

**Entwicklung einer VR-Anwendung
zur Simulation des autonomen
Einzel- und Gruppenverhaltens
von Lebewesen in einer Unterwasserwelt**

Diplomarbeit

vorgelegt von
Sebastian Jockel

 UNIVERSITÄT
KOBLENZ · LANDAU
Institut für Computervisualistik
Arbeitsgruppe Computergraphik

 [vertigo]²
systems gmbh

Prüfer: Prof. Dr.-Ing. Stefan Müller
Betreuer: Dipl.-Inf. Frank Hasenbrink

August 2005

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass die vorliegende Arbeit selbstständig verfasst wurde und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet wurden. Die Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Koblenz, den 02.08.2005

Kurzbeschreibung

Um eine virtuell erzeugte Welt mit Leben zu bereichern, ist es notwendig, Charaktere zu erzeugen, die sich in dieser virtuellen Welt möglichst frei und autonom bewegen können. Damit die Charaktere auf Änderungen in dieser Welt reagieren können, müssen sie ein eigenes Verhalten besitzen. Hierfür können diverse, vorab definierte Verhaltensmuster verwendet werden, welche die Art des Handelns eines Charakters beschreiben.

In dieser Arbeit wird ein Framework¹ zur Erstellung von autonomen Einzel- und Gruppenverhalten im Sinne einer Bewegungsänderung von Charakteren in einer Virtual-Reality-Anwendung vorgestellt. Die Funktionalität des Frameworks wird am Beispiel einer Unterwasserwelt mit autonomen Fischen verdeutlicht.

Die Schwerpunkte liegen auf der Navigation der Charaktere und deren Entscheidungsfindung zur Auswahl eines bestimmten Verhaltens. Das entwickelte Konzept erlaubt die schnelle Strukturierung von Verhaltensweisen zu einer Verhaltenshierarchie und die einfache Verknüpfung der Navigationsmechanismen an Bedingungen. Dadurch lassen sich mit relativ wenig Aufwand auch komplexere Verhaltensweisen erstellen. Darüber hinaus ist es möglich, Beziehungen zwischen den Charakteren zu definieren, damit diese beispielsweise zwischen Freunden oder Feinden unterscheiden können. Die definierten Beziehungen können dann bei der Auswahl der Verhaltensmuster Berücksichtigung finden und das Verhalten eines Charakters beeinflussen.

¹ Der Begriff „Framework“ bezeichnet in dieser Arbeit eine Sammlung von Klassen zu einem bestimmten Thema, die als Basisklassen für weitere Klassen dienen. Ein Framework hat somit das Ziel, eine einheitliche Anwendungsarchitektur vorzugeben.

Inhaltsverzeichnis

Kapitel 0. Einleitung	1
Kapitel 1. Thematik	3
1.1 Problemstellung	3
1.2 Rahmenbedingungen.....	5
1.2.1 Vertigo Systems	5
1.2.2 Das Anwendungsszenario: living	5
1.2.3 Anforderungslisten	8
Kapitel 2. Theorie	9
2.1 Verwandte Arbeiten	9
2.1.1 Craig W. Reynolds	9
2.1.2 Xiaoyuan Tu	20
2.1.3 Demetri Terzopoulos	24
2.1.4 John David Funge	25
2.1.5 Diskussion der Arbeiten	26
2.2 Techniken aus der KI	28
2.2.1 Reaktive Verhaltensregeln	29
2.2.2 Endliche Automaten	29
2.2.3 Neuronale Netze	30
2.2.4 Genetische Algorithmen.....	32
2.2.5 Diskussion der Techniken	33
2.3 Bestehende OpenSource-Projekte	35
2.3.1 OpenAI	35
2.3.2 FEAR.....	35
2.3.3 OpenSteer	36
2.3.4 MetaAgent.....	39
2.3.5 Diskussion der OpenSource-Projekte	39
Kapitel 3. Das Konzept	41
3.1 Entscheidungsfindung.....	41
3.2 Konzeptdesign	42

3.3 Funktionsspezifikation.....	43
Kapitel 4. Realisierung des Gesamtsystems / Implementierung	48
4.1 Überblick.....	48
4.2 Beschreibung der wichtigsten Klassen	50
4.2.1 Die ThinkingVehicleMixin-Klasse	50
4.2.2 Die ThinkingVehicle-Klasse.....	53
4.2.3 Die ShadowVehicle-Klasse	53
4.2.4 Die Need-Klassen	54
4.2.5 Die Action-Klassen.....	56
4.3 Beschreibung der graphischen Benutzeroberfläche	65
Kapitel 5. Beispielanwendung: Eine Unterwassersimulation mit Fischen	68
5.1 Szenariobeschreibung	68
5.2 Das Fischverhalten	69
5.3 Die Fischarten.....	72
5.4 Animation der Fische	73
Kapitel 6. Zusammenfassung und Ausblick	74
6.1 Erweiterungsmöglichkeiten	74
6.2 Fazit	79
Anhang A: Pseudocode für einen Simulationsschritt	81
Literaturverzeichnis	84
Internetquellen	86
Abbildungsverzeichnis	87
Abkürzungsverzeichnis	89

Kapitel 0. Einleitung

Mit dem nicht enden wollenden Boom der Computergraphik, der durch ständig anwachsende Computerleistungen und schnellere Algorithmen zu immer mehr echtzeitfähigen und realistischeren Graphiken führt, wächst seit spätestens Anfang der neunziger Jahre auch der sogenannte „Artificial Life“ - Bereich zu einem immer bedeutender werdenden Teilgebiet der Informatik heran. Hinter diesem Begriff verbirgt sich ein interdisziplinäres Forschungsfeld, das Techniken aus der Computergraphik mit Methoden aus der KI, der künstlichen Intelligenz, verbindet.

Durch Zusammenführung der beiden Forschungsgebiete können künstliche Computercharaktere erzeugt werden, die ein autonomes Verhalten besitzen, das ihnen erlaubt, sich eigenständig in einer virtuellen 3D-Welt zurechtzufinden. Unter einem „Charakter“ werden in diesem Fall aber keineswegs nur Menschen mit einer ausgeprägten Persönlichkeit verstanden. Auch Tiere und Insekten, ja sogar Flugzeuge und Autos können als „Charaktere“ bezeichnet werden, sofern sie in einer Computersimulation ein entsprechendes autonomes Verhalten aufweisen.

Die Einsatzmöglichkeiten solcher künstlichen Lebensformen sind vielseitig. Generell können sie beliebige virtuelle Welten mit vermeintlichem Leben bereichern. Deshalb finden sie sowohl in der Wissenschaft und Forschung Verwendung als auch in der Industrie, Pädagogik und der Unterhaltungsbranche.

In Computerspielen werden die „Artificial Life“ - Techniken gerne zur Erzeugung sogenannter Non-Player Charaktere eingesetzt. Also für Charaktere, die zwar auf die Spielfigur des Benutzers reagieren und mit ihr interagieren können, aber nicht direkt vom Benutzer gesteuert werden. Einen Schritt weiter in Richtung künstlichen Lebens geht beispielsweise die Computerspielreihe *Creatures*¹, in der sich die virtuellen Charaktere eigenständig fortpflanzen und durch Kombination ihrer „Gene“ neue Arten entstehen.

Bei Filmproduktionen werden sie heutzutage bevorzugt in sogenannten Massenszenen eingesetzt, in denen viele Charaktere - zumeist Menschen - einer ähnlichen Tätigkeit nachgehen. Mittlerweile kommen die großen Kinospektakel kaum noch ohne ihre virtuellen Schauspieler aus, die immer häufiger die Rollen

¹ *Creatures 1, Creatures 2 und Creatures 3* von Gameware Development Ltd. 1998-2004

der Statisten übernehmen. So trugen sie den größten Teil der Schlachten um Mittelerde in der „Der Herr der Ringe“¹ Trilogie aus oder kämpften in den neuen „Star Wars“ Episoden² um das Universum. Die ersten autonomen Charaktere, mit denen die Eroberung der Kinoleinwände begann, kamen allerdings aus der Tierwelt. Es waren Fledermausschwärme und watschelnde Pinguinhorden, die 1992 in „Batmans Rückkehr“³ erstmalig ihren Auftritt hatten. Wenn man das rasante Tempo bedenkt, mit der die Komplexität der autonomen Charaktere seit 1992 stetig zugenommen hat, darf man gespannt sein, wie sich deren schauspielerischen Leistungen in naher Zukunft entwickeln werden.

Wie bereits erwähnt, gibt es auch außerhalb des Entertainmentbereichs für autonome Charaktere vielfältige Verwendungsmöglichkeiten. Verhaltensforscher versuchen mit ihrer Hilfe, theoretische Verhaltensmodelle zu beweisen. Auf Grund der Ergebnisse, kann beispielsweise ein Architekt in einem geplanten Gebäude eine Massenpanik simulieren und dadurch entscheiden, ob genügend Notausgänge vorhanden sind (vergleiche [9]). In der Verkehrsplanung halfen autonome Charaktere bereits dabei, das Entstehen von Staus besser zu verstehen (siehe [18]), woraufhin entsprechende Gegenmaßnahmen getroffen wurden. Auf Grund dieser Erkenntnisse ist es in Deutschland zur Pflicht geworden, sich als Autofahrer bei Sperrung eines Fahrstreifens erst kurz vor der Sperrung in den fließenden Verkehr einzufädeln, weil so längere Rückstaus vermieden werden können.

¹ Originaltitel: „The Lord of the Rings I: The Fellowship of the Ring“ von Warner Brothers, 2001

Originaltitel: „The Lord of the Rings II: The Two Towers“ von Warner Brothers, 2002

Originaltitel: „The Lord of the Rings III: The Return of the King“ von Warner Brothers, 2003

² Originaltitel: „StarWars: Episode I - The Phantom Menace“ von Lucasfilm, 1999

Originaltitel: „StarWars: Episode II - The Clone Wars“ von Lucasfilm, 2002

Originaltitel: „StarWars: Episode III - Revenge of the Sith“ von Lucasfilm, 2005

³ Originaltitel: „Batman Returns“ von Warner Brothers, 1992

Kapitel 1. Thematik

Die Verwendung eines „Artificial-Life“ Charakters besitzt gegenüber einem herkömmlich animierten Charakter viele Vorteile. Besonders hervorzuheben ist, dass er weder auf aufwendige Einzelbildanimationen, noch auf starre, vordefinierte Pfade angewiesen ist (vergleiche [15]). Statt dessen kann er selbstständig seinen Weg bestimmen und zu einer Bewegung eine passende Animation herleiten. Da sein Verhalten mitunter von sehr vielen Faktoren beeinflusst wird, ist eine Vorhersage des exakten Animationsablaufes kaum möglich. Somit ergibt sich für jeden Charakter stets ein individuelles Verhalten. Dies verleiht ihm Lebendigkeit und macht ihn für eine Interaktion mit dem Benutzer interessant.

Dieses Kapitel befasst sich mit den allgemeinen Problemen, die es beim Prozess der Erstellung eines künstlichen Charakters zu lösen gilt. Hierfür werden zunächst einige Definitionen eingeführt und die Thematik spezifiziert. In Abschnitt 1.2 werden die Rahmenbedingungen erläutert, die für den Entwurf des Frameworks maßgeblich waren.

1.1 Problemstellung

Die Erzeugung eines sich autonom bewegendes „Artificial Life“ - Charakters birgt drei zentrale Probleme (vergleiche [13]).

Das erste Problem ist die Auswahl eines Verhaltens. Hierzu gehört die Fragestellung, welche Motivation oder welches Ziel den Charakter dazu verleitet, ein bestimmtes Verhalten anzuwenden und wie der Entscheidungsfindungsprozess dazu gestaltet ist. Das zweite Problem ist die Navigation des Charakters. Hierfür muss er in der Lage sein, seine Entscheidung durch Steuerungsanweisungen umzusetzen. Das dritte Problem ist die Umsetzung der Steuerungsanweisungen in eine konkrete Animation seines Körpers.

Gegenstand dieser Arbeit ist es, eine Lösung der ersten beiden Probleme anzubieten. Das Ergebnis ist ein leicht zu modifizierendes und vielseitig einsetzbares Verhaltensframework, dass auf der OpenSource-Software OpenSteer aufbaut. Das Konzept hierzu wird in Kapitel 3 ausführlich vorgestellt

und die Implementierung anschließend in Kapitel 4 beschrieben. Für das dritte Problem, der Animation des Charakters, werden nur allgemeine Lösungsansätze angegeben, die in Kapitel 5.4 auf Seite 73 geschildert sind.

Ein weiterer Schwerpunkt liegt auf der Generierung von höherem Verhalten. Deshalb ist es wichtig, an dieser Stelle eine klare Unterscheidung zwischen höherem und niederem Verhalten einzuführen.

In der Literatur (siehe [4] oder [13]) wird niederes Verhalten im Zusammenhang mit autonomen Charakteren oft als ein Verhaltensprimitiv bezeichnet, welches sich nicht weiter in einzelne Bausteine zerlegen lässt. Der Ausdruck des höheren Verhaltens findet häufig bei einer Kombination im Sinne einer gleichzeitigen Anwendung mehrerer niederer Verhalten Verwendung.

Da sich aber auch mehrere höhere Verhalten wieder zu einem neuen Verhalten kombinieren lassen, sollte das neue dementsprechend von höherer Ordnung sein, als seine Teilkomponenten. Daraus könnte man eine relative Angabe in Form von „höher als ...“ ableiten, bei der dann generell die Teilkomponenten eines Verhaltens als niedere Verhaltensweisen eingestuft werden. Da diese Vorgehensweise keine klare Abgrenzung der Begriffe zulässt, soll von dieser Definition für höheres Verhalten Abstand genommen werden.

Im Rahmen dieser Arbeit besitzt der Begriff des niederen Verhaltens die eingangs genannte Bedeutung. Der Begriff des Verhaltensprimitivs wird meist synonym dazu verwendet, er kann aber auch die Kombination einiger weniger niederer Verhalten bezeichnen.

Von höherem Verhalten wird nur dann gesprochen, wenn ein Charakter auf eine Situationsänderung angemessen reagiert. Insbesondere kann es auch dann als höheres Verhalten bezeichnet werden, wenn die Reaktion durch ein einziges Verhaltensprimitiv erfolgt. Diese Definition von höherem Verhalten setzt voraus, dass ein Charakter sein Verhalten ändern kann. Welche Reaktion in einer Situation als angemessen erscheint, obliegt letztendlich der Beurteilung des Betrachters und ist vom Anwendungsfall abhängig.

Zur Aufgabenstellung gehört es, die höheren Verhaltensweisen am Beispiel einer Unterwassersimulation mit Fischen zu veranschaulichen. Diese wird in Kapitel 5 beschrieben. Hierzu befinden sich auf der beliegenden CD weitere Bilder und Videos.

1.2 Rahmenbedingungen

Diese Diplomarbeit wurde in Zusammenarbeit mit der Vertigo Systems GmbH aus Köln durchgeführt. Dadurch entstanden einige Anforderungen an das Framework, die aus der Unternehmensstruktur und den Vorgaben der Firma hervorgehen. Aus diesem Grund wird die Firma zunächst kurz vorgestellt. Danach wird das Anwendungsszenario beschrieben, für das die Unterwassersimulation erstellt werden soll.

Am Ende des Abschnitts werden die sich aus der Zusammenarbeit mit Vertigo und dem Anwendungsszenario ergebenden Anforderungen in Stichpunkten zusammengefasst.

1.2.1 Vertigo Systems

Die Firma Vertigo Systems GmbH aus Köln ist ein SpinOff-Unternehmen des Bonner Fraunhofer IMK, das 1999 gegründet wurde (vergleiche [IQ 9]). Vertigo Systems ist spezialisiert auf die Erstellung von immersiven Virtual-Reality-Anwendungen für den Entertainment- und Infotainmentbereich.

Der Hauptkundenstamm besteht aus Firmen, Veranstaltern und Museen, die eine VR-Installation als Highlight oder besonderen Blickfang nutzen möchten. Die Anforderungen der Kunden an ein Endprodukt können sich von Auftrag zu Auftrag stark unterscheiden, wobei häufig nur ein Entwicklungszeitraum von wenigen Monaten zur Verfügung steht.

Die für ein Produkt benötigten 3D-Modelle und Animationen innerhalb der Modelle werden normalerweise von einer Partnerfirma erstellt. Vertigo Systems bindet diese Modelle in die VR-Umgebung ein und steuert sie. Da die Steuerung bei früheren Projekten durch die wechselnden Anforderungen oft einen nicht unerheblichen Teil der Entwicklungszeit beansprucht hat, entstand der Wunsch nach einer allgemeineren Lösung. Daraus resultierte die Aufgabenstellung der Diplomarbeit.

1.2.2 Das Anwendungsszenario: living

Im Verlauf der Diplomarbeit zeichnete sich schon früh ein konkretes Produkt ab, in dem die Verhaltensframework zum Einsatz kommen sollte. Dieses ist das

sogenannte **living** System, dass von Vertigo System in Kooperation mit rmh new media und der mediaMotion AG entwickelt wurde.

Das **living** System besteht aus einer Projektion an Decke, Wand oder Fußboden (siehe Abbildung 1), in der sich virtuelle Charaktere in einer dreidimensionalen Umgebung bewegen. Durch eine Schattenerkennung auf der projizierten Fläche kann ein Benutzer Einfluss auf das Geschehen in der virtuellen Welt nehmen.

Die Schattenerkennung funktioniert über eine Infrarotlampe, die in der Nähe des Projektors (Nr. 1 in Abbildung 1) angebracht ist und einer infrarotempfindlichen

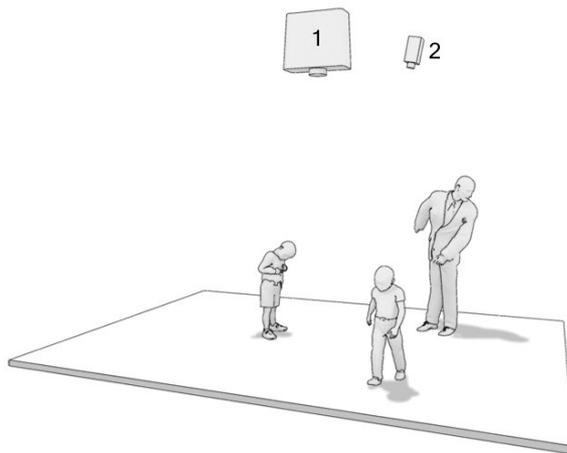


Abbildung 1: Beispielaufbau des **living** Systems

Kamera (Nr. 2 in Abbildung 1). Sobald ein Benutzer in den Projektionsstrahl des Projektors tritt, wirft er neben seinen vom Projektor verursachten Schatten gleichzeitig einen für ihn unsichtbaren Schatten im Infrarotbereich. Dieser wird von der Kamera aufgenommen und über ein Trackingverfahren erkannt. Der Schatten wird dann an das System weitergegeben und die virtuellen

Charaktere können auf ihn - also auf den Benutzer - reagieren.

Die Interaktion mit den Charakteren und anderen Teilen der Szenerie lässt beim Benutzer die Illusion entstehen, in das Geschehen einer realen, lebendigen Welt eingreifen zu können. Anders als bei einem Computerspiel manipuliert der Benutzer die virtuelle Welt also mit seinem eigenen Körper. Die Charaktere in der Projektion haben deshalb keine Möglichkeit, aktiv auf die Bewegungen des Benutzers Einfluss zu nehmen, wie es bei einer Figur in einem Spiel möglich wäre. In der Regel handelt es sich bei dem dargestellten Inhalt um ein mehr oder weniger abstraktes Szenario mit spielerischen Elementen.

Das Beispiel aus Abbildung 2 auf Seite 7 zeigt das System als Teil des Messestandes des Landes NRW während der Hannovermesse 2005 auf dem Technik für Brennstoffzellen präsentiert wurde. Zu sehen ist ein auf den Fußboden projizierter Pool mit virtuellem Wasser. In ihm bewegen sich große, gelbe und kleinere, rote Moleküle, die die Messebesucher durch Bewegungen über dem Pool verscheuchen konnten. Die gelben Moleküle bewegen sich langsamer durch das

Wasser und sind weniger agil als die roten.



Abbildung 2: Das *living System* im Einsatz

Das Beispiel zeigt, dass das Verhalten der Charaktere keiner Überprüfung auf biologische oder physikalische Korrektheit standhalten muss. Es handelt sich vielmehr um tier-ähnliche Verhaltensweisen. Die Simulation von Gestik oder Mimik, die den inneren Gefühlszustand eines Charakters widerspiegeln könnten, ist nicht geplant.

Es wird also für eine Unterwasserwelt mit Fischen nicht nötig sein, den Artenreichtum oder das Verhaltensspektrum eines kompletten Ökosystems naturgetreu nachzubilden. Statt dessen genügt eine abstraktere, vereinfachte Simulation, die dem Betrachter auf den ersten Blick als lebendig erscheint. Das bedeutet, dass ein Fisch nicht unbedingt essen muss, um zu überleben. Er muss sich auch nicht fortpflanzen, um nicht auszusterben. Statt dessen muss er aber auf Knopfdruck einem anderen Fisch hinterherjagen können, damit ein ankommender Besucher unterhalten wird.

Aus diesen Bedingungen ergeben sich einige Freiräume, da man nicht gezwungen ist, biologisch korrektes Verhalten zu erstellen, sondern nur ein natürlich *wirkendes* Verhalten. Hieraus ergibt sich gleichzeitig das Problem, dass man darauf vorbereitet sein muss, auch unnatürliches oder unlogisches Verhalten erstellen zu können. Zum Beispiel wäre es vorstellbar, dass ein Fisch, der sonst immer vor seinen Feinden davongeschwommen ist, plötzlich kehrt machen soll, damit er von einem anderen Fisch gefressen werden kann. Das Verhaltensframework muss solche Sonderbehandlungen zulassen.

Die Anzahl der zu simulierenden Charaktere wird in Abhängigkeit der Individuengröße und mit Größe der Projektionsfläche schwanken. Im Beispiel waren es nur ca. 15 Moleküle, die gleichzeitig den Pool bevölkerten. Für andere Szenarien wären auch Größenordnungen von hundert bis dreihundert Individuen denkbar. Da die Charaktere aber nicht beliebig klein werden können, um von einem Benutzer noch erkannt zu werden, sind größere Zahlen unwahrscheinlich.

1.2.3 Anforderungslisten

Folgende formale Anforderungen wurden von Vertigo Systems, an die im Rahmen dieser Diplomarbeit anzufertigende Software, gestellt:

- Die Software ermöglicht die Erstellung autonomer Charaktere für eine Unterwasserwelt.
- Sie wird mit C++ entwickelt.
- Sie ist zu Linux kompatibel.
- Die Software kann mit geringem Arbeitsaufwand auf andere Anwendungsfälle als eine Unterwasserwelt übertragen werden.
- Falls sie auf eine bereits vorhandene Software aufbaut, dann soll diese aus dem OpenSource-Bereich stammen.

Folgende inhaltliche Anforderungen ergeben sich aus dem Anwendungsszenario:

- Es soll primär tierähnliches Verhalten erstellt werden. Ausnahmen sollen zugelassen werden.
- Das Verhalten äußert sich durch eine Bewegungsänderung des Charakters, nicht durch Gestik oder Mimik.
- Feineinstellungen wie die Geschwindigkeit, Agilität, etc. eines Charakters müssen auch kurzfristig an örtliche Bedingungen (z.B. Größe der Projektion) angepasst werden können.

Kapitel 2. Theorie

Dieses Kapitel erläutert verschiedene Techniken, die zur Erstellung von autonomen Charakteren herangezogen werden können.

Zunächst werden die Arbeiten einzelner Personen vorgestellt, die als Grundlage für den weiteren Verlauf dienen, da sie mitunter richtungsweisende Techniken hervorgebracht und neue Maßstäbe bei der Generierung von Verhalten gesetzt haben. Anschließend werden allgemeine Verfahrensweisen aus der KI, die zur Erstellung von Charakteren mit höherem Verhalten geeignet sind, vorgestellt. Im letzten Abschnitt erfolgt ein Überblick über derzeit verfügbare OpenSource Projekte, die sich mit der Steuerung autonomer Charaktere auseinandersetzen oder bei deren Erstellung hilfreich sein könnten.

Am Ende jedes Abschnitts werden die vorgestellten Inhalte im Hinblick auf ihren möglichen Nutzen für die eigene Aufgabenstellung diskutiert. Die Diskussionen bilden die Grundlage für die im nachfolgenden Kapitel beschriebene Entscheidungsfindung zum Konzeptdesign.

2.1 Verwandte Arbeiten

In der wissenschaftlichen Literatur finden sich zahlreiche Arbeiten, die sich mit der Steuerung von autonomen Charakteren auseinandersetzen. Viele davon sind durch die Robotik motiviert, und die Frühesten von ihnen reichen bis in die 1940er Jahre zurück. Auf Grund der Fülle an Arbeiten wäre es utopisch an dieser Stelle einen auch nur annähernd kompletten Überblick geben zu wollen. Deshalb soll in diesem Abschnitt ein Einblick in die Forschungen derjenigen Personen gegeben werden, die den größten Einfluss auf das in Kapitel 3 und 4 beschriebene Framework hatten.

2.1.1 Craig W. Reynolds

Bereits 1987 entwickelte Reynolds ein Modell für komplexes Schwarmverhalten im dreidimensionalen Raum (siehe [11]). Dabei besitzt jeder „Boid“, wie Reynolds seine autonomen Charaktere nennt, drei einfache Regeln, durch deren Anwendung ein gemeinsames Schwarmverhalten simuliert wird. Diese Regeln

sind *Separation*, *Cohesion*, und *Alignment* durch die ein Boid stets „bemüht“ ist, sich in dieselbe Richtung wie seine Nachbarn zu bewegen (*Alignment*), während er einen gewissen Abstand zu ihnen einhält (*Separation*) und sich an eine ähnliche Position wie diese begibt (*Cohesion*). Ab Seite 11 wird genauer auf diese Regeln eingegangen.

Ein Jahr später fügt Reynolds seinen Boids eine Regel zur Vermeidung von Hindernissen hinzu (siehe [12]). Diese Regel steht in ihrer Hierarchie über den Regeln zur Schwarmbildung. Das bedeutet, dass die Schwarmregeln nur angewandt werden, wenn eine Hindernisvermeidung nicht notwendig ist.

Das besondere am Boid-Modell ist, dass es unabhängig von der konkreten Animation der Bewegungen ist und somit für jede Art von Charakter, ob Mensch, Auto, Fisch oder Flugzeug, einsetzbar ist. Reynolds legte damit den Grundstein für die sogenannte „Behavioral Animation“, der verhaltensgesteuerten Animation. Ein Charakter ist dabei selbstständig in der Lage, zu entscheiden, *wie* eine Animation oder Bewegung ausgeführt wird. Der Animator muss dann lediglich vorgeben, *was* der Charakter tun soll. Zum Beispiel braucht ein virtueller Fisch lediglich die Anweisung „weiche Hindernissen aus“ von höherer Ebene erhalten und ist danach selbstständig in der Lage diesen Befehl auszuführen. Dies erleichtert die Arbeit des Animators, da er auf einem höheren Abstraktionsniveau arbeiten kann.

1999 veröffentlicht Reynolds „Steering Behaviors For Autonomous Characters“ (siehe [13]), in dem er die Regelmenge zur Steuerung seiner Boids, die hier Vehikel genannt werden, erheblich erweitert. Das Paper bildet gleichzeitig die Grundlage für sein 2003 begonnenes OpenSource-Projekt „OpenSteer“, welches in Kapitel 2.3.3 genauer beschrieben ist.

