Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Andreas Pongs

Lebensechte Simulation von Zierfischen

Artifishial Intelligence

Zeitabhängige Medien - Games
Fakultät Design, Medien und Information
Department Medientechnik

eingereicht am
Fachhochschul-Masterstudiengang

ZEITABHÄNGIGE MEDIEN / GAMES

Hochschule für Angewandte Wissenschaften Hamburg

im Januar 2013

http://www.gamesmaster-hamburg.de

Inhaltsverzeichnis

Teil I

Einführung

Kapitel 1

Einleitung

Die Idee des künstlichen Lebens beschäftigt die Menschheit schon seit der Antike. So finden sich bereits in der griechischen Mythologie zahlreiche künstliche Kreaturen, die den Menschen und Halbgöttern zur Seite stehen. Die technische Realisierung solcher künstlicher Kreaturen stellte seit jeher eine besondere Herausforderung für Künstler und Wissenschaftler dar. Dank diverser technologischer Durchbrüche entstanden bereits im 17. und 18. Jahrhundert zahlreiche mechanische Automaten und Mensch-Maschinen. Mit Erfindung des Digitalcomputers bot sich ab der Mitte des 20. Jahrhunderts die Möglichkeit, Verhaltensweisen natürlicher Lebewesen per Algorithmus nachzubilden und grafisch darzustellen. Dieser als „Artificial Life“ bezeichnete Teilbereich im Feld der computerbasierten künstlichen Intelligenz findet seine Anwendung heute beispielsweise im Bereich der Grafik-Animation.
Die vorliegende Arbeit entstand im Rahmen des Studienprojektes “Realistic Aquarium”, einem Aquarium-Simulator mit Social-Game Elementen, bei dem der Fokus auf dem kreativen Einrichten des Aquariums sowie der möglichst lebensechten Darstellung der Fische liegt. Um dabei nach dem Vorbild der japanischen Tamagotchi-Spielzeuge auch ein Gefühl der Verantwortung und Fürsorge beim Spieler zu erwecken, ist es dabei wichtig, auch die Grundbedürfnisse der Fische zu simulieren, inklusive Hunger und Schlafbedürfnis. Das Betrachten der Fische sollte interessant sein, ihr Verhalten stets nachvollziehbar und nach Möglichkeit so abwechslungsreich wie in der Realität. Aktuelle Aquarienspiele bilden das Verhalten der Fische nur sehr rudimentär ab. In der Regel werden als notwendige Grundlage für die Spielmechanik nur das Hungergefühl und die Fortpflanzung simuliert. Alle mir bekannten Titel verzichten bereits auf die Implementierung von Ruhephasen oder einen simulierten Tag/Nacht-Wechsel. Zum Zeitpunkt der Fertigstellung dieser Arbeit existierte außerdem noch kein einziges kommerzielles Aquarienspiel für Webbrowser oder mobile Endgeräte, dessen Spielumgebung komplett in 3D gehalten wurde. Lediglich die Fische werden mittlerweile zur besseren Darstellung als 3D-Objekte gerendert, die sich aber immer vor einem 2D-Hintergrund bewegen. Realistic Aquarium wird dagegen komplett in 3D gehalten werden, da sich zum einen das kreative Einrichten des Aquariums unter 2D sehr unbefriedigend anfühlt. 3D eröffnet hier neue gestalterische Möglichkeiten, da alle Objekte nun rotiert und sogar physikalisch korrekt gestapelt werden können und dank Shader-Beleuchtung auch realistischer aussehen. Zum anderen wurde mir von Aquarienbesitzern immer wieder bestätigt,

dass ein großer Anreiz der Aquaristik darin besteht, ein Stück Natur zu Hause zu erschaffen und zu pflegen, und es den Fischen darin so gemütlich wie möglich zu machen. Um diesen Reiz wenigstens ansatzweise übertragen zu können, müssen unsere virtuellen Fische spürbar auf die Einrichtung des Aquariums reagieren, indem sie sich beispielsweise in geeigneten Pflanzenbewuchs oder unter Steine zurückziehen. Auch wenn noch nicht alle dieser Verhaltensweisen innerhalb dieser Arbeit umgesetzt werden können, so kommen wir in den folgenden Kapiteln um echtes 3D als Grundlage für die zu untersuchenden Techniken nicht herum.

Um die genannten Ziele erreichen zu können, betrachten wir im ersten Teil dieser Arbeit vor allem die Bewegungsabläufe, mit denen sich Fische in der Realität durch ihre Umgebung bewegen, sowie ihre typischen Verhaltensweisen. Außerdem stelle ich die drei wichtigsten Projekte der letzten 20 Jahre vor, deren Ziel die möglichst lebensechte Simulation von Fischen war. Im zweiten Teil gehe ich auf die konkreten Techniken ein, die heute in Simulationen und kommerziellen Spielen bei diesen und ähnlichen Anforderungen zum Einsatz kommen. Diese Anforderungen teilen sich grob in die Bereiche Animation, Navigation und Aktionswahl auf. Jedem dieser Teilbereiche widme ich ein eigenes Kapitel, an dessen Ende ich die jeweils vorgestellten Techniken im Hinblick auf die Anforderungen meines Projekts evaluiere. Dabei spielt auch die Performanz eine besondere Rolle, da zu den Zielplattformen von Realistic Aquarium auch mobile Endgeräte mit vergleichsweise schwachen Prozessoren gehören. Dementsprechend gilt es, in jedem Teilbereich den besten Kompromiss zwischen Realitätsnähe und Performanz zu finden.

Kapitel 2

Fischkunde

Der erste Schritt zur Nachbildung der Fische ist die Untersuchung möglichst aller ihrer natürlichen Eigenschaften, die sich unmittelbar auf ihre visuelle Darstellung auswirken. Dazu gehört in erster Linie die Technik, mit der sich Fische durch ihre Umgebung bewegen. Durch eine gute Nachbildung dieser Muskelkontraktionen erreichen wir einen glaubhaften Eindruck beim ersten flüchtigen Betrachten. Um beim Spieler auch einen Willen zur Fürsorge auszulösen, muss der Eindruck von Leben auch längerfristig aufrechterhalten werden. Dazu ist es notwendig, dass wir auch die wichtigsten Verhaltensweisen und Instinkte der Fische nachmodellieren. Diese werden in Sektion 2.2 vorgestellt.

2.1 Fischlokomotion

Als Lokomotion bezeichnet man allgemein die Fähigkeit eines Lebewesens, sich physikalisch durch seine Umgebung fortzubewegen. Die Kompressions-Eigenschaften und die hohe Dichte des Wassers als Medium spielten dabei in der Evolution der Fische eine wichtige Rolle (vgl. [Lin78]). Die Wasserdichte beträgt ungefähr das 800fache der Luftdichte und ist nahe genug an der Körperdichte der Fische, um die Gravitation fast vollständig auszugleichen. Durch die relativ geringe Rolle, die demzufolge der Ausgleich der Schwerkraft innerhalb der Fischlokomotion spielt, konnte die Evolution eine erstaunliche Vielfalt an verschiedenen Antriebsmöglichkeiten entwickeln. Im Folgenden betrachten wir die grundlegenden Varianten.

2.1.1 Periodisches Schwimmen und Übergangsbewegungen

Alle Bewegungen eines Fisches können laut [WW83] grundsätzlich in zwei verschiedene Kategorien eingeteilt werden:

1. Periodisches Schwimmen
   Diese Bewegungen zeichnen sich durch ihre zyklische Wiederholung aus und dienen dazu, große Distanzen mit einer relativ konstanten Geschwindigkeit zurückzulegen.
2. Übergangsbewegungen
   Sie beinhalten ruckartiges Beschleunigen, Ausweichmanöver und enge Drehungen.

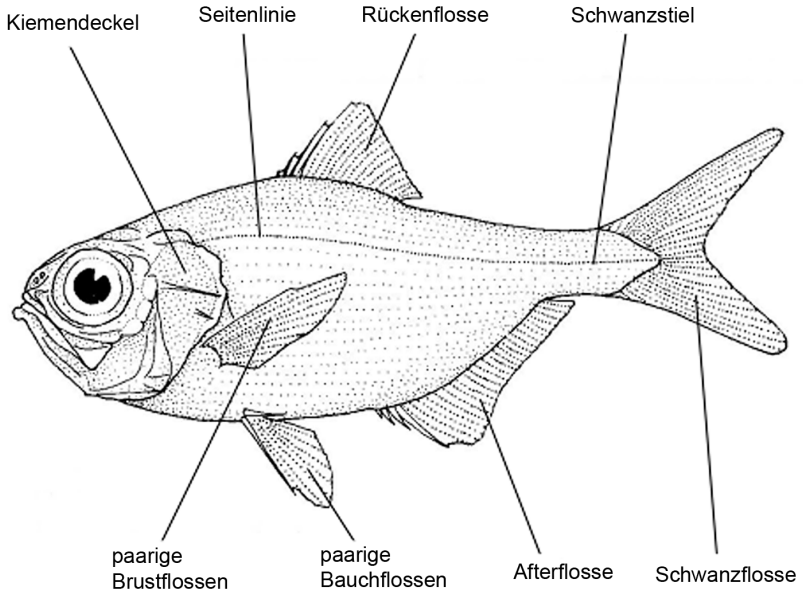


Abbildung 2.1: Fisch-Anatomie (adaptiert von wikimedia.org)

Das Periodische Schwimmen ist wissenschaftlich am besten untersucht, da es sich im Gegensatz zu den Übergangsbewegungen in Versuchsaufbauten gut wiederholen, analysieren und überprüfen lässt (vgl. [MS99]). In unserem Projekt ist dem Periodischen Schwimmen große Aufmerksamkeit zu widmen, da sich die Fische die meiste Zeit über in dieser Fortbewegungsvariante bewegen werden und unrealistische Animationen bei langsamen und regelmäßigen Bewegungen besonders störend auffallen.

2.1.2 Schwimmvarianten

Die Fortbewegungsart eines Fisches kann weiterhin laut [M.B26] jeweils einer von zwei Kategorien zugeordnet werden, die im Folgenden vorgestellt werden.

2.1.2.1 BCF-Lokomotion

Die meisten Fische generieren Vorwärtsantrieb, indem sie Teile ihres Körpers zu einer Welle biegen und diese Welle über die restliche Körperlänge hin bis über die Schwanzflosse hinweg auswandern lassen. Diese Technik wird BCF-Lokomotion genannt [J.G32] („body and/or caudal fin“). Abbildung 2.2 zeigt die BCF-Lokomotion eines Aals in elf Einzelbildern. Die positive und negative Amplitude der Wellenbewegung ist dabei auf jedem Bild markiert, sodass sich das rückwärtige Auswandern der Welle gut nachverfolgen lässt.

Physikalisch entsteht der Vorwärtsantrieb bei der BCF-Lokomotion durch die Gegenkräfte, die durch das vertikale Auswandern der Welle entstehen (siehe Abbildung2.3). Jeder äußere Punkt n_i des Fisches drückt dabei gegen die ihn umgebenden Wassermoleküle. Dadurch wirkt auf den Fischkörper an dieser

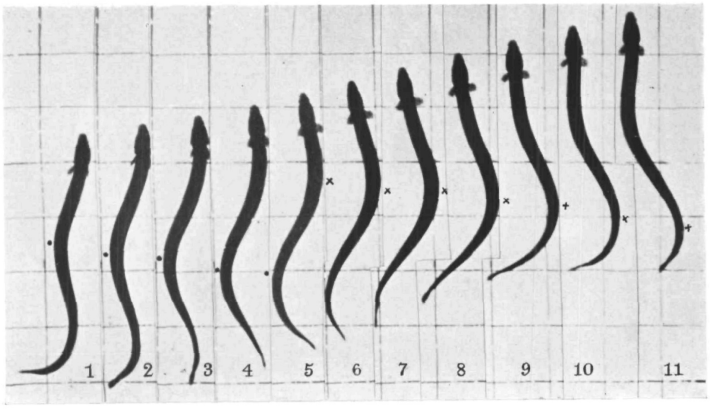


Abbildung 2.2: BCF-Lokomotion eines Aales in Einzelbildern (Quelle: [J.G32])

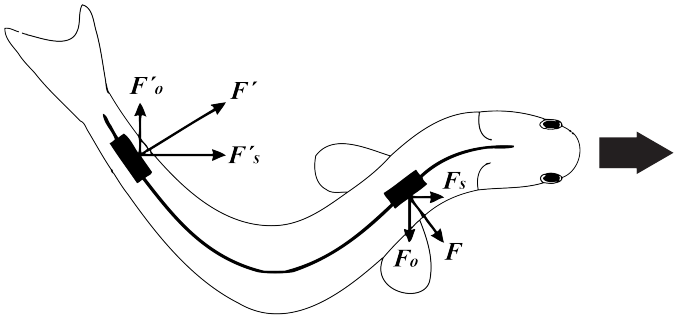


Abbildung 2.3: BCF-Lokomotion: Generierung der Schubkraft (adaptiert von [MS99])

Position eine Gegenkraft F , die in eine vorwärts gerichtete Komponente F_s und eine orthogonale Komponente F_o aufgeteilt werden kann. Die Komponente F_s ist die Schubkraft, die den Fisch gemäß seiner Körper-Ausrichtung nach vorne schiebt. Die seitlichen Komponenten wirken über die gesamte Fischlänge gesehen in entgegengesetzte Richtungen, sodass sie sich über die gesamte Länge hinweg ausgleichen.

