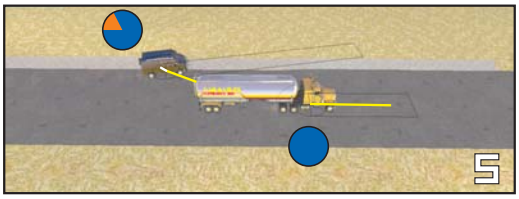
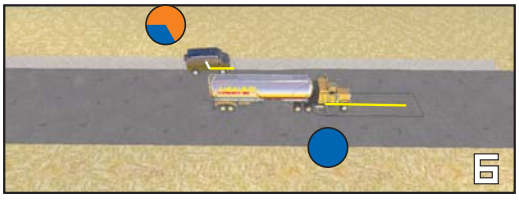
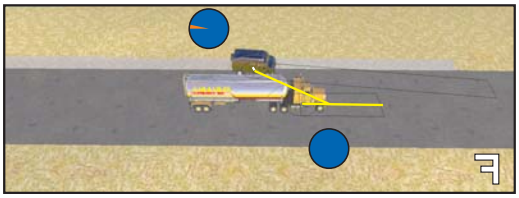
Der PKW steuert wieder auf den alten Kurs zurück, da er kein Hindernis mehr vor sich hat.

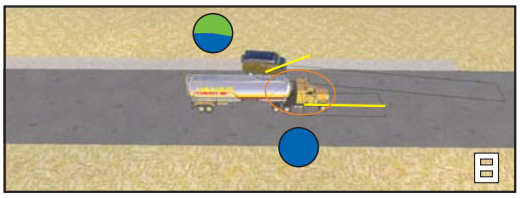


Ende der Sequenz

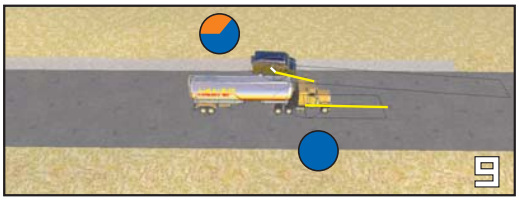
Ausweichvorgang abgeschlossen.

7.1.2 Überholen

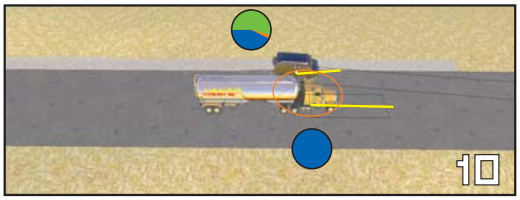
Bild	Beschreibung
	Start der Sequenz, Keines der Vehikel befindet sich in den Sensoren des anderen.
	Der Van weicht dem LKW-Anhänger aus.
	Der Van steuert auf seinem alten Kurs, da der Anhänger ausserhalb seiner Sensoren ist.
	Der Van befindet sich nahe am LKW-Anhänger. Er weicht wieder aus, da sein Hindernissensor ihn wieder erfasst hat. Zudem dazu, stößt *Separation* den Van vom Anhänger ab (weiß). Die erzeugte Ausweichkraft ist relativ groß.
	Der LKW-Anhänger ist außerhalb der Van-Sensoren des Vans. Er steuert auf seinen alten Kurs zu. *Separation* verhindert, dass er das auch tut, indem eine bestimmte Abstoßungskraft erzeugt wird.
	Separation hält den Van auf Kurs.
	Der Van hat den LKW-Anhänger zum größten Teil überholt und steuert auf seinen alten Kurs zu.



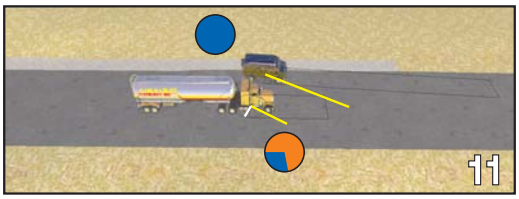
Der Van hat die LKW-Zugmaschine im Sensor und weicht aus.



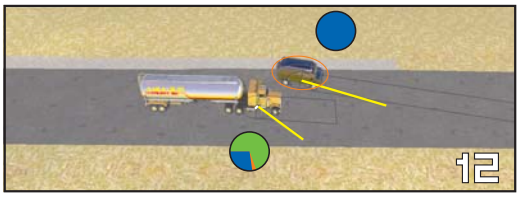
Die Zugmaschine befindet sich außerhalb der Sensoren, doch *Separation* stößt den Van von der Zugmaschine ab, während er wieder auf seinem alten Kurs steuert.