Um die Vehikel zu konkretisieren, aber ohne den Anspruch auf Allgemeingültigkeit zu verlieren, stellt Reynolds ein einfachstes Vehikelmodell vor. Nach diesem Modell entspricht jedes Vehikel einem Massepunkt im Raum, der durch seine Masse, seine Position, seiner Orientierung im Raum und einen Geschwindigkeitsvektor definiert ist. Zusätzlich gibt es eine Höchstgeschwindigkeit und eine maximale Kraft, die auf das Vehikel wirken kann.

Aus jeder Anwendung einer Regel, die auch als „steering behavior“ bezeichnet wird, resultiert ein Kraftvektor, in dessen Richtung das Vehikel idealerweise steuern muss, um die Regel zu erfüllen. Dieser Kraftvektor wird durch die

maximale Kraft des Vehikels begrenzt und durch dessen Masse dividiert. Nach Addition des Ergebnisses mit der alten Geschwindigkeit und Begrenzung mit der Höchstgeschwindigkeit, erhält man die neue Geschwindigkeit des Fahrzeugs. Die neue Position ergibt sich durch Addition der neuen Geschwindigkeit zur alten Position.

Um die Orientierung des Vehikels an den neuen Geschwindigkeitsvektor anzupassen, stellt Reynolds einen heuristischen Ansatz vor. Dabei wird davon ausgegangen, dass sich die Orientierung von einem Berechnungsschritt zum nächsten nur geringfügig ändert. Ein ungefährender Wert für den neuen „Oben“-Vektor des Vehikels ergibt sich durch Addition des neuen Vorwärtsvektors mit dem globalen „Oben“-Vektor (üblicher Weise die Y-Achse). Um das lokale Koordinatensystem des Vehikels wieder orthogonal auszurichten, wird das Kreuzprodukt der Näherung des „Oben“-Vektors mit dem neuen Vorwärtsvektor gebildet. Daraus ergibt sich der neue „Seiten“-Vektor des Vehikels. Der tatsächliche „Oben“-Vektor wird durch das Kreuzprodukt des neuen „Seiten“-Vektors mit dem Vorwärtsvektor berechnet.

Die wichtigsten neuen Regeln, die Reynolds in seiner Arbeit vorstellt, werden im Folgenden beschrieben. Um mit der Namensgebung von Reynolds und der von OpenSteer Konformität zu wahren, wurden die anglizistischen Namensgebungen der Verhaltensregeln beibehalten.

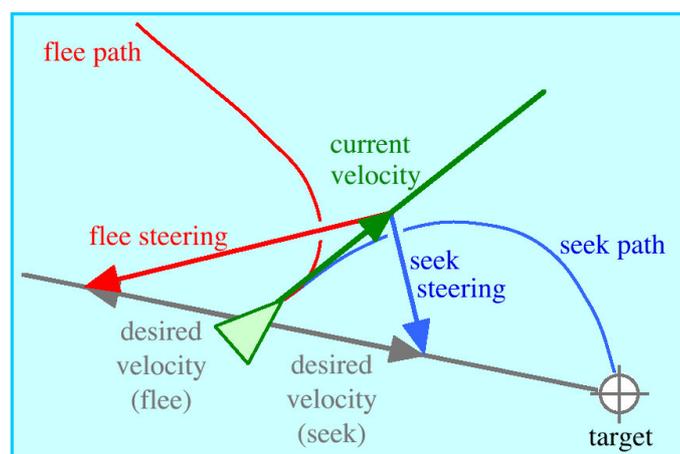


Abbildung 3: Seek und Flee Verhalten

Seek

Das Seek Verhalten steuert das Vehikel in Richtung eines bestimmten Zieles. Die gewünschte Geschwindigkeit ergibt sich aus dem Vektor zwischen Ziel und

Vehikel. Der Steuerungsvektor, den dieses Verhalten liefert, ist der Differenzvektor zwischen der gewünschten und der aktuellen Geschwindigkeit.

Es muss beachtet werden, dass ein Vehikel, welches das angesteuerte Ziel erreicht, dort nicht etwa abbremst und stehen bleibt. Auf Grund seiner Masseträgheit wird es über das Ziel hinaussteuern, danach wenden und dann wieder auf das Ziel zusteuern. Das Ergebnis ähnelt einer Motte, die um eine Lampe kreist. Will man diesen Effekt vermeiden, muss statt dem Seek das Arrival Verhalten (siehe Seite 13) angewendet werden.

Flee

Der Fluchtvektor, den dieses Verhalten liefert, errechnet sich ähnlich zum Seek-Vektor. Lediglich der Vektor für die gewünschte Geschwindigkeit verläuft entgegengesetzt.

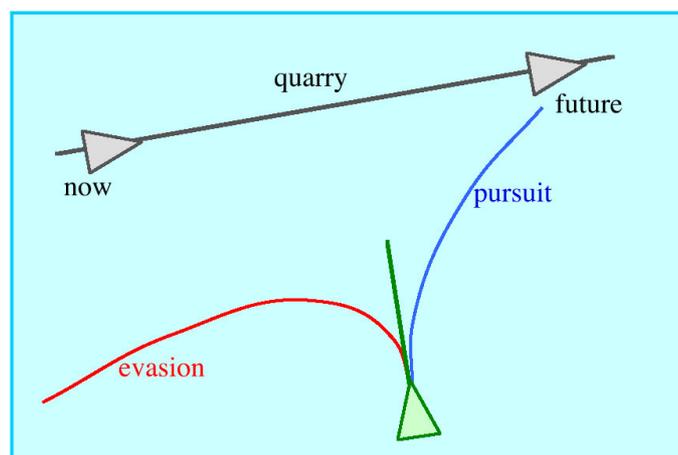


Abbildung 4: Pursuit und Evasion Verhalten

Pursuit

Pursuit liefert einen Vektor zum Verfolgen oder Abfangen eines anderen Vehikels. Es funktioniert ähnlich zum Seek Verhalten, nur dass es hierbei gilt, ein bewegliches Ziel zu erreichen, wofür die Eigenbewegung des anderen Vehikels geschätzt werden muss. Zur Vereinfachung wird angenommen, dass sich das andere Vehikel auf einer geraden Linie bewegt (siehe Abbildung 4). Die zukünftige Position des Fahrzeugs im nächsten Zeitschritt errechnet sich durch Addition seiner aktuellen Position und seiner aktuellen Geschwindigkeit. Um letztere zu ermitteln, kann entweder die Differenz zwischen aktueller und vorheriger Position errechnet werden, oder diese dem anderen Fahrzeug durch eine Art Gedankenübertragung mitgeteilt werden. Nachdem die zukünftige Position bekannt ist, kann das Seek Verhalten auf diese Position angewendet werden.

Evasion

Entgegen dem Abfangen und Verfolgen von anderen Vehikeln durch das Pursuit Verhalten, können diese mit Evasion gemieden werden. Die Berechnungen verlaufen analog, nur dass auf die zukünftige Position des anderen Vehikels das Flee Verhalten angewendet wird.

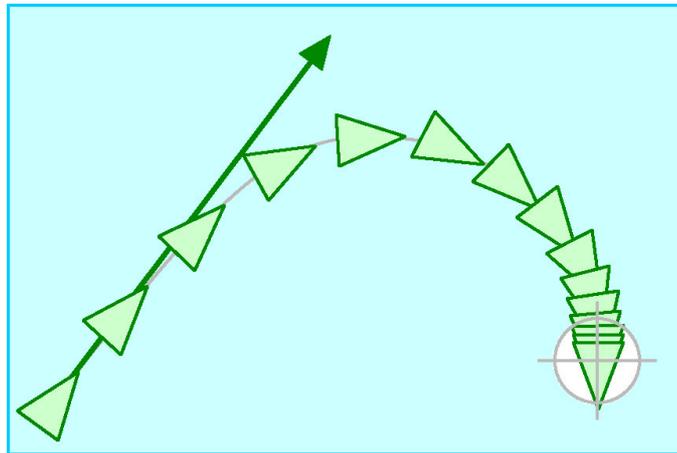


Abbildung 5: Arrival Verhalten

Arrival

Das Verhalten hat eine ähnliche Wirkung wie das Seek-Verhalten. Es bewirkt, dass das Vehikel auf eine bestimmte Position zusteuert und an dieser zum Stehen kommt (siehe Abbildung 5).

Die Distanz, ab der das Vehikel zu bremsen anfängt, wird als Parameter angegeben. Wenn der Abstand des Vehikels kleiner als die Distanz ist, wird dessen Geschwindigkeit immer weiter verringert, bis es den Zielpunkt erreicht hat. Die Geschwindigkeit im Zielpunkt ist gleich Null. Die Verringerung kann linear oder auch exponentiell geschehen.

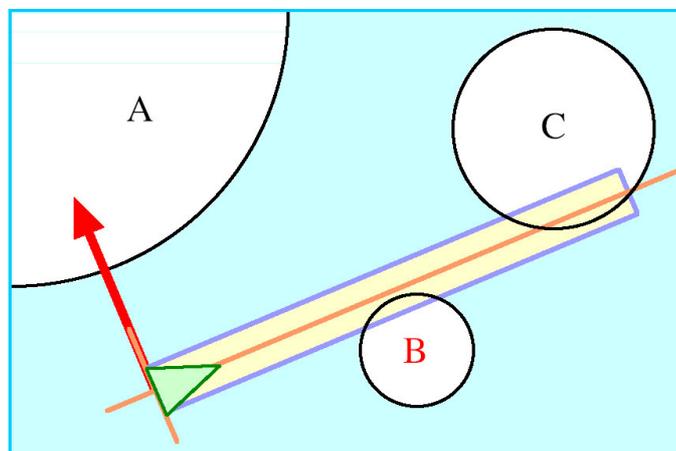


Abbildung 6: Obstacle Avoidance Verhalten

Obstacle Avoidance

Dieses Verhalten dient der Vermeidung von Hindernissen. Um die Berechnungen zu vereinfachen, nimmt Reynolds sowohl für das Vehikel als auch für das Hindernis eine kugelförmige Oberfläche an (siehe Abbildung 6).

Das Obstacle Avoidance Verhalten versucht stets einen zylindrischen Korridor in Fahrtrichtung des Vehikels frei zu halten. Wenn sich Hindernisse innerhalb dieses Korridors befinden, wird der Ausweichvektor berechnet (in Abbildung 6 geschieht dies für Hindernis B). Dieser ergibt sich aus der Negation der Projektion des Hindernismittelpunktes auf den in Bewegungsrichtung zeigenden Vektor des Fahrzeugs. Seine Länge beträgt die maximale Kraft des Vehikels.

Aus Sicht des Vehikels entspricht das einem Ausweichen nach links, wenn der Hindernismittelpunkt auf der rechten Seite des Vehikels liegt. Um sich bei der Form des Hindernisses nicht auf Kugeln beschränken zu müssen, kann für beliebige konkave Objekte, die sich innerhalb des Sichtkorridors befinden, ein Senkrecht zur Bewegungsrichtung zeigender Vektor zum Ausweichen verwendet werden, der aus Sicht des Vehikels in Richtung der Oberflächennormalen des nächsten Schnittpunktes des Objektes mit dem Sichtkorridor zeigt.

Reynolds weist darauf hin, dass das Obstacle Avoidance Verhalten lediglich eine Hindernisvermeidung bietet und weder eine Kollisionserkennung noch eine Kollisionsbehandlung darstellt. Wenn die Geschwindigkeit eines Vehikels im Verhältnis zu seiner maximalen Kraft zu groß ist, kann ein Hindernis durchbrochen werden, wodurch sich das Vehikel danach in oder hinter dem Hindernis befindet. Da das Vehikel selbst dies nicht bemerkt, steuert es nicht wieder von alleine zurück. Damit dieses Fehlverhalten möglichst selten oder gar nicht auftritt, müssen die Masse, die maximale Kraft und die maximale Geschwindigkeit des Vehikels „sinnvolle“ Werte besitzen. Die Entscheidung, wann die Werte in einem sinnvollen Verhältnis zueinander stehen, überlässt Reynolds der Entscheidungskraft des Programmierers.

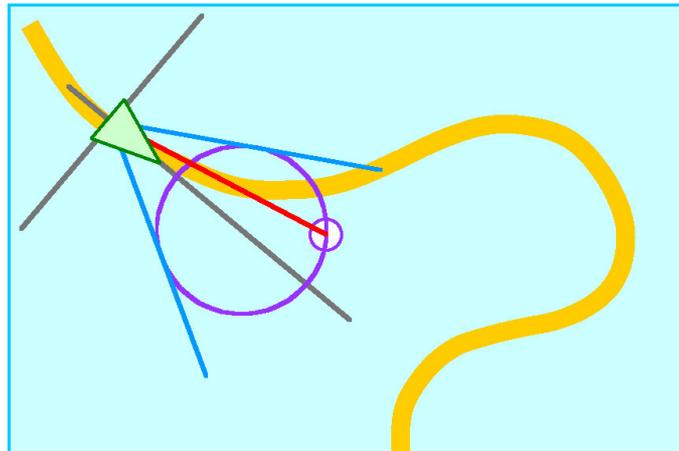


Abbildung 7: Wander Verhalten

Wander

Hierdurch wird ein Umherwandern des Vehikels ohne bestimmtes Ziel ermöglicht. Es ist vergleichbar mit einem Tier, das sich auf Futtersuche befindet und dafür in der Umgebung umherstreift.

Im Prinzip liefert das Wander Verhalten einen zufälligen Steuerungsvektor zurück. Der Vorteil des hier vorgestellten Verfahrens im Gegensatz zu vollkommen zufällig gewählten Richtungsänderungen, ist ein natürlicheres Erscheinungsbild. Hierzu werden pro Simulationsschritt nur kleine Richtungsänderungen zugelassen. Dies wird erreicht, indem der Steuerungsvektor stets von der Position des Vehikels aus, auf einen zufälligen Punkt auf einem gedachten Kreis vor dem Vehikel verweist. Die Größe des Kreises bestimmt die Größe der Kurven, die das Vehikel beim Umherwandern beschreibt. Ein weiterer, kleinerer Kreis beschränkt die maximale Änderung des Steuerungsvektors auf dem größeren Kreis pro Simulationsschritt (siehe Abbildung 7). Durch seine Größe wird bestimmt, wie schnell ein Vehikel seine aktuelle Steuerungsrichtung ändern kann.

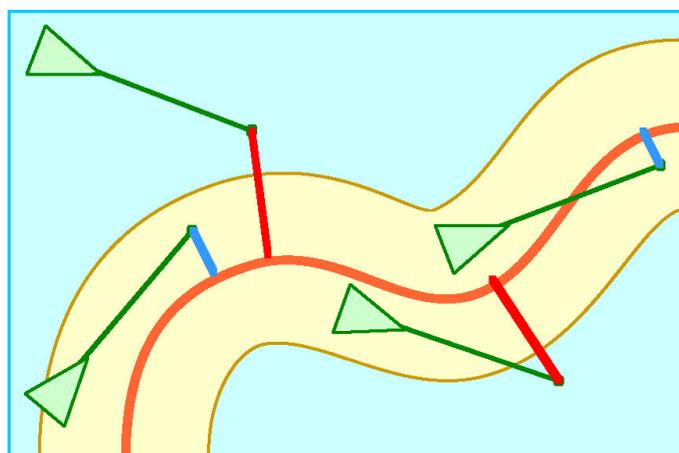


Abbildung 8: Path Following Verhalten

Path Following

Path Following erlaubt es einem Vehikel einem vorher definierten Pfad zu folgen. Es ist vergleichbar mit Menschen, die sich auf dem Bürgersteig bewegen oder Fischen, die durch ein Rohr schwimmen.

Der Pfad wird durch einen Spline und seinen Radius definiert. Ein Vehikel versucht, sich immer innerhalb des Radius des Splines zu bewegen. Dies ist über die Schätzung der zukünftigen Position des Vehikels und der Projektion dieser auf den nächsten Punkt der Spline möglich. Ist der Abstand der zukünftigen Position von der Projektion größer als der Pfadradius, wird das Seek-Verhalten auf den Projektionspunkt angewendet. Ist der Abstand größer als der Pfadradius, ist keine Korrektur nötig.

Aus den gleichen Gründen, wie sie bereits beim Obstacle Avoidance Verhalten geschildert wurden (siehe Seite 14), kann es passieren, dass das Vehikel den Pfad verlässt. Anders als beim Obstacle Avoidance Verhalten würde das Vehikel in einem solchen Fall selbstständig zum Pfad zurückkehren.

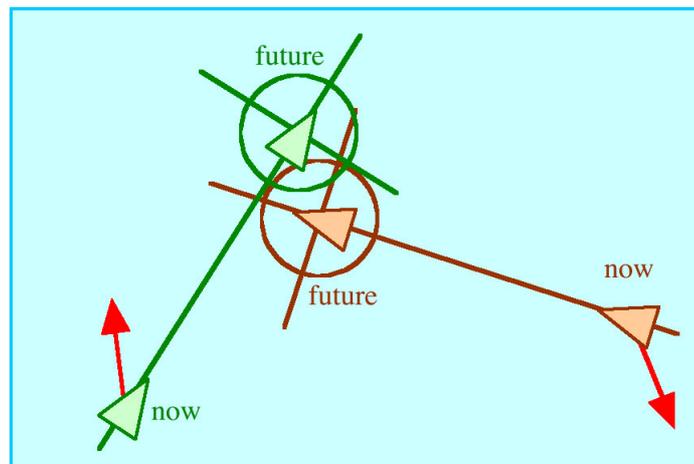


Abbildung 9: Unaligned collision avoidance Verhalten

Unaligned collision avoidance

Das Verhalten versucht, die Kollision mit anderen Vehikeln zu vermeiden.

Als potentielle Kollisionspartner werden alle Vehikel betrachtet, die sich zum aktuellen Zeitpunkt innerhalb einer bestimmten Distanz und in einem bestimmten

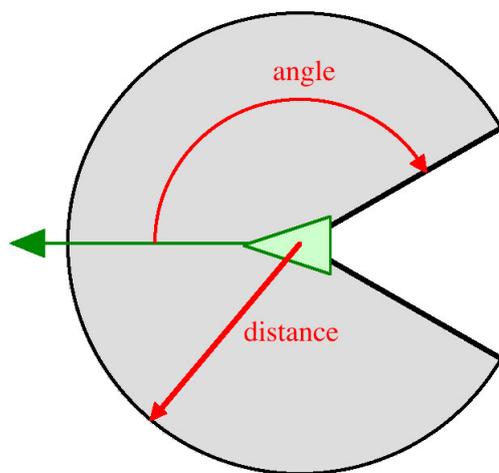


Abbildung 10: die Vehikel Nachbarschaft

Winkel zur Position des Vehikels befinden. Von diesen Vehikeln werden die zukünftigen Positionen zu einem als Parameter einstellbaren Zeitpunkt geschätzt. Wenn sich der Radius des nächsten Vehikels mit dem eigenen Radius überschneidet, besteht eine potentielle Kollision. In Abhängigkeit der Lage der Kollision aus Sicht des Vehikels, wird das Vehikel zur Kollisionsvermeidung seine Geschwindigkeit erhöhen oder verlangsamen. In jedem Fall wird es in

entgegengesetzte Richtung zum Kollisionspunkt steuern. Als Beispiel wird in Abbildung 9 das grüne Vehikel versuchen, seine Geschwindigkeit zu erhöhen und nach links auszuweichen, während das Andere seine Fahrt verlangsamt und ebenfalls nach links auszuweichen versucht.

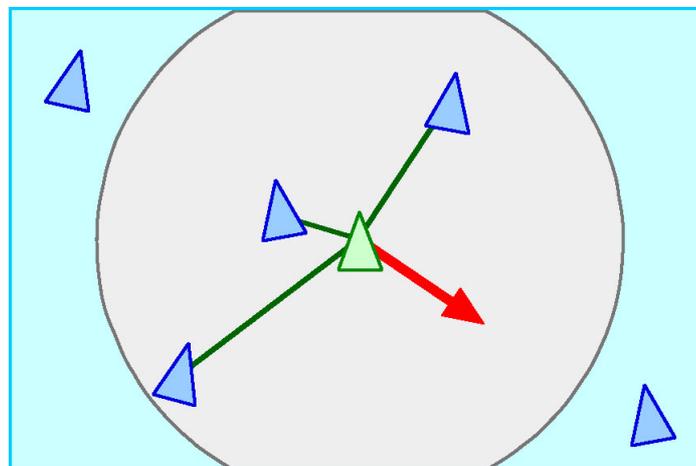


Abbildung 11: Separation Verhalten

Separation

Das Verhalten sorgt dafür, dass ein gewisser Abstand zu den Nachbarn des Vehikels eingehalten wird. Die Nachbarschaften ergeben sich wie in Abbildung 10 beschrieben. Der Steuerungsvektor ergibt sich aus der Summe aller Ausweichvektoren für jedes Vehikel. Diese berechnen sich durch Division des Vektors, der von der eigenen Position zur Nachbarposition zeigt, durch das negierte Skalarprodukt dieses Vektors. Dadurch wird der Abstand der beiden

Vehikel voneinander berücksichtigt und die Länge des Ausweichvektors nimmt bei reduzierter Distanz zu.

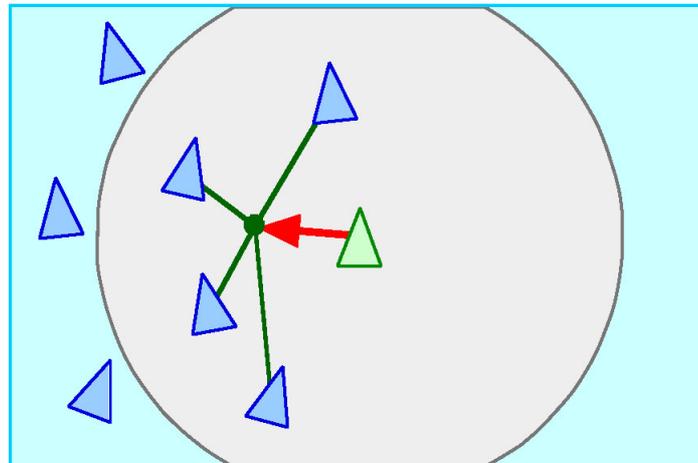


Abbildung 12: Cohesion Verhalten

Cohesion

Dieses Verhalten dient dazu, dass sich ein Vehikel in die Nähe anderer Vehikel begibt. Die Nachbarschaften ergeben sich wie in Abbildung 10 beschrieben. Der Punkt, an den sich das Vehikel begeben soll, ergibt sich aus dem Mittelwert aller Positionen der Nachbarvehikel. Auf diesen Punkt wird das Seek-Verhalten angewendet.

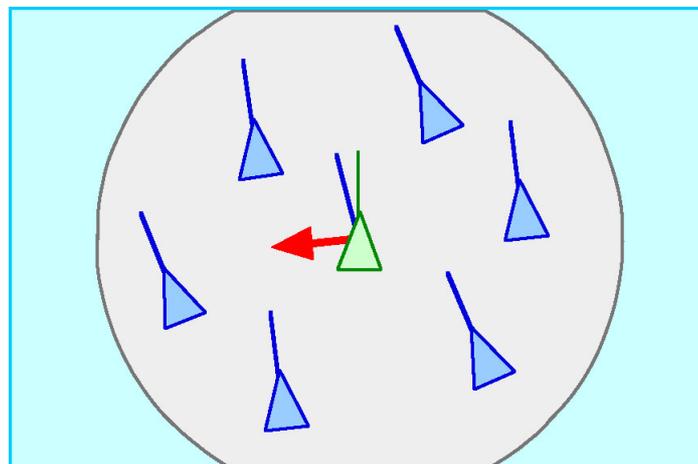


Abbildung 13: Alignment Verhalten

Alignment

Bei diesem Verhalten versucht das Vehikel seine Orientierung an die seiner Nachbarn anzupassen. Die Nachbarschaften ergeben sich wie in Abbildung 10 beschrieben. Um die gewünschte Ausrichtung zu erhalten, müssen zunächst alle Orientierungen der Nachbarn gemittelt werden. Davon wird die eigene Orientierung subtrahiert und der Ergebnisvektor normiert.

Flocking

Wenn Flocking auf mehrere Vehikel angewendet wird, lässt sich ein komplexes Schwarmverhalten simulieren. Es besteht, wie am Anfang des Kapitels bereits erwähnt, aus einer Kombination von Separation, Cohesion und Alignment, deren Ergebnisvektoren addiert werden. Für jede der drei Verhalten werden die Parameter zum Radius und Winkel der Nachbarschaftssuche getrennt angegeben. Ebenso ist es sinnvoll die einzelnen Verhalten mit einem Gewichtungsfaktor zu versehen. Reynolds schlägt vor, die Ergebnisvektoren vor der Gewichtung zu normieren. Dies soll eine bessere Kontrolle über das Schwarmverhalten ermöglichen.

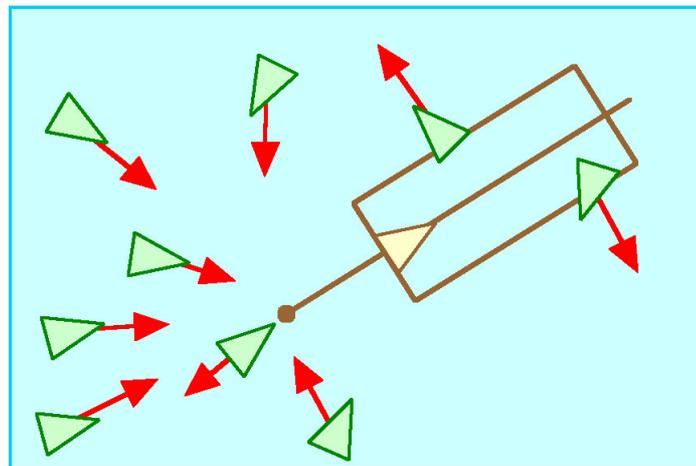


Abbildung 14: *Leader Following Verhalten*

Leader Following

Bei Anwendung dieses Verhaltens ist das Vehikel danach bestrebt, einem anderen als Anführer gekennzeichnetes Vehikel in einem gewissen Abstand zu folgen. Zusätzlich ist das Vehikel darum bemüht, den Weg des Anführers nicht zu versperren und Kollisionen mit anderen Vehikeln zu vermeiden.

Die Verfolgung des Anführers wird durch Anwendung des Arrival Verhaltens auf einen Punkt hinter dem Anführer erzielt. Sollte sich das Vehikel allerdings in einem vorher definierten quaderförmigen Bereich vor dem Anführer befinden (siehe Abbildung 14), würde es erst seitlich von der Vorwärtsrichtung des Anführers wegsteuern, bevor es das Arrival Verhalten wieder aufnimmt. Um den anderen Vehikeln auszuweichen wird das Separation Verhalten verwendet.

Reynolds gibt an, dass eine Applikation, die mehrere der Verhalten sinnvoll anwenden soll, ein intelligentes Auswahlverfahren für die Verhaltensweisen

benötigt. Dieses muss zwischen Verhalten, die sich gegenseitig ausschließen und Verhalten, die gleichzeitig aktiv sein können unterscheiden. Beispielsweise kann ein Reh, das auf einer Wiese äst, nicht gleichzeitig vor einem Wolf davonlaufen. Es kann aber sehr wohl vor dem Wolf fliehen und gleichzeitig einem Baum ausweichen wollen.