Je nach Fischgattung unterscheidet sich die Länge des Körperbereiches, der von der Wellenbewegung durchlaufen wird. Während sie beispielsweise beim Aal mit konstanter Amplitude den kompletten Körper durchläuft, so ist bei anderen Fischen nur die hintere Körperhälfte beteiligt. Je kürzer der an der Lokomotion beteiligte Bereich ausfällt, desto stärker geht die ursprüngliche wellenförmige Bewegung (*'undolatorische Lokomotion'*) in ein periodisches Schwingen über (*'Oszillierende Lokomotion'*). Einige wenige Fischarten erzeugen den Antrieb ausschließlich durch periodisches Schlagen mit der Schwanzflosse.

2.1.2.2 MPF-Lokomotion

Einige Fischarten haben alternative Schwimmtechniken entwickelt, die auf der Bewegung ihrer Medianflossen (Rückenflosse, Afterflosse) und paarigen Flossen

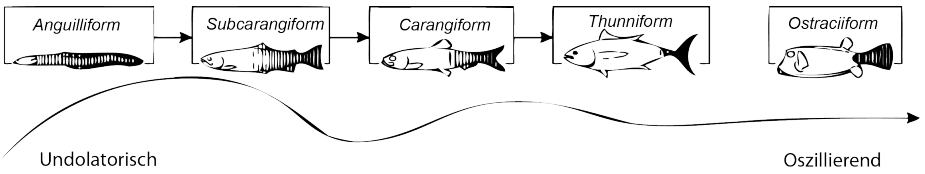


Abbildung 2.4: Übersicht: Varianten der BCF-Lokomotion geordnet nach dem Anteil der beteiligten Körperlänge (adaptiert von [Lin78])

(Brustflossen, Bauchflossen) basieren. Diese Variante wird als MPF-Lokomotion („Median/Paired Fin“) bezeichnet. MPF-Lokomotion wird nur von etwa 15% aller Fische eingesetzt, die dank ihr auf das präzise, aber langsame Schwimmen in ruhigen Gewässern oder Korallenriffen spezialisiert sind.

Die MPF-Lokomotion existiert in verschiedenen Varianten, wobei der Bewegungsablauf und die Art und Anzahl der eingesetzten Flossen je nach Fischgattung variiert (siehe Abbildung 2.4). Die Antriebskraft wird jeweils durch periodische Bewegungen der beteiligten Flossen erzeugt. Ähnlich zur BCF-Lokomotion kann eine beteiligte Flosse dabei über ihre gesamte Länge von einer wellenförmigen Bewegung durchlaufen werden, wie etwa beim Rochen oder Messerfisch [GVL96]. Bei anderen Fischarten geht die Bewegung in ein periodisches Rudern oder Flattern über, bei dem der Wasserwiderstand der Flossen in der Rückwärtsbewegung gering gehalten wird, beispielsweise durch Kippen oder Drehen der Flossenmembran (siehe Abbildung 2.5).

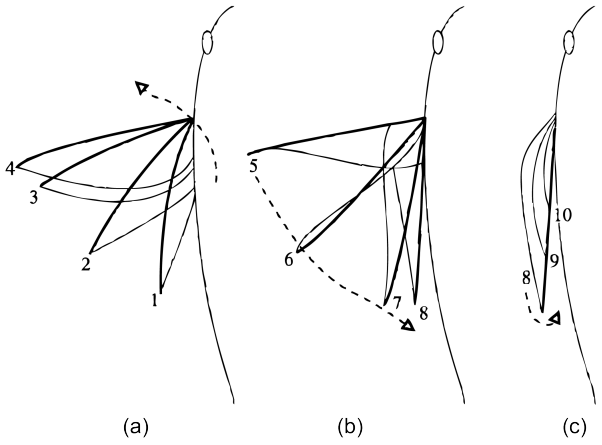


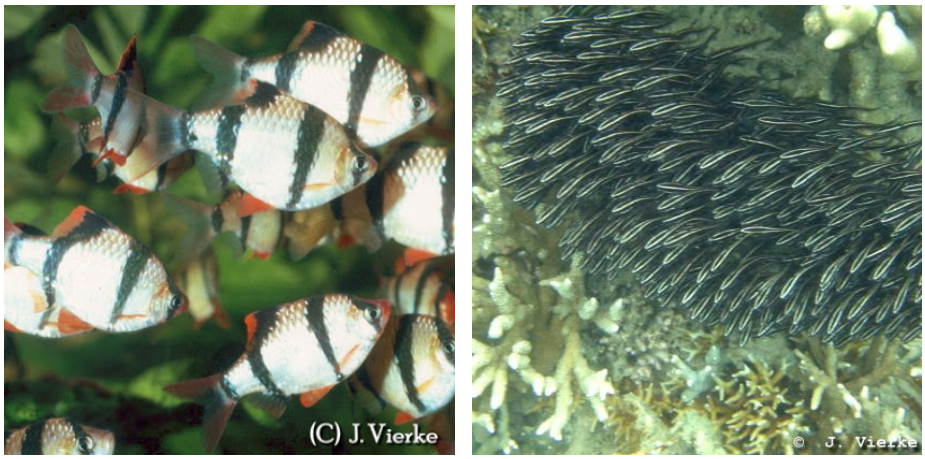
Abbildung 2.5: Rückenansicht der Flossenbewegungen während der MPF-Lokomotion zum Zeitpunkt des Abspreizens (a), des Anziehens (b), sowie am Umkehrpunkt der Bewegung (c). (Adaptiert von [Web73])

2.2 Verhaltensweisen der Fische

In unserem Projekt sollte die Simulation der Fische nicht nur auf den ersten Blick lebensecht wirken, sondern auch möglichst viele ihrer typischen Verhal-

tensweisen beinhalten. Diese Verhaltensweisen unterscheiden sich zum Teil sehr stark von Spezies zu Spezies und sind ausführlich und teils sehr spezialisiert in zahlreichen Büchern und Veröffentlichungen dokumentiert [HG06] [Vie94]. Aufgrund des zeitlichen Rahmens muss ich mich zunächst auf die Simulation der wichtigsten und typischen Verhaltensweisen beschränken. Falls genug Interesse an dem Projekt besteht, hoffe ich diese Auswahl später erweitern und verfeinern zu können.

Während der Entwicklung des Projekts habe ich einige Zeit in Zoohandlungen, öffentlichen Aquarien und auf YouTube-Kanälen verbracht, um ein Gefühl für die auffälligsten und häufigsten dieser Verhaltensweisen zu bekommen. Die folgende Liste stellt speziesübergreifend das Minimum der zu implementierenden Handlungsalternativen dar, um ein ausreichend realistisches und abwechslungsreiches Gesamtbild zu erreichen. Ein späteres Erweitern und Verfeinern der Verhaltensweisen sollte später möglich sein, ohne tiefgreifende Änderungen an der KI-Architektur vornehmen zu müssen.



(a) Sumatrabarben (b) junge Korallenwelse

Abbildung 2.6: Schwarmverhalten (Quelle: [Vie94])

Schwarmverhalten Eine weit verbreitete Überlebensstrategie vieler Fischgattungen ist das Bilden von Schwärmen und kleinen Gruppen. Schwarmfische bleiben dabei in möglichst großen Gruppen dicht zusammen und bewegen sich weitgehend parallel zueinander. So ist das Individuum in der Gruppe besser geschützt, da es Angreifer wesentlich schwerer haben, sich auf einzelne Beutefische

Abbildung 2.7: südamerikanischer Messerfisch beim Schlafen im Pflanzendickicht (Quelle: [Vie94])

zu konzentrieren.

- Schwarmfische bleiben je nach Spezies unterschiedlich dicht zusammen, bewegen sich in die gleiche Richtung und richten sich oft auch in Ruhephasen in eine gemeinsame Richtung aus (siehe Abbildung 2.6)
- Außerhalb seiner Aktivzeiten zieht sich der Fisch an einen geeigneten Ruheplatz (z.B. Grund, Pflanzendickicht) zurück und verbringt dort mehrere Stunden (siehe Abbildung 2.7)
- Ausgeruhte Fische durchstreifen das Aquarium auf der Suche nach Nahrung und Abwechslung.
- Fische legen auch außerhalb ihrer Ruhephasen längere Pausen ein, an denen sie sich nicht bewegen.
- Hungrige Fische steuern Nahrung sofort an. Je hungriger der Fisch ist, desto schneller und aggressiver wird er sich dem Futter nähern und desto weitere Wege wird er in Kauf nehmen.
- Kommen sich zwei Rivalen zu nahe, verscheucht der stärkere den schwächeren mit einem kurzen angedeuteten Angriff. Gelegentlich kommt es auch zu längeren Verfolgungsjagden, die sich über mehrere Sekunden hinziehen können.
- Müde und schwache Fische versuchen, Auseinandersetzungen zu vermeiden.
- Ein ausreichend hungriger Fisch, der einen Rivalen jagt, wird von diesem ablassen, sobald er Futter entdeckt.
- Gerät ein Fisch in Panik, so ergreift er die Flucht, unabhängig von der gerade ausgeführten Handlung. Je größer die Panik bzw. Bedrohung, desto höher die Fluchtgeschwindigkeit.

2.2.1 Emotionen

Die Liste der zu simulierenden Verhaltensweisen in der letzten Sektion gibt vor, dass wir uns zusätzlich mit den wichtigsten Empfindungen der Fische als Motivation für ihre Verhaltensweisen auseinander setzen müssen.

Hunger Das Hungergefühl muss simuliert werden, um entscheiden zu können, ob der Fisch ein Nahrungsobjekt ansteuert oder ignoriert. Der Grad des Hungers bestimmt dabei, welche Wege und Risiken der Fisch in Kauf nehmen wird, wie z.B. das Verlassen des Schwarms.

Stress Wie alle Wirbeltiere reagieren Fische mit der Ausschüttung von Stresshormonen auf äußere Reize, die als Gefahr interpretiert werden. Der Stresspegel steigt dabei proportional zum empfundenen Gefährlichkeitsgrad der aktuellen Situation. Fische mit hohem Stresspegel reagieren schreckhafter und ergreifen bei einem ausreichend starken Reiz die Flucht. Außerhalb der Gefahrenzone sinkt der Pegel.

Müdigkeit / Erschöpfung Jede Fischgattung unterliegt festen Aktiv- und Ruhezeiten, zu denen sich der Fisch an einen geeigneten Ruheplatz zurückzieht (vgl. [Vie94]). Je nach Fischgattung kann dies z.B. ein schattiger Platz, der sandige Grund des Aquariums, Pflanzendickicht oder eine Felsspalte sein. Manche Zierfische wie z.B. Dornaugen sind ausschließlich nachtaktiv. Auch während der Aktivphase legt der Fisch Ruhephasen ein, zu denen er sich wenig bis gar nicht bewegt.

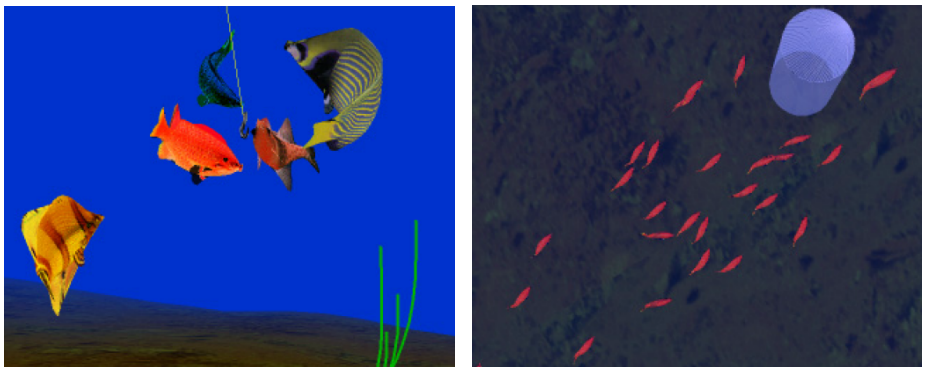
Kapitel 3

Verwandte Arbeiten

Zur Simulation von Fischen entstanden seit den 1990er Jahren mehrere größere Projekte, die ich im Folgenden vorstelle.