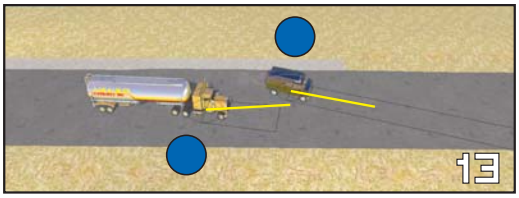
Die Zugmaschine befindet sich wieder in den Van-Sensoren und er weicht wieder aus.



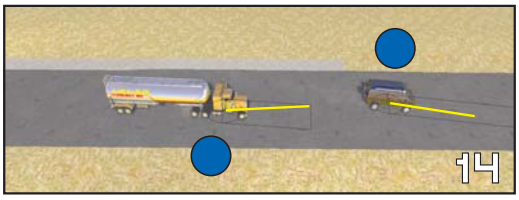
Der Van hat den LKW zum größten Teil überholt. Der LKW hat den Van nun direkt neben sich. Er stößt sich vom Van ab, da er zu nah ist.



Der Van befindet sich innerhalb der LKW-Sensoren. Der LKW weicht aus.



Alle Vehikel befinden sich ausserhalb der Sensoren, sie kehren auf ihren alten Kurs zurück.



Ende der Sequenz.
Überholvorgang abgeschlossen.

7.2 Testumgebung

Zur Demonstration der Fähigkeiten dieses Steuerungssystems wurde eine interaktive Testumgebung (Level) erstellt, die einige häufig benötigte Verhaltensweisen wie Pfad- und Vehikelverfolgung beinhaltet. Hier kann man bestimmte Standardsituationen beobachten, die durch die Interaktion zwischen Vehikeln ständig auftreten.

Die Fahrzeuge in diesem Level verfolgen verschiedene Aufgaben. Die erste Gruppe der Fahrzeuge zeigt den normalen Strassenverkehr. Sie sind damit beschäftigt von einem Punkt im Level in einen anderen zu fahren. Dabei kommt es häufig zu Ausweichmanövern. Zusätzlich befinden sich spezielle Fahrzeuge auf den Straßen. Sie sollen den normalen Straßenverkehr etwas stören, indem sie schneller oder langsamer fahren. Zu dieser Fahrzeuggruppe gehören der Van und der Truck. Der Van ist mit einer State Machine ausgestattet, die ihn immer wieder eine bestimmte Route abfahren lässt (Patrouille). Mit diesen Fahrzeugen kommt es häufiger zu Überholmanövern. All diese Fahrzeuge verwenden die Steuerungsverhalten *Obstacle Avoidance, Separation* und *Pathfollow,* um sich im Gelände zu bewegen.

Der Benutzer hat die Möglichkeit selbst ein Vehikel zu steuern. Dieses Vehikel (*Player-Van*) wird durch zwei Polizeiwagen und einen Polizeihubschrauber verfolgt. Diese Fahrzeuge versuchen stets in der Nähe des Player-Vans zu bleiben, wobei Zusammenstöße vermieden werden sollen. Die Verfolgervehikel verwenden die Steuerungsverhalten *Separation* und *Arrival*.

Der Level bietet drei verschiedene Modi, um das Geschehen zu verfolgen. Der erste Modus, in dem der Benutzer startet, nennt sich *Free-Cam*. Der Benutzer kann hier durch die Szene schweben und sie von beliebigen Blickwinkeln betrachten. Der zweite Modus (*Vehicle-Cam*) bietet eine leichtere Verfolgung von Fahrzeugen. Hier hängt die Kamera direkt am Fahrzeug und der Betrachter kann in einem bestimmten Abstand frei um das Fahrzeug rotieren. Der dritte Modus (*Top-Cam*) bietet eine Draufsicht auf das Level und gewährt dem Betrachter ein Maximum an Überblick.

Darüberhinaus stehen weitere Möglichkeiten zur Steuerung der Zeit zur Verfügung. Das Geschehen kann eingefroren oder verlangsamt werden. Das ermöglicht eine genaue Analyse von Abläufen, die andernfalls zu schnell für das Auge wären.

Auf diese Funktionalität wird über das Menü zugegriffen. Zusätzlich können hier können die einzelnen Vehikel ausgewählt werden, die näher untersucht werden sollen. Zu jedem dieser Vehikel können zusätzliche Steuerungsinformationen angezeigt werden. Diese Informationen reichen von der Steuerungsrichtung über eingesetzte State Machines und Steuerungsverhalten bis zur Anzeige der Sensordaten von einzelnen Verhalten. Im Folgenden wird die Tastaturbelegung zur Navigation im Level vorgestellt.