Um mehrere gleichzeitig aktive Verhaltensweisen zu realisieren, schlägt Reynolds zwei Lösungsansätze vor. Der Erste ist eine einfache Addition der Steuerungsvektoren, welche die Verhalten zurückliefern. Falls jedem Verhalten ein unterschiedliches Maß an Bedeutsamkeit zugeordnet werden soll, könnten die Steuerungsvektoren gewichtet werden. Als Nachteile werden hierbei die mögliche Aufhebung der Steuerfaktoren und ein Performanceverlust durch die kontinuierliche Berechnung der Vektoren aufgeführt.

Deshalb lautet der zweite Vorschlag von Reynolds, den Verhaltensweisen unterschiedliche Prioritäten zuzuweisen und niederrangige Verhalten nur dann zu berechnen, wenn alle der Höherrangigen einen Null-Vektor zurückliefern.

Eine Hybrid-Technik zwischen beiden Ansätzen besteht darin, die Verhalten nach Prioritäten zu sortieren und jedem eine Wahrscheinlichkeit zuzuordnen, mit der es ausgewertet wird. Das Verhalten mit der höchsten Priorität wird also nur mit einer gewissen Wahrscheinlichkeit berechnet. Ist die Wahrscheinlichkeit gegeben und liefert es einen Wert zurück, wird dieser verwendet. In allen anderen Fällen wird das nächste Verhalten mit einer gewissen Wahrscheinlichkeit berechnet und so weiter.

Nach der Beschreibung von Reynolds ist es in den meisten Fällen ausreichend, die zuerst vorgestellte Additions-Methode oder die Hybrid-Technik zu verwenden.

2.1.2 Xiaoyuan Tu

Zwischen 1993 und 1995 veröffentlichte Xiaoyuan Tu diverse Artikel über die Erstellung natürlich wirkender virtueller Fische. Dies geschah häufig in Zusammenarbeit mit Demetri Terzopoulos, auf dessen Arbeiten ab Seite 24 näher eingegangen wird. Die Thematiken, mit denen sich die Abhandlungen beschäftigen, reichen von der Umsetzung der Biomechanik und der Bewegungsabläufe, bis hin zur Wahrnehmung und des Verhaltens der Fische. Eine Art Zusammenfassung aller Veröffentlichungen findet man in der Doktorarbeit von Xiaoyuan Tu [18], die sie 1996 an der Universität Toronto beendet hat.

Die Ergebnisse ihrer Arbeiten beeindrucken durch ein hohes Maß an Realismus, sowohl im Verhalten als auch in den Bewegungen der Fische. Besonders hervorzuheben ist, dass sich die Fische vollständig autonom in ihrer Welt bewegen können, ohne dass vorherige Anweisungen von einem Animateur oder durch Skripte erfolgen müssten.

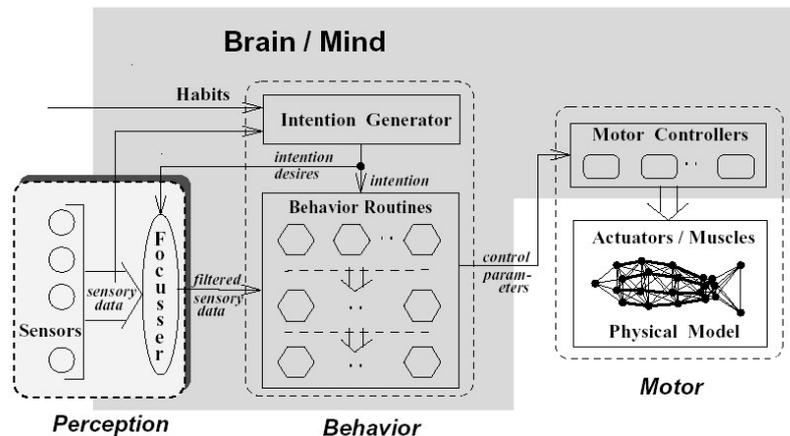


Abbildung 15: Aufbau eines Fisches

Der Aufbau eines virtuellen Fisches von Tu ist in drei voneinander weitgehend unabhängige Bausteine gegliedert. Diese sind die Wahrnehmung (Perception), das Verhalten (Behavior) und der Bewegungsapparat (Motor), die in Abbildung 15 dargestellt sind. Die in der Abbildung grau hinterlegten Teile der drei Bausteine sollen nach den Angaben von Tu als generelles Framework für virtuelle Tiere in nahezu beliebigen Ökosystemen genutzt werden können.

Der Bewegungsapparat

Das Fischmodell von Tu besteht im Wesentlichen aus mehreren Massepunkten von denen einige durch Federn miteinander verbunden sind. Die Federn dienen als Muskeln und können durch Kontraktion den Fisch bewegen. Dabei werden die Wasserverdrängung und die daraus resultierende Positionsänderung des Fisches nach den physikalischen Gesetzen korrekt berechnet. Die Kontraktion der Muskeln wird von den sogenannten Motor Controllern gesteuert. Jeder Motor Controller ist für eine bestimmte Bewegungsart zuständig und spricht eine bestimmte Muskelgruppe an. Diese sind z.B. *swim-MC* zum geradeaus schwimmen, *left-turn-MC* für eine Drehung nach links oder *ascend-MC* und *descend-MC* zum auf- und abwärts schwimmen. Insgesamt gibt es neun dieser Controller, die alle ihre festgelegte Muskelgruppe in einer bestimmten Frequenz und Amplitude ansprechen, um im Falle des *swim-MC* eine gewisse

Geschwindigkeit oder im Falle des *left-turn-MC* eine bestimmte Winkeländerung des Fisches zu erreichen. Die Kalibrierung der Controller geschah bei Tu manuell. Demetri Terzopoulos et. al. stellen in [16] und [17] eine Methode vor, wie ein Fisch nahezu selbstständig seine Motor-Controller zu nutzen lernt.

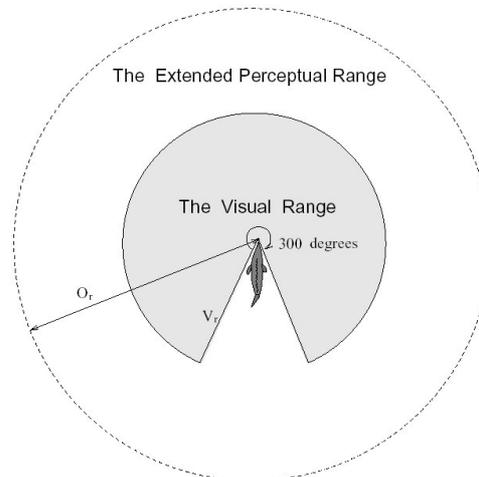


Abbildung 16: Wahrnehmungsbeschränkung

Die Wahrnehmung

Die Fische in der Unterwasserwelt von Tu besitzen zwei Sensoren zur Wahrnehmung. Der erste ist ein Temperaturfühler durch den der Fisch seine aktuelle Umgebungstemperatur messen kann. Dieser wird benötigt, da sich jede Fischart bevorzugt in einer bestimmten Wassertemperatur aufhält.

Der zweite und weitaus wichtigere Sensor entspricht einem zyklopischen Auge durch den ein Fisch seine Umwelt wahrnehmen kann. Die Sicht eines virtuellen Fisches ist, wie bei den meisten echten Fischen auch, auf einen 300° Winkel beschränkt. Zusätzlich wird die Reichweite der Sicht durch einen Radius eingeschränkt, welcher der Trübung des Wassers gerecht werden soll. Darüber hinaus erhält jeder Fisch einen erweiterten Sichtradius, in dem er nur die Präsenz von Beute, nicht aber die von anderen Objekten wahrnehmen kann. Ein Fisch wird von einem anderen Fisch erkannt, sobald sein Körpermittelpunkt innerhalb des Sichtradius des anderen Fisches liegt und mindestens einer seiner Massepunkte, aus denen das Fischmodell besteht, von keinem Hindernis verdeckt ist.

Tu schlägt zudem vor, die Größe eines Fisches bei seiner Erkennung zu berücksichtigen, damit größere Fische früher und kleinere Fische später erkannt werden. Sie führt dies aber nicht weiter aus.

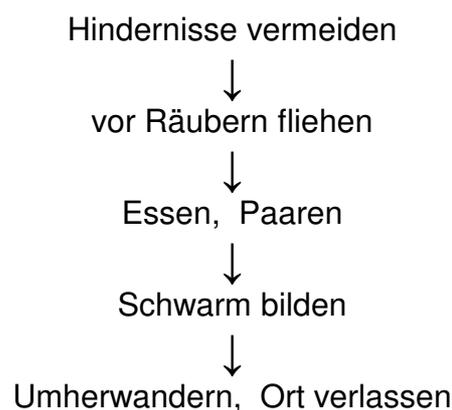
Nach dem Erkennen werden die Objekte zur weiteren Filterung an den sogenannten Fokussierer geschickt. Dieser leitet nur dasjenige Objekt an die

Verhaltensroutinen aus dem Verhaltenssystem weiter, welches zur Erfüllung der aktuellen Intention (Erläuterungen zum Intentionsgenerator siehe unten) benötigt wird. Lautet die Intention beispielsweise *Vermeide Hindernis*, wird nur das Hindernis weitergegeben, mit dem eine Kollision ohne Steuerungskorrektur zu erwarten ist. Zusätzlich wird eine *Bewegungspräferenz* erstellt und ebenfalls an die Verhaltensroutinen übergeben. Eine *Bewegungspräferenz* bezeichnet die generelle Richtung, in die sich ein Fisch bewegen würde, wenn er nicht im Moment seine aktuelle Intention erfüllen würde. Beispielsweise würde ein hungriger Fisch, dessen aktuelle Intention es ist, vor einem Hindernis auszuweichen, die *Bewegungspräferenz* „rechts“ haben, wenn sich zusätzlich zu dem Hindernis ein Beutefisch rechts vom Fisch aufhält.

Das Verhalten

Der Verhaltensapparat besteht aus dem Intentionsgenerator und den Verhaltensroutinen. Im Intentionsgenerator wird zunächst auf Grund der äußeren Einflüsse, die der Fisch durch seine Sensoren ermittelt hat und der inneren *Wünsche*, die jeder Fisch besitzt, eine Intention erstellt. Diese löst eine Verhaltensroutine aus, die wiederum die Motor Controller im Bewegungsapparat anspricht.

Die *Wünsche* sind hierarchisch geordnet und abhängig von der Art des Fisches. Das folgende Diagramm zeigt die *Wünsche* und ihre Ordnung:



Sie begründet das hierarchische Prinzip mit der Annahme von Vertretern der Ethologie¹, dass sich das Verhalten von Tieren hierarchisch ordnen lässt. Dabei

¹ Tu bezieht sich auf:

Lorenz, K.. *The Foundations of Ethology*. Springer-Verlag. New York, 1981,
Tinbergen, N.. *The Study of Instinct*. Clarendon Press, Oxford, England, 1950,

stehen Aktionen, die der Befriedigung primitiver Bedürfnisse dienen, in der Hierarchie über Aktionen, die höhere Bedürfnisse erfüllen.

In der Unterwasserwelt von Tu existieren drei verschiedene Fischtypen. Diese sind Jäger-, Beute- und Pazifistenfische. Allen Fischtypen ist gemeinsam, dass sie die *Wünsche* „Hindernisse vermeiden“, „Essen“ und „Umherwandern“ besitzen. Optional ist der *Wunsch* „Ort verlassen“. Dieser kann aktiviert werden, wenn der Fisch eine *Vorliebe* für warmes, kaltes, helles oder dunkles Wasser besitzen sollte, was sich für jedes Individuum einzeln angeben lässt.

Falls ein Jäger Hunger bekommt, wird er versuchen einen Fisch vom Typ Beutefisch zu fressen. Die Pazifisten- und die Beutefische ernähren sich hingegen von Plankton, das in der Unterwasserwelt in Form von kleinen Partikeln vorhanden ist. Die Beutefische besitzen als einzige die *Wünsche* „vor Räubern fliehen“ und „Schwarm bilden“. Die einzigen Fische mit Paarungstrieb sind die Pazifistenfische. Ob Beutefische die Flucht vor einem Feind ergreifen oder sich zu einem Schwarm zusammenfinden, ist abhängig von ihrer aktuellen Furcht, die durch eine Variable ausgedrückt wird. Die Variable kann beliebige Werte zwischen 0 und 1 besitzen und wird größer, je näher ein Jäger einem Beutefisch kommt. Ist ein Jäger noch weit entfernt, wird das Schwarmverhalten aktiviert. Erst wenn er näher kommt, ergreifen die Fische die Flucht.

Auch der Hunger und die Paarungsbereitschaft eines Fisches werden über Variablen gesteuert. Die Größe des Hungers ist allein von der Zeit der letzten Mahlzeit abhängig. Die Paarungsbereitschaft wird zusätzlich durch die aktuelle Verfügbarkeit eines potentiellen Partners beeinflusst.

2.1.3 Demetri Terzopoulos

Demetri Terzopoulos gilt als einer der Pioniere im „Artificial Life“ Sektor. Zusammen mit Radek Grzeszczuk betrieb er Mitte der neunziger Jahre intensive Forschungen im Bereich der Wahrnehmung ([16]), sowie dem eigenständigen Erlernen von Bewegungsabläufen ([8]) bei künstlichen Lebewesen. Als Testumgebung diente ihnen die von Tu erstellte und in Abschnitt 2.1.2 (ab S. 20) beschriebene Fischsimulation.

Um die Wahrnehmung eines Fisches zu simulieren, wird ausgehend von jedem Auge des Fisches ein Retinabild erzeugt. Über Bildverarbeitungsalgorithmen werden in den Bildern beispielsweise die Formen oder Farben von anderen Fischen erkannt. Auch die Position der erkannten Objekte wird auf Grund der Differenz beider Bilder zurückgerechnet. Ein Teilziel der Forschungen war es, die Objekterkennungsverfahren später in autonomen Robotern einsetzen zu können oder auch theoretische Wahrnehmungskonzepte zu überprüfen.

Für das Erlernen von Bewegungsabläufen wird auf das von Tu verwendete Feder-Masse Modell eines Fisches zurückgegriffen. Zunächst werden die einfachen Motorcontroller, die ein Fisch zum Vorwärts- oder Seitwärtsschwimmen benötigt, trainiert. Hierfür wird dem Fisch ein Zielpunkt vorgegeben, den er in einer bestimmten Zeit erreichen soll. Nach Ablauf der Zeit um, wird die Entfernung gemessen, die der Fisch zurückgelegt hat, und er wird wieder an seine Startposition gesetzt. Seine Muskeln werden dabei über ein neuronales Netz (siehe hierzu auch Abschnitt 2.2.3 ab S.30) gesteuert, das die Frequenz der Muskelbewegungen iterativ anpasst. Dadurch „lernt“ der Fisch mit jedem Iterationsschritt seine Muskeln besser einzusetzen.

Kompliziertere Bewegungsabläufe wie ein Sprung aus dem Wasser werden nach einem ähnlichen Prinzip eingeübt. Diesmal ist es die geeignete Reihenfolge und Dauer der Motorcontroller, die von dem Fisch iterativ erlernt wird.

2.1.4 John David Funge

John David Funge promovierte im Jahr 1998 an der Universität von Toronto in Kanada, an der zur selben Zeit Terzopoulos und Tu aktiv tätig waren. Teile seiner Arbeit sind durch seine Zusammenarbeit mit den beiden inspiriert. Seine Abhandlungen und Bücher (z.B. [3], [4] und [6]) behandeln die Fragestellung, wie Techniken aus der KI für Computeranimationen und Spiele verwendet werden können.

Ein wichtiges Ergebnis seiner Arbeit mit Tu und Terzopoulos ist die kognitive Modellierungssprache CML (engl.: cognitive modeling language), die in [6] beschrieben wird. Mit Hilfe von CML kann das Verhalten eines Charakters auf einer höheren Abstraktionsebene beschrieben werden. Dadurch wird es einfacher, Verhalten zu planen und zu verwalten. CML ermöglicht es einem Charakter, Wissen über seine Umgebung zu sammeln, um daraus Rückschlüsse auf die

Auswirkungen seines eigenen Verhaltens zu ziehen. Dies befähigt ihn, seine Handlungen zur Erfüllung konkreter Ziele zu planen.

In [6] werden zwei verschiedene Szenarien geschildert, in denen jeweils ein Charakter mit Hilfe von CML zu höheren, kognitiven Denkprozessen befähigt wird. Funge schildert, dass teilweise Performanceprobleme auftraten, wenn der Charakter komplexere Planungsprozesse ausführte. Die verwendete CML Version war in C++ implementiert. Auf der Homepage von Funge [IQ 3] wird eine CML Version für die Programmiersprache Java zum Download angeboten. Die C++ Version scheint nicht öffentlich zugänglich zu sein.

2.1.5 Diskussion der Arbeiten

Die Arbeit von Tu besitzt den engsten Bezug zu der eigenen, geplanten Unterwassersimulation. Ihr System hat bewiesen, dass es realistisches Verhalten von Fischen erzeugt und kann als eine Art Maßstab für ein eigenes Framework genommen werden.

Allerdings wird in Frage gestellt, ob der Aufbau des Systems wirklich so leicht als Framework für nahezu beliebige Ökosysteme genutzt werden kann, wie von Tu angegeben. Besondere Probleme werden im Wahrnehmungssystem und im Bewegungsapparat vermutet.

Im Bewegungsapparat wird nicht näher beschrieben, welche Schritte notwendig sind, um die zur Fortbewegung benötigten Motorcontroller auf andere Lebewesen als Fische zu übertragen. Es bleibt auch unklar, ob alle oder nur einige der Controller in einem anderen Tier verwendet werden können oder überhaupt sollen. Vermutlich sollen die Controllerarten beibehalten und deren Funktionalität an das jeweilige Bewegungsmodell des Tieres angepasst werden. Um aber alle Controller sinnvoll verwenden zu können, müsste sich das Tier in einer 3D Umgebung bewegen. Ein Pferd beispielsweise, das sich auf einer Wiese fortbewegt, würde keine Verwendung für die *ascend* und *descend* Motorcontroller haben.

Das Bewegungsmodell von Reynolds hingegen ist sehr allgemein gehalten. Es kann einfach und flexibel an eine Vielzahl verschiedener Charaktere angepasst werden. Eventuell ließen sich die Steuerungsvektoren von Reynolds auf die Motorcontroller von Tu umrechnen und damit das von Tu erstellte Feder-Masse-Modell eines Fisches steuern. Eine einfachere Möglichkeit zur Animation besteht darin, auf vorgefertigte Animationen zurückzugreifen, und diese an die

Geschwindigkeit von Reynolds Vehikel anzupassen. Die visuell beeindruckenden Ergebnisse von Tu wären mit dieser Methode allerdings nicht zu erwarten.

Um ein künstliches Lebewesen zu erstellen, welches selbstständig erlernt, seine Bewegungsmöglichkeiten zu nutzen, so wie es Terzopoulos beschreibt, ist leider eine Simulation der physikalischen Gesetzmäßigkeiten, die zur Fortbewegung führen, erforderlich. Ohne eine Berechnung der Wasserverdrängung ist der Ansatz von Terzopoulos in einer eigenen Unterwassersimulation also nicht umsetzbar. Da es das **living** System vorsieht, diverse Charaktere mit verschiedensten Fortbewegungsarten zu simulieren, wäre auch immer eine entsprechende physikalische Simulation und ein Bewegungsmodell notwendig. Beides ist sehr aufwendig und für Vertigo Systems erst bei einer mehrfachen Verwendung rentabel.

Das Wahrnehmungssystem von Tu besitzt den Nachteil, dass ein mehrfacher Austausch von Daten mit dem Intentionsgenerator des Verhaltensapparats stattfindet. Somit ist die Wahrnehmung abhängig vom Verhalten des Fisches. Dies erschwert für ein Framework den Aufbau einer modularen Struktur. Das Wahrnehmungskonzept von Terzopoulos entspricht zwar annähernd den natürlichen Sehgewohnheiten eines Fisches, ist für den Einsatz in virtuellen Welten aber zu rechenintensiv. Reynolds stellt zwar keinen Ansatz zur Wahrnehmung vor, lässt die Möglichkeit zur Wahrnehmungsart aber offen.

Zur Simulation des Verhaltens geht Tu von den Grundbedürfnissen eines Fisches aus und strukturiert diese in einer hierarchischen Ordnung. Das Verfahren bietet den Vorteil, dass es sehr einfach und verständlich ist. Allerdings scheint die Reihenfolge der *Wünsche* bei Tu fest im Programmcode verankert zu sein. Ein Framework sollte eine flexible Umstellung erlauben.

Die hierarchische Strukturierung entspricht dem zweiten Vorschlag von Reynolds zur Auswahl der Steuerverhalten. Dahingegen steht sein erster Vorschlag, stets alle Steuervektoren zu berechnen und zu addieren, im Widerspruch zu den von Tu auf Seite 23 beschriebenen ethologischen Erkenntnissen zur natürlichen Aktionsauswahl von Tieren. Ein weiterer Nachteil ist, dass sich in einem ungünstigen Fall die Steuervektoren gegenseitig aufheben könnten. Außerdem werden stets alle Vektoren berechnet, was zu einem Performanceverlust führen kann. Die erwähnte Hybrid-Technik mit einer zugeordneten Wahrscheinlichkeit für die Ausrechnung eines Steuerungsvektors besitzt den Nachteil, dass es mit

steigender Zahl der Steuerungsvektoren schwieriger werden wird, ein sinnvolles Zusammenspiel der Wahrscheinlichkeiten einzustellen. Zudem erscheint der Umgang mit Wahrscheinlichkeiten im Zusammenhang mit Verhaltensweisen wenig intuitiv.

Die von Funge geschilderte kognitive Modellierungssprache CML würde die Spezifikation des Verhaltens eines Charakters erheblich vereinfachen. Doch auch wenn sie für die Programmiersprache C++ verfügbar wäre, müsste beachtet werden, dass mit steigender Komplexität des Charakters Performanceeinbußen hinzunehmen wären. Da die Erstellung einer Sprache mit der Funktionalität von CML eine eigene Diplomarbeit füllen würde, soll von ihr nur der gedankliche Ansatz übernommen werden, das Verhalten eines Charakters auf einer höheren Abstraktionsebene beschreiben zu wollen, sowie eine konkrete Hilfestellung beim Planen und Verwalten des Verhaltens eines Charakters zu geben.

Die Erzeugung verschiedener Fischarten erreicht Tu, indem sie diese durch ihre Bedürfnisse und der Art, wie sie aufeinander reagieren, unterscheidet. Die Existenz der verschiedenen Spezies und deren Zusammenspiel beeinflussen entscheidend den hohen Grad an Realismus in ihren Simulationen. Dabei sind die Beziehungen der Fische untereinander relativ einfach gehalten und mit nur drei verschiedenen Fischtypen leicht überschaubar. Allerdings scheinen auch hier die Besonderheiten einer Spezies fester Bestandteil des Programmcodes zu sein, so dass das Hinzufügen neuer Spezies von mal zu mal aufwendiger wird. Für ein Framework ist dies nicht akzeptabel.

2.2 Techniken aus der KI

In diesem Abschnitt werden zunächst einige gängige Verfahren vorgestellt, die sich prinzipiell zur Erstellung von höherem Verhalten eignen. Da im Rahmen dieser Arbeit nur die Konzepte der wichtigsten Techniken umrissen werden können, sei für eine umfassendere Übersicht auf Werke von Stuart Russel und Peter Norvig „Artificial Intelligence: A Modern Approach“ [14] und von John David Funge „AI for games and Animation“ [3] verwiesen. Zum schnellen Nachschlagen eignet sich auch das Werk von Alan Bundy „Artificial Intelligence Techniques“ [1]. Im Anschluss an die Vorstellung werden die Techniken im Hinblick auf ihren Nutzen für die gegebene Problemstellung diskutiert.

2.2.1 Reaktive Verhaltensregeln

Der Begriff des reaktiven Verhaltens wird verwendet, wenn das Verhalten eines Charakters allein von seiner Wahrnehmung der aktuellen Situation abhängig ist (siehe [3]). Er besitzt also keine Form von Erinnerungsvermögen. Die von Reynolds in 2.1.1 beschriebenen Verhaltensweisen können als reaktive Verhaltensregeln bezeichnet werden.

Ein reaktiver Charakter besitzt eine Menge an Aktionen, deren Ausführung an bestimmte Bedingungen geknüpft ist. Ist die Bedingung für eine Aktion erfüllt, wird sie ausgeführt.

Der Vorteil von reaktiven Verhaltensweisen ist ihre Einfachheit, da sich prinzipiell bereits durch die Verwendung weniger Regeln ein komplexes Verhalten erreichen lässt (z.B. das von Reynolds beschriebene Schwarmverhalten). Die Schwierigkeit besteht darin, die richtigen Regeln zu definieren.

Der Nachteil eines rein reaktiven Charakters ist, dass sämtliche Regeln vor der Laufzeit festgelegt sein müssen. Eine Reaktion des Charakters auf unvorhergesehene Ereignisse ist deshalb nicht möglich. Je größer die Regelmenge eines reaktiven Charakters wird, desto schwieriger ist es, sein Verhalten für eine genaue Situation festzulegen, da mehrere Verhaltensweisen aktiv werden können. Zusätzlich ergibt sich das Problem, dass die Vorbedingungen der Aktionen korrekt parametrisiert sein müssen, damit ein Charakter das gewünschte Verhalten zeigt.

2.2.2 Endliche Automaten

Ein Endlicher Automat (EA) besteht aus einer Menge an Zuständen, einer Menge an Ein- und Ausgabewerten, sowie einer Menge an Übergangsfunktionen um von einem Zustand in den anderen zu gelangen. In einer Verhaltenssimulation kann er dazu verwendet werden, um die Bedingungen festzulegen, in denen ein Charakter von einem Zustand in den Anderen wechselt.

Ein EA befindet sich zu einem Zeitpunkt in genau einem Zustand. Sind die Übergangsfunktionen stets eindeutig, das heißt, führen sie genau zu einem Zustand, wird von einem deterministischen Verhalten gesprochen. Die Übergangsfunktionen können mit einer Aktion oder einem auftretenden Ereignis gleichgesetzt werden.

Ein EA kann als Zustand einen anderen EA besitzen. Die Automaten befinden sich dann in einer Hierarchie. Deshalb spricht man in einem solchen Fall auch von einem hierarchischen Endlichen Automaten (HEA).

Der Umgang mit HEA ist ähnlich einfach, wie der mit reaktiven Verhaltensweisen (siehe [3]). Sie besitzen aber den Vorteil, dass sie durch die Einteilung in verschiedene Zustände, eine bessere Kontrolle über das Verhalten eines Charakters ermöglichen. Zudem lassen sich auch innere Charakterzustände wie Hunger oder Müdigkeit darstellen, die nicht unmittelbar auf äußere Einflüsse zurückzuführen sind. Deshalb werden hierarchische endliche Automaten zur Steuerung des autonomen Verhaltens von Charakteren in den meisten Computerspielen bevorzugt (siehe [22]). Dies nicht zuletzt, weil sie leicht zu implementieren und zu verstehen sind, sondern auch weil sie die Verwendung von anderen Technologien nicht einschränken, da diese als ein Zustand des Automaten verwendet werden können.

Der Nachteil eines jeden Automaten ist es, dass er nur Zustände erzeugen kann, die vorher definiert wurden. Somit lassen sich die Bewegungen eines Charakters auf eine gewisse Art vorausahnen. Nach Funge besteht ein weiteres Problem bei der Verwendung von HEA darin, dass diese nur schwer an neue Bedingungen angepasst werden können, da teilweise Zustände und Übergänge neu definiert werden müssen.