3.1 Artificial Fishes (1994)

Das im Jahr 1994 von Xiaoyuan Tu und Demetri Terzopoulos vorgestellte Framework *Artificial Fishes* stellt den bis heute bei weitem ambitioniertesten Versuch dar, Fische möglichst naturgetreu per Computer nachzubilden [TT94]. Dabei wurde nicht nur das Verhalten der Fische, sondern insbesondere auch ihre Fortbewegung so wirklichkeitsnah nachempfunden, dass die berechneten Animationen trotz veralteter und detailarmer Polygongrafik auch heute noch sehr beeindruckend wirken. Erreicht wurde dies durch ein aufwendiges Nachbilden der Muskelbewegungen und der hydrodynamischen Kräfteverhältnisse auf Basis eines physikalischen Modells. Der elastische Fischkörper wurde als Feder-Masse-System nachmodelliert, bei dem sämtliche benachbarten Eckpunkte des Drahtgittermodells mit virtuellen Federn verbunden waren, wie in Abbildung 3.2 zu sehen ist. Bestimmten Verbindungen konnten dabei Sequenzen von Parameteränderung übergeben werden, wodurch sich kontinuierliche Bewegungsabläufe der Muskeln simulieren ließen. Die dadurch entstehende Übertragung der Kräfte auf die umgebenden Wassermoleküle wurde ebenso simuliert wie der Energieverbrauch der Muskeln. Auf Basis dieser Daten konnten Tu und Terzopoulos nun berechnen, welche Muskelbewegungen den Fisch dabei am effektivsten, d.h. mit dem geringsten Energieverbrauch durch das Wasser gleiten ließen. Dies wurde durch einen genetischen Algorithmus erreicht, bei dem der Fisch – wie im echten Leben nach seiner Geburt – sich zunächst versuchsweise mit willkürlichem und wenig effektivem Zappeln bewegt und seine Bewegungen dann zufallsbasiert mehr oder weniger stark variiert, um schließlich am Ende der Lernphase eine maximal effiziente Abfolge beizubehalten. Ein Fisch-Modell bestand damals aus 23 Masse-Punkten, die mit 91 Federn verbunden waren, von denen 12 als Muskeln kontrahierbar waren. Die daraus resultierenden Kräfteverhältnisse wurden über die Lagrang'schen Bewegungsgleichungen definiert und mit einer numerisch stabilen impliziten Euler-Methode implementiert. Auf einem damaligen High-End-System wurden mit 10 Fischen und 5 Hindernissen eine Framerate von etwa 4 Bildern pro Sekunde erreicht.



(a) flexible Fischkörper in Bewegung (b) Schwarmverhalten

Abbildung 3.1: 'Artificial Fishes' von Tu und Terzopulos

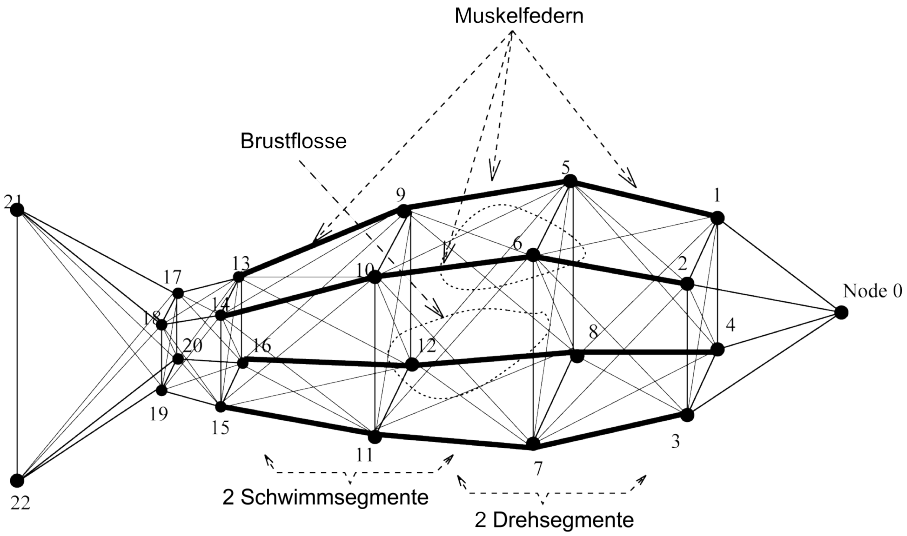


Abbildung 3.2: Terzopoulos: Masse/Feder-System zur Simulation der Muskeln (adaptiert von [TT94])

3.2 Das Virtuelle Ozeanarium (1994-1996)

Das Virtuelle Ozeanarium wurde von 1996 bis 1998 am Fraunhofer Institut für grafische Datenverarbeitung für die Weltausstellung in Lissabon, EXPO'98 entwickelt [Kla97] [Mül97] [Frö97] [Frö00]. Koordiniert wurde es vom Centro de Computacao Graphica in Coimbra, Portugal. Aufgabe war es, das Ozeanarium als Hauptattraktion der EXPO in Form einer virtuellen Umgebung darzustellen. Diese wurde in einem eigenen Auditorium auf der EXPO'98 einem großen Publikum präsentiert. Ein wichtiges Ziel war es dabei, das reale Ozeanarium so exakt wie möglich darzustellen, inclusive des marinen Lebens in seinen fünf großen Salzwasser-Aquarien. Im Gegensatz zu den *Artificial Fishes* von Tu und Terzopoulos lag der Fokus dieses Projekts klar auf der großen Anzahl der darzustellenden Fische und nicht auf der naturgetreuen Nachbildung natürlicher Vorgänge. Die Fische bewegten sich nicht anhand von reaktiver Navigation (siehe Kapitel 5), sondern anhand von Kurvenbahnen, die für längere Strecken vorberechnet wurden. Kam es dabei zu Kollisionen mit dem Grund oder anderen Objekten, wurden die Kurvenkoeffizienten nach dem Prinzip von Versuch-und-Irrtum verändert. Die Animation der Fische basierte auf manuell erstellten Keyframe-Animationen, die passend zur Fortbewegungsgeschwindigkeit des virtuellen Fisches mit variabler Framerate abgespielt wurden.

Abbildung 3.3: Blick in das Hauptbecken des virtuellen Ozeanariums

3.3 Dream Aquarium (2005-2010)

Der unter der Domain *dreamaquarium.com* kommerziell vertriebene Bildschirmschoner von Alan Kapler bietet bis heute eine der überzeugendsten grafischen Animationen virtueller Fische, die heutzutage auf einem handelsüblichen PC in Echtzeit berechnet werden können. Neben der äußerst detailgetreuen Animation einzelner Elemente wie der Augen und der Flossen beeindruckt hier vor allem das Biegen des kompletten Fischkörpers in schnellen Kurvenbewegungen. Dabei wird in jedem Frame die physikalisch korrekte Position aller Fisch-Segmente berücksichtigt, sodass der Fisch sich wirklich durch das Wasser zu winden scheint.

Dream Aquarium läuft auf heutigen Mittelklasse-PCs bei einer Darstellung von ca 20 Fischen mit einer Framerate von weit über 100 Bildern pro Sekunde. Dank seiner außerordentlich hohen grafischen Qualität stellt es somit unter allen bekannten Ansätzen den mit Abstand besten Kompromiss zwischen Realismus und Effektivität dar. Der Code zu *DreamAqarium* ist weder öffentlich, noch wurde der verwendete Algorithmus anderweitig dokumentiert. Persönliche Anfragen meinerseits an Alan Kapler ergaben, dass auch er lange Zeit mit physikalischen Modellen experimentiert hatte, jedoch mit mäßigem Erfolg. Seine jetzige Lösung basiere stattdessen aus einer Kombination aus Keyframe-Animation und prozeduraler Animation. Ein Teilziel meiner Arbeit bestand deshalb darin, Kaplers Algorithmus möglichst gut nachzuempfinden, bzw. einen ähnlichen gut funktionierenden zu implementieren. Das Ergebnis stelle ich in Kapitel 4.4 vor.

Abbildung 3.4: Dream Aquarium von Alan Kapler

Teil II

Konzeptionierung und Implementierung

Nachdem wir uns in den vorangegangenen Kapiteln mit den Lokomotionsvarianten und Verhaltensweisen echter Fische beschäftigt haben, stellt sich nun die Frage, in welche Komponenten sich unsere Künstliche Intelligenz aufteilen lässt, um die verschiedenen Anforderungen möglichst modular und redundanzfrei abbilden zu können. Wie Mat Buckland in [Buc05] darlegt, lässt sich die Bewegung von autonomen virtuellen Lebewesen oder Robotern - im KI-Jargon auch als *autonome Agenten* bezeichnet - in die drei folgenden Ebenen unterteilen:

1. **Aktionswahl:** Dieser Teil der KI ist dafür verantwortlich, die Ziele des autonomen Agenten festzulegen und welcher Plan dabei ausgeführt wird.

2. **Steuerung / Navigation:** In dieser Ebene wird der Bewegungsverlauf des Agenten berechnet, um die Ziele erreichen zu können, die durch die übergeordnete Ebene definiert wurden.

3. **Lokomotion:** Diese unterste Ebene beschreibt den mechanischen Teil zur Bewegung des Agenten. Während der übergeordente Steuerungsteil zu jedem Zeitpunkt die gewünschte Bewegungsrichtung vorgibt, beschäftigt sich der Lokomotionsteil also eher damit, wie genau die technischen Antriebsmöglichkeiten des Agenten eingesetzt werden müssen, um die gewünschte Richtung zu erreichen bzw beizubehalten.

Der Aufbau meiner Fisch-KI und der folgenden Kapitel orientiert sich an diesen drei Ebenen, denen ich jeweils ein eigenes Kapitel widme. Wir beginnen mit der Lokomotion bzw. Animation der Fische, zumal diese unterste Ebene wichtig für den ersten optischen Eindruck ist und sich in unserem Fall weitestgehend unabhängig von den übergeordneten Ebenen konzipieren und implementieren lässt.

Kapitel 4

Animation der Fische

Von allen Teilaspekten der Fisch-Simulation ist die Animation der Fische derjenige, dessen Qualität für jeden Betrachter auf den ersten Blick ersichtlich ist. Dieser Bereich verdient deshalb besonders viel Sorgfalt und Aufwand, um potentielle Spieler nicht schon in den ersten Sekunden zu vergraulen. Die Arbeiten von Tu und Terzopoulos [TT94] haben hier gezeigt, dass durch eine realistische Animation auch bei einfachster grafischer Darstellung sofort ein sehr lebensechter Eindruck entsteht. Umkehrt zerstört eine schlechte Animation sofort die realistische Wirkung, die detaillierte Modelle und aufwendige Shader-Effekte in Standbildern erzielen können. Hier hat das im letzten Kapitel beschriebene *Dream Aquarium* gezeigt, dass vor allem auf ein passendes Biegen der Fischgeometrie in Kurvenbewegungen geachtet werden muss. Durch falsche Drehpunkte oder zu steife Geometrie kann beim Betrachter das Gefühl von Masse und Trägheit sehr schnell zerstört werden. In dieser Hinsicht ist zu erwarten, dass die realitätsnahe Darstellung der BCF-Lokomotion relativ aufwendig wird, da diese Variante eine geometrische Veränderung des gesamten Fischkörpers mit sich bringt. Solange der Fisch geradeaus schwimmt, stellt diese Animation kein Problem dar, aber sobald der Fisch sich dreht oder enge Kurven schwimmt, muss die Wellenanimation an das Biegeverhalten des Körpers angepasst werden.

4.1 Bone-Animation

Im Bereich der Computerspiele hat sich zur Animation von 3D-Modellen spätestens seit der Veröffentlichung des Spiels *Half-Life* (Valve Corporation) im Jahr 1999 die Technik der Bone-Animation (oder auch *Skeletal-Animation*) durchgesetzt. Mit Hilfe dieser Technik lassen sich beliebig viele Knoten eines Polygonmodells animieren, ohne dabei jeden Knoten einzeln ansprechen zu müssen. Stattdessen wird, wie bei einem biologischen Skelett, zunächst eine Struktur von Knochen-Objekten in das Polygonmodell eingefügt. Jedem Knochen dieses Skeletts lässt sich nun innerhalb eines 3D-Editors über diverse Werkzeuge diejenige Menge an Knoten im Drahtgittermodell zuordnen, die durch diesen Knochen bewegt werden sollen. Dabei sind auch prozentuale Gewichtungen pro Knochen und Knoten möglich, was bei Bewegungen einzelner Knochen weiche Übergänge statt harter Kanten entstehen lässt. Der gesamte Prozess der Skelett-Erstellung und Gewichtung der Polygonknoten wird als *Rigging* bezeichet, das Skelett auch

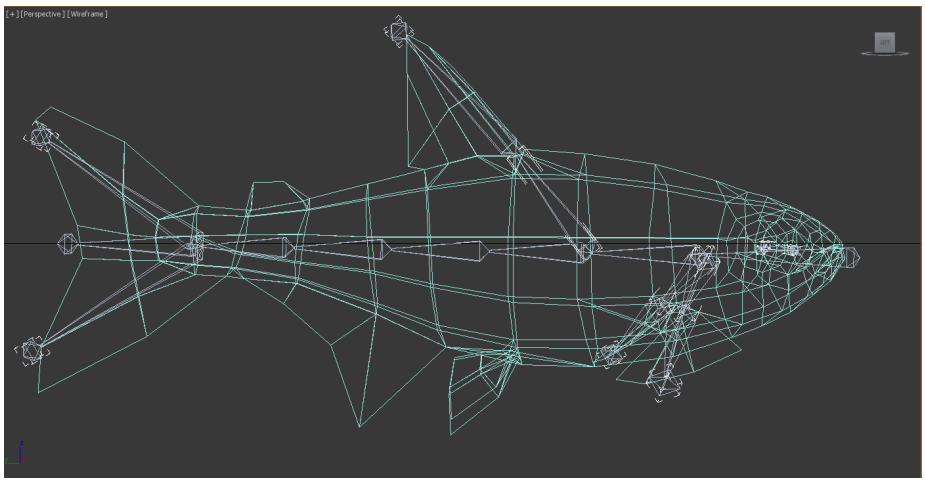


Abbildung 4.1: Bone-Struktur und Polygonmodell eines Neon-Tetras als Basis für die Bone-Animation

als *Rig*. Bevor wir uns mit der Animationen der Fische beschäftigen, brauchen wir zunächst ein entsprechendes 3D-Modell inklusive Rig. Unabdingbar für eine fließende Nachbildung der BCF-Lokomotion wird dabei vor allem eine ausreichend fein abgestufte Wirbelsäule sein. Dabei hat sich in meinen Versuchen eine Wirbelsäule von etwa fünf bis sieben Segmenten als ausreichend herausgestellt. Abbildung 4.1 zeigt das fertige Rig für einen Fisch der Gattung Neon-Tetra.