7.2.1 Tastenbelegung

Navigation im Menu

F1	– Aufruf des Menüs
Pfeiltasten	– Anwahl der Menupunkte
Enter	– Auswahl bestätigen

Navigation im Level

Tasten W,S,A,D	– Steuerung der Kamera
Mouse	– Steuerung der Kamera
TAB	– Umschalten der Zeit (Start, Stop)

7.2.2 Minimale Systemanforderungen

Der Level basiert auf der YAGER-Engine und hat daher einige Anforderungen an die eingesetzte Hardware. Für eine flüssige Darstellung der Szene werden mindestens benötigt:

- Windows 98, 98SE, ME, XP
- 1,5 Ghz Intel Pentium III oder AMD Athlon Prozessor
- 256 MB RAM
- DirectX 9.0
- DirectX 8.1 kompatible Grafikkarte mit TnL Unterstützung
  (NVIDIA GeForce Serie, ATI Radeon 32MB oder bessere Grafikarte empfohlen)

7.3 Probleme

Im Laufe der Entwicklung kam es zu einigen Problemen. Das Hauptproblem bestand in der Stabilität der Steuerung. In bestimmten Situationen kommt es zu ruckartigen Korrekturbewegungen, die durch eingeschränkte Sensoren bedingt sind. So wird in Frame 1, am Beispiel des *Obstacle Avoidance* Verhaltens, ein Hindernis erkannt. Es erfolgt eine korrigierende Steuerung, um das Vehikel aus dem Kollisionskurs zu bringen. Im zweiten Frame befindet sich das Hindernis dann außerhalb des Sensors. Dadurch kommt es zu Korrektursteuerungen, die das Vehikel auf den ursprünglichen Kurs zurückbringen. Das Hindernis befindet sich somit in Frame 3 wieder innerhalb der Sensoren. Dieses Überschwungverhalten ist das Hauptproblem, mit dem die Steuerungsverhalten zu kämpfen haben. Dieser Effekt wird durch eine gewisse Reaktionsträgheit der zugrundeliegenden Fahrphysik verstärkt . Durch die Kombination von Steuerungsverhalten und physikalisch simulierten Vehikeln, ist ein vollständig stabiles Vehikelverhalten, ohne überflüssige Bewegungen, nur durch den Einsatz zusätzlicher Sensoren zu realisierbar.

Ein weiteres Problem besteht in der Übersetzung einiger Kräfte. Vehikel, die auf einfachen Fahrzeugmodellen basieren, sind in der Lage, sich seitwärts zu bewegen. Am Beispiel des *Separation* Verhaltens, das u.a. seitlich wirkende Abstoßungskräfte erzeugt, werden die Einschränkungen deutlich, die durch ein realistischeres Fahrzeugmodell entstehen. Die generierten Abstoßungskräfte können nicht direkt umgesetzt werden, da sich ein Vehikel nicht seitwärts bewegen kann. Sie müssen auf sinnvolle Steuerungsbefehle umgesetzt werden, die das Vehikel mittels Wendemanövern aus dem Einflussgebiet bringen. Das ist nicht immer möglich. Aufgrund der eingeschränkten Bewegungsfreiheit bedarf es größerer Vorausplanung, was dem Gedanken von Steuerungsverhalten an einigen Stellen widerspricht.

7.4 Fazit und Ausblick

Trotz der oben genannten Probleme wurden die Hauptanforderungen Flexibilität, Erweiterbarkeit, Kontrollierbarkeit, Speichersparsamkeit und Glaubwürdigkeit an das Steuerungssystem erfüllt worden.

In dieser Arbeit wurde ein System zur Steuerung von Vehikeln mit unterschiedlicher Fortbewegungsart, Agilität und Geschwindigkeit entworfen. Dieses Steuerungssystem basiert auf Steuerungsverhalten, die die Grundlage komplexer Verhaltensweisen bilden. Die Methode, Vehikel durch Steuerungsverhalten zu steuern, hat sich inzwischen bei vielen Projekten[14] und Computerspielen bewährt.

Die Forderungen nach Flexibilität und Erweiterbarkeit sind bereits zur Entwurfszeit berücksichtigt worden und führten zu einem System, das eine breite Palette unterschiedlicher Vehikeltypen unterstützt, deren Verhalten leicht erweitert und bis zu einem gewissen Grad kombiniert und ausgetauscht werden können.