2.2.3 Neuronale Netze

Neuronale Netze ermöglichen einem System das selbstständige Erlernen von Fertigkeiten. Dadurch lassen sich zum Beispiel intelligente Texterkennungsprogramme erstellen, die die Handschrift von Menschen zu erkennen lernen. Eine weite Verbreitung finden neuronale Netze in neueren Computerspielen. Hier werden sie zur Verbesserung der Intelligenz computergesteuerter Gegenspieler eingesetzt, da sie es ermöglichen, dass sich der Computergegner im Spielverlauf auf die Taktik des menschlichen Spielers einstellt. In einer Unterwasserwelt könnte ein virtueller Fisch lernen, einem Hindernis auszuweichen. Denkbar wäre es auch, dass ein Fisch selbstständig herausfindet, in welche Richtung er schwimmen muss, um einem Feind zu entkommen. Ebenso könnte er lernen, in welchen Situationen es sinnvoller ist, vor einem Feind zu fliehen oder sich vor ihm zu verstecken.

Das Prinzip der neuronalen Netze ist der Funktionsweise der Neuronen im menschlichen Gehirn nachempfunden. In der Informatik existieren mittlerweile mehrere verschiedene Arten neuronaler Netze. Um Lernprozesse zu simulieren, wird üblicherweise die „back propagation“ Methode verwendet. Vereinfacht ausgedrückt besitzt bei ihr jedes Neuron genau eine Verbindungsleitung zu einem anderen Neuron. Über diese Leitung kann es ein Signal (Null oder Eins) an das andere Neuron senden. An jedem Neuron können beliebig viele dieser Leitungen enden. Neben Signalleitungen von Neuron zu Neuron können Signale in Form von Eingabeparameter aus der Applikation an ein Neuron gesendet werden. Ein Neuron versieht jede der bei ihm ankommenden Leitungen mit einem Gewichtungsfaktor. Auf Grund der Summe der gewichteten einkommenden Signale entscheidet das Neuron, ob es ein Null- oder Einssignal weitersendet. In dem einfachen Beispielnetz aus Abbildung 17 sind sieben Eingabeparameter x_0 bis x_6 an sechs Neuronen (mit dem Summenzeichen gekennzeichnet) angeschlossen. Das Neuron ganz rechts in der Abbildung liefert einen Ausgabewert zurück. Prinzipiell sind beliebig viele Ausgabewerte möglich.

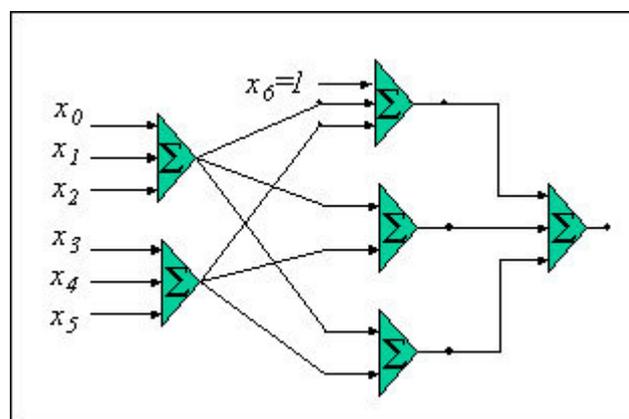


Abbildung 17: Neuronales Netz

Es ist evident, dass eine Veränderung der Gewichtungen zu einer Änderung der Ausgabewerte führen kann. Wenn die Ausgabewerte vom Benutzer durch eine Funktion vorgegeben werden, kann das neuronale Netz durch „back propagation“ (~Rückentwicklung oder Rückverfolgung) die Gewichtungen der Signalleitungen anpassen, damit die Ausgabewerte der Funktion möglichst nahe kommen. Die Anpassung geschieht durch viele Iterationsprozesse, in denen nach den richtigen Gewichtungen gesucht wird. Da sich die Ergebnisse mit fortschreitendem Iterationsprozess immer mehr den vorgegebenen Werten nähern, wird die Suche auch als Lernprozess bezeichnet.

Bei der Erstellung eines solchen Systems gilt es, vor allem zwei Probleme zu lösen. Das erste und größte Problem ist die Erstellung des neuronalen Netzes selber. Da es keine formale Definition gibt, wie die Architektur des Netzes für ein gegebenes Problem gestaltet sein muss, ist ein Programmierer darauf angewiesen, dies durch Versuche herauszufinden. Das Zweite ist die Definition der Ergebnisfunktion, für die die passenden Gewichtungen gefunden werden sollen. Auch hierfür gibt es keine allgemeine Lösung, da die Zielfunktion vollständig vom Verwendungszweck der Applikation abhängig ist.

2.2.4 Genetische Algorithmen

Genetische Algorithmen funktionieren nach einem darwinistischen Prinzip: Nur Individuen, die am Besten ans "Überleben" angepasst sind, können ihre "Gene" an die nächste Generation weitergeben.

Was in einer Simulation unter "Überleben" zu verstehen ist, hängt von der Art der Applikation ab. In einer Simulation mit Fischen könnte jeder Fisch, der nach einer gewissen Zeitspanne noch nicht gefressen wurde oder verhungert ist, als Überlebender betrachtet werden. In einem Autorennen könnte man jedes Auto, welches in einer bestimmten Zeit die Ziellinie überquert, als Überlebenden bezeichnen.

Ein "Gen" bezeichnet eine Eigenschaft des Individuums. Dies könnten bei Fischen z.B. die Größe, Masse, Kraft, Schnelligkeit, Farbe des Kopfes, Farbe der Flossen oder die Vorliebe für einen bestimmten Beutefisch sein. Ein Rennwagen könnte die PS-Zahl, seine Reifen oder die Fahrweise als spezifische Eigenschaften besitzen.

Durch die Kombination der Gene von zwei (oder mehr) überlebenden Individuen entsteht ein neues Individuum. Auf diese Art werden nur die "Siegeregene" weitervererbt und unnütze Gene sterben mit der Zeit aus.

Beim Vererben ist es möglich, dass ein Gen mutiert und ein völlig neues Gen entsteht. Dadurch wird sichergestellt, dass mit voranschreitendem Ausleseprozess die Menge der Kombinationsmöglichkeiten nicht zu klein wird.

2.2.5 Diskussion der Techniken

In den beschriebenen Verfahren werden zwei verschiedene Ansätze zur Erstellung autonomer Charaktere verfolgt. Die reaktiven Verhaltensregeln und die hierarchischen, endlichen Automaten verfolgen das Ziel, mittels vom Benutzer vordefinierten Regeln zu höherem Verhalten zu gelangen. Die genetischen Algorithmen und neuronalen Netze versuchen dies, allein durch eine Zielvorgabe vom Benutzer möglichst eigenständig zu erreichen.

Allgemein scheint es von Vorteil zu sein, wenn einem bei der Generierung des Verhaltens einige Arbeiten vom Computer abgenommen werden könnten. Deswegen sollen zunächst die beiden zuletzt vorgestellten Verfahren diskutiert werden.

Um genetische Algorithmen in einer Unterwasserwelt erfolgreich benutzen zu können, wäre man gezwungen, zuerst einige grundlegende, in der Natur vorherrschende Zusammenhänge zwischen den Parametern eines Fisches in das System zu implementieren. Beispielsweise müsste sich die Masse eines Fisches automatisch erhöhen, wenn seine Kraft gesteigert wird, da sich sonst Fische mit einer Masse nahe Null und einer extrem hohen Kraft durchsetzen würden, weil diese am schnellsten schwimmen könnten. Um wie viel die Masse genau gesteigert werden müsste, ließe sich vermutlich nur durch Testen herausfinden, da die in Fachbüchern beschriebenen biologischen Zusammenhänge sich nicht vollständig auf die abstrahierte Simulation übertragen lassen. Neben der möglichen Entstehung solcher „Superfische“, müssten auch visuelle Aspekte berücksichtigt werden. Denn ob die Bewegungen und das Verhalten eines Fisches als natürlich erscheinen, muss letztendlich nicht der Computer sondern der Betrachter entscheiden.

Vermutlich wäre es von Vorteil, wenn in der virtuellen Unterwasserwelt eine Art korrektes, ökologisches Gleichgewicht herrschen würde. Doch der Aufwand hierfür entspricht derzeit nicht dem Nutzen, den Vertigo Systems daraus zieht. Um das System an einen anderen Anwendungsfall anzupassen, müssten erneut die Zusammenhänge geprüft und definiert werden. Voraussetzung hierfür wäre, dass sich die Individuen in der Applikation überhaupt in einem Konkurrenzkampf zueinander befinden, damit ein Ausleseprozess stattfinden kann. Bei einer Simulation mit Autos oder Flugzeugen ist das nur schwer vorstellbar.

Von den vorgestellten Verfahren lassen sich mit Hilfe neuronaler Netze die höchsten Formen intelligenten Verhaltens erzeugen. Sie ermöglichen es einem Charakter, sich durch Lernprozesse immer wieder neu an seine Umwelt anzupassen. Der Nachteil ist, dass es sich im Vorfeld nicht genau abschätzen lässt, wie schwierig es wird, ein passendes neuronales Netz zu finden. Ebenso lässt sich der Aufwand nicht vorhersagen, der notwendig ist, um die passende Ergebnisfunktion zu finden, auf Grund derer das Verhalten des Charakters adaptiv angepasst werden soll. Beide Nachteile wären vertretbar, wenn sie einmalig auftreten würden. Da die im **living** System dargestellten Inhalte und somit auch die Charaktere häufig wechseln, müssten jedes mal das neuronale Netz und die Ergebnisfunktion angepasst werden. Für Vertigo Systems ist es aber enorm wichtig, dass sich der Arbeits- und Zeitaufwand einschätzen lassen, zumal sich das Verhalten direkt am Einsatzort des **living** Systems (siehe Anforderungsliste in Kapitel 1.2.3) an die gegebenen Umstände anpassen lassen soll.

Deshalb kommen für eine mögliche Verwendung im eigenen Framework vorerst nur noch die reaktiven Verhaltensweisen und die hierarchischen, endlichen Automaten in Betracht. Der größte Nachteil beider Verfahren ist es, dass sich nur vordefinierte Verhaltensweisen erzeugen lassen, die nicht auf unvorhergesehene Ereignisse reagieren können. Es stellt sich die Frage, wie schwerwiegend dieser Nachteil in Bezug auf das Anwendungsszenario ist. Da die Möglichkeiten eines Benutzers des **living** Systems, Einfluss auf die virtuelle Welt auszuüben, derzeit auf seinen Schatten begrenzt ist, werden sich die hierfür erforderlichen Regeln in einem überschaubaren Rahmen bewegen. Die Regeln zur Verhaltenssteuerung der Charaktere untereinander können zwar vielschichtiger sein, werden aber schätzungsweise die Komplexität des von Tu beschriebenen Verhaltenssystems annehmen. Dadurch ist es durchaus gerechtfertigt, wenn die Charaktere auf vordefinierte Aktionen zurückgreifen.

Damit sich das Verhalten leicht erstellen, steuern und verstehen lässt, muss ein intelligenter Strukturierungsmechanismus verwendet werden, der die Verwaltung der Aktionen erleichtert. Dies schließt eine alleinige Verwendung reaktiver Verhaltensweisen aus. Stattdessen bietet es sich an, diese in einem hierarchischen endlichen Automaten zu ordnen. Damit ein Benutzer mit den Möglichkeiten des Automaten nicht überfordert wird, sollte eine sinnvolle Beschränkung der Zustandsübergänge vorgenommen werden. Die letztendliche

logische Zusammenstellung von Verhaltensregeln muss aber in der Obhut des Benutzers bleiben.

2.3 Bestehende OpenSource-Projekte

Dieses Kapitel soll einen Überblick über bestehende OpenSource-Systeme geben, die im Internet zu finden sind und die sich mit der Erstellung autonomer Agenten oder künstlicher Intelligenzen allgemein befassen. Das Ziel ist es, die Systeme im Hinblick auf die Nutzung als Grundlage für ein eigenes Verhaltenssystem beurteilen zu können. Deswegen wurden zur Betrachtung auch nur Systeme herangezogen, die mit der Programmiersprache C++ verfasst wurden.

Da in manchen Fällen schnell ersichtlich ist, dass eine Nutzung als Arbeitsgrundlage aus verschiedenen Gründen nicht möglich ist, werden diese nur sehr kurz vorgestellt.

2.3.1 OpenAI

OpenAI hat sich das ehrgeizige Ziel gesetzt, eine Art OpenGL für den KI Bereich zu werden. Es ist seit Mai 2001 online. Neben Tools für KI Anwendungen sollen Konfigurations- und Kommunikationsstandards geschaffen werden. Bisher wurde an Tools zur Erstellung von Neuronalen Netzen, Genetischen Algorithmen, Endlichen Automaten und Mobilen Agentensystemen, sowie einer graphischen Benutzeroberfläche gearbeitet. Mit Ausnahme der Neuronalen Netze befinden sich alle Programmteile in einem relativ frühen Entwicklungsstadium. Die Implementierung erfolgte in Java. Entsprechende Versionen in C++ sollten folgen, doch leider ist das Projekt seit April 2003 für unbestimmte Zeit eingefroren.

2.3.2 FEAR

FEAR ist die Abkürzung für Flexible Embodied Animat architecture. Es handelt sich hierbei um ein OpenSource Projekt, das Entwickler bei der Erstellung intelligenter Kreaturen für den Spielbereich unterstützen soll. Die Funktionen decken ein breites Spektrum von Techniken aus dem KI-Bereich ab und sind nach Angaben der Projektseite [IQ 1] fast vollständig entwickelt. Unter anderem wird die Erzeugung von neuronalen Netzen oder genetischen Algorithmen unterstützt.

Neben reaktiven Steuerungsverhalten, wie Reynolds sie benutzt, werden auch Verfahren zur Planung von Pfaden bereitgestellt.

FEAR ist zum Großteil in C++ geschrieben und auf Windows- und Linuxplattformen lauffähig. Allerdings benötigt das User Interface ein Windows Betriebssystem. Als Testumgebungen für die Charaktere dienen Level aus dem Computerspiel Quake II, welches zum Testen installiert sein muss.

Leider sind die auf der FEAR-Homepage (siehe [IQ 1]) angegebenen Verweise zur Dokumentation und zum Forum seit längerem nicht verfügbar (Stand Juli 2005), so dass es schwer ist, einen genaueren Überblick zu geben. Zur Informationsbeschaffung, musste deswegen teilweise auf [IQ 2] ausgewichen werden.

2.3.3 OpenSteer

Im Jahr 2002 entwickelte Craig Reynolds in der Forschungs- und Entwicklungsabteilung bei Sony Computer Entertainment America eine erste Version von OpenSteer. Seit Mai 2003 ist dessen Sourcecode als OpenSource-Projekt öffentlich.

Der Grundgedanke von OpenSteer besteht darin, eine plattformunabhängige C++ Bibliothek zur Steuerung von autonomen Charakteren zur Verfügung zu stellen, um diese in Computerspielen oder Multi-Agentensystemen einsetzen zu können. Neben der Steuerungsbibliothek wird eine von der letztendlich verwendeten Graphikengine unabhängige Demo-Umgebung bereitgestellt, die das schnelle Testen des Verhaltens von erstellten Charakteren über sogenannte Plug-Ins¹ ermöglicht.

Die Funktionen von OpenSteer entsprechen zum Großteil den von Reynolds in [13] und in Kapitel 2.1.1 (ab S. 11) beschriebenen Steuerungsmechanismen. Nicht oder noch nicht implementiert sind das Arrival und das Leader Following Verhalten.

Die hier betrachtete Version von OpenSteer wurde im Februar 2005 der Codeverwaltung des Projekts (dem CVS) entnommen. Sie entspricht überwiegend der Version 0.8.2, die seit dem Oktober 2004 als offizielle, lauffähige Version zum Download angeboten wird (Stand: Juli 2005). Allerdings bietet die aktuellere

¹ siehe hierzu auch die Beschreibung der Plug-Ins auf Seite 39

Version erweiterte Möglichkeiten zur Hindernisvermeidung, weswegen sie zur Betrachtung herangezogen wird.

Funktionsweise

Der Zusammenhang der wichtigsten Klassen der OpenSteer Architektur, die zum Verständnis der Funktionsweise notwendig sind, wird in Abbildung 18 dargestellt. Alle Klassen, die individuell vom Benutzer erstellt werden, und nicht zum OpenSteer System gehören, besitzen ein „Own“ (engl.: *eigene(s)*) vor ihrem Namen. Die einzige Ausnahme bildet die Klasse *SimpleVehicle*. Wehalb ihr kein Suffix vorangestellt ist, wird im weiteren Textverlauf deutlich.

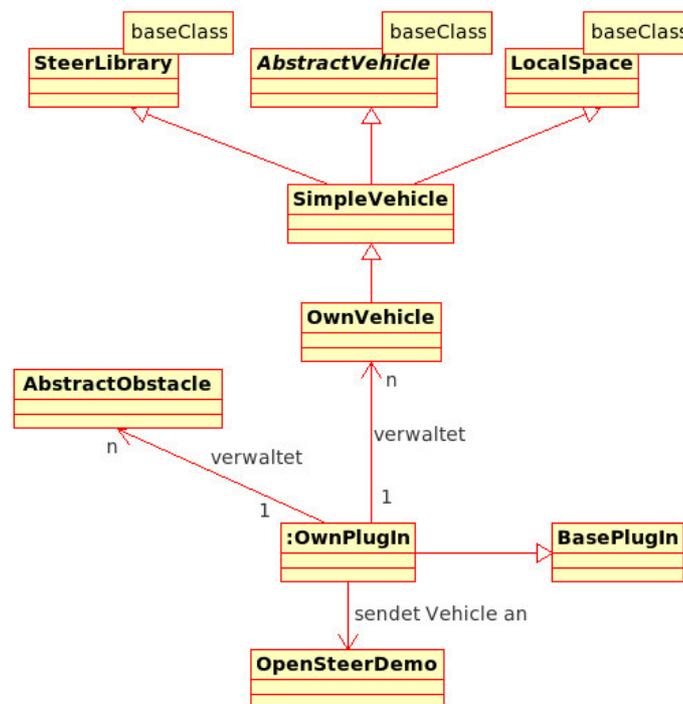


Abbildung 18: Beispielarchitektur zur Verwendung eines OpenSteer Vehikels

Alle Steuerungsmechanismen mit Ausnahme derer, die zur Hindernisvermeidung dienen, sind in der *SteerLibrary* untergebracht. Die Steuerungsmechanismen liefern einen Null-Vektor zurück, wenn sie nicht ausführbar sind, ansonsten den Steuerungsvektor, der zur Ausführung des Verhaltens notwendig ist. Um die Steuerungsmechanismen nutzen zu können, muss ein eigenes Vehikel wenigstens von den Klassen *AbstractVehicle* und *LocalSpace* abgeleitet sein. Die *AbstractVehicle*-Klasse fügt das Gerüst für ein einfachstes Vehikel-Modell hinzu, wie es von Reynolds in [13] (siehe auch S. 10) beschrieben wird. Außerdem setzt sie voraus, dass das Bounding Volume jedes Vehikel durch einen Kreis oder

Kugel angenähert werden kann. Die Klasse selber verfügt über keinerlei Funktionalität, sondern stellt nur ein Methodeninterface zur Verfügung. Diese wird erst in der *SimpleVehicle*-Klasse implementiert. Die *LocalSpace*-Klasse verwaltet für das *SimpleVehicle* ein lokales Koordinatensystem. Dadurch können die Berechnungen in der *SteerLibrary* mit abstrakten Achsenbezeichnungen wie „Oben“ oder „Vorne“ vollzogen werden.

Wie bereits erwähnt wurde, gehört das *SimpleVehicle* streng genommen nicht zur Kernarchitektur von OpenSteer. Allerdings ist es ein fester Bestandteil der Beispiel-Plug-Ins, die in jeder Version von OpenSteer enthalten sind. In ihm ist bereits eine sinnvolle Positionsänderung des Vehikels auf Grund seiner Geschwindigkeit, Masse, Kraft und der vergangenen Zeit implementiert. Zudem besitzt es weitere nützliche Funktionen, wie die Stärke der Kurvenneigung, die in einer Vielzahl verschiedenster Vehikel Verwendung finden können. Deshalb ist es wahrscheinlich, dass ein eigenes Vehikel vom *SimpleVehicle* abgeleitet wird, um diese Grundfunktionalität nicht selber implementieren zu müssen. Notwendig ist dies allerdings nicht. Ein eigenes Vehikel könnte auch direkt vom *AbstractVehicle* abgeleitet werden.

Die *OwnVehicle*-Klasse steht symbolisch für eine eigene Vehikelklasse. Hier wird die Logik und somit das gesamte Verhalten des Vehikels bestimmt. Hierfür werden die Bedingungen festgelegt, unter denen das Vehikel einen der Steuerungsmechanismen aus der *SteerLibrary* anwendet.

Die *AbstractObstacle*-Klasse beinhaltet die Basismethoden für alle Typen von Hindernissen. Diese erlauben es, dass ein Vehikel ein Hindernis „fragen“ kann, ob in naher Zukunft eine Kollision mit ihm stattfinden wird. Das Hindernis liefert dann einen Steuerungsvektor zurück, der zum Ausweichen des Vehikels notwendig ist. In der genannten Version aus dem CVS sind vier verschiedenen Arten von Hindernissen implementiert. Diese sind Kugel, Quader, Ebene und Rechteck. In der Version 0.8.2 existiert nur das Kugelhindernis. Für jedes Hindernis kann angegeben werden, in welchen Fällen es von einem Vehikel als Hindernis erkannt werden soll. Dies kann geschehen, wenn sich das Vehikel von außen und/oder innen nähert.

Wenn die Demo-Umgebung von OpenSteer benutzt wird, muss ein eigenes Plug-In erstellt werden, welches vom *BasePlugIn* abgeleitet ist ¹. Zur Visualisierung der Hindernisse und Vehikel, muss es diese an die *OpenSteerDemo* weitergeben. Diese verwaltet Kameraparameter und kann Bildschirmtext ausgeben. Ebenso ist sie für das Voranschreiten der Simulationszeit verantwortlich.

Nachbarschaftsfunktionen

OpenSteer stellt zwei Datenstrukturen zur Nachbarschaftssuche zur Verfügung. Die einfachere ist eine BrutForce-Methode dessen Aufwand quadratisch mit der Anzahl der zu vergleichenden Objekte steigt. Die zweite Methode unterteilt die Szene in gleichgroße Voxelgitter, wodurch der Aufwand zum Suchen verringert werden kann.

2.3.4 MetaAgent

Der MetaAgent wurde im Mai 2003 von Jonathan de Halleux ins Leben gerufen. Er ist auf die Windowsplattform fixiert und hat es sich zum Ziel gesetzt, die von Reynolds in [13] beschriebenen Steuerungsverhalten (siehe auch Kapitel 2.1.1) zu implementieren. Der Unterschied zu OpenSteer (siehe S. 36) soll in einer verbesserten Ausnutzung der Funktionalitäten von C++ liegen.

Der MetaAgent liegt in seiner Entwicklung derzeit weit hinter OpenSteer zurück. Als einziges Verhalten ist das Wander Verhalten implementiert.

2.3.5 Diskussion der OpenSource-Projekte

OpenSteer, der MetaAgent und FEAR stellen Funktionen zur Navigation autonomer Charaktere mittels reaktiver Verhaltensweisen zur Verfügung. FEAR bietet darüber hinaus zusammen mit OpenAI Techniken für die Planung von Handlungen und das Erlernen von Fertigkeiten an. Die intelligenteren Charaktere ließen sich also mit Hilfe von FEAR und OpenAI erzeugen.

Leider befindet sich OpenAI unter C++ in einem noch nicht verwendbaren Entwicklungsstadium und scheidet deshalb für weitere Untersuchungen aus.

¹ Die Bezeichnung Plug-In ist an dieser Stelle etwas irreführend, da Plug-Ins normalerweise zur Laufzeit hinzugefügt werden können. Dies ist aber auf Grund der Architektur des *BasePlugIn* nicht möglich. Diese sieht ein Hinzufügen während des Linkvorgang vor.

Ebenso ist der MetaAgent auf Grund noch fehlender Funktionalitäten und einer fehlenden Kompatibilität zu Linux für eine nähere Betrachtung uninteressant.

FEAR und OpenSteer hingegen besitzen eine genügende Entwicklungsreife, um als Grundlage für ein eigenes Framework dienen zu können. In einem ersten Vergleich bietet FEAR gegenüber OpenSteer deutlich bessere Nutzungsmöglichkeiten an, da es sowohl Unterstützung bei der Navigation der Charaktere anbietet, als auch verschiedene Strategien zur Entscheidungsfindung beim Charakter bereitstellt.

Gegen den Einsatz von FEAR spricht allerdings die fehlende Dokumentation und dessen hohe Affinität zu Ego-Shooter-Anwendungen insbesondere zu Quake II. Die von FEAR unterstützten Strategien zum Waffenwechseln, Ducken, Springen oder Schießen, sind auf allgemeine autonome Charaktere, wie sie im **living** System Verwendung finden sollen, nicht anwendbar. Für einen universelleren Einsatz müsste der Sourcecode umgeschrieben werden. Dadurch wäre zu befürchten, dass der neue Code mit einer späteren Version von FEAR inkompatibel sein könnte und weitere Anpassungen erforderlich wären.

Um den Aufwand zur Anpassung zu rechtfertigen, müsste Vertigo Systems einen essentiellen Nutzen aus der Verwendung hoch intelligenter Charaktere mit Lern- und Planungseigenschaften haben. Da das Interaktionskonzept des **living** Systems es derzeit vornehmlich vorsieht, dass die Veränderungen in der virtuellen Welt durch äußere Einflussnahme des Benutzers stattfinden, wäre eine Steuerung der Charaktere mittels eines einfacheren Verfahrens durchaus hinreichend. Der Aufwand zur Erstellung von wirklich intelligenten Verhalten der Charaktere ist erst dann gerechtfertigt, wenn die Charaktere in einer Art Wettstreit gegen den Benutzer antreten sollten.

Unter diesem Gesichtspunkt bietet OpenSteer eine hinreichende Funktionalität an, die mit akzeptablem Aufwand zur Erstellung höherer Verhaltenweisen erweitert werden kann. Durch den modularen Aufbau bleibt die Möglichkeit erhalten, eventuell zu einem späteren Zeitpunkt andere Techniken wie neuronale Netze oder genetische Algorithmen zu integrieren.

Kapitel 3. Das Konzept

In diesem Kapitel wird das Konzept vorgestellt, das der im darauffolgenden Kapitel beschriebenen Implementierung zu Grunde liegt. Zuvor werden die Entscheidungen begründet, die zur Konzipierung führten.

3.1 Entscheidungsfindung

Durch Nutzung von OpenSteer kann wertvolle Entwicklungszeit gespart werden, da bereits primitive Verhaltensweisen und eine Testumgebung vorhanden sind. Außerdem bestehen gute Chancen, dass das Projekt von der OpenSource Gemeinde auch zukünftig weiterentwickelt werden wird und die Software somit ohne Mehrkosten auf einem technisch aktuellen Stand bleiben kann. Deshalb soll OpenSteer als Grundlage verwendet werden.

Das Verhalten

Das Verhaltenssystem der in 2.1.2 beschriebenen Arbeit von Tu liefert sehr gute Ergebnisse. Es geht von den Grundbedürfnissen eines Fisches aus und strukturiert sie in einer hierarchischen Ordnung. Zudem gibt es verschiedene Fischtypen, die sich durch ihre Bedürfnisse und der Art, wie sie aufeinander reagieren, unterscheiden. Das eigene Framework soll diese Möglichkeiten ebenfalls anbieten.