4.2 Nachbildung der MPF-Lokomotion

Die Nachbildung der in Kapitel 2.1.2.2 vorgestellten MPF-Lokomotion ist in unserem vereinfachten Anwendungsfall relativ leicht zu realisieren, da hier nur jeweils einzelne Flossen betroffen sind, die als Kindknoten unabhängig von der restlichen Skelettstruktur betrachtet und animiert werden können. Hinzu kommt, dass bei der Darstellung der Flossenbewegungen schon aufgrund der geringen Darstellungsgröße weniger Details nötig sind, um ein überzeugendes Ergebnis zu erreichen. Parametrisierte Keyframe-Animationen der Flossen reichen hier aus, wobei die Amplitude der Bewegung und die Abspielgeschwindigkeit an die momentane Geschwindigkeit des Fisches gekoppelt werden.

4.3 Nachbildung der BCF-Lokomotion

In Kapitel 2.1.2.1 haben wir betrachtet, wie Fische Vorwärtsantrieb generieren, indem sie Sinus-förmige Wellen durch Teile ihres Körpers achtern auswandern lassen. In [TT94] wurde diese Technik extrem aufwendig und exakt nachgebildet, wie in Kapitel 3.1 beschrieben. Ich stelle im folgenden Kapitel einen Ansatz vor, der das Feder-Masse-System von Tu und Terzopoulos soweit abstrahiert und reduziert, dass es auch auf schwächeren Endgeräten für eine Vielzahl von virtuellen Fischen berechnet werden kann und gleichzeitig noch ein zufriedenstellend realistisches Ergebnis liefert.

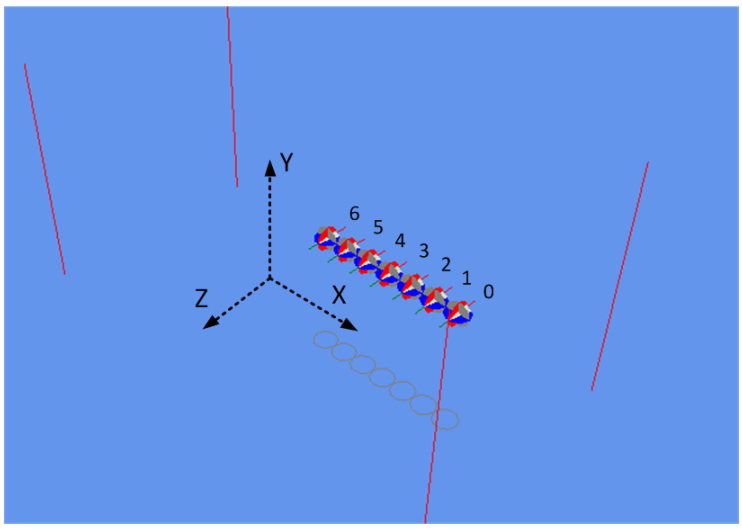


Abbildung 4.2: abstrahiertes Modell der schwingenden Fisch-Segmente (Ruhezustand)

4.3.1 Zweidimensionale Modellierung der Segmente

Für erste Experimente implementierte ich dazu mit Hilfe des XNA-Frameworks eine einfache Applikation unter .NET, in der ich sieben Fisch-Segmente als einfache Massepunkte modellierte und entlang der x-Achse anordnete (siehe Abbildung 4.2). Diese Segmente ließ ich entlang der z-Achse schwingen, wobei ich die Auslenkung entlang der z-Achse anhand des Zeit-Orts-Gesetz der harmonischen Schwingung berechnete:

$$z(t) = \hat{z} \cdot \sin(\omega t)$$

Um eine auswandernde Sinus-Welle zu generieren, wurden die Schwingungen um einen Wert $t_{delayProSegment}$ gegeneinander verzögert. Ein optionaler Parameter $gainProSegment$ sorgte für eine ansteigende Amplitude zum Heck des Schwingungskörpers hin. Daraus ergab sich folgende Formel für die horizontale Auslenkung z eines Segments i zum Zeitpunkt t:

$$z_i(t) = \hat{z}_i \cdot \sin(2\pi\Delta f + t_{delayProSegment} \cdot i) + gainProSegment \cdot i$$

Der Algorithmus ist mit einer Laufzeit von $O(n)$ sehr genügsam, und liefert eine zufriedenstellende Animation, solange der Fisch geradeaus schwimmt und nicht beschleunigt. Eine Änderung der Schwingfrequenz führt jedoch sofort zu einer schlagartigen Positionsänderung sämtlicher Segmente. Im zweiten Ansatz werden die Segmente deshalb nun mit Hilfe eines virtuellen Feder-Masse-Systems miteinander verbunden.

4.3.2 Periodische Bone-Animation durch Feder-/Masse-System

Das in Kapitel 3.1 vorgestellte System der vernetzen Polygonknoten liefert zwar beeindruckende Ergebnisse, ist jedoch für die anvisierten Zielplattformen bei

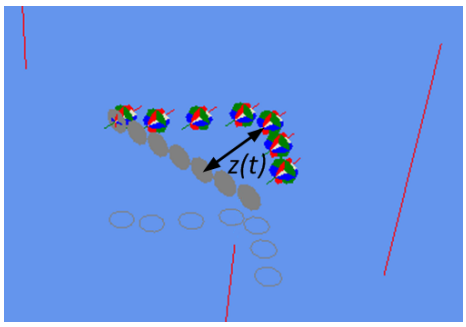


Abbildung 4.3: Modell der Fisch-Segmente: Schwingung entlang der z-Achse

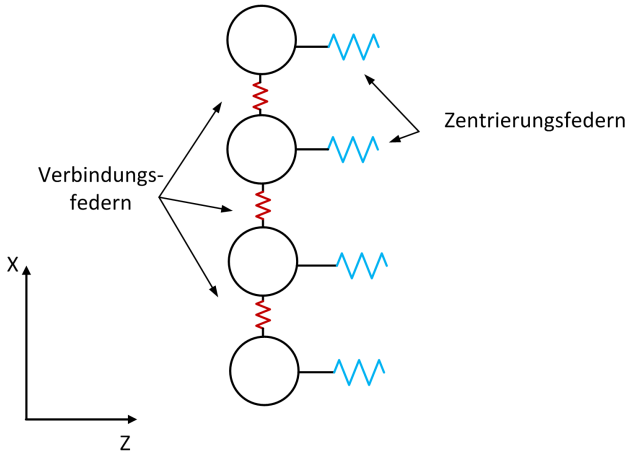


Abbildung 4.4: 2-dimensionales Feder/Masse-System zur Modellierung der BCF-Lokomotion

weitem zu aufwendig. Stattdessen habe ich versucht, eine vereinfachte Form dieses Ansatzes auf das in Unity3D verwendete Bone-Animations-System zu übertragen. Die Segmente des in der letzten Sektion vorgestellten Modells habe ich dazu mit zwei Paaren von virtuellen gedämpften Federn verbunden, wie in Abbildung 4.4 dargestellt.

In meinem Modell geht der Kontraktionsimpuls ausschließlich vom Kopfsegment aus. Dadurch wird es möglich, durch einfache zeitlich verzögerte Steuersignale (links/rechts/aus) die BCF-Lokomotion des gesamten Fisches zu steuern. Ausgehend vom Kopf ist dabei jedes Segment mit seinem Kindsegment vertikal durch eine virtuelle Feder verbunden, die den Kontraktionsimpuls an ihr Kindsegment überträgt. Dabei ist zu beachten, dass die Kraftübertragung nur in Richtung des Kindsegments ausgewertet wird, damit sich die Wellenbewegung nur in Richtung der Schwanzflosse ausbreiten kann und nicht zurück zum Kopf. Da sich die Segmente in unserem Modell nur horizontal bewegen können, wird außerdem nur die horizontale Komponente der Zugkraft ausgewertet und die vertikale Komponente ignoriert. Der horizontale Versatz ist in Abbildung 4.5 dargestellt. Damit ein Segment horizontal nicht unendlich weit auswandert und nach einem Kontraktionsimpuls wieder langsam zentriert wird, sind an jedem

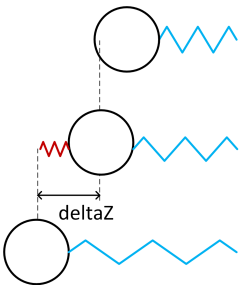


Abbildung 4.5: Berechnung des relativen horizontalen Segment-Versatzes

Segment gedämpfte Zentrierungsfedern angebracht. Durch eine entsprechende Parametrisierung sind damit schnelle steife Bewegungen ebenso möglich wie eine sehr ausladende Schwingung der Segmente, die der Bewegung von Wasserschlangen sehr nahe kommt. Die Kraft F eines gedämpften Feder-/Masse-Systems berechnet sich entsprechend des Hook'schen Gesetzes dabei aus der Summe der Federkraft F_S und der entgegen gerichteten Dämpfungskraft F_D. Der Federkoeffizient k (N/m) der Feder S beschreibt die Kraft, die durch die Feder bei einer Auslenkung von einem Meter wirkt. Der Betrag von F_D ist abhängig von der Geschwindigkeit des Massepunktes, also unseres Fisch-Segments. Sie wird durch den Dämpfungskoeffizienten c beschrieben, der die Masse in Kilogramm angibt, die bei einer Geschwindigkeit von einem Meter pro Sekunde gedämpft wird.

$$F = F_S + F_D$$

$$F_S = -kx$$

$$F_D = -cv = -c\frac{dx}{dt} = -c\dot{x}$$

F_s	Federkraft (N)
x	Auslenkung der Feder (m)
k	Federkonstante (N/m)
c	Dämpfungs-Koeffizient (kg/s)

Die Kraft, die ein Elternsegment *parent* unserer Fischwirbelsäule auf ein Kindsegment i wirkt, errechnet sich gemäß Abbildung 4.5 wie folgt:

$$F_{parent} = k_{parent} \cdot deltaZ - c_{parent} \cdot v_i$$

Auf Basis unseres Modells können wir nun zu jedem Zeitpunkt für jedes Segment einen Winkel α berechnen, um den das Kindsegment dieses Segments im lokalen Koordinatensystem gedreht werden muss, um eine entsprechende Ausrichtung der Wirbelsäule in Unity3D zu erzielen (siehe Abbildung 4.6).

4.3.3 Laufzeit und Optimierungsmöglichkeiten

Der Algorithmus ist bei einer Laufzeit von $O(n)$ immer noch sehr genügsam, zumal sich das n hier nur auf die Anzahl von maximal zehn Segmenten bezieht. Dadurch, dass sich die Federkräfte nur auf die Animation auswirken und

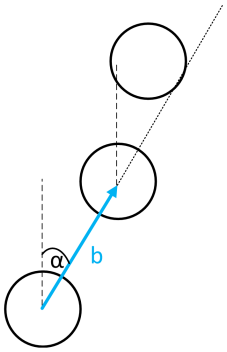


Abbildung 4.6: Berechnung des Bone-Rotationswinkels auf Basis des Versatzvektors zweier Segmente

auch größere Ungenauigkeiten hier keine sichtbaren Fehler nach sich ziehen, könnten wir bei der Sinus-Berechnung auf einfache Lookup-Tabellen zurückgreifen, in der einige wenige Sinuswerte mit geringer Präzision gespeichert und interpoliert werden. Während man mit dieser Technik noch vor wenigen Jahren auf Java-basierten mobilen Endgeräten eine signifikante Einsparung von CPU-Zyklen erzielen konnte, bleibt zu testen, inwieweit diese Optimierung auf unseren aktuellen Zielplattformen noch messbare Verbesserungen bringt.

Interferenzen Ein Nachteil des Feder-Masse-Systems in Kombination mit der einfachen Links/Rechts-Steuerung des Kopfsegmentes ist, dass die Schlagfrequenz auf die Federkonstanten der Verbindungsfedern abgestimmt sein muss. Passt die Frequenz nicht optimal, bilden sich Interferenzen in der Wellenbewegung, die unnatürlich aussehen und im Extremfall zu stehenden Wellen führen können. Die Federkonstanten müssten also dynamisch zur Frequenz berechnet und angepasst werden. Alternativ könnte auf Basis des Feder/Masse-Systems eine Reihe von Keyframe-Animationen generiert werden, die dann passend zur Fisch-Beschleunigung kombiniert und mit variabler Geschwindigkeit abgespielt werden könnte.