Das Agentenverhalten ist trotz eines gewissen Grades an Autonomität kontrollier- und steuerbar. Durch Scriptbefehle und den *Data-Driven* Ansatz ist man in der Lage, zur Laufzeit Verhaltensparameter oder gleich das gesamte Verhalten beeinflussen zu können. Dadurch entsteht ein Agent, der dem Leveldesigner die „Low-Level“ Arbeiten wie Hindernisbehandlung abnimmt und stattdessen durch „High-Level“ Befehle wie Zielvorgaben gesteuert wird.

Der konsequente Einsatz von einfachen Algorithmen sorgt für weitestgehende Performanz und Speichersparsamkeit. Der Umstand, dass Steuerungsverhalten keine Weltzustände speichern, kommt dem zugute. Sollte sich herausstellen, dass ein Verhalten durch den Einsatz zusätzlicher Sensoren oder besserer Algorithmen stabiler und zuverlässiger gestaltet werden kann, so ist das trotzdem jederzeit möglich.

14 In vielen Filmen wie z.B. „Der Herr der Ringe“ werden Steuerungsverhalten für Massenszenen verwendet, in denen Tausende Einheiten dargestellt werden.

Bisherige Implementationen konzentrierten sich auf die Steuerung von Vehikeln mit relativ einfachen Fahrzeugmodellen, mit denen Vehikel auf der Stelle wenden und bei Bedarf sofort stehen bleiben können. Die dort zugrunde liegenden Steuerungsverhalten sind deshalb in der Lage, die Vehikel direkt zu steuern, indem sie ihre Geschwindigkeit und Ausrichtung setzen.
Im Steuerungssystem dieser Arbeit liegt der Fokus jedoch auf physikalisch simulierten Fahrzeugen, die nur indirekt gesteuert werden können. Die Herausforderung besteht darin, trotz dieser eingeschränkten Möglichkeiten ein glaubwürdiges Vehikelverhalten zu kreieren. Glaubwürdigkeit ist das eigentliche Problem des Steuerungssystems. Hier gibt es in der Regel immer etwas zu verbessern, da es ein subjektives, d.h. vom Betrachter abhängiges Kriterium ist.

Mit Abschluss der vorliegenden Arbeit ist die Entwicklung des Steuerungssystems keineswegs beendet. Es wurde die erste Ausbaustufe des Steuerungssystems verwirklicht. Die nächste Ausbaustufe sieht Gruppenverhalten und die Kommunikation zwischen einzelnen Agenten vor. Eine eigenständige aktive Planung und das Setzen von strategischen Zielen auf mehreren Ebenen (Squad-Level etc.) ist eine weitere Option.
Zudem kann jede Ebene erweitert und verbessert werden. Im Bereich der Steuerungsebene ist eine Erweiterung der Steuerungsverhalten geplant. Erweiterte Verhaltensweisen wie *Verstecken* und *Deckung suchen* sind hier denkbar. Einige Steuerungsverhalten können hinsichtlich der Stabilität optimiert werden. Das kann zum einen durch den Einsatz zusätzlicher Sensoren geschehen, die das Verhalten beeinflussen und die Wahrnehmung in kritischen Situationen verbessern. Zum anderen kann eine Stabilisierung durch den Einsatz von Methoden aus der Reglungstechnik erzielt werden. Zu nennen wären hier PID-Controller, die dazu beitragen können das Überschwungverhalten zu minimieren.
Zusätzlich kann die Umstellung einiger Verhalten von Boundingspheres auf Polygone zur Repräsentation ihrer Ausmaße umgesetzt werden. Dadurch wird die Genauigkeit von Verhalten wie Obstacle Avoidance zusätzlich erhöht.
Auf der Fortbewegungsebene sind fortgeschrittene Techniken im Bereich Animation geplant. Ein Animationssystem für Charaktere erleichtert die Kommunikation komplexer Steuerungs- und KI-Vorgänge nach aussen.