Da Tu keine Lösung zur flexiblen Umgestaltung der Verhaltensweisen vorstellt, wird vermutet, dass sowohl die Ordnung der Bedürfnisse, als auch die Eigenarten der Fischtypen, fest im Programmcode verankert sind. Dies ist für ein flexibel an verschiedenste Bedingungen anzupassendes Verhaltensframework inakzeptabel. Eine weiterer Kritikpunkt bei Tu ist, dass ein mehrfacher Austausch von Daten zwischen dem Wahrnehmungs- und dem Verhaltenssystem stattfindet. Dies kann durch die Nutzung des Wahrnehmungskonzepts von OpenSteer vermieden werden.

Tu hat gezeigt, dass die wechselseitigen Beziehungen unter den Bewohnern eines simulierten Ökosystems eine wichtige Rolle bei der glaubhaften Darstellung von Lebewesen spielen. Deshalb sollen sich im eigenen Framework solche

Beziehungen möglichst leicht definieren lassen. Die Beziehungen sollen flexibel angegeben werden und sich zur Laufzeit ändern können.

Um das Prinzip der Verhaltensbausteine aus OpenSteer beizubehalten, deren Handhabung aber intuitiver zu gestalten, wird dem Konzept für das eigene Verhaltenssystem eine Metapher zu Grunde gelegt, die der bei Tu verwendeten ähnelt. Anders als bei Tu soll aber nicht von *Wünschen*, *Intention* und *Verhaltensroutinen* gesprochen werden. Statt dessen soll ein Charakter *Bedürfnisse* besitzen, die er durch das Ausführen von *Aktionen* befriedigen kann. Die Metapher entspricht dem gedanklichen Ansatz, dass jede Handlung eines Individuums durch ein inneres Bedürfnis ausgelöst wird. Die Handlung kann also als Mittel zur Erfüllung eines konkretes Ziel betrachtet werden. Die Aktion legt damit fest, wie ein Bedürfnis oder zumindest ein Teil des Bedürfnisses befriedigt werden kann. Diese Denkweise soll einem Anwender die Konzeption eines Verhaltensschemas erleichtern.

Um den flexiblen Anforderungen gerecht zu werden und kürzere Entwicklungszeiten zu erreichen, sollen die Aktionen und Bedürfnisse während der Laufzeit des Programms hinzugefügt, entfernt und parametrisiert werden können. Zusätzlich sollen die Aktionen gegenüber den einfachen Verhaltensroutinen von OpenSteer verbessert werden. Es sei daran erinnert, dass in OpenSteer lediglich ein Null-Vektor zurückgegeben wird, wenn eine Verhaltensroutine keinen Steuerungsvektor ermitteln konnte (vergleiche das Kapitel 2.3.3 „OpenSteer“ auf S. 37). Da dies bei Addition mehrerer Vektoren in ungünstigen Fällen zu Fehlinterpretationen führen kann, wird eine eindeutige Lösung bevorzugt. Deshalb sollen die Aktionen klar zwischen der Ausführung und der Prüfung, ob die Möglichkeit zur Ausführung der Aktion überhaupt besteht, unterscheiden können.

3.2 Konzeptdesign

Aus den getroffenen Entscheidungen ergibt sich folgender Aufbau für das Gesamtsystem:

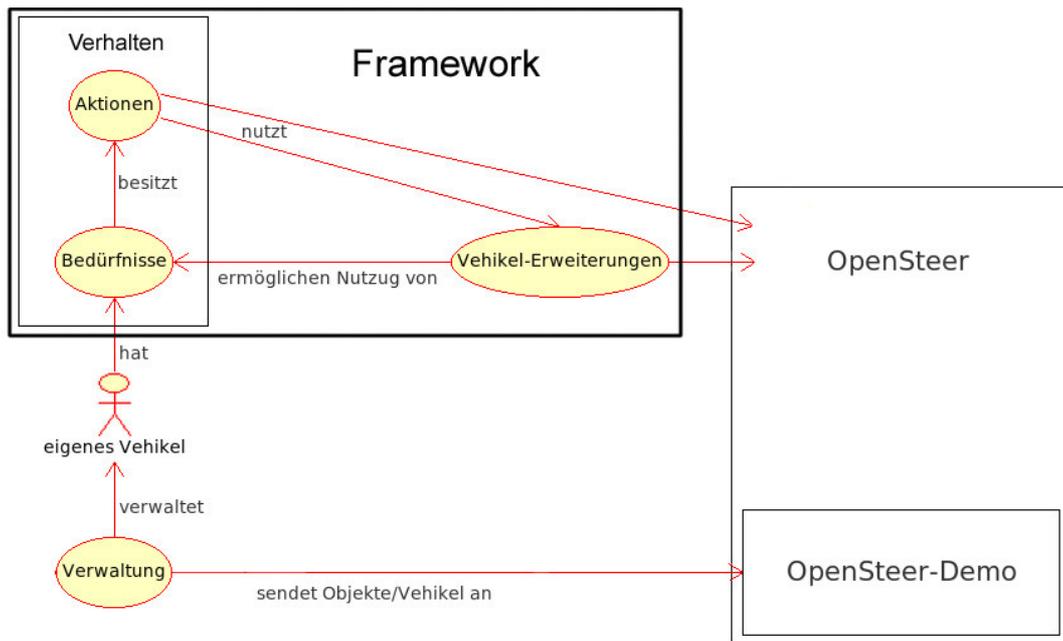


Abbildung 19: Anwendungsdiagramm

Ein Vehikel soll Bedürfnisse besitzen und diese verwalten können. Ebenso sollen Beziehungen zu anderen Vehikeln definiert werden können. Diese Funktionalität wird jedem Vehikel durch eine Verbindung zu einer Erweiterungsklasse aus dem Framework zur Verfügung gestellt. Auch fehlende Funktionalität von OpenSteer wird in diesem Erweiterungsteil untergebracht.

Die Bedürfnisse verwalten wiederum eine Menge von Aktionen, in denen die Beschreibung einer Handlung niedergelegt ist. Die Handlungsbeschreibung greift auf die von OpenSteer, sowie auf die vom Framework eingeführten Funktionen zurück.

Da mehrere Vehikel erstellt werden sollen, ist es notwendig, diese zu verwalten. Die Verwaltung findet zwar von OpenSteer getrennt statt, bleibt aber dennoch zur Schnittstelle der OpenSteer-Demo kompatibel.

3.3 Funktionsspezifikation

Nachfolgend werden die Funktionsweisen der Bedürfnisse, der Aktionen und der Beziehungen genau spezifiziert und jeweils mittels eines anschaulichen Beispiels erklärt. Die Spezifikationen dienen als Grundlage der in Kapitel 4 beschriebenen Implementierung.

Die Beziehungen

Ein Vehikel kann eine oder mehrere Beziehungen zu einem anderen Vehikel besitzen. Die Beziehungen werden Relationen genannt. Diese beschreiben sein Verhältnis, in dem es zu dem anderen Vehikel steht. Eine Relation lässt sich zur Laufzeit ändern. Ein Vehikel kann in keiner Beziehung zu sich selber stehen.

Beispiel:

Ein Mann besitzt zwei Haustiere, eine Katze und einen Hamster, die ihm gegenüber beide sehr zutraulich sind. Leider versucht die Katze ständig den Hamster zu fressen.

Vehikel = { Katze, Hamster, Mann }

mögliche Relationen = { Feind, Beute, Freund, Haustier }

Relationen der Katze = (Hamster, Beute) , (Mann, Freund)

Relationen des Hamsters = (Katze, Feind), (Mann, Freund)

Relationen des Mannes = (Katze, Haustier), (Hamster, Haustier)

Die Bedürfnisse

Ein Bedürfnis entspricht einem Verlangen oder einem Ziel, nach dem ein Vehikel seine Aktionen auswählt.

Ein Bedürfnis kann einem Vehikeln zugeordnet werden. Innerhalb eines Vehikels befinden sich die Bedürfnisse in einer hierarchischen Ordnung. Es gibt somit kein Bedürfnis, dass gleichwertig einem anderen gegenüber ist. Ein Bedürfnis kann entweder befriedigt (engl.: satisfied) oder unbefriedigt sein. Wann ein Bedürfnis befriedigt ist, hängt von der Art des Bedürfnisses ab (siehe hierzu auch Abschnitt 4.2.4 auf Seite 54). Einem Bedürfnis können eine beliebige Anzahl von Aktionen zugeordnet werden, mindestens jedoch eine (siehe Abbildung 20). In jedem Simulationsschritt wird versucht, dass unbefriedigte Bedürfnis mit der höchsten Priorität zu erfüllen. Dies geschieht, indem eine oder mehrere der Aktionen des Bedürfnisses ausgeführt werden.

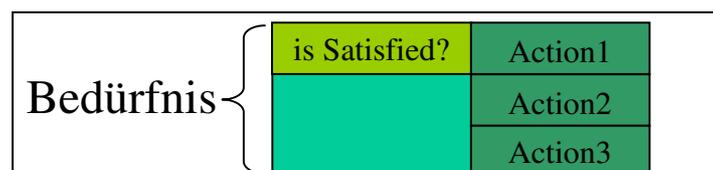


Abbildung 20: Bedürfnisbaustein

Beispiel:

Der Hamster aus dem obigen Beispiel, führt ein hartes Leben. Ständig muss er aufpassen, nicht von der Katze gefressen zu werden. Dabei muss er sich doch selber immer wieder auf Futtersuche begeben, um nicht zu verhungern. Nebenbei ist er sehr darauf bedacht, dass sein Fell sauber bleibt. Ansonsten zieht er sich in sein Hamsterhaus zum Schlafen zurück.

Daraus folgt:

Bedürfnisse = Überleben, Hunger, Putzen, Schlaf,

„Überleben“ befriedigt, wenn Katze außer Sicht

„Hunger“ befriedigt, wenn letzte Mahlzeit noch keine X Minuten her ist

„Putzen“ befriedigt, wenn Fell sauber ist

„Schlaf“ ist niemals befriedigt

Die Bedürfnisse müssen sich innerhalb des Vehikels in der aufgeführten Reihenfolge befinden.

Als erstes wird geprüft, ob das Überlebensbedürfnis befriedigt ist. Ist dies nicht der Fall, wird er die entsprechenden Aktionen des Bedürfnisses (oben nicht dargestellt) ausführen. Andernfalls werden der Reihe nach alle weiteren Bedürfnisse geprüft. Als letztes wird das Schlafbedürfnis geprüft, welches stets unbefriedigt ist.

Das Standardbedürfnis

Jedes Vehikel besitzt genau ein Standardbedürfnis. Dieses besitzt prinzipiell die selben Eigenschaften wie ein normales Bedürfnis, nur das es stets unbefriedigt ist und es nicht zwangsweise eine Aktion besitzen muss. In jedem Simulationsschritt werden die möglichen Aktionen des Standardbedürfnisses zusätzlich, also gleichzeitig, zu dem unbefriedigten Bedürfnis mit der höchsten Priorität ausgeführt. Das Standardbedürfnis hat den Zweck, die von Reynolds beschriebene Addition der Steuerungsvektoren verschiedener Verhaltensweisen (siehe S. 20) zu ermöglichen. Die Addition kann absichtlich nicht an beliebiger Stelle geschehen, damit das erstellte Verhaltensschema möglichst nah an natürlichen Vorbildern bleibt.

Beispiel:

Der Hamster ist in der Lage, während der Befriedigung eines seiner Bedürfnisse, gleichzeitig Hindernissen auszuweichen.

Daraus folgt:

Bedürfnisse = Überleben, Hunger, Putzen, Schlaf

Aktion im Standardbedürfnis = Hindernissen ausweichen

Die Aktionen

Eine Aktion beschreibt eine Handlung in Form eines Steuerungsvektors. Eine Aktion kann einem oder mehreren Bedürfnissen eines Vehikels hinzugefügt werden. Innerhalb eines Bedürfnisses befinden sich die Aktionen in einer eindeutigen hierarchischen Ordnung.

Ob die Möglichkeit zur Ausführung (engl: execution) der Handlung besteht, wird in einer Vorbedingung (engl: precondition) festgelegt (siehe Abbildung 21). Ist diese nicht erfüllt, kann die Handlung nicht ausgeführt werden.

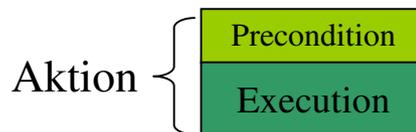


Abbildung 21: Aktionsbaustein

Der Steuerungsvektor jeder Aktion kann beliebig gewichtet werden. Ist die Gewichtung Null, gilt die Vorbedingung automatisch als nicht erfüllt.

Beispiel:

Der Hamster ist hungrig (Bedürfnis). Wenn er kein Futter sehen kann, wird er anfangen, in seinem Käfig umherzustreifen. Sobald er etwas fressbares entdeckt, eilt er dorthin, um es zu essen.

Daraus folgt:

Bedürfnis = Hunger

Aktionen = Iss Futter, Gehe zu Futter, Streife umher

Vorbedingung von „Iss Futter“ = Futter befindet sich in Reichweite

Vorbedingung von „Gehe zu Futter“ = kann Futter sehen

Vorbedingung zu „Streife umher“ = keine

Die Aktionen befinden sich innerhalb des Hungerbedürfnisses in der aufgeführten Reihenfolge. Das Bedürfnis ist so eingestellt, dass stets nur die erste mögliche Aktion ausgeführt wird.

Die Aktion „Streife umher“ wird ausgeführt werden, wenn der Hamster sich nicht in Reichweite zum Futter befindet und auch kein Futter sehen kann. Sobald er aber welches erblickt, wird er sich dem Futter solange nähern, bis die Vorbedingung

von „Iss Futter“ erfüllt ist, also sich das Futter in unmittelbarer Nähe zu ihm befindet. Erst dann wird er essen. Sollte nach dem Essen das Hungerbedürfnis noch nicht befriedigt ist, werden die Aktionen von neuem der Reihenfolge nach geprüft und ausgeführt.

Informationen zum genauen Zusammenspiel eines Vehikels mit seinen Bedürfnisse und Aktionen wird in Anhang A auf Seite 81 gegeben. Dort wird der Pseudocode für einen Simulationsschritt vorgestellt.

Kapitel 4. Realisierung des Gesamtsystems / Implementierung

Dieses Kapitel beschreibt das Framework mit seinen Funktionen und wie es implementiert wurde. Als erstes wird ein Überblick über das Gesamtsystem gegeben und das Zusammenspiel aller Klassen erläutert. Daran schließt eine genaue Beschreibung der Klassen, die für die Erstellung des Verhaltens eines Vehikels wichtig sind, an. Am Ende wird die Funktionalität der erstellten Benutzeroberfläche aufgezeigt.

4.1 Überblick

Die Umsetzung des Konzepts erfolgte, wie in dem auf Seite 49 dargestellten Klassendiagramm. Die einzelnen Teile des Konzepts aus dem vorherigen Kapitel sind farbig dargestellt.

Die Schnittstellen zu OpenSteer bestehen, wie von Reynolds vorgesehen, aus einer eigenen Vehikelklasse und wahlweise einem Plugin zur Steuerung der OpenSteerDemo-Applikation. Dadurch ist zu erwarten, dass die Kompatibilität auch zu zukünftigen Versionen von OpenSteer bestehen bleibt.

Im Verwaltungsteil wurde eine klare Trennung zwischen der allgemeinen Verwaltung von Objekten und den Details, die sich auf die OpenSteerDemo-Applikation beziehen, vorgenommen. Alle Elemente, die zur Nutzung der OpenSteerDemo benötigt werden - dieses sind insbesondere Kameraeinstellungen, Tatstaturbelegungen, Mausfunktionen und OpenGL-Darstellungsparameter - sind in der *OwnPlugin*-Klasse untergebracht. Die *Application*-Klasse befasst sich primär mit der Verwaltung der Vehikel und Initialisierung der Hindernisse (engl.: *obstacle*). Um das zu vereinfachen, bedient sie sich der *Species*- und der *Environment*-Klassen. Die *Species*-Klasse, kann eine beliebige Anzahl von Vehikeln mit gleichen Eigenschaften zusammenfassen. Die *Environment*-Klasse verwaltet die Hindernisse.

Die Klasse *OwnVehicle* steht exemplarisch für eine benutzereigene Vehikel-Klasse, der Mittels der von der *ThinkingVehicle*-Klasse geerbten Funktionalitäten ein Verhalten zugewiesen werden kann

Das ThinkingVehicle ermöglicht einem Vehikel das Hinzufügen und Nutzen der Bedürfnisse. Obwohl ein ThinkingVehicle nicht im eigentlichen Sinne denken

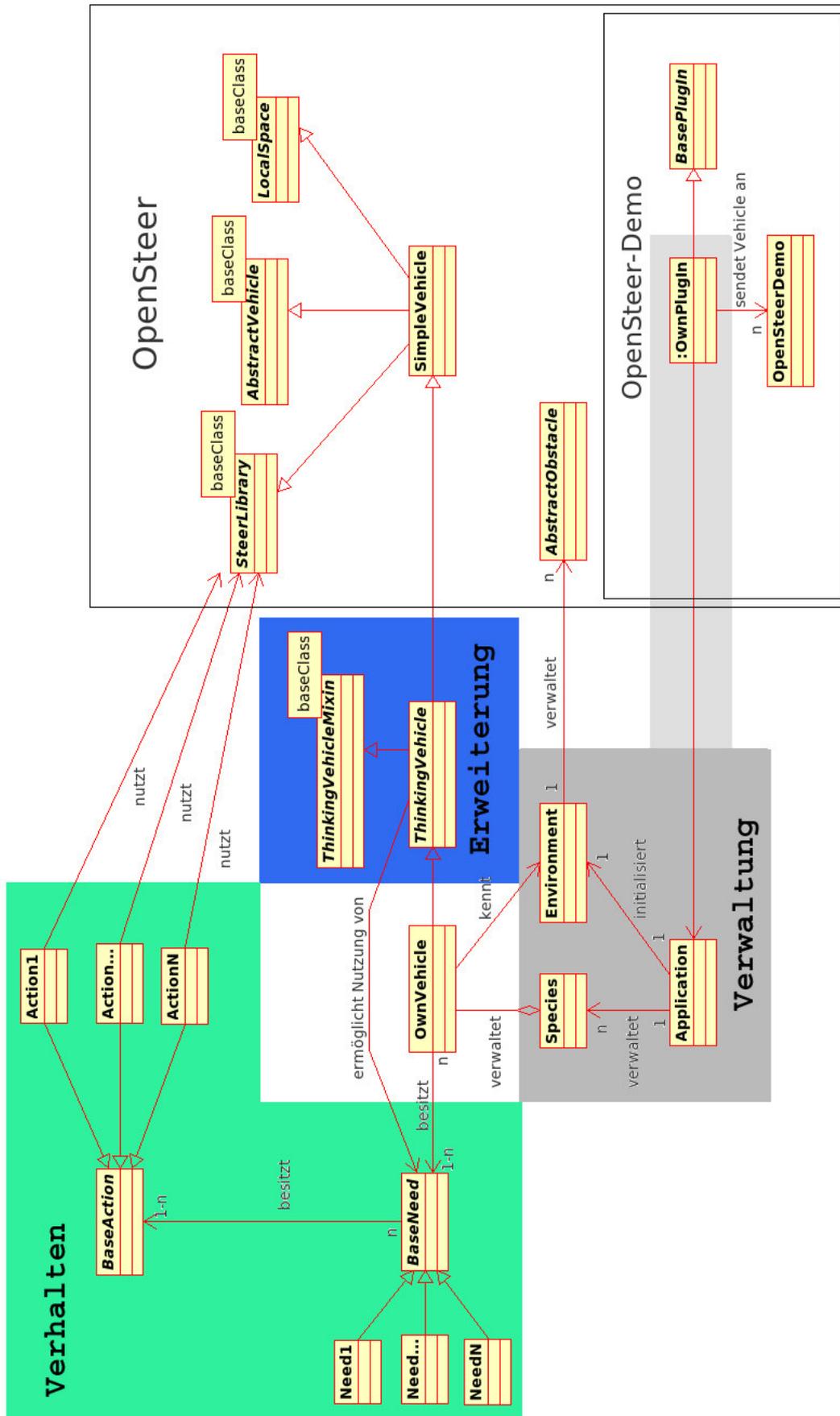


Abbildung 22: Klassendiagramm des Gesamtsystem

kann, soll durch die Bezeichnung die Verbesserung gegenüber dem SimpleVehicle von OpenSteer hervorgehoben werden. Davon getrennt werden alle anderen wichtigen, erweiterten Vehikelfunktionen, die auch ohne die Bedürfnisse nutzbar sind, nach dem von OpenSteer verwendeten Prinzip in einer entsprechenden Templateklasse, dem *ThinkingVehicleMixin*, aufbewahrt.

Die *BaseNeed*- und die *BaseAction*-Klasse fungieren als Basisklassen für alle Aktionen und Bedürfnisse. Sie sorgen für einheitliche Schnittstellen zur ThinkingVehicle-Klasse. Diesen wird ein Zeiger auf das ThinkingVehicle übergeben, über den sie auf die Steuerungsverhalten aus der *SteerLibrary*-Klasse von OpenSteer zugreifen können.

4.2 Beschreibung der wichtigsten Klassen

4.2.1 Die ThinkingVehicleMixin-Klasse

Die *ThinkingVehicleMixin*-Klasse stellt einige wichtige Funktionen zur Verfügung.

Die Geschwindigkeiten

Die *ThinkingVehicleMixin*-Klasse ermöglicht einem Vehikel die Nutzung von zwei verschiedenen Geschwindigkeitsstufen. Die eine ist die Standardgeschwindigkeit, die im Normalfall als Höchstgeschwindigkeit des Vehikels gesetzt wird. Die Zweite ist die Boostgeschwindigkeit, die in besonderen Fällen zum Einsatz kommen kann. Die Boostfunktion ist ein wichtiges Instrument zur Erzeugung realistisch wirkender Verhaltensweisen. Bei Aktivierung des Boostmodus wird die Höchstgeschwindigkeit des Vehikels für einen bestimmten Zeitraum erhöht. Die maximal mögliche Boostgeschwindigkeit lässt sich frei einstellen.

Bei der Aktivierung können drei Parameter angegeben werden: die Boostzeit, die Beruhigungszeit und der Skalierungsfaktor für die maximale Boostgeschwindigkeit. Ist der Skalierungsfaktor Eins, wird die maximale Boostgeschwindigkeit als neue Höchstgeschwindigkeit gesetzt. Ist er Null, wird die Standardgeschwindigkeit verwendet. Werte zwischen Null und Eins ergeben entsprechende Abstufungen. Die Boostzeit bestimmt, wie lange die neue Höchstgeschwindigkeit gesetzt bleibt. Nach Ablauf der Boostzeit wird die aktuelle Geschwindigkeit des Vehikels wieder gedrosselt, bis sie der Standardgeschwindigkeit entspricht. Besaß das Vehikel am Ende der Boostzeit

die höchstmögliche Geschwindigkeit, benötigt es genau die angegebene Beruhigungszeit, um zur Standardgeschwindigkeit zurückzukehren. War es langsamer, beruhigt es sich entsprechend schneller. Dadurch ist gewährleistet, dass es nach Ablauf der Boostzeit zu keiner Geschwindigkeitszunahme über der Standardgeschwindigkeit kommt.

Ist im weiteren Verlauf nichts anderes angegeben, wird von einer Boostzeit von 0 Sekunden und einem Skalierungsfaktor von Eins ausgegangen. Dies sind die Standardwerte für den Boostmodus.

Die Agilität

Bei der Arbeit mit dem SimpleVehicle hat sich gezeigt, dass es oft sehr mühsam ist, die Agilität eines Vehikels allein über seine Masse und maximalen Kraft zu steuern. Die Bewegungen erscheinen schnell als hektisch, wenn die Kraft zu groß und die Masse zu klein wird. Andererseits kann das Vehikel oft nicht schnell genug beschleunigen, wenn man die Einstellungen ändert.

Deshalb wurde die Möglichkeit gegeben, die maximale seitliche Auslenkung eines Vehikels pro Simulationsschritt zu beschränken. Die Einschränkung kann frei zwischen Null und 180 Grad gewählt werden. Dadurch kann ein ehemals hektisches Vehikel beruhigt werden, ohne ihm seine Fähigkeit zur schnellen Beschleunigung zu nehmen.

Da es spezielle Aktionen wie dem Flüchten eines Vehikels gibt, während denen man die Einschränkung kurzzeitig aufheben will, wurde ein entsprechender Modus eingeführt. Dieser wurde in Anlehnung an die Comicfigur des Hulk, der in Notsituationen unglaubliche Kräfte entwickeln kann, der Hulkmodus genannt. Der Hulkmodus deaktiviert die Einschränkung und schaltet sich nach jedem Simulationsschritt automatisch wieder ab.

Die Kurvenneigung

Im SimpleVehicle ist bereits eine Möglichkeit, die seitliche Schräglage eines Vehikels zu berechnen, implementiert. Die Funktionsweise ist in Abschnitt 9 (auf S.11) erklärt. Leider erlaubt die Methode keine Parametrisierung, weswegen sie in der ThinkingVehicleMixin-Klasse erneut implementiert werden musste.

Die Parameter sind die Intensität, mit der sich das Vehikel nach einer Kurve wieder aufrichtet, die Stärke, mit der es sich in die Kurve neigt und ein zur

Simulationszeit proportionaler Faktor, der die Überblendungsgeschwindigkeit der Aufrichtung bestimmt.

Die Beziehungen (zu anderen Vehikeln)

Um es einem Vehikel zu ermöglichen, Beziehungen zu anderen Vehikeln zu definieren, verwaltet die `ThinkingVehicleMixin`-Klasse eine Liste, in der die verschiedenen, möglichen Arten von Relationen aufgelistet sind. Derzeit sind dies die Relationsarten Feind, Beute, Freund, Mutter und Kind. Die Liste kann durch weitere Einträge leicht um neue Arten erweitert werden.

Eine konkrete Relation wird über den Namen des anderen Vehikels und der Art der Relation definiert. Für jede Relationsart verwaltet deshalb jedes Vehikel eine eigene individuelle Liste, in der die Namen der Vehikel mit entsprechender Relation gespeichert sind. Da der Name eines Vehikels nicht eindeutig sein muss, kann hierüber auch die Relation zu einer ganzen Gruppe von Vehikeln definiert werden.

Die Möglichkeiten zur Definition von Beziehungen entsprechen den im Konzeptteil auf Seite 44 genannten.

Die Nachbarschaften

Die Nachbarschaftsdatenbank von `OpenSteer` ermöglicht es, alle Vehikel herauszufinden, die sich innerhalb eines frei bestimmbaren Radius um das Vehikel befinden. Die Abfrage der Nachbarschaftsdatenbank geschieht zu Beginn jedes Simulationsschritts. Alle späteren Nachbarschaftsberechnungen beruhen auf den Vehikeln, die in diesem Schritt gefunden wurde. Deshalb wird der Radius für diese Suche vom `ThinkingVehicleMixin` als `Sichtradius` bezeichnet.

Mögliche Verdeckungen von Vehikeln durch Hindernisse werden in der Nachbarschaftsdatenbank nicht berücksichtigt. Deshalb stellt die `ThinkingVehicleMixin`-Klasse eine entsprechende Methode zur Verfügung. Diese funktioniert bislang nur für kugelförmige Hindernisse, die von außen betrachtet werden, da eine Prüfung für andere Hindernisarten bislang nicht notwendig war.