4.4 Krümmung des Fischkörpers in Kurven

Dream Aquarium (siehe Kapitel 3.3) hat gezeigt, dass sich sehr realistische Bewegungsabläufe auch wesentlich sparsamer als im Feder-/Masse-System von Tu und Terzopoulos erzielen lassen. Auffällig bei Kaplers Ansatz ist, dass der realistische Eindruck auch bei Übergangsbewegungen (siehe Kapitel 2) nicht verloren geht. Werden in Computerspielen zur Animation von Schlangen und Fischen einfache aufgesetzte Keyframe-Animationen abgespielt, so neigen diese dazu, in Kurvenbewegungen den Eindruck der Masseträgheit zu zerstören, da die einzelnen Segmente während der Bewegung keine feste Bahn im Welt-Koordinatensystem beschreiben. Kaplers Algorithmus berücksichtigt hingegen diese korrekte Positionierung, sodass niemals ein unnatürlicher Eindruck entsteht. Diese Funktionalität habe ich in einem eigenen Algorithmus namens *FishCurveBender* nachimplementiert, den ich im Folgenden vorstelle.

4.4.1 Verwaltung von Stützstellen

Für eine Anpassung des Körpers an die Kurvenbahn ist es zunächst nötig, diese mit Hilfe periodisch gespeicherter Stützstellen zu sichern. Die erste Stützstelle entspricht dabei immer der aktuellen Position des äußersten Kopf-Endes des Fisches, da der Kopf beim Schwimmen von Kurven an einer festen relativen Position bleibt, und in meiner Implementierung deshalb der aktuellen Position des gesamten Fisch-Objektes entspricht. Die übrigen Stützstellen werden anhand der zuletzt zurückgelegten Wegstrecke des Fisches in festen räumlichen Abständen gespeichert. Anhand dieser Stützstellen lässt sich innerhalb des Algorithmus die zuletzt beschriebene Kurvenbahn rekonstruieren. Dazu verwende ich momentan einfache lineare Interpolation. Eine optisch sauberere Alternative bei gleicher oder geringerer Anzahl von Stützstellen böten hier die B-Spline-Funktionen ([Nür89]), deren Evaluation ich aus Zeitmangel auf später verschiebe. Für ein optisch ansprechendes Ergebnis muss der Abstand der Stützstellen in meinem linearen Ansatz deshalb ausreichend klein sein. Zu große Abstände verfälschen den Bewegungsverlauf und führen zu einer eckigen Krümmung des Fischkörpers, der sich dann wie auf kantigen Schienen zu bewegen scheint. Zu kleine Abstände führen hingegen zu einer steigenden CPU-Last, da der Algorithmus über jedes einzelne Wegsegment iteriert. Als Kompromiss wähle ich als Abstand der Stützstellen die Hälfte der kürzesten Knochenlänge des animierten Fisches. Am Ende des Algorithmus werden überschüssige Stützstellen aussortiert, deren Positionen sich im Bewegungsverlauf jenseits des Fischschwanzes befinden. Als Containerklasse für die gesammelten Stützstellen wähle ich eine verkettete Liste, da sich außer dem Anfang und Ende des Inhaltes jeweils nur die Index-Positionen der Elemente verändern.

4.4.2 Bestimmung der Segment-Rotationen

Im nächsten Schritt muss nun der Winkel der einzelnen Segment-Knochen dem Bewegungspfad angeglichen werden. Dazu beginnen wir am Kopf-Ende des Fisches und legen eine gedachte Kugel um den Kopf, dessen Mittelpunkt am äußersten Schnauzen-Ende des Fisches liegt und dessen Durchmesser der Länge des Kopfsegments entspricht. Der Schnittpunkt der Kugel mit dem Bewegungspfad ergibt die äußerste End-Position des folgenden Segments. Der Winkel zwischen den beiden Endpunkten der Segmente stellt den Winkel dar, um den das Kopfsegment gedreht werden muss, damit sein Kindsegment an die korrekte Position wandert (siehe Abbildung 4.7). Zum Berechnen der Schnittpunkte werden dabei jeweils Strahlen (in Unity3D in der Klasse *Ray* implementiert) an der Position des letzten Bones generiert, die in Richtung der nächst-älteren Stützstelle zeigen. Während beim Kopfsegment dabei als Ergebnis immer nur ein einzelner Schnittpunkt existiert, so ist es bei den folgenden Bones möglich, dass die Intersektions-Methode zwei Schnittpunkte mit dem Wegpunkt-Strahl liefert. Dies ist der Fall, wenn die Stützstellen in so großen Abständen generiert wurden, dass eine Stützstelle außerhalb des Bone-Radius liegt. In diesem Fall muss überprüft werden, welcher der beiden Schnittpunkte der Richtige ist. Da spitze Winkel innerhalb des Fischkörpers nicht auftreten, ist derjenige Schnittpunkt der richtige, dessen Schnittpunkt weiter von der vorletzten Bone-Position entfernt liegt. Alternativ kann hierzu auch der Richtungsvektor zwischen Schnittpunkt und letzter Bone-Position gebildet und per Punktprodukt mit der Ausrichtung des letzten Bones

verglichen werden. Der Schnittpunkt, der in der falschen Richtung liegt, wird hierbei zu einem negativen Wert führen. Ist der gewählte Abstand der Wegpunkte kleiner als die Länge der Segmente, so ist es möglich, dass die Strahlen mehrerer aufeinander folgender Pfad-Abschnitte einen Schnittpunkt mit der jeweiligen Segment-Kugel liefern. Aus diesem Grund wird pro Segment so lange weiter über die Wegabschnitte iteriert, bis kein Schnittpunkt mehr gefunden wird. Der vorherige Wegabschnitt liefert dann mit seinem Schnittpunkt den Winkel des folgenden Segments-Bones. Der FishCurveBender-Algorithmus ist beendet, wenn die Position des letzten Knochens berechnet wurde und/oder keine weiteren Schnittpunkte mit dem Bewegungspfad mehr erkannt werden. Der letzte Fall tritt ein, wenn der Fisch aus einer Ruheposition heraus losschwimmt, da zu diesem Zeitpunkt noch nicht genug Stützstellen existieren.

Während der Implementierung stellte ich fest, dass eine fehlerhafte Konertierung zwischen Linke-Hand- und Rechte-Hand-Koordinatensystemen beim Import von Modellen unter Unity3D die Verwendung von 3D-Modellen, die im FBX-Format exportiert wurden, mit dem beschriebenen Algorithmus praktisch unmöglich macht. Eine zweite Version des FishCurveBenders basiert deshalb auf einer Knochenstruktur, bei dem die einzelnen Glieder der Wirbelsäule nicht hierarchisch miteinander verbunden sind. Dies hat den Vorteil, dass jedes Segment nun unabhängig vom letzten positioniert und rotiert werden kann. Statt des Rotationswinkels werden über die Schnittpunkte der Segment-Kugeln mit dem Bewegungspfad nun direkt die Positionen der nachfolgenden Segmente ermittelt.

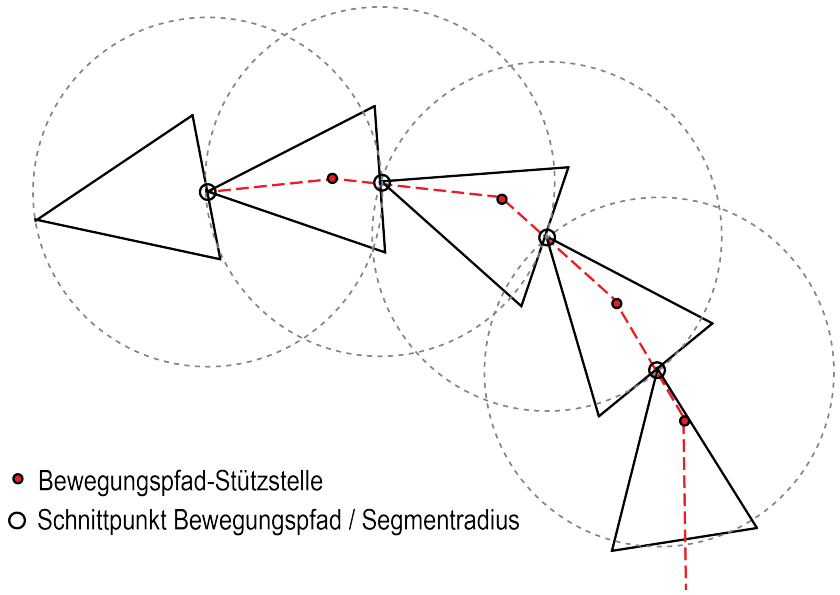


Abbildung 4.7: einfacher Curve-Bender-Algorithmus: Bestimmung der lokalen Rotationswinkel der Wirbel durch Bestimmung der Schnittpunkte ihrer Radien mit dem angenäherten Bewegungspfad

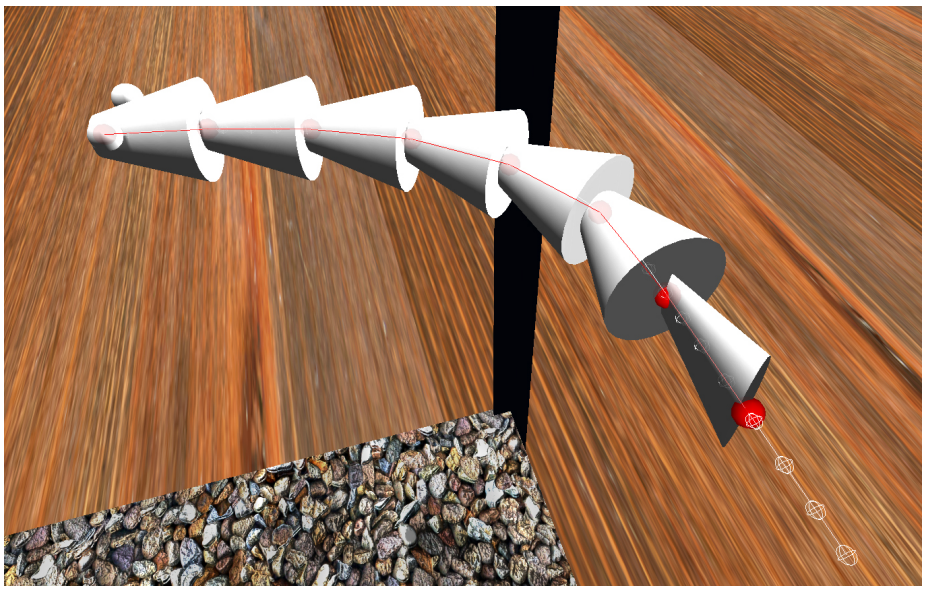


Abbildung 4.8: Curve-Bender-Algorithmus: Darstellung des Bewegungspfades (weiße Punkte), der lokalen Rotationswinkel (rote Linien) und der Bone-Segmente anhand eines Testmodells innerhalb von Unity3D

4.4.3 Laufzeit und Optimierungsmöglichkeiten

Wie das Feder-/Masse-System zur Nachbildung der BCF-Lokomotion muss auch der zuletzt beschriebene FishCurveBender-Algorithmus bei einer Laufzeit von $O(n)$ grundsätzlich nur über die Anzahl der Segmente iterieren. Die kürzeste Laufzeit kann erzielt werden, wenn der Abstand der Stützstellen dabei in etwa der Knochenlänge entspricht, was allerdings zu einem kantigen Gesamtbild führt. Ein weicher Verlauf könnte bei einer geringen Anzahl von Stützstellen mit den bereits in Abschnitt 4.4.1 erwähnten B-Splines erzielt werden.

4.5 Kombination von BCF-Lokomotion und Körper-Krümmung

Der bisher beschriebene FishCurveBender-Algorithmus krümmt die Bones des Fisches vollständig auf den zuletzt beschriebenen Bewegungspfad. In der nächsten Iteration des Algorithmus muss nun zusätzlich die vom Feder-/Masse-System berechnete horizontale Auslenkung der Segmente berücksichtigt werden, um beide Effekte miteinander zu kombinieren. Ziel dabei ist, die Schwingung der BCF-Lokomotion über einen Prozentwert stufenlos auf den Bewegungspfad addieren zu können. Dadurch kann bei Bedarf die Schwingung der Segmente stufenlos ausgeblendet werden, was beispielsweise in engen Kurvenlagen zu einer realistischeren Animation beiträgt.