8. Anhang

8.1 Abbildungsverzeichnis

8.2 Quellcodeverzeichnis

8.3 Literaturverzeichnis

[ALT&KING03] Alt, Greg & King, Kristin, Intelligent Movement Animation for NPCs, 2003, AI Game Programming Wisdom 2, CharlesRiver Media, Hingham, Massachusetts, Hrsg: Steve Rabin, ISBN: 1-58450-289-4

[COOPER98] Cooper, James W., The Design Patterns Java Companion, 1998, , Addison Wesley, , Hrsg: , ISBN:

[DARBY03] Darby, Alex, Racing Vehicle Control using Insect Intelligence, 2003, AI Game Programming Wisdom 2, CharlesRiver Media, Hingham, Massachusetts, Hrsg: Steve Rabin, ISBN: 1-58450-289-4

[FU&HOU03] Dan Fu & Ryan Houlette, The Ultimate Guide to FSMs in Games, 2003, AI Game Programming Wisdom 2, CharlesRiver Media, Hingham, Massachusetts, Hrsg: Steve Rabin, ISBN: 1-58450-289-4

[GOF96] Erich Gamma, Richard Helm, Ralph Johnson, John Ulissides, Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software, 1996, , Addison Wesley, , Hrsg: , ISBN: 0-201-63361-2

[GREEN00] Green, Robin, Steering Behaviours, August 2000, ACM SIGGRAPH 2000 Conference Proceedings, Anaheim,California (USA)

[KENT03] Kent, Tom, Multi-Tiered AI Layers and Terrain Analysis for RTS Games, 2003, AI Game Programming Wisdom 2, CharlesRiver Media, Hingham, Massachusetts, Hrsg: Steve Rabin, ISBN: 1-58450-289-4

[LARAMEE03] Laramée, Francois D., Dead Reckoning in Sports and Strategy Games, 2003, AI Game Programming Wisdom 2, CharlesRiver Media, Hingham, Massachusetts, Hrsg: Steve Rabin, ISBN: 1-58450-289-4

[LIDEN03] Liden, Lars, Artificial Stupidity: The Art of Intentional Mistakes, 2003, AI Game Programming Wisdom 2, CharlesRiver Media, Hingham, Massachusetts, Hrsg: Steve Rabin, ISBN: 1-58450-289-4

[RABIN00] Rabin, Steve, A* Speed Optimizations, 2000, Game Programming Gems, Charles River Media Inc., Hingham, Massachusetts, Hrsg: Mark Deloura, ISBN: 1-58450-049-2

[RAMSEY03] Ramsey, Micheal, Designing a Multi-Tiered AI Framework, 2003, AI Game

Programming Wisdom 2, CharlesRiver Media, Hingham, Massachusetts, Hrsg: Steve Rabin, ISBN: 1-58450-289-4

[REYNOLDS87] Reynolds, Craig W., Flocks, Herds, and Schools: A Distributed Behavioral Model, Juli 1987, ACM SIGGRAPH 1987 Conference Proceedings, Anaheim,California (USA)

[REYNOLDS88] Reynolds, Craig W., Not Bumping Into Things, August 1988, ACM SIGGRAPH 1988 Conference Proceedings, Anaheim,California (USA)

[REYNOLDS99] Reynolds, Craig W., Steering Behaviors for Autonomous Characters, März 1999, Game Developers Conference,

[TOMLINSON04] Tomlinson, Simon L., The Long and Short of Steering in Computergames, März, 2004, UKSIM CONFERENCE ON COMPUTER SIMULATION, St Catherine's College, Oxford, England.

8.4 Verzeichnis der Online Quellen

Alle Quellen wurden zuletzt am 07.11.2004 auf Erreichbarkeit überprüft.

[O-1] Craig Reynolds Homepage
http://www.red3d.com/cwr

[O-2] Gamasutra
http://www.gamasutra.com

[O-3] Coordinated Unit Movement (Gamasutra)
http://www.gamasutra.com/features/game_design/19990122/movement_01.htm

[O-4] Implementing Coordinated Unit Movement (Gamasutra)
http://www.gamasutra.com/features/game_design/19990129/implementing_01.htm

[O-5] Eberley, D, Magic Software library.
http://www.magic-software.com

[O-6] AI Wisdom
http://www.aiwisdom.com

[O-7] The Star of the Search Algorithms (A Star)
http://www.gamasutra.com/features/19990212/sm_04.htm

[O-8] www.steeringbehaviours.com
http://www.steeringbehaviors.com/

[O-9] „Simulating Crowd-Motion using Behavioural Animation“
http://www.kolve.com/thesis/thesis.htm

[O-10] Open Steer
http://opensteer.sourceforge.net/

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen Hilfsmittel als die angegebenen Quellen benutzt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, habe ich als solche gekennzeichnet. Weiter erkläre ich, die Diplomarbeit in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt zu haben.

Berlin, 10. 11. 2004 Stephan Ziep