In den Berechnungen wird ein Vehikel als nicht sichtbar angesehen, sobald seine `Bounding Sphere` oder sein `Bounding Circle` (der Umfang jedes Vehikels wird als Kreis/Kugelförmig angesehen, siehe S. 37) vollständig von einem Hindernis verdeckt ist.

Eine Verdeckung kann direkt ausgeschlossen werden, wenn die Länge des Vektors \vec{d}_o vom Vehikel zum Hindernis, größer als der Länge des Vektors \vec{d}_v vom zu betrachtenden Vehikel ist. Ebenso ist keine Verdeckung möglich, wenn der Winkel zwischen dem Hindernismittelpunkt und dem Mittelpunkt des anderen Vehikels mehr als 90° beträgt. Der eigentliche Test lässt sich dann durch Prüfung des Vehikelabstandes d mit

$$d = \frac{|\vec{d}_o \times \vec{d}_v|}{|\vec{d}_v|}$$

auf die verlängerte Linie durch den Mittelpunkt des Hindernisses durchführen. Ist $d + r_v < r_o$, ist das Vehikel vom Hindernis verdeckt.

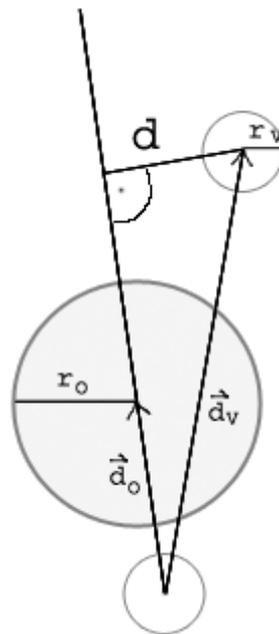


Abbildung 23: Verdeckungsberechnung

4.2.2 Die ThinkingVehicle-Klasse

In der ThinkingVehicle-Klasse ist die in Abschnitt 3.3 (S. 45) beschriebene Funktionalität zur Verwendung der Bedürfnisse und des Standardbedürfnisses implementiert. Die Bedürfnisse werden in einer Liste verwaltet.

4.2.3 Die ShadowVehicle-Klasse

Die ShadowVehicle-Klasse kann nur im Zusammenhang mit dem in Kapitel 1.2.2 (S. 5) beschriebenen **living** Systems von Vertigo Systems sinnvoll eingesetzt

werden. Deshalb ist sie in dem Klassendiagramm auf Seite 49 auch nicht aufgeführt. Sie besitzt keinerlei Funktionalität für das Verhaltensframework, ist aber für das Verständnis einiger Aktionen, die in Kapitel 4.2.5 beschrieben werden, sehr wichtig, weswegen sie hier erklärt werden soll.

Das ShadowVehicle ist direkt vom SimpleVehicle abgeleitet. Ein Vehikel vom Typ ShadowVehicle wird immer dann erzeugt, wenn die Kamera des **living** Systems einen neuen Schatten auf der Projektionsfläche detektiert.

Die virtuelle Kamera befindet sich auf der Y-Achse und benutzt eine perspektivische Projektion. Da die Projektionsebene die X-Z-Ebene ist, wird die Y-Position des ShadowVehicle auf Null gesetzt und die X-Z-Position an die entsprechende Position des realen Schattens angepasst. Die Ausdehnung des Schattens wird durch den Radius des Vehikels approximiert. Eine Positionsänderung des realen Schattens bewirkt eine Positionskorrektur des Vehikels.

Ein Objekt vom Typ ShadowVehicle kann also als zweidimensionale Repräsentation eines Schattens in der X-Z-Ebene betrachtet werden.

4.2.4 Die Need-Klassen

Alle Bedürfnisklassen (engl. = need) sind von der abstrakten Klasse BaseNeed abgeleitet. Sie ermöglicht das Hinzufügen und Entfernen der Aktionen in einer hierarchischen Ordnung. Das BaseNeed zwingt alle abgeleiteten Klassen dazu, eine Methode zu implementieren, in der festgelegt wird, wann das Bedürfnis als befriedigt gilt. Außerdem erlaubt sie die Auswahl, ob zur Befriedigung des Bedürfnisses nur die erste, mögliche Aktion oder aber alle möglichen Aktionen gleichzeitig ausgeführt werden sollen. Ist letzteres der Fall, wird von einem *additiven* Bedürfnis gesprochen.

Nachfolgend werden alle bisher erstellten Bedürfnisse und ihre Eigenarten beschrieben. Allen Bedürfnissen ist gemein, dass sie sehr universell eingesetzt werden können und als Basisklassen für weitere Bedürfnisse genutzt werden können.

Das DefaultNeed

Das DefaultNeed ist das einfachste aller Bedürfnisse. Es gilt immer als unbefriedigt. Deshalb ist es sinnvoll, dieses Bedürfnis als letztes in der Hierarchie

einzufügen, damit gewährleistet ist, dass immer einem Bedürfnis nachgegangen werden kann. Das in Abschnitt 3.3 (S. 45) beschriebene Standardbedürfnis des ThinkingVehicle (siehe Abschnitt 4.2.2 Seite 53) ist beispielsweise von diesem Typ.

ActionBasedNeed

Ein Bedürfnis von diesem Typ ist nur dann befriedigt, wenn keine seiner Aktionen ausführbar ist. Es kann immer dann eingesetzt werden, wenn die Bedingung zur Befriedigung des Bedürfnisses hinreichend durch die Vorbedingungen seiner Aktionen umschrieben ist. Dies ist meist bei Bedürfnissen gegeben, die an das Eintreten externer Ereignisse gebunden sind.

Zum Beispiel könnte ein Sicherheitsbedürfnis befriedigt sein, solange sich alle Feinde weiter als einhundert Meter entfernt befinden. Die Vorbedingung für die Aktion „Fliehe vor Feind“ könnte ebenfalls voraussetzen, dass sich ein Feind in einem Umkreis von einhundert Metern befinden muss. In diesem Fall sind die Bedingungen identisch, und es kann auf eine erneute Definition verzichtet werden. Dadurch werden unnötige doppelte Bedingungsdefinitionen überflüssig und die Aktionen können problemlos benutzt werden.

Hinweis: Zur effizienten Auswertung der Vorbedingungen in den Aktionen sie auch im Anfang von „Die Action-Klassen“ auf Seite 56.

LimitBasedNeed

Das Bedürfnis besitzt einen Grenzwert und eine Variable, sowie die Einstellungsmöglichkeit, ob es sich um einen oberen oder unteren Grenzwert handelt. Über- oder unterschreitet die Variable den Grenzwert (je nach Einstellung), gilt das Bedürfnis als unbefriedigt.

Es können also Bedingungen formuliert werden, die sich durch eine Größer- / Kleinerrelation ausdrücken lassen. Dies eignet sich besonders für die Simulation von Bedürfnissen, die durch eine innere Motivation heraus entstehen. Zum Beispiel könnte das Hungerbedürfnis eines Vehikels imitiert werden, indem die Variable mit Voranschreiten der Zeit immer höher gesetzt wird. Überschreitet die Variable den Grenzwert, begibt sich das Vehikel auf Futtersuche.

4.2.5 Die Action-Klassen

Alle Aktionen sind von der abstrakten Klasse `BaseAction` abgeleitet. Sie ermöglicht die auf Seite 46 geforderten Funktion, der Gewichtung des Steuerungsvektors.

Sie zwingt abgeleitete Klassen dazu, zwei Methoden zu implementieren. Die eine berechnet den Steuerungsvektor, den das Vehikel zur Ausführung der Aktion benötigt. Die andere dient der Prüfung, ob die Vorbedingungen zur Ausführung der Aktion gegeben sind.

Die `BaseAction` besitzt einen Mechanismus, durch den die Vorbedingung der von ihr abgeleiteten Klassen in einem Simulationsschritt nicht doppelt ausgewertet werden. Sie merkt sich das Ergebnis der letzten Auswertung und prüft, ob Simulationszeit seit der letzten Prüfung vergangen ist. Dies ist besonders für einen Effizienten Einsatz des `ActionBasedNeeds` wichtig, welches von sich aus stets alle Vorbedingungen pro Simulationsschritt überprüft.

Bisher kamen die nachfolgend beschriebenen Aktionen bei der Verwendung des **living** Systems zum Einsatz: `ObstacleAvoidanceAction`, `SeekAction`, `SeparationAction`, `NonShadowSeparationAction`, `WanderAction`, `FlockAction`, `FlockWithRelationTypeAction`, `ShadowEvasionAction`, `EnemyFleeAction`, `HuntForPreyAction` und `HideFromEnemyAction`.

Diese werden nachfolgend in der aufgeführten Reihenfolge detailliert erklärt. Einige der Aktionen sind nicht viel mehr als eine „Hülle“, um die Funktionen der `SteerLibrary` von `OpenSteer` in Verbindung mit den Bedürfnissen nutzen zu können. Deshalb wird mit der Beschreibung der einfachsten Aktionen begonnen.

ObstacleAvoidanceAction

Mit Hilfe dieser Aktion kann ein Vehikel Hindernissen ausweichen.

Hierfür greift sie auf die entsprechende Funktion aus der `SteerLibrary`-Klasse zurück, deren Funktionsweise in Kapitel 2.1.1 (S. 14) beschrieben ist. Die Vorbedingung zur Ausführung ist erfüllt, wenn die aufgerufene Funktion keinen Null-Vektor zurückliefert.

SeekAction

Die Aktion dient dem Steuern zu einem festgelegten Ziel. Das Ziel kann als Parameter übergeben werden.

Eine SeekAction kann immer ausgeführt werden. Die Vorbedingung ist also stets erfüllt.

Zur Berechnung des Steuerungsvektors greift die Aktion auf die entsprechende Seek-Funktion aus der SteerLibrary zurück, deren Funktionsweise in Kapitel 2.1.1 (S. 11) nachgelesen werden kann.

Die Aktion kann dazu verwendet werden, um Vehikel gezielt zu steuern. Beispielsweise wurde es dazu eingesetzt, die Molekül-Vehikel aus dem Beispiel in Abschnitt 1.2.2, von einer Seite des Pools auf die andere zu navigieren.

SeparationAction

Bei Anwendung dieser Aktion, nimmt ein Vehikel Abstand zu allen Nachbarvehikeln. Als Parameter können der Winkel und Radius angegeben werden, in dem nach Nachbarn gesucht werden soll. Zur Berechnung des Steuerungsvektors greift die Aktion auf die entsprechende Separation-Funktion aus der SteerLibrary zurück, deren Funktionsweise in Kapitel 2.1.1 (S. 17) nachgelesen werden kann.

Die Vorbedingung ist erfüllt, wenn sich Nachbarn im angegebenen Suchraum befinden.

NonShadowSeparationAction

Die Aktion ist von der SeparationAction abgeleitet. Sie dient dazu, dass ein Vehikel Abstand zu allen Vehikeln mit Ausnahme von Schattenvehikeln (siehe Seite 53) hält. Die Extrabehandlung der Schattenvehikel ist notwendig, da die einfache SeparationAction sich mit der ShadowEvasionAction (siehe hierzu auch Seite 61) überlagern könnte.

WanderAction

Die Aktion ermöglicht das Umherwandern eines Vehikels. Sie baut hierfür auf das in der SteerLibrary-Klasse implementierte Wanderverhalten auf.

Zusätzlich wird ein Parameter verwendet, der bestimmt, wie hoch die Wahrscheinlichkeit ist, dass bei einem Aufruf der Aktion kein Steuerungsvektor berechnet wird. Diese Funktion erwies sich als sinnvoll, weil auch Vehikel erzeugt werden sollten, die nur sporadisch und nicht kontinuierlich ihre Richtung ändern.

Eine Vorbedingung zur Ausführung dieser Aktion ist nicht notwendig, deshalb wird bei einem entsprechenden Test immer *wahr* zurückgegeben.

FlockAction

Die FlockAction ermöglicht die Bildung eines Schwarms. Das Grundprinzip entspricht dem in Kapitel 2.1.1 (S. 19) beschriebenen Verfahren von Reynolds, mit dem sich durch geschickte Anwendung des Cohesion-, Alignment- und Separation-Verhaltens die Vehikel in einer Gruppe fortbewegen können, ohne miteinander zu kollidieren oder sich voneinander zu entfernen. Die Parameter für jedes der drei Verhalten sind der Radius und der Winkel der zu betrachtenden Nachbarschaften, sowie ein Gewichtungsfaktor. Einen Anhaltspunkt für eine sinnvolle Einstellung der Parameter bietet zum Glück ein implementierter Flocking-Algorithmus im SimpleVehicle. In diesem sind die Parameter wie folgt gesetzt:

	Radius	Winkel	Gewichtung
Separation	r	135°	$g \cdot \frac{3}{2}$
Alignment	$r \cdot 1,5$	45°	g
Cohesion	$r \cdot 2$	99°	g

Tabelle 1: Standardwerte der FlockAction

Da es sehr schwer ist, sich das Zusammenwirken der drei Verhaltensweisen vorzustellen, werden diese in Abbildung 27 visualisiert.



Abbildung 24: Separation Nachbarschaft

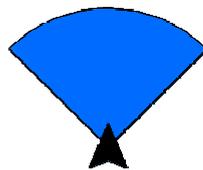


Abbildung 25: Alignment Nachbarschaft

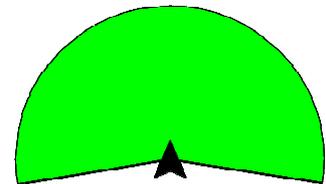


Abbildung 26: Cohesion Nachbarschaft

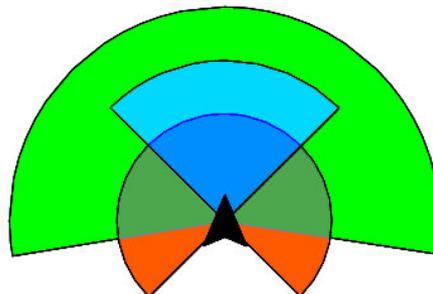


Abbildung 27: alle Nachbarschaften

Besonders bei der Verwendung von wenigen Vehikeln oder am Rand von Schwärmen ergaben sich mit dem klassischen Flocking-Algorithmus und den obenstehenden Einstellungen einige Schwierigkeiten.

Ein Problem tritt auf, wenn zwei Vehikel gleichzeitig eine Separation ausführen. In diesem Fall kann es geschehen, dass sie sich soweit voneinander abwenden, dass sie den jeweils anderen weder in ihrer Alignment-, noch in ihrer Cohesion-Nachbarschaft wiederfinden. Befinden sich auch keine anderen Vehikel in der Nachbarschaft, würden die Vehikel voneinander wegsteuern ohne wieder umzukehren. Durch eine Erhöhung des Cohesion-Radius könnte dieses Problem zwar behoben werden, doch würden sich dann auch die vordersten Vehikel eines Schwarms stets zum Mittelpunkt der anderen Vehikel begeben wollen. Das Ergebnis wäre ein Schwarm, der sich kaum noch fortbewegt, da alle Vehikel an dieselbe Stelle streben.

Deshalb wird immer, wenn in der Cohesion- und der Alignment-Nachbarschaft keine Vehikel gefunden werden können, sich aber welche in der Separation Nachbarschaft befinden, die Alignment-Nachbarschaftssuche auf 180° erweitert. Dadurch ist das Separation-Verhalten niemals alleine aktiv. Dies bewirkt, dass sich die Vehikel zwar noch gegenseitig abstoßen, sich aber nicht mehr so stark wie vorher voneinander abwenden und stets Cohesion oder Alignment aktiv bleiben.

Ein anderes Problem tritt auf, wenn die Vehikel eines Schwarms zum Beispiel durch einen plötzlich auftretenden Feind auseinandergetrieben werden. In diesem Fall geschieht es häufig, dass sich die Vehikel zwar noch in der Nähe zueinander befinden, aber in keiner der drei Nachbarschaften gefunden werden können, da sie sich zu stark voneinander abgewendet haben. Visuell entsteht der Eindruck, als wenn die Vehikel die Orientierung verloren hätten.

Zur Lösung des Problems wird immer dann eine Cohesion mit einem Radius von 180° angewendet, wenn die anderen beiden Steuerungsverhalten nicht angewendet werden konnten, sich auch im normalen Cohesion Radius keine Vehikel befinden und der Abstand zum nächsten Vehikel nicht größer als der Radius des Alignment-Verhaltens ist. Letzteres verhindert, dass das Verfahren angewendet wird, wenn sich das Vehikel an der Spitze eines Schwarms befindet und als Anführer für andere Vehikel fungiert.

Beide Verfahren haben sich zum besseren Zusammenhalt von Schwärmen als sehr nützlich erwiesen.

Zur Verbesserung des natürlichen Eindrucks bei der Schwarmbildung wurde ein weiteres Verfahren eingeführt, welches bewirkt, dass ein einzelner Fisch, der sich

in zu großer Entfernung zum Schwarm befindet, seine Geschwindigkeit erhöht, um zum Schwarm aufzuschließen. Natürlich soll er seine Geschwindigkeit nur erhöhen, wenn sich der Schwarm vor ihm und nicht hinter ihm befindet. Dieser Mechanismus verleiht der Schwarmbildung ein vielfaches Maß an Lebendigkeit.

Realisiert wurde dies, indem ein Vehikel in den Boostmodus (siehe Kapitel 4.2.1 S. 50) versetzt wird, wenn der Abstand zum nächsten Vehikel kleiner als der Radius des Alignment-Verhaltens ist. Als Parameter des Boostmodus werden für die Boostzeit 0 Sekunden, die Beruhigungszeit 0.8 Sekunden und für den Skalierungsfaktor 0.5 verwendet. Das Vehikel beschleunigt also maximal bis zur halben Boostgeschwindigkeit und bremst langsam ab, sobald es den Alignment-Radius erreicht hat. Die Abfrage, ob sich der nächste Nachbar vor oder hinter dem Vehikel befindet, kann durch den Kosinus des Winkels zwischen der Vorwärtsachse und dem Vektor vom Vehikel zum Nachbarn berechnet werden. Für die folgende Berechnung wird vorausgesetzt, dass der Vorwärtsvektor *forward* des Vehikels, der die Vorwärtsachse des Vehikels in globalen Koordinaten bezeichnet, normiert ist.

$$\vec{o}_{ffset} = neighborPos - vehiclePos$$

$$\cos \alpha = \frac{\vec{o}_{ffset} \bullet forward}{|\vec{o}_{ffset}| \cdot 1}$$

Ist der Kosinus kleiner als Null, befindet sich der Nachbar hinter dem Vehikel, ansonsten davor.

Die Vorbedingung zur Anwendung dieser Aktion ist, dass sich mindestens ein Vehikel in einem der drei Nachbarschaftsbereiche inklusive der möglichen Erweiterung dieser Bereiche durch eine der Verbesserungen befindet.

FlockWithRelationTypeAction

Diese Aktion ermöglicht genau wie die FlockAction die Bildung eines Schwarms. Sie ist auch von ihr abgeleitet und greift auf ihre Methoden zurück. Allerdings wird nur ein Schwarm mit Vehikeln gebildet, die entweder denselben Namen besitzen oder zu denen eine bestimmte Relation besteht. Die Relationen können beliebig gewählt werden.

Sinnvoll wäre es zum Beispiel alle Vehikel, zu denen eine „Friend“-Relation besteht, in die Schwarmbildung zu integrieren.

ShadowEvasionAction

Die ShadowEvasionAction ermöglicht das Fliehen vor einem ShadowVehicle. Wie in 4.2.2 (Seite 53) bereits erwähnt, repräsentiert ein ShadowVehicle den Schattenwurf eines Nutzers auf die Projektionsfläche des **living** Systems.

Als Parameter kann die minimale Fluchtdistanz eingestellt werden. Ist der Abstand des Vehikels zum in die X-Z-Ebene des Vehikels projizierten ShadowVehicle kleiner als diese Distanz, wird ein Fluchtverhalten wie in Abschnitt 2.1.1 (S. 12) auf den projizierten Mittelpunkt des ShadowVehicles ausgeführt.

Der Abstand d vom Vehikel zum projizierten Schatten kann dann durch Anwendung der Strahlensätze herausgefunden werden (siehe auch Abbildung 28). Wenn c der Abstand der Kamera zum Nullpunkt ist und die Kamera, wie auf Seite 53 beschrieben, auf der Y-Achse ausgerichtet wird, ist der projizierte Mittelpunkt des Schattenvehikels

$$S' = \begin{pmatrix} \left(\frac{S_x}{c} \cdot (V_y + c) \right) \\ V_y \\ \left(\frac{S_z}{c} \cdot (V_y + c) \right) \end{pmatrix}$$

V bezeichnet die Position des Vehikels. Durch den Vektor $\overrightarrow{S'V}$, der vom projizierten Schattenmittelpunkt zur Position des Vehikels zeigt, kann der Punkt P auf dem Radius r_s des Schattens ermittelt werden, der den kürzesten Abstand zum Vehikel besitzt.

$$P = S + \frac{\overrightarrow{S'V} \cdot r_s}{|\overrightarrow{S'V}|}$$

Der Punkt P wird wie folgt in die Ebene des Vehikels projiziert:

$$P' = \begin{pmatrix} \left(\frac{P_x}{c} \cdot (V_y + c) \right) \\ V_y \\ \left(\frac{P_z}{c} \cdot (V_y + c) \right) \end{pmatrix}$$

Der Abstand d vom Vehikel zu diesem nahsten Punkt P' ergibt sich durch

$$d = \frac{V - P'}{|V - P'|} - r_v$$

mit r_v als Radius des Vehikels.

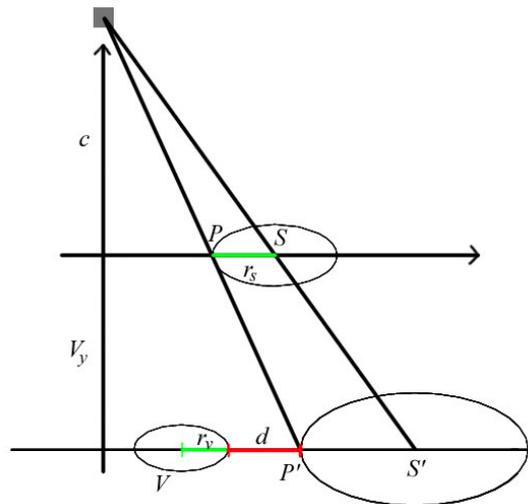


Abbildung 28: Berechnung des Schattenabstandes

Wird der minimale Abstand unterschritten, befindet sich das Vehikel innerhalb des Radius des Schattenvehikels. In diesem Fall kann, wenn gewünscht, der Hulkmodus, sowie der Boostmodus des Vehikels genutzt werden, um sich besonders schnell vom Schatten zu entfernen. Der Boostmodus wird nur aktiv, wenn sich der Schatten hinter dem Vehikel befindet. Andernfalls könnte das Vehikel in Richtung des Schattens beschleunigen, was zu einem optisch unnatürlichen Ergebnis führen würde.

EnemyFleeAction

Diese Aktion ermöglicht die Flucht vor allen Vehikeln, zu denen eine Feindrelation besteht. Die Flucht kann im Boostmodus geschehen und der Hulkmodus kann benutzt werden. Der Boostmodus bleibt noch eine Sekunde nach Eintreten der Aktion aktiv, damit sich ein Vehikel noch weiter von seinem Feind entfernen kann. Die anschließende Beruhigungszeit kann als Parameter übergeben werden.

Ein Vehikel flieht nur vor dem nahsten Feind. Die Berechnung des Steuerungsvektors kann in Abschnitt 2.1.1 (S. 12) nachgelesen werden.

Die Vorbedingung ist erfüllt, sobald sich ein entsprechendes Vehikel in der Nachbarschaft befindet.

HuntForPreyAction

Ein Vehikel, das diese Aktion verwendet, wird versuchen, ein Vehikel zu fangen, zu dem es eine Beuterelation besitzt. Die Jagd geschieht im Boost- und im Hulkmodus. Die anschließende Beruhigungszeit ist auf eine halbe Sekunde

festgelegt. Im Gegensatz zur EnemyFleeAction wird der Boostmodus nach Beendigung der Aktion nicht weiter fortgeführt. In einer Simulation wirkt dies, als wenn ein Jäger seine Beute aufgibt, weil diese ihm entkommen ist.

Als Jagdobjekt wird allgemein das nahste Vehikel mit Beuterelation angesteuert, welches sich innerhalb eines frei wählbaren Radius befindet. Die Position der Beute wird leicht vorausgeschätzt, um das Vehikel „abzufangen“. Damit andere Vehikel einen Nutzen davontragen, dass sie sich zu einem Schwarm zusammengefunden haben, kann ein sogenannter Verwirrungsradius angegeben werden. Durch ihn wird die mittlere Position aller Vehikel berechnet, die sich im Radius um das nahste Vehikel befinden. Dieser Punkt wird mit der Position des nahsten Vehikels im Verhältnis 3:2 verrechnet und zum Jagen angepeilt. Das Jagen selber entspricht dem Seek Verhalten von Abschnitt 2.1.1 (S. 11).

Als weitere Besonderheit besitzt die HuntForPreyAction einen Gedächtnisspeicher, in dem die Position und der Geschwindigkeitsvektor des zuletzt gesichteten Beutevehikels festgehalten werden. Ist kein Beutevehikel in Sicht, wird der Gedächtnisspeicher abgefragt. Kann sich der Jagende noch an ein Vehikel „erinnern“, wird die voraussichtliche Position der Beute auf Grund des gemerkten Geschwindigkeitsvektors geschätzt. Der Geschwindigkeitsvektor wird zum Schätzen halbiert, da es sonst passieren kann, dass sich der Jäger unrealistisch weit von dem Punkt entfernt, an dem er den letzten Blickkontakt zu seiner Beute hatte. Die Vorhaltezeit des Speichers wird aus demselben Grund auf zwei Sekunden begrenzt.

Durch den Gedächtnisspeicher können im Zusammenspiel mit der im Anschluss beschriebenen HideAction sehr spannende Simulationen erzielt werden (siehe hierzu auch Abbildung 30 auf Seite 65).

Die Vorbedingung der HuntForPreyAction ist erfüllt, wenn sich ein entsprechendes Beutevehikel in der Nachbarschaft befindet oder sich noch ein Vehikel mit Beuterelation im Gedächtnisspeicher befindet.

HideFromEnemyAction

Die Aktion erlaubt es einem Vehikel, sich vor einem anderen Vehikel mit Feindrelation zu verstecken. Das Aufsuchen des Verstecks geschieht prinzipiell im Boostmodus. Der Hulkmodus kann optional als Unterstützung dazugeschaltet werden. Die Beruhigungszeit des Hulkmodus ist als Parameter wählbar.

Die Aktion wird erst ausgelöst, wenn ein Feind eine frei wählbare Fluchtdistanz zum Vehikel unterschreitet. Geschieht dies, sucht das Vehikel die Umgebung nach geeigneten Hindernissen zum Verstecken ab. Derzeit werden nur sphärische Hindernisse als Verstecke betrachtet, da andere Formen noch keine Anwendung fanden. Die Idee lässt sich aber auch leicht auf quaderförmige und rechteckige Hindernisse ausweiten.