Ausgehend von dem Sektion 4.3.1 vorgestellten Feder-/Masse-System ändert sich durch die Kombination mit dem FishCurveBender-Algorithmus die Grundlinie, zu denen die Segmente orthogonal schwingen. Statt senkrecht zu

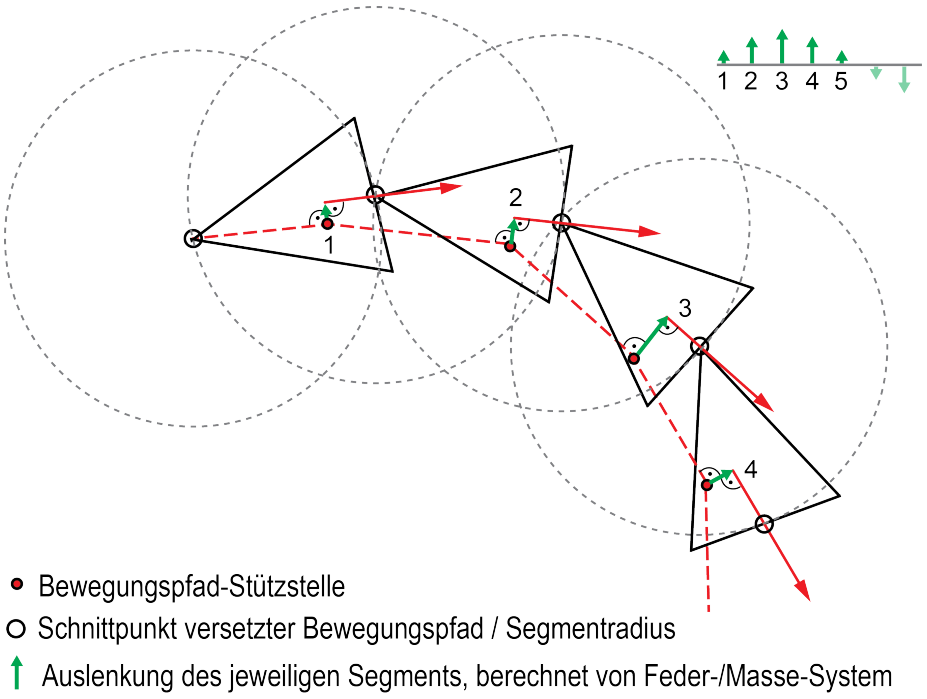


Abbildung 4.9: Kombination von BCF-Lokomotion und Krümmungs-Algorithmus

einer gemeinsamen Geraden erfolgt die Schwingung jedes einzelnen Segments nun senkrecht zum Bewegungspfad des Fisches. Ausgehend von dieser Überlegung muss unser FishCurveBender-Algorithmus nun so erweitert werden, dass die Radien der Segmente nicht direkt mit den Abschnitten des Bewegungspfaden geschnitten werden, sondern mit Strahlen, die um die Z-Auslenkung des jeweiligen Segments versetzt sind, und die parallel zum jeweiligen Teilabschnitt des Bewegungspfades verlaufen. Der angepasste Algorithmus ist in Abbildung 4.9 dargestellt. Oben rechts in der Abbildung ist die momentane Z-Auslenkung der Segmente im aktuellen Render-Frame dargestellt, um die ein jeweiliger Pfad-Abschnitt versetzt werden muss. Die Richtung dieses Versatz-Vektors errechnet sich jeweils durch das Kreuzprodukt des momentan betrachteten Pfadabschnitts-Richungsvektors mit dem Einheitsvektor der positiven Y-Achse im lokalen Koordinatensystem des Fisches. Ein solcher versetzter Strahl wird für jede Stützstelle des Bewegungspfades generiert und übernimmt als Richtungsvektor den des vorherigen Pfadabschnitts. Wie in der vorherigen Version des FishCurveBender-Algorithmus wird wieder über sämtliche Fischsegmente und alle Stützstellen des Bewegungspfades iteriert, ausgehend vom Kopf-Segment. Der jeweils letzte Wegpunkt, für den noch ein Schnittpunkt mit dessen versetzten Strahl und der Kugel um das letzte Fischsegment ermittelt werden konnte, bestimmt mit diesem Schnittpunkt die Position des folgenden Fischsegments. Am Ende des Algorithmus wird schließlich der lokale Richtungsvektor der einzelnen Segmente berechnet. Dieser errechnet sich jeweils durch Subtraktion des Segment-Ortsvektors mit dem des benachbarten Bones.

Kapitel 5

Reaktives Navigieren

Nachdem wir im vorangegangenen Kapitel die Lokomotion der Fische einigermaßen glaubwürdig visualisieren konnten, beschäftigen wir uns nun mit der Frage, auf Basis welcher Informationen die momentane Bewegungsrichtung des Fisches bestimmt werden soll. Zu den Anforderungen an diese Ebene der KI gehört beispielsweise das Umschwimmen von Hindernissen, das Beibehalten von Schwarmformationen oder die Flucht vor Fressfeinden.
Den wichtigsten Ansatz in diesem Bereich der Künstlichen Intelligenz stellen die von Craig W. Reynolds vorgestellten *Steering Behaviors* (bzw. eingedeutscht: Bewegungsmuster) dar. Den Grundstein für diese Sammlung von 16 verschiedenen Steuerungstechniken legte Reynolds im Jahr 1987 mit seiner Veröffentlichung *Flocks, Herds, and Schools: A Distributed Behavioral Model* [Rey87]. Reynolds stellte dabei einen Algorithmus vor, mit dem sich das Schwarmverhalten von Fisch- und Vogelschwärmen auf Basis von drei einfachen Regeln erstaunlich realistisch im Computer nachbilden ließ. Der Algorithmus basiert auf einem einfachen physikalischen Bewegungsmodell der autonomen Agenten, die er in seiner Arbeit als *Boids* bezeichnet. Auf jeden Boid wirken dabei bis zu drei Teilkräfte, die von der Geschwindigkeit, der Rotation und der Position der Boids in dessen Umgebung abhängen. Reynolds Schwarm-Algorithmus ist für unseren Anwendungsfall bestens geeignet und wird in Sektion 5.3.1 genauer vorgestellt. Wir betrachten im folgenden Abschnitt zunächst das Bewegungsmodell, das Reynolds in seinem Ansatz verwendet. Dieses bildet auch die Grundlage für weitere Steering Behaviors, die Reynolds 1999 in seiner ergänzenden Veröffentlichung *Steering Behaviors for Autonomous Characters* [Rey99] vorstellte. Mit den wichtigsten dieser Steering Behaviors beschäftigen wir uns in Sektion 5.2. Auf den von Reynolds vorgestellten Techniken basiert eine Open-Source-Implementierung namens *OpenSteer*, die 2003 von Sony Computer Entertainment America veröffentlicht und seitdem von verschiedenen Parteien weiterentwickelt wurde [Ame03]. Zu OpenSteer existieren Portierungen für .NET und Unity3D, wobei letztere auch die Grundlage für meine eigene Implementierung bildet.

Masse	float
Position	Vector3
Geschwindigkeit	Vector3
Maximale Geschwindigkeit	float

Tabelle 5.1: Variablen des Bewegungsmodells

5.1 Physikalisches Bewegungsmodell

Zur Implementation der Bewegungsmuster muss der Boid über ein physikalisches Modell verfügen, auf das die durch die einzelnen Bewegungsmuster generierten Teilkräfte wirken können. Aus der Berechnung jedes Bewegungsmusters resultiert jeweils genau eine Teilkraft, wobei theoretisch beliebig viele dieser einzelnen Teilkräfte zu einer resultierenden Gesamtkraft addiert werden können. Verschiedene Ansätze zum Aufsummieren mehrerer Teilkräfte stelle ich in Abschnitt 5.4 vor. Die Summe der Teilkräfte ergibt die resultierende Steuerkraft, die auf die Masse des virtuellen Fisches wirkt. Basierend auf der Masse des Fisches, der Dauer des letzten Frames und der Geschwindigkeit und Position des Fisches im letzten Frame kann somit zu jedem Zeitpunkt die aktuelle Position des Fisches berechnet werden. In Reynolds Modell ist eine Bewegung des Boids nur geradeaus, sprich entlang dessen Vorwärts-Ausrichtungsvektors möglich. In jedem Frame wird der Boid deshalb nach Aufsummieren der Steuerkräfte an der aktualisierten Bewegungsrichtung ausgerichtet. Um unnatürlich schnelle Drehungen zu vermeiden, kann außerdem eine maximale Rotationsgeschwindigkeit in Radians pro Sekunde festgelegt werden. Ich verzichte in meiner Implementierung auf diese Einschränkung, da auch extrem ruckartige Richtungsänderungen wie beispielsweise von kleinen Schwarmfischen möglich sein sollen. Da diese schnellen Bewegungen nur in relativ seltenen Schreckmomenten auftreten, und nicht ständig, wie beispielsweise bei der Einhaltung der Schwarmformation, federe ich unnatürlich schnelle Drehungen stattdessen direkt in den einzelnen Bewegungsmustern ab, wie in Sektion 5.5.1.2 erläutert.

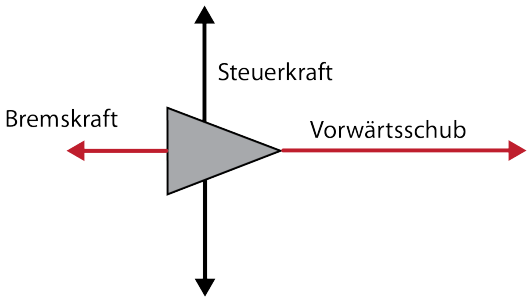


Abbildung 5.1: Bewegungsmodell zur Implementierung der Bewegungsmuster

5.2 Bewegungsmuster für autonome Agenten

In den folgenden Abschnitten gehe ich nun konkret auf die einzelnen Bewegungsmuster ein, die Reynolds in seinen Arbeiten vorgestellt hat. Von Reynolds ursprünglich 16 Bewegungsmustern stelle ich 11 vor, die für unser Aquarien-Projekt besonders geeignet sind.

5.2.1 Seek

Eine wichtige Anforderung, die auch die Grundlage zu verschiedenen komplexeren Bewegungsmustern darstellt, ist das Ansteuern eines beliebigen Punktes in der Umgebung des Agenten. Das Ergebnis der Seek-Behavior ist eine Lenkkraft, die den momentanen Kurs des Agenten zur Richtung des Zielpunktes hin zieht. Dazu ist es zunächst erforderlich, den gewünschten Richtungsvektor zu errechnen, der vom Agenten aus direkt auf das Ziel zeigt. Dies geschieht durch einfache Subtraktion der Position unseres Fisches von der Position seines Zieles. Durch Normalisieren des Vektors und Skalieren mit der maximal möglichen Geschwindigkeit des Fisches erhalten wir einen Vektor, der die gewünschte Bewegungsrichtung und -geschwindigkeit repräsentiert. Im nächsten Schritt muss von diesem Vektor nun nur noch die tatsächliche aktuelle Geschwindigkeit des autonomen Agenten subtrahiert werden, um die korrigierende Lenkkraft zu erhalten. Je genauer die aktuelle Geschwindigkeit mit der gewünschten Geschwindigkeit übereinstimmt, desto kleiner fällt die Länge der Lenkkraft aus.

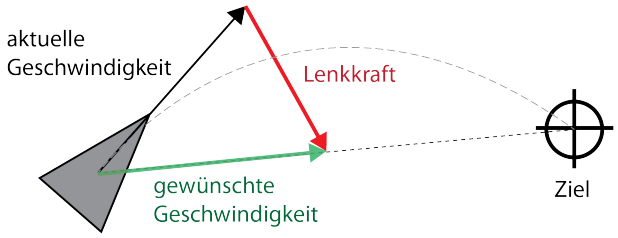


Abbildung 5.2: Berechnung der Teilkraft des Seek-Bewegungsmusters

5.2.2 Flee

Das Gegenteil zur eben betrachteten Seek-Behavior stellt die Flucht vor einem bestimmten Punkt im Raum dar. Die Berechnung erfolgt hierbei fast genau wie bei *Seek*, mit dem einfachen Unterschied, dass der Vektor der gewünschten Geschwindigkeit in die entgegengesetzte Richtung zeigt.

5.2.3 Pursuit

Verfolgt ein Fisch einen Beutefisch oder ein anderes bewegliches Objekt, so reicht ein einfaches Anwenden der Seek-Behavior oft nicht aus, um die Beute abzufangen. Ein stures Zuschwimmen auf die aktuelle Position der Beute führt bei einer ausreichend hohen Fluchtgeschwindigkeit der Beute stattdessen dazu, dass

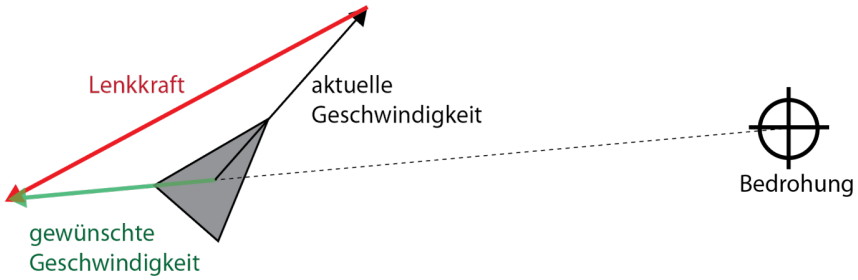


Abbildung 5.3: Berechnung der Teilkraft des Flee-Bewegungsmusters

der Jäger nur hinter seine Beute gelangen wird und ihr schließlich nur hinterher schwimmt. Um dies zu verhindern, muss stattdessen der Punkt angesteuert werden, an dem sich das Ziel zum Zeitpunkt des Auftreffens wahrscheinlich befinden wird. Sind Position und Geschwindigkeit v des Ziels bekannt, muss dazu eine Vorhaltezeit gewählt werden, auf deren Basis die zukünftige Position der Beute extrapoliert wird. Diese Vorhaltezeit ist proportional zum Abstand zur Beute und umgekehrt proportional zu den Bewegungs-Geschwindigkeiten der beiden Objekte. Durch diese Anforderungen ergibt sich folgende Formel zur Berechnung der Vorhaltezeit t_0:

$$t_0 = \frac{\vec{Pos}_{beute} - \vec{Pos}_{jaeger}}{\vec{Vmax}_{jaeger} + \vec{V}_{beute}}$$

Mit Hilfe der Vorhaltezeit kann nun die Zielposition extrapoliert werden:

$$\vec{Pos}_{ziel} = (\vec{Pos}_{beute} + \vec{V}_{beute}) \cdot t_0$$

Diese zukünftige Zielposition kann nun mit der bereits vorgestellten Seek-Behavior angesteuert werden, um das Zielobjekt abzufangen.