Ein Hindernis gilt dann als geeignet, wenn dessen Entfernung nicht größer ist, als die eingestellte Fluchtdistanz. Außerdem darf es nicht in derselben Richtung gelegen sein, in der der Feind gesichtet wurde, es sei denn, der Abstand d zum Hindernis ist geringer als $\frac{3}{4}$ der Länge des Abstandes zum Feind (siehe Abbildung 29). Um herauszufinden, auf welcher Seite relativ zum Feind das Hindernis gelegen ist, wird der Winkel zwischen dem Vektor vom Vehikel zum Hindernis \vec{v}_o und dem Vektor vom Feind zum Vehikel \vec{e}_v berechnet. Der Kosinus des Winkels kann nach dem bereits in der FlockAction verwendeten Schema wie folgt berechnet werden:

$$\begin{aligned}\vec{e}_v &= \text{vehiclePosition} - \text{enemyPosition} \\ \vec{v}_o &= \text{obstacleCenter} - \text{vehiclePosition} \\ \cos \alpha &= \frac{\vec{e}_v \cdot \vec{v}_o}{|\vec{e}_v| \cdot |\vec{v}_o|}\end{aligned}$$

Ist der Kosinus größer als Null, befindet sich das Hindernis auf der vom Feind abgewandten Seite.

Ist ein geeignetes Hindernis gefunden, befindet sich der bestmögliche Ort zum Verstecken auf der dem Feind gegenüberliegenden Seite des Hindernisses. Dieser Punkt errechnet sich durch

$$\begin{aligned}\vec{e}_o &= \text{obstacleCenter} - \text{enemyPosition} \\ \text{Zielpunkt} &= \frac{\vec{e}_o}{|\vec{e}_o|} \cdot (|\vec{e}_o| + r_o + r_v)\end{aligned}$$

mit r_o als dem Radius des Hindernisses und r_v dem Radius des Vehikels. In Abbildung 29 ist dieser Punkt dargestellt. Auf ihn wird das in Abschnitt 2.1.1 (S. 11) beschriebene Seek Verhalten angewendet.

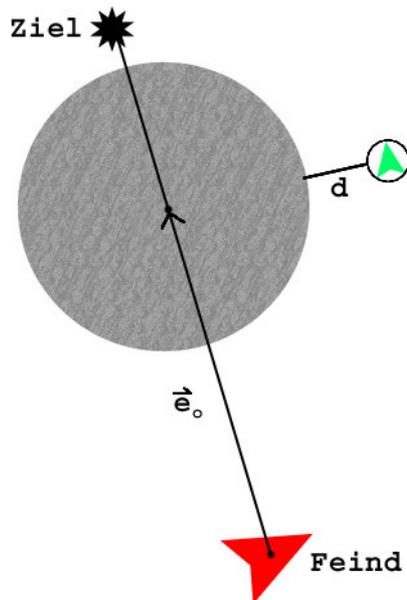


Abbildung 29: Prinzip der HideAction

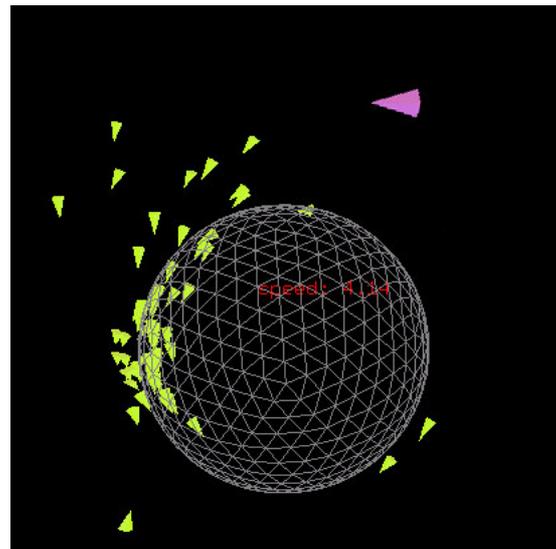


Abbildung 30: Anwendung der HideAction in der OpenSteerDemo

Damit ein Vehikel auch für längere Zeit vor dem Feind versteckt bleibt, besitzt die HideFromEnemyAction einen Gedächtnisspeicher. In diesem wird die Position und der Geschwindigkeitsvektor des zuletzt gesichteten Feindes festgehalten. Die Vorhaltezeit des Speichers kann als Parameter eingestellt werden. Solange sich noch ein Feind im Gedächtnis befindet, wird dessen aktuelle Position mit Hilfe seiner letzten Geschwindigkeit geschätzt. Hierfür wird nur die halbe Länge des Geschwindigkeitsvektors verwendet, weil sonst der Fehler des Schätzwertes zu groß wird. Auf Grund der geschätzten Position wird der Zielpunkt zum Verstecken korrigiert. Dadurch wandert das Vehikel langsam am Hindernis entlang, als wenn es seinen Verfolger abzuschütteln versucht.

Die Vorbedingungen, damit die Aktion ausgeführt werden kann, sind ein geeignetes Hindernis und ein Feind innerhalb des Fluchradius oder im Gedächtnisspeicher.

4.3 Beschreibung der graphischen Benutzeroberfläche

Die graphische Benutzeroberfläche, die im Folgenden mit GUI (engl.: Graphical User Interface) abgekürzt wird, ist ein wichtiges Hilfsmittel, um die Kontrolle über das Verhalten der Charaktere zu behalten. Denn es hat sich als eine der schwersten Aufgaben herausgestellt, alle Parameter im Verhalten so einzustellen, dass sich ein vernünftiges Zusammenspiel ergibt.

Da die Erstellung der GUI kein Teil der eigentlichen Aufgabenstellung ist und sie zur Erstellung autonomer Charaktere auch nicht zwingend erforderlich ist, sollen hier nur ihre Funktionen aufgezeigt, nicht aber ihre Funktionsweise erklärt werden. Es sei an dieser Stelle lediglich erwähnt, dass sie mit Hilfe des QT-Frameworks der Firma Trolltech generiert wurde.

Die GUI erlaubt es, alle wichtigen Eigenschaften eines Vehikels zur Laufzeit zu verändern. Dadurch kann die Entwicklungszeit, die vom ersten Entwurf bis zum perfekt ans Szenario angepassten Vehikel vergeht, ungemein verkürzt werden. Es ist wichtig, dass es dem Benutzer stets möglich ist, verschieden parametrisierte Arten von Vehikeln gleichzeitig zu steuern. Nur so kann das Zusammenwirken verschiedener „Spezies“ getestet werden. In Abbildung 31 erkennt man, wie sich mit der GUI verschiedene Spezies gleichzeitig verwalten lassen. In dem Beispiel wurden vier Spezies mit den Namen Hai, Hering, Forelle und Lachs erzeugt. Für jede Art lassen sich die Anzahl der Individuen (Quantity) und deren allgemeinen Parameter einstellen.

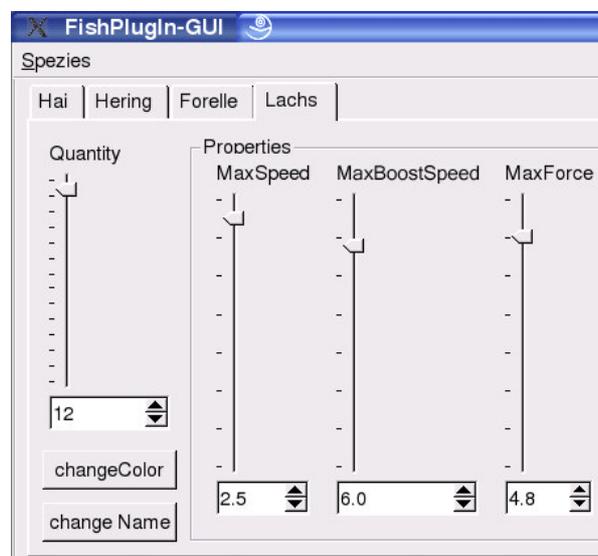


Abbildung 31: Kontrolle über die Spezies

Um den Zusammenhang von Bedürfnissen und Aktionen zu erleichtern, wird der Aufbau des Verhaltens visualisiert (siehe Abbildung 32 auf S. 67). Durch die Darstellung aller verfügbaren Aktionen und Bedürfnisse, werden dem Benutzer seine Nutzungsmöglichkeiten aufgezeigt. Neue Aktionen können zur Laufzeit zu einem Bedürfnis hinzugefügt werden. Wird eine bereits hinzugefügte Aktion mit

der Maus ausgewählt, erscheint automatisch ein neues Fenster, in dem sich alle Parameter der Aktion einstellen lassen.

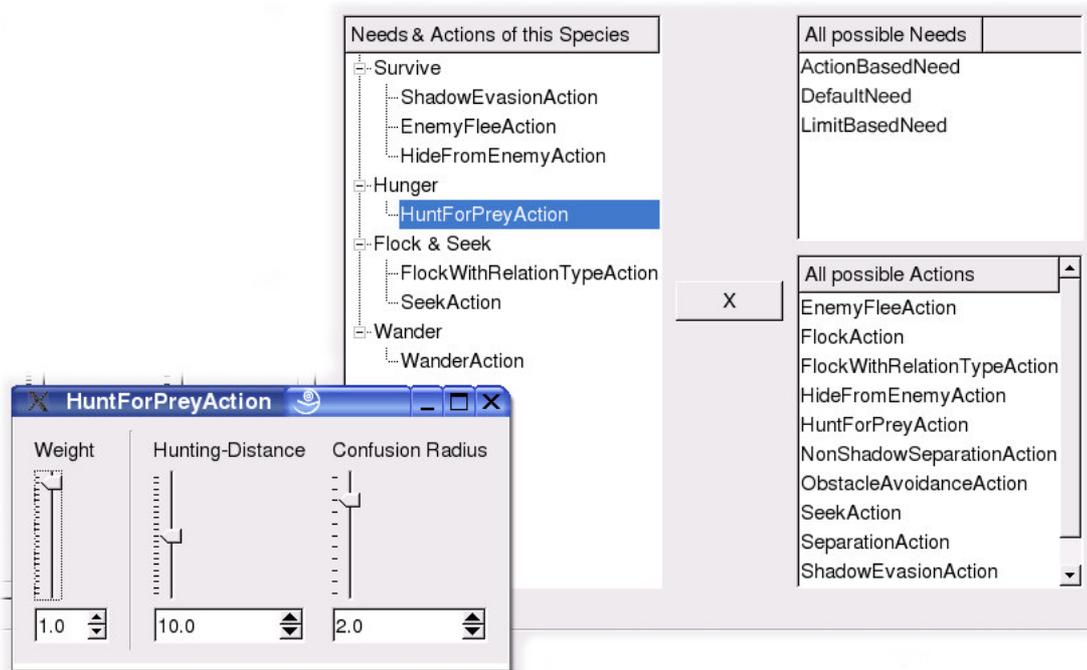


Abbildung 32: Bearbeitung der Bedürfnisse und Aktionen

Die Relationen von einer Spezies zu einer anderen lassen sich ebenfalls leicht einstellen. Für die ausgewählte Spezies werden alle verfügbaren Relationen angezeigt. Über Kontrollkästchen können diese intuitiv aktiviert werden (siehe Abbildung 33 und Abbildung 34).

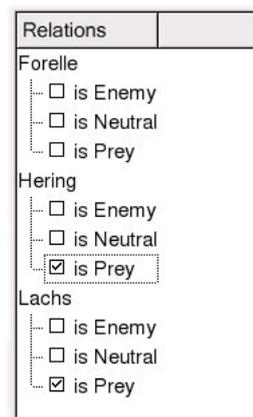


Abbildung 33: Relationen eines Hais

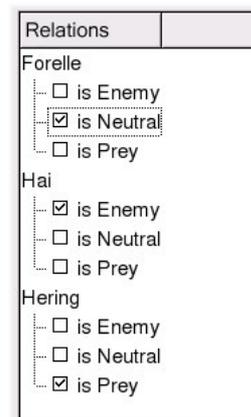


Abbildung 34: Relationen eines Lachses

Kapitel 5. Beispielanwendung: Eine Unterwassersimulation mit Fischen

Dieses Kapitel stellt ein Anwendungsbeispiel für das Verhaltensframework im **living** System vor. Beschrieben wird eine Simulation von Fischen in einer Unterwasserwelt. Diese wurde im Juli 2005 von Vertigo Systems, rmh new media und der mediaMotion AG mit Hilfe des Frameworks entwickelt.

In diesem Zusammenhang wird ein generelles Verhaltensmodell eines Fisches beschrieben, welches für vielfältige Zwecke einsetzbar ist. Im Anschluss daran werden die für das Anwendungsbeispiel erstellten Fischarten erläutert und beschrieben. Zunächst wird der Aufbau des Szenarios erläutert.

Auf der beiliegenden CD befinden sich zusätzliche Bilder und einige Videos der Unterwassersimulation, die die Ergebnisse anschaulich präsentieren.

5.1 Szenariobeschreibung

Erstellt wurde eine Unterwassersimulation mit drei verschiedenen Fischarten. Diese befinden sich in einer Art Pool oder Wasserbecken, welches aus der Vogelperspektive betrachtet wird (siehe auch Abbildung 35 auf S. 69). Der Pool wird auf eine ca. 3x4 Meter große Fläche auf den Fußboden projiziert. Ein Benutzer kann sich frei über diese Fläche bewegen. Mit Hilfe der Schattenerkennung des **living** Systems wird seine Position lokalisiert und die virtuelle Wasseroberfläche des Pools an seinem Standpunkt zum Bewegen angeregt. Über seinen Schatten (in Abbildung 35 als schwarzer Kreis dargestellt) ist es dem Benutzer möglich die Fische zu verscheuchen und durch das Wasser zu jagen. Werden die Fische in Ruhe gelassen, finden sie sich in Abhängigkeit ihrer Art wieder zu Schwärmen zusammen oder streifen als Einzelgänger durch das Becken. Es ist auch möglich, dass ein Fisch Jagd auf andere Beutefische macht (siehe Abbildung 36 auf S. 71). Sind virtuelle Hindernisse in dem Szenario vorhanden, können sich die Fische dahinter verstecken.

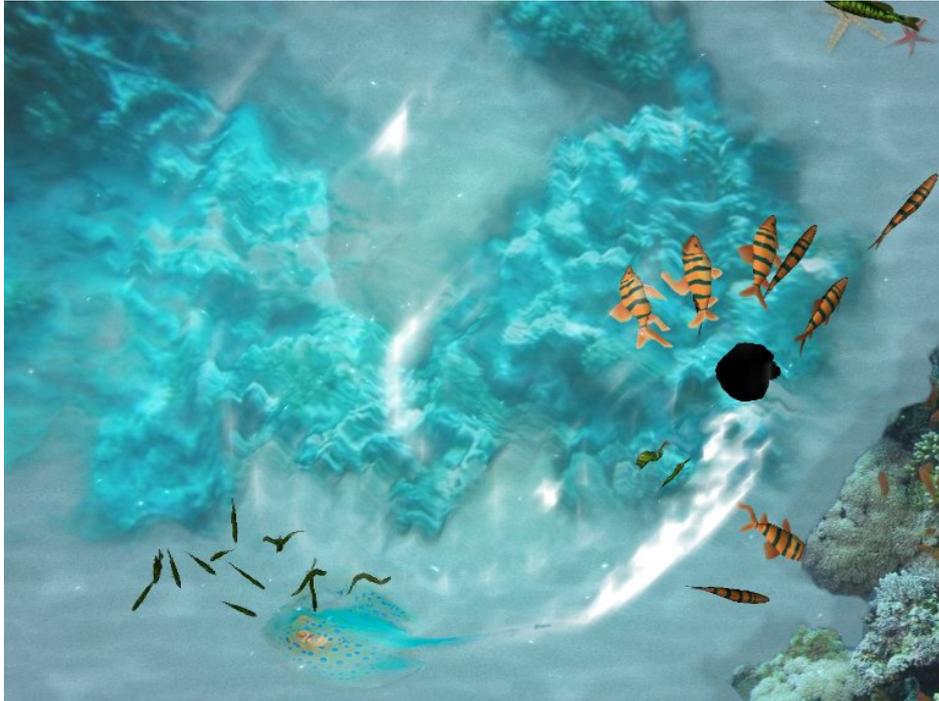


Abbildung 35: Ein Schatten (Benutzer) verscheucht Fische

Den Fischen ist es möglich, den sichtbaren Bereich des Pools zu verlassen. Wenn dies geschieht, werden sie entweder von einem für den Benutzer nicht sichtbaren Hindernis dazu verleitet, wieder in den Pool zu schwimmen, oder das **living** System setzt sie an einer anderen Stelle außerhalb des Pools wieder ein. Dies suggeriert dem Benutzer nur einen kleinen Ausschnitt einer größeren, komplexeren Unterwasserwelt zu betrachten und trägt zu einem realistischeren Gesamtempfinden bei.

5.2 Das Fischverhalten

Prinzipiell besitzen alle Fischarten die gleichen Bedürfnisse mit denselben Aktionen. Durch eine unterschiedliche Parametrisierung und verschiedene Relationen entstehen individuelle Fischarten. Dadurch, dass alle Fische vom Ansatz her gleich sind, wird es leichter das „Ökosystem“ Pool um weitere Fischarten zu erweitern, da man keine Besonderheiten einzelner Fische berücksichtigen muss. Trotzdem bietet das Grundverhalten genügend Möglichkeiten, um ein interessantes, vielseitiges Verhalten der Fische zu simulieren.

Der Aufbau des Fischverhaltens ist an die Erfordernisse des **living** Systems angepasst und besitzt keinerlei Anspruch auf biologische Korrektheit. Es wurde darauf abgestimmt, einem Betrachter oder Benutzer des Systems eine

abwechslungsreiche, stimmungsvolle Unterwasserwelt zu präsentieren. Dennoch finden sich die typischen Grundbedürfnisse eines jeden Lebewesens wie Überleben und Nahrungsaufnahme im Aufbau wieder. Eine Übersicht über den Grundaufbau ist in Tabelle 2 gegeben.

	Bedürfnisse	Aktionen
	StandardNeed:	ObstacleAvoidanceAction
①	ActionBasedNeed:	ShadowEvasionAction EnemyFleeAction HideFromEnemyAction
②	LimitBasedNeed:	HuntForPreyAction WanderAction
③	ActionBasedNeed: (isAnAdditiveNeed = true)	FlockWithRelationTypeAction SeekAction
④	DefaultNeed:	WanderAction

Tabelle 2: *Verhaltensaufbau eines Fisches*

Die Aktion zur Hindernisvermeidung (`ObstacleAvoidanceAction`) ist dem Standardbedürfnis hinzugefügt. Das bedeutet, dass sie, wenn nötig, auch zusätzlich zu einer anderen Aktion ausgeführt wird. Dadurch können Hindernisse vermieden und gleichzeitig andere Aktionen ausgeführt werden.

In dem ersten Bedürfnis sind alle Aktionen untergebracht, die das Überleben eines Fisches sicherstellen. Das Bedürfnis selbst ist ein `ActionBasedNeed`. Das bedeutet, dass das Bedürfnis nur befriedigt ist, wenn keine seiner Aktionen ausführbar ist. Die oberste Priorität besitzt hierbei die Flucht vor dem Schatten des Benutzers. Dies verdeutlicht noch einmal, dass der Benutzer im Zentrum des Interesses eines jeden Fisches stehen muss. Erst danach wird den Feinden des Fisches Beachtung geschenkt. Die Aktion zum Verstecken vor Feinden steht in der Hierarchie unter der Fluchtaktion. Dadurch wird sich ein Fisch nur dann verstecken, wenn der Feind dem Fisch nicht bereits zu nahe gekommen ist.

Das zweite Bedürfnis aktiviert bei einem Fisch den Jagdtrieb. Da das Bedürfnis ein `LimitBasedNeed` ist, wäre es leicht möglich, den Trieb in Abhängigkeit der vergangenen Zeit zum letzten Beutefang zu aktivieren. Im Anwendungsbeispiel

wird er aber durch eine externe Logik des **living** Systems gesteuert, da man die Jagdzeiten eines Fisches genau kontrollieren will. Man möchte zum Beispiel nicht, dass ein Fisch mit dem Jagen beginnt, solange sich noch zu viele Benutzer auf dem Pool befinden, da sonst das Szenario zu hektisch wird. Stattdessen kann es gezielt dazu eingesetzt werden, das Fischbassin interessanter zu machen, wenn längere Zeit kein Benutzer den Pool betreten hat. Das `LimitBasedNeed` wird in diesem Fall also wie ein Schalter zum de- oder aktivieren des Bedürfnisses verwendet. Die `WanderAction` am Ende des Bedürfnisses stellt sicher, dass auch eine Aktion ausgeführt wird, wenn der Fisch keine Beute zum Jagen finden kann.



Abbildung 36: *Ein Jäger auf Beutefang*

Das dritte Bedürfnis ist wieder vom Typ `ActionBasedNeed`. Es gilt demnach als unbefriedigt, sobald eine seiner Aktionen ausführbar ist. Zudem wurde es als *additives* Bedürfnis gekennzeichnet, wodurch stets alle möglichen Aktionen des Bedürfnisses ausgeführt werden. Das Bedürfnis ermöglicht es, eine Schwarmbildung zusammen mit einem `Seek`-Verhalten anzuwenden. Diese Kombination erscheint zunächst etwas ungewöhnlich, zumal die `SeekAction` keine Vorbedingung benötigt und somit immer ausgeführt werden kann. Ein Sinn ergibt sich dann, wenn die `SeekAction` zur Steuerung des Schwarms verwendet wird. Das Ziel des `Seek`-Verhaltens wird durch das **living** System vorgegeben. So können Schwärme von einer Poolseite auf die andere navigiert werden oder sich die Fische gezielt um den Schatten eines Besuchers sammeln. Wird keine

Steuerung mehr benötigt, wird die Gewichtung der SeekAction auf Null gesetzt und die Vorbedingung der Aktion gilt automatisch als nicht erfüllt.

Das vierte und letzte Bedürfnis ist ein DefaultNeed und dadurch immer unbefriedigt. Es beinhaltet nur eine WanderAction, die, da sie keine Vorbedingung benötigt, immer ausgeführt werden kann. Ein Fisch wird also im Becken umherstreifen, sobald alle seine anderen Bedürfnisse befriedigt sind. Dadurch ist gewährleistet, dass ein Fisch immer mindestens eine Aktion ausführt.

5.3 Die Fischarten

Für die Beispielanwendung wurden drei verschiedene Fischarten erstellt. Wie bereits erwähnt, besitzen alle dasselbe, oben beschriebene Verhalten mit verschiedenen Parametereinstellungen und Relationen.

Die erste Unterscheidung der Fische wird durch Festlegung ihrer Namen getroffen. Im verwendeten Beispiel wurden die Namen Hai, Lachs und Hering vergeben. Allen Fischen mit dem Namen Hai wurde eine Beuterelation zum Lachs zugewiesen. Die Lachse und Heringe besitzen eine Feindrelation zu den Haien. Es sei darauf hingewiesen, dass die Namen nur dem Zwecke der Anschaulichkeit dienen und keine realen Beziehungen zwischen diesen Tieren widerspiegeln sollen. Auch die zur Animation verwendeten Fischmodelle entsprechen nicht dem Aussehen der Tiere.

Für die Haie wurde die Gewichtung zur Schwarmbildung auf Null gesetzt. Sie bewegen sich also als Einzelgänger durch das Becken. Da sie keine Feinde besitzen, werden sie vor anderen Fischen kein Fluchtverhalten zeigen und sich nicht verstecken. Da sie die größten Fische im Becken sind, bewegen sie sich „gemächlich“ durch das Wasser. Wird ihr Jagdtrieb aktiviert, sind sie schneller, aber nicht so wendig wie die anderen Spezies.

Bei den Lachsen wurde die Gewichtung des Versteck-Verhaltens auf Null gesetzt. Der Kohäsionsradius ihres Schwarmverhaltens ist so groß, dass er das ganze Becken umfasst. Dadurch finden sie sich stets zu einem Schwarm zusammen. Durch eine erhöhte Gewichtung des Alignmentverhaltens bilden sie strenge Schwarmformationen.

Die Heringe sind die kleinsten Fische im Pool. Sie bilden lose Schwärme, die mehr kleinen „Fischwolken“ ähneln. Dies wird durch eine niedrigere Gewichtung des Alignmentverhaltens und einen kleineren Alignment Winkel erreicht.

Weil alle Fische überschüssige Aktionen mit sich führen, ist es natürlich möglich, dass durch eine unnötige Prüfung derer Vorbedingungen etwas Rechenzeit in Anspruch genommen wird. In der Simulation hat sich deswegen aber kein Performanceproblem eingestellt. Sollte dennoch einmal eine Optimierung notwendig sein, können wahlweise die Gewichtungen der Aktionen auf Null gesetzt oder ein neues Verhaltensschema ohne überschüssige Aktionen verwendet werden. Letzteres hat den Nachteil, dass beim Hinzufügen neuer Spezies eventuell die Bedürfnisse bereits bestehender Fische angepasst werden müssten. Mit der verwendeten Methode ist es hingegen kein Problem auch noch eine vierte Spezies einzuführen, die durch eine entsprechende Relation sofort als Beutefische der Lachse fungieren könnte.

5.4 Animation der Fische

Um die Bewegungen des Fisches zu simulieren, wurden vorgefertigte Animationen verwendet. Die Geschwindigkeit der Animation wird automatisch an die aktuelle Geschwindigkeit und Beschleunigung des Fisches angepasst. Bislang bleibt es unberücksichtigt, wenn ein Fisch eine Kurve schwimmt. Dies könnte durch eine verbesserte Animation oder ein bio-mechanisches Fischmodell, wie es Tu verwendet (siehe Abschnitt 2.1.2 ab S. 20), behoben werden. Das visuelle Ergebnis ist aber hinreichend, solange die Fische nicht zu groß werden und keine Nahaufnahmen erfolgen. Dieses ist jedoch beim Einsatz des **living** Systems nicht vorgesehen.

Weitere Vorschläge zur Verbesserung der Animationen werden in Kapitel 6.1 ab Seite 78 gegeben.

Kapitel 6. Zusammenfassung und Ausblick

Dieses Kapitel beschreibt Überlegungen, wie die Funktionalität des Frameworks weiter ausgebaut werden kann und welche technologischen Neuerungen eine sinnvolle Fortführung der bisherigen Arbeit darstellen könnten.

Im Anschluss daran wird eine Zusammenfassung der gesamten Arbeit gegeben.

6.1 Erweiterungsmöglichkeiten

Die potentiellen Erweiterungsmöglichkeiten des Frameworks sind vielfältig. Allgemein kann zwischen Erweiterungen auf drei verschiedenen Ebenen unterschieden werden. Die erste sind Erweiterungen durch neue Aktionen und Bedürfnisse. Diese können durch die bisherige Funktionalität des Frameworks erzeugt werden und erweitern die Menge der simulierbaren Verhaltensweisen. Auf der zweiten und dritten Ebene befinden sich Erweiterungen, die in die Struktur des Frameworks selber eingreifen. Erweiterungen der zweiten Ebene bauen auf bereits bestehenden Funktionen auf und verbessern diese. Hingegen stellen die der dritten Ebene vollständig neue Funktionalität zur Verfügung.

Zunächst werden die Erweiterungsmöglichkeiten für alle drei Ebenen gegeben. Im Anschluss folgt eine kurze Diskussion, wie die Animationen der Charaktere verbessert werden können.

Erweiterungen der ersten Ebene

Die Erweiterungen in dieser Ebene sind stark auf den Anwendungshintergrund der Charaktere bezogen. Deshalb werden nur wenige Beispiele genannt, die Anregungen geben sollen, wofür das Framework bereits jetzt problemlos verwendet werden kann.

Ein Mutter-Kind-Verhalten, wie es bei einigen Meeressäugern, zum Beispiel Delfinen, zu beobachten ist, kann dadurch erreicht werden, dass ein Kind-Vehikel stets versucht in der Nähe eines anderen Vehikels zu bleiben, zu dem es eine Mutterbeziehung besitzt.

Eine Neugier-Aktion kann bewirken, dass alle Vehikel, zu denen noch keine Relation besteht, „untersucht“ werden. Wurde das andere Vehikel lange genug

untersucht, wird eine Ist-bekannt-Beziehung hinzugefügt. Dies würde sich eignen, um die Vehikel zum Schatten eines Benutzers des **living** Systems zu locken.

Erweiterungen der zweiten Ebene

Diese Erweiterungen setzen an den bisherigen Funktionen des Frameworks an und verbessern diese gezielt.

Beispielsweise kann für die Definition komplizierter Beziehungen und Abhängigkeiten unter den Charakteren, wie es zum Beispiel zur Simulation eines echten Ökosystems notwendig wäre, das Prinzip der Relationen erweitert werden. Anstatt jedem Vehikel einen einzelnen Namen zu geben, über den die Art der Beziehung aufgelöst wird, können Namen für verschiedene Individualisierungskategorien vergeben werden. In der Biologie werden Tiere beispielsweise durch eine Einteilung in Stamm, Klasse, Ordnung, Familie, Gattung, Art und Unterart unterschieden (siehe [7]). Natürlich sind auch einfachere Strukturen, die lediglich zwischen einem Speziesnamen und einem individuellen Namen unterscheiden, denkbar. Dadurch können Beziehungen auf einem höheren Abstraktionsniveau definiert werden.

Eine sinnvolle Strukturierungshilfe beim Zusammenstellen der Aktionen in einem Bedürfnis ist eine Meta-Aktion, der beliebig viele andere Aktionen hinzugefügt werden können. Sie ist ausführbar, sobald eine oder wahlweise auch alle ihrer Aktionen ausführbar ist. Der zurückgegebene Steuerungsvektor entspricht dem gewichteten Rückgabevektor einer oder aller Aktionen. Dieser Meta-Aktion kann der Name *combinedAction* gegeben werden, da sie mehrere Aktionen zusammenfasst. Ein sinnvolles Anwendungsszenario ist eine Aktion, die innerhalb eines Bedürfnisses nur in Kombination mit einzelnen anderen Aktionen ausgeführt werden soll. Dies gilt insbesondere dann, wenn die Aktion keine Vorbedingung zur Ausführung benötigt. Ein Beispiel ist eine *SeekAction*, die nur in Verbindung mit einer *WanderAction* eingesetzt werden soll.

Um das Verhalten eines Charakters weniger berechenbar zu gestalten, ist es sinnvoll die Aktionen in einem Bedürfnis auf Wunsch auch nach einem stochastischen Verfahren auswählen zu lassen, ähnlich wie es bereits jetzt bei einem additiven Bedürfnis (siehe S. 44) möglich ist. Dabei muss beachtet werden, dass eine Aktion immer über einen bestimmten Zeitraum ausgeführt wird, weil sonst ein optisches Zittern eines Charakters auftreten kann, wenn dieser zu schnell zwischen verschiedenen Aktionen wechselt. Eingesetzt werden kann dies

insbesondere im hierarchisch untersten Bedürfnis eines Charakters. Zum Beispiel könnte ein Fisch, wenn alle seine übrigen Bedürfnisse befriedigt sind, es sich aussuchen, ob er im gesamten Becken umherstreift oder eine bestimmte Ecke des Beckens aufsucht.

Erweiterungen der dritten Ebene

Derzeit beruht das Verhalten eines Charakters vollständig auf den Vorgaben des Benutzers. Um dem entgegenzuwirken, kann eine Verwendung der in Kapitel 2.2.4 (ab S. 32) beschriebenen und in Abschnitt 2.2.5 (ab S. 33) diskutierten genetischen Algorithmen in Erwägung gezogen werden. Als genetische Bausteine können die Aktionen und Bedürfnisse, sowie deren Parameter verwendet werden. Wie bereits in der Diskussion erwähnt wird, ist es hierfür notwendig, dass sich Kriterien spezifizieren lassen, nach denen eine genetische Auslese der Charaktere stattfinden soll. Ein mögliches Kriterium könnte es sein, nur diejenigen Charaktere überleben zu lassen, die in der Lage sind, dem Schatten des Benutzers des **living** Systems auszuweichen, ohne dabei den Pool zu verlassen. Zur Erfüllung des Kriteriums sind verschiedene Kombinationsmöglichkeiten von Bedürfnissen und Aktionen, bei gleichen Vehikeleigenschaften denkbar. Dadurch könnten verschiedenen Charaktertypen erzeugt werden.

In diesem Zusammenhang kann die automatische Kalibrierung von Parametern diskutiert werden. Die in Abschnitt 4.3 ab Seite 65 beschriebene Benutzeroberfläche verkürzt zwar die Zeit, die zur Anpassung der verschiedenen Parameter notwendig ist, dennoch wäre es hilfreich, wenn auf einige Parameter verzichtet werden kann. Mit einem genetischen Ansatz ist es ebenfalls möglich, die Höchstgeschwindigkeiten, die Kraft oder die Masse eines Charakters automatisch zu kalibrieren, indem diese als Gene von den überlebenden Charakteren weitervererbt werden. Das oben genannte Überlebenskriterium ist hierfür aber sicherlich nicht hinreichend. Es muss eine Bedingung gefunden werden, die die gewünschte Wendigkeit und Schnelligkeit eines Charakters umschreibt, damit optisch ansprechende Simulationen erzielt werden. Ein Lösungsvorschlag hierfür liegt aber noch nicht vor.

Für Vertigo Systems wird empfohlen, von einer Anpassung der Charaktere während eines Einsatzes des **living** Systems abzusehen, da das Verhalten ungewünschte Zustände annehmen könnte. Stattdessen sollten die genetischen Algorithmen als Unterstützung bei der Entwicklung von Charakteren angesehen

werden. In jedem Fall muss noch eine hinreichende Kontrolle über das Verhalten beim Benutzer verbleiben.

Den Charakteren, die mit dem Framework erstellt werden können, ist es derzeit nur möglich auf eine vorherrschende Situation zu reagieren. Sie sind nicht dazu in der Lage, ihr Verhalten zu planen oder zu reflektieren. Die Voraussetzung hierfür wäre, dass es ihnen möglich ist, Wissen zu speichern und daraus Erkenntnisse abzuleiten. Derzeit wird Wissen lediglich in einzelnen Aktionen gespeichert und ist für andere Aktionen nicht zugänglich. Eine einheitliche, allgemein zugängliche Schnittstelle zu gesammeltem Wissen oder Erkenntnissen ist notwendig, damit das Erstellen von Strategien oder das vorausschauende Planen von Handlungen möglich wird. Generell eignen sich hierfür die in Abschnitt 2.2.3 beschriebenen neuronalen Netze. Wie in Abschnitt 2.2.5 ab S. 33 aber bereits diskutiert wird, ist deren Verwendung nicht trivial. Außerdem ist mit Performanceeinbußen zu rechnen (siehe Abschnitt 2.1.4 auf S. 25). Weitere Techniken werden in [3] und [14] vorgestellt. Der Aufwand, um einen Charakter mit kognitiven Fähigkeiten auszustatten, ist für das **living** System erst dann gerechtfertigt, wenn die Charaktere als Mit- oder Gegenspieler des Benutzers fungieren sollen. In diesem Fall sind die jetzigen reaktiven Verhaltensweisen nicht mehr hinreichend.

Es sei darauf hingewiesen, dass das Verständnis des Benutzers für den Zusammenhang verschiedener Aktionen negativ beeinträchtigt werden kann, da mit einer zentralen Wissensverwaltung eine gegenseitige Beeinflussung von Aktionen möglich ist. Zudem muss im Vorfeld eine genaue Untersuchung stattfinden, wie gut das entstehende Verhalten weiterhin steuerbar bleibt.

Ein Schwachpunkt im Gesamtsystem ist leider noch die Hindernisvermeidung von OpenSteer, die in einigen Fällen, insbesondere bei schnellen aber wenig agilen Vehikeln, versagt (siehe auch Abschnitt 2.1.1 ab S. 14). Um das Problem zu behandeln ist zunächst eine Kollisionsabfrage notwendig. Zur Kollisionsbehandlung werden zwei verschiedene Lösungsansätze vorgeschlagen. Der erste Ansatz lässt es zwar zu, dass ein Hindernis von einem Vehikel durchbrochen wird, liefert aber auch nach dem Durchbrechen weiterhin einen Steuerungsvektor, der zum Verlassen des Hindernisses führen wird. Ein zweiter Ansatz wäre es, das Durchdringen eines Hindernisses von einem Vehikel ganz zu verhindern. Es bleibt die Frage, ob die Kräfte des Vehikels, die zum Durchdringen

des Hindernisses geführt hätten, in irgendeiner Form als Rückprall oder Reibungsverlust dem Vehikel wieder zurückgeführt werden sollen.

Verbesserungen der Animation

Wie in Kapitel 5.4 auf Seite 73 bereits erwähnt wird, wurden zur Darstellung der Fische Animationen verwendet, die lediglich an die Geschwindigkeit und Beschleunigung des Fisches angepasst wurden. Der realistische Eindruck ist hinreichend, solange ein Fisch ausreichend klein dargestellt wird.

Eine Verbesserung der Animation kann durch mehrere Teilanimationen für das Schwimmen von Kurven oder Auf- und Abwärtsschwimmen erreicht werden. Das Ergebnis der Gesamtanimation ist dabei von der Güte der Teilanimationen abhängig. Es ist zu beachten, dass der Gesamtaufwand mit der Zahl der Teilanimationen steigt.

Ein anderer Ansatz, um realistische Animationen zu erstellen wird in Kapitel 2.1 beschrieben. Tu verwendet ein bio-mechanisches Modell, das zusammen mit der Anwendung physikalischer Gesetzmäßigkeiten zu beeindruckenden, visuellen Ergebnissen führt. Das Modell und dessen Funktionsweise wird detailliert in [19] beschrieben. Das ab Seite 24 vorgestellte Verfahren von Demetri Terzopoulos, ermöglicht es, dass ein Charakter seinen Körper eigenständig zu beherrschen lernt. Dieses beruht auf den von Tu vorgestellten Motorcontrollern, die ein künstliches Lebewesen zur Fortbewegung verwendet. Um die Ansätze im vorgestellten Framework zu verwenden, müssen die Steuerungsvektoren, auf entsprechende Motorcontroller umgerechnet werden.

Mit einem physikalisch, bio-mechanisch motivierten Ansatz sind realistischere Ergebnisse zu erwarten. Man muss aber beachten, dass die Modelle sehr stark an physikalische Gesetzmäßigkeiten gebunden sind. Dies kann die Nutzungsmöglichkeiten der Charaktere einschränken. Der Vorteil von vorgefertigten Animationen ist, dass der Aufwand, der für ihre Erstellung notwendig ist, skalierbar ist. Der Aufwand für ein bio-mechanisches Modell bleibt stets gleich. Es muss also im Vorfeld abgestimmt werden, wie viel Aufwand insgesamt zur Animation der Charaktere verwendet werden soll, bevor eine Entscheidung über die Wahl des Verfahrens getroffen wird.

6.2 Fazit

In dieser Arbeit wurde ein Framework vorgestellt, das die Erstellung autonomer Charaktere ermöglicht. Dieses greift auf die Funktionalität des OpenSource Projektes OpenSteer zurück.

Den Kern des Frameworks bilden sogenannte Bedürfnisse und Aktionen, mit deren Hilfe sich leicht eine hierarchische Gliederung von Verhaltensweisen aufbauen lässt. Obwohl die Gliederung nach einfachen, biologischen Prinzipien funktioniert, lassen sich abwechslungsreiche und vielfältige Charaktere erzeugen, die ein interessantes Eigenleben zu besitzen scheinen. Die gegenseitige Beeinflussung verschiedener Charaktere kann durch die Definition von Relationen einfach ausgedrückt werden. Dadurch sind lebendige Simulationen möglich. Dies wurde am Beispiel einer Unterwassersimulation mit verschiedenen Fischarten bewiesen.

Das erstellte Verhalten ist überschaubar, kann schnell angepasst werden, lässt sich leicht wiederverwerten und kann auch zur Laufzeit frei parametrisiert werden. Dadurch sind kurze Entwicklungszeiten selbst für neuentwickelte Charaktere möglich. Dies macht es für die Verwendung in einem System, indem häufig neue Arten von Charakteren benötigt werden und das auf eine ständige Kontrolle der Charaktere angewiesen ist, besonders interessant. Als ein solches System wurde das **living** System in Abschnitt 1.2.2 (ab S. 5) vorgestellt.

Das Kapitel 4 beschreibt mehrere Aktionen und Bedürfnisse, mit denen bereits eine Vielzahl interessanter Charaktere erstellt werden können. Die Funktionalität ist jederzeit durch beliebige, spezialisiertere Aktionen und Bedürfnisse erweiterbar. Eine Einschränkung des Frameworks ist, dass ein Charakter derzeit nur vordefiniertes Verhalten wiedergeben kann. Die Verhaltensweisen sind im Kern reaktiv und erlauben es nicht, dass mehrere Aktionen im Voraus geplant werden. Dies macht die Charaktere für eine Verwendung als intelligente Mit- oder Gegenspieler in einem Computerspiel uninteressant. Um die Funktionalität in diese Richtung zu erweitern, wurde in Kapitel 2 ein Überblick über weiterführende Verfahren gegeben. Weitere Verbesserungsvorschläge wurden in 6.1 beschrieben.

Insgesamt kann ein positives Fazit gezogen werden. Die in Abschnitt 1.2.3 (auf S. 8) aufgeführten Anforderungen, insbesondere Hinsichtlich der Flexibilität und

Wiederverwendbarkeit der zu erstellenden Software, konnten mit dem entwickelten Verhaltensframework ausnahmslos erfüllt werden.

Anhang A: Pseudocode für einen Simulationsschritt

Der nachfolgende Pseudocode gibt einen genauen Überblick darüber, wie es zur Auswertung der Bedürfnisse und Aktionen in einem Vehikel kommt. Übergebene Parameter wurden aus Gründen der Übersichtlichkeit im Code weggelassen, können aber leicht anhand der Klassendiagramme rekonstruiert werden. Zum besseren Verständnis sollte das Klassendiagramm von Seite 43 herangezogen werden. Ebenso sind Kenntnisse in einer objektorientierten Programmiersprache hilfreich.

Der Mechanismus zur Nutzung einer Aktion in mehreren Bedürfnissen wird in dem Pseudocode zur Vereinfachung nicht erklärt.

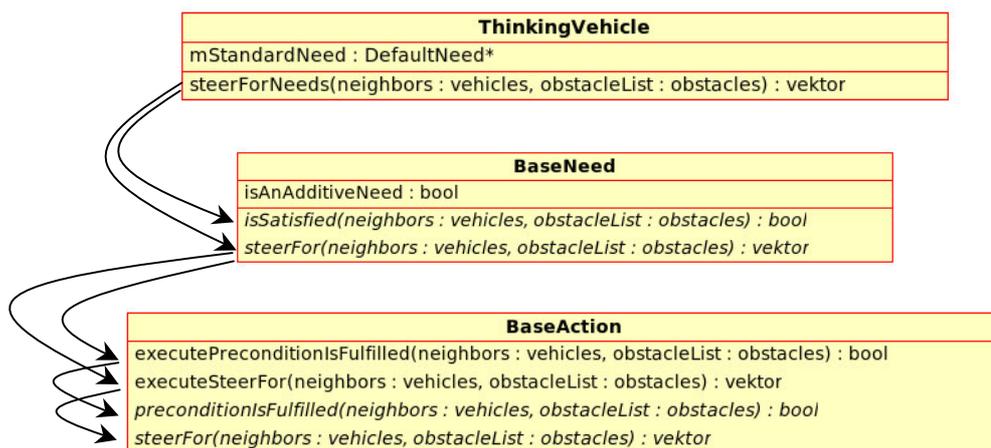


Abbildung 37: Klassendiagramm für Simulationsschritt

```

/* Methode versucht die Bedürfnisse des Vehikels zu befriedigen
void OwnVehicle::update() {

    // befriedige Bedürfnisse
    steering = steerForNeeds();

}
  
```

```

/* Führt Bedürfnisse und Standardbedürfnis aus. Liefert Steuerungsvektor
zurück, der zur Befriedigung des obersten Bedürfnisses führt. */
vector ThinkingVehicle::steerForNeeds(neighbors, obstacleList) {

    // führe das Standardbedürfnisses aus
    steering += standardNeed.steerFor();
  
```

```
//laufe der Reihe nach über alle anderen Bedürfnisse dieses Vehikels
for all needs do {

    //prüfe, ob das Bedürfnis unbefriedigt ist
    if (!need.isSatisfied()) {
        // versuche es zu befriedigen und gib das Ergebnis zurück
        return steering += need.steerFor();
    }
}

/* individuelle Methode für jedes Bedürfnis */
bool NeedN :: isSatisfied() {

    // Prüfe, ob das Bedürfnis befriedigt ist.
    return answer;
}

/* Gibt den ungewichteten Steuerungsvektor zurück, der zur Befriedigung
des Bedürfnisses führt */
vector BaseNeed :: steerFor() {

    // prüfe, ob zur Befriedigung des Bedürfnisses alle möglichen
// Aktionen ausgeführt und zusammenaddiert werden sollen.
if ( !isAnAdditiveNeed ) {          // Wenn nein....

    // ... laufe über alle Aktionen
for all actions do {

        // und beende, sobald die erste mögliche Aktion ausgeführt wurde.
if ( action.preconditionIsFulfilled() ) {
            steering = action.execute()
        }
    }
}

// Wenn es sich um kein additives Bedürfnis handelt ....
else {

    // ... laufe über alle Aktionen
for all actions do {

        // und führe, alle möglichen Aktion aus.
if ( action.preconditionIsFulfilled() ) {
            steering += action.execute()
        }
    }
return steering;
}
}
```

```
/* Prüft die Vorbedingung der eigentlichen Aktion. Verhindert eine
möglicherweise rechenaufwändige Prüfung der Aktion, wenn diese Aktion
sowieso nicht gewichtet werden soll. Gibt die Antwort der Prüfung
zurück.*/
bool BaseAction :: executePreconditionIsFulfilled () {

    //prüfe Gewichtung
    if ( weight > 0 ) {
        // führe die eigentliche Abfrage aus
        return preconditionIsFulfilled();
    }
    // Die Vorbedingung ist nicht erfüllt, wenn die Gewichtung Null oder
    geringer ist
    else
        return false;
}

/* Liefert den gewichtet Steuerungsvektor der Aktion zurück*/
vector BaseAction :: steerFor() {

    //führe die eigentliche Aktion aus und gewichte diese
    return steerFor() * weight;
}
```

Literaturverzeichnis

- [1] Bundy, Alan. *Artificial Intelligence Techniques. A Comprehensive Catalogue. Fourth Revised Edition.* Springer-Verlag Berlin Heidelberg, 1997
- [2] Feilkas, Thomas. Schnellhammer, Christian. *Steering Behaviors.* Fachhochschule Regensburg, Fachbereich Informatik, 2002
- [3] Funge, John David. *AI for Games and Animation: A Cognitive Modeling Approach.* A K Peters, Ltd, Natick, Massachusetts, 1999
- [4] Funge, John David. *Making them behave. Cognitive Models for Computer Animation.* Department of Computer Science, University of Toronto, 1998
- [5] Funge, John David. *Representing Knowledge within the situation Calculus using Interval-valued Epistemic Fluents.* Journal of Reliable Computing, 5(1), 1999
- [6] Funge, John David. Tu, Xiaoyuan. Terzopoulos, Demetri. *Cognitive Modeling: Knowledge, Reasoning and Planning for Intelligent Characters.* Aus *Computer Graphics Proceedings, Annual Conference Series: SIGGRAPH '99*, August 99
- [7] Göthel, Helmut. *Farbatlas Meeresfauna. Fische. Rotes Meer Indischer Ozean (Malediven).* Eugen Ulmer GmbH & Co., Stuttgart, 1994
- [8] Grzeszczuk, Radek. Terzopoulos, Demetri. *Automated Learning of Muscle-Actuated Locomotion Through Control Abstraction.* Aus *Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH 1995*, Seite 63-70, 1995
- [9] Helbing, Dirk. Farkas, Illes. Vicsek, Tamas. *Simulating dynamical features of escape panic.* Nature 407, S. 487-490, 2000
- [10] Hornung, Nils. *Simulation von Schwarmverhalten autonomer Objekte - Entwicklung einer Programmbibliothek.* Universität Koblenz - Landau, Fachbereich 4, 2004
- [11] Reynolds, Craig W. *Flocks, Herds, and Schools: schools: A distributed behavioral model.* Computer Graphics 21 (4), S. 25-34, 1987
- [12] Reynolds, Craig W. *Not Bumping Into Things.* In the notes for the SIGGRAPH 88 course Developments in Physically-Based Modeling, S. G1-G3. ACM SIGGRAPH, 1988
- [13] Reynolds, Craig W. *Steering Behaviors For Autonomous Characters.* Game Developers Conference 1999, 1999
- [14] Russell, Stuart. Norwig, Peter. *Artificial Intelligence. A Modern Approach.* Second Edition. Pearson Education, Inc. 2003
- [15] Terzopoulos, Demetri. *Artificial Life for Computer Graphics.* Communication of the ACM, Vol. 42, No.8, 1999

- [16] Terzopoulos, Demetri. Rabie, Tamer. Grzeszczuk, Radek. *Perception and Learning in Artificial Animals*. Artificial Life 5: Prov fifth Inter. Conf. on the Synthesis and Simulation of Living Systems, Nara, Japan, 1996
- [17] Terzopoulos, Demetri. Tu, Xiaoyuan. Grzeszczuk, Radek. *Artificial Fishes: Autonomous Locomotion, Perception, Behavior, and Learning in a Simulated Physical World*. Artificial Life, 1(4): 327-351, 1994
- [18] Treiber, Martin. Helbing, Dirk. Realistische Mikrosimulation von Straßenverkehr mit einem einfachen Modell. S. 514-520 in: D. Tavangarian and R. Grützner (eds.) 16. Symposium "Simulationstechnik" ASIM 2002, Tagungsband (Rostock), 2003
- [19] Tu, Xiaoyuan. *Artificial Animals for Computer Animation: Biomechanics, Locomotion, Perception, and Behavior*. Department of Computer Science, University of Toronto, 1996
- [20] Tu, Xiaoyuan. Terzopoulos, Demetri. *Artificial Fishes: Physics, Locomotion, Perception, Behavior*. Department of Computer Science, University of Toronto. Aus *Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH 1994*, Seite 43-50, 1994
- [21] Tu, Xiaoyuan. Terzopoulos, Demetri. *Perceptual modeling for behavioral animation of fishes*. Proc. 2nd Pacific Conf. on Computer Graphics, Beijing, China, 1994.
- [22] Wood, Oliver Edward. *Autonomous Characters in Virtual Environments: The technologies involved in artificial life and their affects of perceived intelligence and playability of computer games*. Department of Computer Science, University of Durham, 2004

Internetquellen

- [IQ 1] FEAR. <http://fear.sourceforge.net/>. Stand: Juli 2005
- [IQ 2] FEAR. Eine Beschreibung zu FEAR von Graham Mann. http://www.cs.duke.edu/courses/cps124/current/notes/13_ai/index.html. Stand: Juli 2005
- [IQ 3] Funge, John David. Eigene Homepage mit Sourcecode für CML. <http://www..cs.totonto.edu/~funge/cml>. Stand: Juli 2005
- [IQ 4] Macri, Dean. *An Introduction to Neural Networks with an Application to Games*. <http://www.intel.com/cd/ids/developer/asmo-na/eng/newsletter/20426.htm>. Stand: Juli 2005
- [IQ 5] MASSIVE. <http://www.massivesoftware.com/>. Stand: Juli 2005
- [IQ 6] MetaAgent. <http://sourceforge.net/projects/metaagent/>. Stand: Juli 2005
- [IQ 7] OpenAI. <http://openai.sourceforge.net/>. Stand Juli 2005
- [IQ 8] OpenSteer. <http://opensteer.sourceforge.net/>. Stand: Juli 2005
- [IQ 9] Vertigo Systems. <http://www.vertigo-systems.de/>. Stand: Juli 2005

Abbildungsverzeichnis

- Abbildung 1: **Beispielaufbau des living Systems** - bereitgestellt von rmh new Media
- Abbildung 2: **Das living System im Einsatz** - aus [IQ 9]
- Abbildung 3: **Seek und Flee Verhalten** - aus [13]
- Abbildung 4: **Pursuit und Evasion Verhalten** - aus [13]
- Abbildung 5: **Arrival Verhalten** - aus [13]
- Abbildung 6: **Obstacle Avoidance Verhalten** - aus [13]
- Abbildung 7: **Wander Verhalten** - aus [13]
- Abbildung 8: **Path Following Verhalten** - aus [13]
- Abbildung 9: **Unaligned collision avoidance Verhalten** - aus [13]
- Abbildung 10: **die Vehikel Nachbarschaft** - aus [13]
- Abbildung 11: **Separation Verhalten** - aus [13]
- Abbildung 12: **Cohesion Verhalten** - aus [13]
- Abbildung 13: **Alignment Verhalten** - aus [13]
- Abbildung 14: **Leader Following Verhalten** - aus [13]
- Abbildung 15: **Aufbau eines Fisches** - aus [18]
- Abbildung 16: **Wahrnehmungsbeschränkung** - aus [18]
- Abbildung 17: **Neuronales Netz** - aus [IQ 4]
- Abbildung 18: **Beispielarchitektur zur Verwendung eines OpenSteer Vehikels** - eigener Entwurf
- Abbildung 19: **Anwendungsdiagramm** - eigener Entwurf
- Abbildung 20: **Bedürfnisbaustein** - eigener Entwurf
- Abbildung 21: **Aktionsbaustein** - eigener Entwurf
- Abbildung 22: **Klassendiagramm des Gesamtsystem** - eigener Entwurf
- Abbildung 23: **Verdeckungsberechnung** - eigener Entwurf
- Abbildung 24: **Separation Nachbarschaft** - eigener Entwurf
- Abbildung 25: **Alignment Nachbarschaft** - eigener Entwurf
- Abbildung 26: **Cohesion Nachbarschaft** - eigener Entwurf
- Abbildung 27: **alle Nachbarschaften** - eigener Entwurf
- Abbildung 28: **Berechnung des Schattenabstandes** - eigener Entwurf
- Abbildung 29: **Prinzip der HideAction** - eigener Entwurf

Abbildung 30: **Anwendung der HideAction in der OpenSteerDemo** - Schnappschuss der OpenSteerDemo

Abbildung 31: **Kontrolle über die Spezies** - Schnappschuss der GUI

Abbildung 32: **Bearbeitung der Bedürfnisse und Aktionen** - Schnappschuss der GUI

Abbildung 33: **Relationen eines Hais** - Schnappschuss der GUI

Abbildung 34: **Relationen eines Lachses** - Schnappschuss der GUI

Abbildung 35: **Ein Schatten (Benutzer) verscheucht Fische** - Schnappschuss aus dem living System

Abbildung 36: **Ein Jäger auf Beutefang** - Schnappschuss aus dem living System

Abbildung 37: **Klassendiagramm für Simulationsschritt** - eigener Entwurf

Abkürzungsverzeichnis

- AI = Artificial Intelligence (englische Abkürzung für KI)
- CML = Cognitive Modeling Language (Beschreibung auf S. 25)
- CVS = Current Version System (Beschreibung auf S. 36)
- EA = endlicher Automat (Beschreibung ab S. 29)
- GUI = Graphical User Interface (deutsch: Graphische Benutzeroberfläche)
- HEA = hierarchischer, endlicher Automat (Beschreibung ab S. 29)
- KI = Künstliche Intelligenz
- VR = Virtual Reality