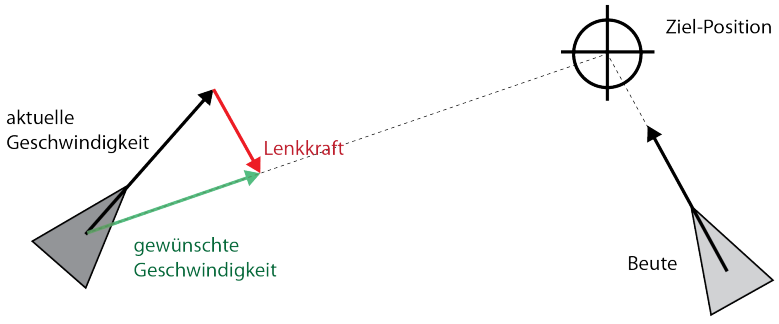


Abbildung 5.4: Berechnung der Teilkraft des Pursuit-Bewegungsmusters

5.2.4 Evade

Ähnlich wie bei *Seek* und *Flee* existiert auch zu *Pursuit* eine Behavior, die dessen Bewegungsrichtung direkt entgegengerichtet ist und deren Berechnung fast gleich erfolgt. Mit Hilfe des Bewegungsmusters Evade kann ein Beutefisch vor einem Jäger entfliehen, wobei analog zu *Pursuit* ebenfalls dessen aktuelle Geschwindigkeit berücksichtigt wird, um eine zukünftige Position vorherzusagen. Ähnlich wie bei *Seek* und *Flee* ändert sich wieder nur die Richtung des Vektors der gewünschten Geschwindigkeit. Der einzige Unterschied zu *Pursuit* besteht bei *Evade* entsprechend darin, dass auf die extrapolierte zukünftige Position statt der Seek- nun die Flee-Behavior aufgerufen wird.

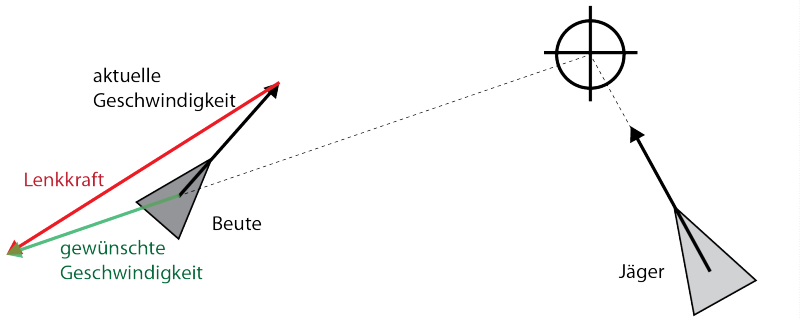


Abbildung 5.5: Berechnung der Teilkraft des Evade-Bewegungsmusters

5.2.5 Wander

Das Wander-Bewegungsmuster berechnet einen zufallsbasierten Bewegungsverlauf und eignet sich dazu, das willkürliche Umherstreifen der Fische im Aquarium nachzubilden. Der einfachste denkbare Ansatz zum Erzeugen einer Zufallsbewegung wäre, in jedem Frame eine zufällige Lenkkraft zu generieren, was jedoch zu einer hektischen und wenig realistischen Bewegung führt. Um gleichmäßigere, konstante Kurvenbewegungen zu ermöglichen, wird bei *Wander* stattdessen die letzte Bewegungsrichtung beibehalten und stattdessen pro Frame ein zufälliger Versatz berechnet. Die gleichmäßigsten Verläufe werden dabei erzielt, wenn die Lenkkraft auf den Rand einer gedachten Kugel beschränkt wird, die sich vor dem Agenten befindet. Der Parameter *Wander Distance* gibt dabei den Abstand zum Mittelpunkt der gedachten Kugel an, und der Parameter *Wander Radius* die Größe der Kugel. Die *Wander Rate* beschreibt den maximal möglichen zufälligen Versatz pro Frame. Durch eine kurze Wander Distance und einen großen Wander Radius neigt der Fisch zu engeren Kurven. Große Werte für die Wander Rate führen zu hektischeren Bewegungswechseln.

5.2.6 Spherical Obstacle Avoidance

Zur Vermeidung von Kollisionen mit der Umgebung existieren verschiedene Bewegungsmuster, die sich in Hinblick auf ihren Anwendungsbereich und ihre

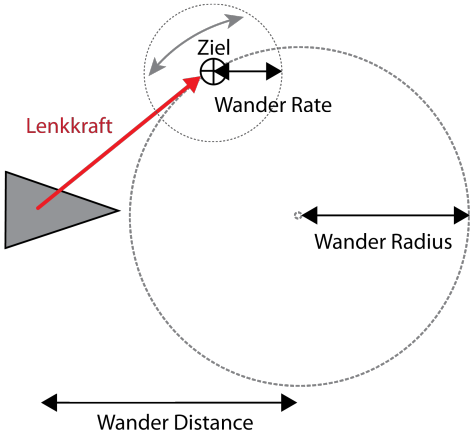


Abbildung 5.6: Berechnung der Teilkraft des Wander-Bewegungsmusters

Komplexität unterscheiden. Ein einfacher, aber effektiver Ansatz ist dabei das Ausweichen von Hindernissen, deren Form durch eine Kugel angenähert werden kann. Der Vorteil ist hierbei, dass der Agent außer Position und Radius auf keine weiteren Informationen über das Hindernis zugreifen muss. Durch die Kombination mehrerer Kugeln lassen sich hierbei auch komplexere Formen nachbilden. Theoretisch wäre es auch denkbar, die komplette Außenwand unseres Aquariums durch kleine Kugeln zu beschreiben. Dies ist jedoch aufgrund des hohen Rechenaufwands nicht ratsam, zumal mit den Bewegungsmustern *Containment* und *Wall Avoidance* effizientere Alternativen zur Verfügung stehen. Ziel von *Spherical Obstacle Avoidance* ist das Umschwimmen einfacher Objekte auf dem kürzesten Weg. Der Algorithmus basiert auf einem gedachten Zylinder, der die zukünftige Bewegungsbahn des Agenten beschreibt und auf Kollisionen mit der Umgebung getestet wird. Wird eine drohende Kollision festgestellt, wird die Bewegungsrichtung des Agenten so geändert, dass sich der Agent ausgehend vom Mittelpunkt des Hindernisses so vorbeibewegt, dass nur eine möglichst geringe Bewegungsänderung nötig ist. Im Gegensatz zu *Flee* steuert der Fisch diesmal also nicht direkt vom Hindernis weg, sondern nur knapp daran vorbei.

5.2.6.1 Test auf Kollision

Hindernisse, die nicht den Zylinder schneiden, stellen keine unmittelbare Bedrohung dar, und werden vom Algorithmus komplett ignoriert. Dieses Aussortieren erfolgt in mehreren Schritten, die in Abbildung 5.8 dargestellt sind. Im ersten Schritt kann durch eine Subtraktion der beiden Ortsvektoren und des Hindernis-Radius schnell festgestellt werden, ob der Abstand eines Hindernisses die Länge des Zylinders unterschreitet. Nur Hindernisse, bei denen dies der Fall ist, stellen eine potentielle Bedrohung dar, und werden für eine weitere Untersuchung markiert. Im nächsten Schritt transformiert der Algorithmus alle markierten Hindernisse in das lokale Koordinatensystem des Agenten. Hindernisse, deren x-Koordinate nach der Transformation negativ ist, befinden sich hinter dem Agenten, und können ebenfalls ignoriert werden (Schritt B). In Schritt C muss

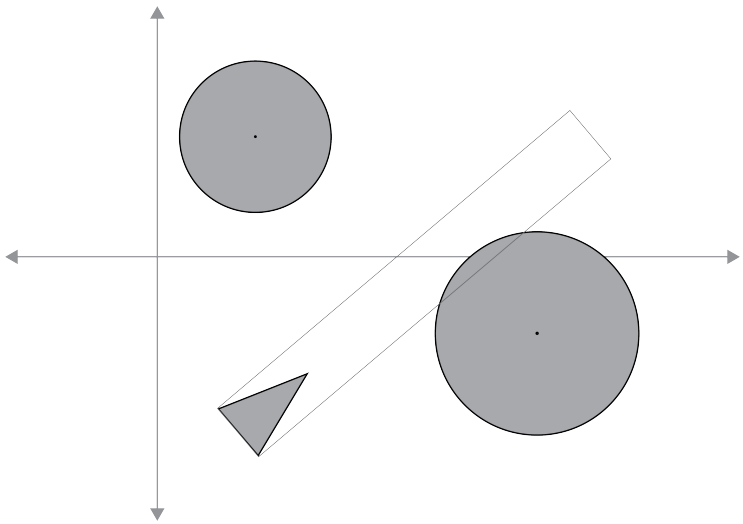


Abbildung 5.7: Ausweichen kugelförmiger Hinternisse (Obstacle Avoidance)

nun bei allen übrig gebliebenen Hindernissen geprüft werden, ob sie den Zylinder schneiden. Hier sind wieder die lokalen Koordinaten hilfreich, da der Radius des Hindernisses jetzt nur noch um die halbe Breite des Kollisions-Zylinders (sprich: den Radius des Agenten) erweitert werden muss. Ist die lokale y-Koordinate des Hindernisses größer als dieser Wert, besteht keine Kollisionsgefahr. Jetzt sind nur noch diejenigen Hindernisse übrig, die tatsächlich den Kollisions-Zylinder schneiden. Das Hindernis mit dem kürzesten Abstand zum Agenten stellt dabei die größte Bedrohung dar, alle anderen Hindernisse werden ignoriert. Im letzten Schritt der *Spherical Obstacle Avoidance* muss nun eine Lenkkraft berechnet werden, um diesem Hindernis auszuweichen.

5.2.6.2 Berechnung der ausweichenden Lenkkraft

Je mittiger sich ein Hindernis auf Kollisionskurs mit dem Agenten befindet, desto stärker muss die Kurskorrektur ausfallen, um dem Objekt auszuweichen. Ein praktikabler Ansatz zum Berechnen der Lenkkraft ist daher, die lokale y-Koordinate des Hindernisses von seinem Radius zu subtrahieren. Die so berechnete Kraft sollte außerdem mit dem Abstand zum Objekt skalieren, da weit entfernte Hindernisse weniger starke Ausweichmanöver erfordern. Im zweiten Schritt wird nun noch eine Bremskraft berechnet, die ebenfalls umgekehrt proportional zum Abstand des Hindernisses skaliert.

Spherical Obstacle Avoidance eignet sich in unserem Anwendungsfall hervorragend dazu, einfachen Gegenständen wie Steinen und Dekorationsobjekten auszuweichen. Zu beachten ist hierbei, dass um das entsprechende Hindernis genug Freiraum ist, damit der Fisch nicht in einem lokalen Minimum stecken bleibt.

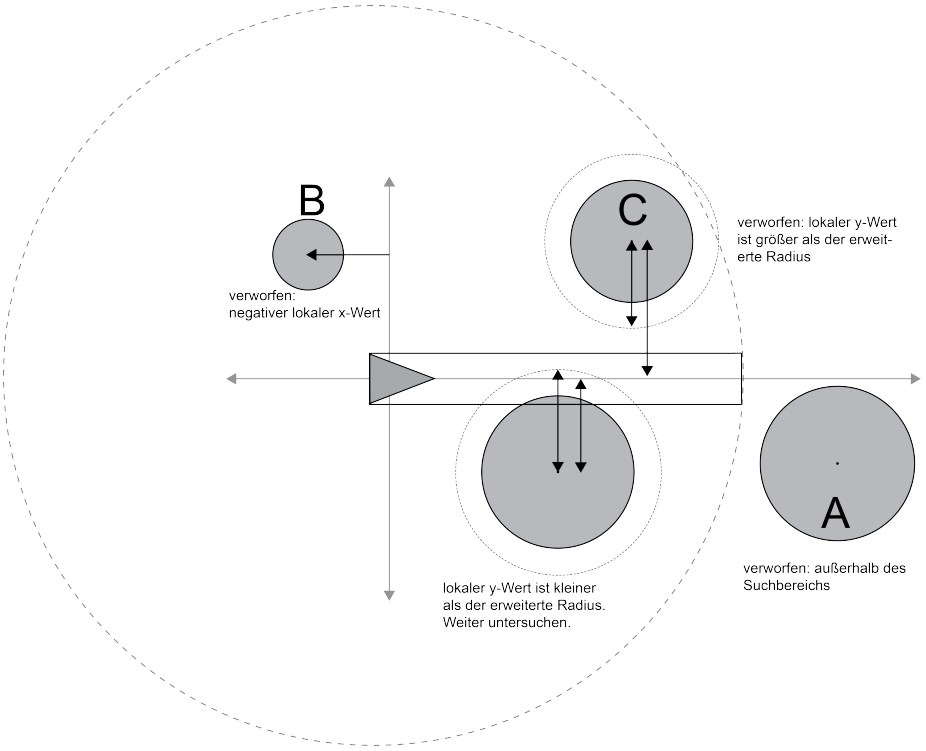


Abbildung 5.8: Obstacle Avoidance: Evaluierung des primären Hindernisses

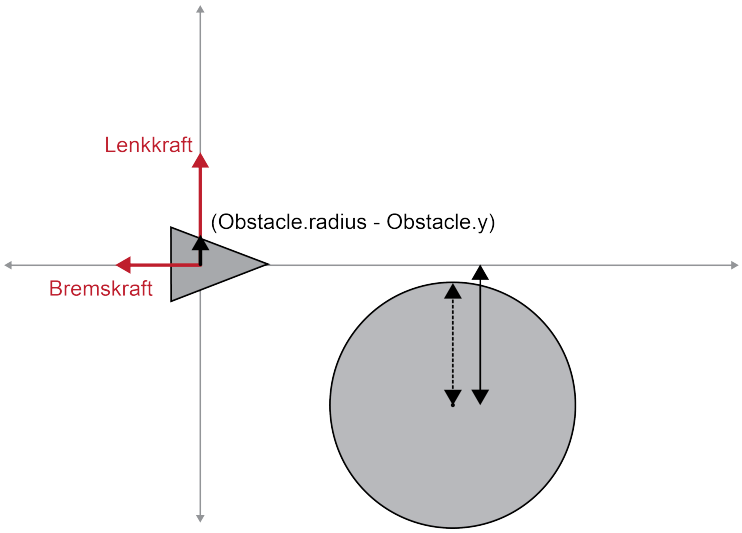


Abbildung 5.9: Obstacle Avoidance: Berechnung der Teilkraft

5.2.7 Wall Avoidance / Containment

Erreicht ein Fisch die Wand eines Aquariums, so muss er seine Schwimmrichtung so angleichen, dass er seine Bewegung parallel zur Wand fortsetzt. Ähnlich wie bei der *Spherical Obstacle Avoidance* muss also geprüft werden, ob bei Beibehaltung der aktuellen Geschwindigkeit eine Kollision mit der Wand droht. Statt eines Kollisions-Zylinders benutzt Reynolds in seinem ursprünglichen Ansatz einen einzelnen Vektor, der als Fühler für Kollisionen dient. Weiterentwicklungen dieses Ansatzes [Buc05] benutzen mehrere dieser Fühler, die zusätzlich seitlich nach vorne zeigen, und dann nacheinander auf Kollisionen getestet werden. Auf Basis des Fühlers, der das Hindernis mit dem kürzesten Abstand zum Agenten meldet, wird daraufhin die Lenkkraft berechnet. Diese muss senkrecht von der Oberfläche weg gerichtet sein, also entlang der Oberflächennormale verlaufen. Die Kraft muss umso größer sein, je steiler und schneller der Fisch auf die Wand zuschwimmt. Da die Länge des Fühler-Vektors in Abhängigkeit zur Geschwindigkeit gewählt wird, eignet sich die Durchdringungstiefe des Fühlers gut zum Skalieren der Lenkkraft.

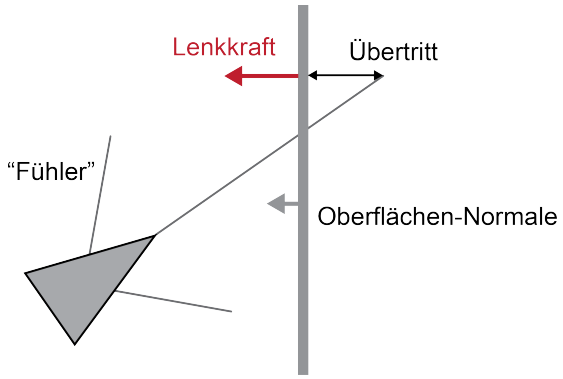


Abbildung 5.10: Berechnung der Teilkraft im Containment-Bewegungsmuster

5.3 Simulation von Schwarmverhalten

Im Bereich der Künstlichen Intelligenz haben sich zur Modellierung des in Kapitel 2.2 erläuterten Schwarmverhaltens zwei verschiedene Ansätze verbreitet, die ich im Folgenden vorstelle.

5.3.1 Flocking

Mit Reynolds' Flocking-Algorithmus lassen sich Schwärme von Vögeln, Fischen oder anderen Lebewesen simulieren oder in modifizierter Form auch Gruppenbewegungen von Militäreinheiten oder Fluggeschwadern. Sein grundlegender Algorithmus zur Nachbildung des Schwarmverhaltens wird heute als klassisches Flocking bezeichnet. Alle Boids können dabei zu einem Augenblick in eine gemeinsame Richtung schwimmen, bis im nächsten Moment ein Außenbereich des Schwarms die Richtung ändert, woraufhin die nachfolgenden Fische diese Bewegung übernehmen und sich die Richtungsänderung als eine Welle von abbiegenden Boids durch den Schwarm ausbreitet. Reynolds Implementation ist dabei führerlos in dem Sinne, dass kein einzelner Boid die Richtung vorgibt. Stattdessen beeinflussen sich die Boids ausschließlich gegenseitig und folgen gleichzeitig der Gruppe, die dadurch eine eigene übergeordnete Schwarmintelligenz zu besitzen scheint. Die durch den Algorithmus entstehende Bewegung wirkt verblüffend echt, insbesondere in Anbetracht der Tatsache, dass dieses Verhalten nur durch die Anwendung von drei einfachen Regeln erzeugt wird:

1. Cohesion (Zusammenbleiben) Jeder Boid steuert auf die durchschnittliche Position seiner Nachbarn zu.

2. Alignment (gemeinsames Ausrichten) Jeder Boid steuert so, dass seine Ausrichtung mit der durchschnittlichen Ausrichtung seiner Nachbarn übereinstimmt.

3. Separation (Abstand halten) Jeder Boid steuert so, dass er nicht mit seinen Nachbarn kollidiert.

5.3.1.1 Bestimmung der Nachbarn

Die drei Flocking-Regeln setzen voraus, dass jeder Boid Kenntnis über seine Umgebung hat, um die Anzahl seiner Nachbarn, deren Position, ihre Bewegungsrichtung sowie den eigenen Abstand zu ihnen berücksichtigen zu können. Um die Regeln anwenden zu können, muss dazu zunächst bestimmt werden, welche Fische in der Umgebung als Nachbarn berücksichtigt werden. Als Nachbar eines Boids B werden dabei alle anderen Boids angesehen, die sich innerhalb eines definierten Radius zu B befinden, und dabei nicht im toten Winkel von B liegen.

Das Sichtfeld eines Boids wird dabei durch den Winkel φ und den Sichtradius r definiert. Bei der Nachbar-Bestimmung lässt sich im ersten Schritt durch einfache Subtraktion der Positions-Vektoren feststellen, ob sich ein Boid im Sichtradius eines anderen befindet. Im zweiten Schritt wird nun zusätzlich geprüft, ob sich ein Boid auch in dessen Sichtwinkel befindet. Dies lässt sich mit Hilfe der normalisierten Vektoren zu Blickrichtung und relativer Position erreichen. Ein Boid liegt im Sichtfeld eines anderen Boids falls gilt:

$$u \cdot v \geq \cos \varphi$$

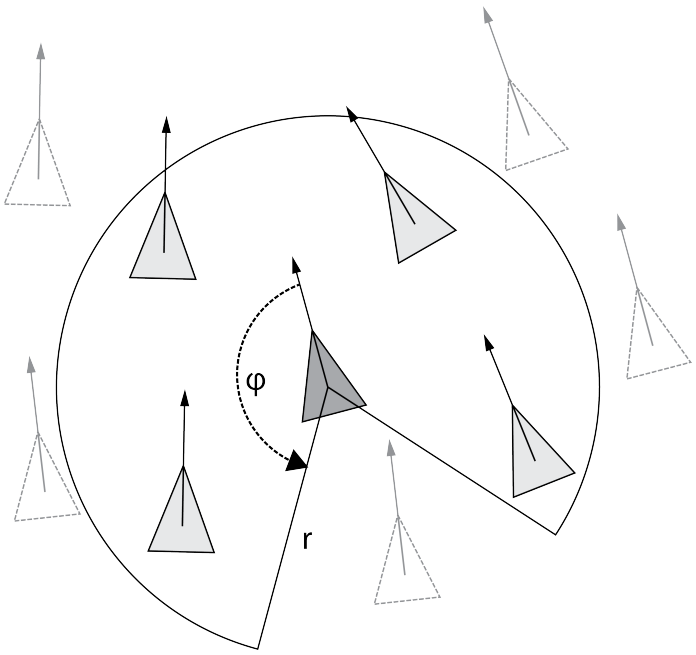


Abbildung 5.11: begrenzter Sichtradius r und Blickwinkel φ im Flocking-Algorithmus

u normalisierter Abstandsvektor

v normalisierte Blickrichtung]

φ Winkel des halben Sichtfeldes

k Index des auf Nachbarschaft zu testenden Boids

5.3.1.2 Cohesion

Die Cohesion-Behavior sorgt für das Zusammenführen der Fische, basierend auf der durchschnittlichen Position der Nachbarn. Basierend auf dieser Position wird ein Richtungsvektor berechnet, der auf die durchschnittliche Position der Nachbarn zeigt. Dieser Richtungsvektor dient als Basis für die Steuerkraft, die den Fisch in die Gruppe zieht. Um die Steuerkraft zu berechnen, werden zunächst alle Ortsvektoren der Nachbarn aufsummiert und dann durch die Anzahl der Nachbarn geteilt.

$$\vec{Pos}_{avrg} = \sum_{k=0}^{n} \vec{Pos}_k \cdot \frac{1}{n}$$

Von diesem Positionsvektor muss nun noch die Position des zu steuernden Boids subtrahiert werden, um den Versatzvektor zu erhalten, der in unserem Bewegungsmodell direkt als Steuerkraft verwendet werden kann. Die Länge dieses Vektors hängt dabei von der Stärke der Abweichung ab, sodass kleine Abweichungen auch nur kleine Steuerkräfte verursachen.

5.3.1.3 Alignment

Alignment beeinflusst die Ausrichtung des Fisches so, dass sie der durchschnittlichen Bewegungsrichtung seiner Nachbarn entspricht. Dadurch wird erreicht, dass sich alle Fische des Schwarms in etwa in die gleiche Richtung bewegen. Die Berechnung verläuft sehr ähnlich zum Cohesion-Algorithmus, mit dem Unterschied, dass anstatt der Ortsvektoren der Fische deren Ausrichtungsvektoren aufsummiert werden. Der dadurch entstehende Vektor wird wieder durch die Anzahl der Nachbarn geteilt, um die gemittelte Ausrichtung aller Nachbarn zu erhalten. Um sich dieser durchschnittlichen Ausrichtung anzugleichen, wird nun die eigene Ausrichtung des zu steuernden Boids subtrahiert. Das Ergebnis ist ein Vektor, der in unserem Bewegungsmodell wieder als Steuerkraft eingesetzt wird, um den Fisch in die gemeinsame Richtung zu drehen.

$$\vec{Forward}_{avrg} = \sum_{k=0}^{n} \vec{Forward}_k \cdot \frac{1}{n}$$

5.3.1.4 Separation

Als dritte Komponente im Schwarmverhalten sorgt *Separation* dafür, dass ein Mindestabstand zu den benachbarten Fischen eingehalten wird. Die Berechnung erfolgt zunächst wieder sehr ähnlich zum bereits betrachteten Cohesion-Algorithmus, zumal auch hier die durchschnittliche Position der Nachbarn betrachtet wird. Da wir eine Kollision vermeiden wollen, wird die Richtung der Steuerkraft diesmal jedoch invertiert, um von den Nachbarn weg zu zeigen. Außerdem muss der Vektor in Abhängigkeit zum Abstand skaliert werden, da sich Cohesion und Separation sonst einfach gegenseitig aufheben würden. Stattdessen muss die Ausweichbewegung umso stärker erfolgen, je kürzer der Abstand zum benachbarten Fisch ist. Dies wird erreicht, indem der invertierte durchschnittliche Ortsvektor der Nachbarn durch das Quadrat des relativen Versatzvektors dividiert wird.

$$\vec{d} = \vec{Pos}_{avrg} - \vec{Pos}_{boid}$$

$$SeparationTeilkraft = \frac{\vec{Pos}_{avrg}}{\left|\vec{d}\right|^2}$$

5.3.1.5 Einstellen der Flocking-Parameter

In der Praxis hat sich gezeigt, dass man ein natürlicheres Gesamtbild erreichen kann, wenn der Sichtwinkel und der Radius nicht pro Boid, sondern pro Teilverhalten definiert wird. Die dadurch erhöhte Kopplung der Teilalgorithmen kann außerdem helfen, CPU-Leistung zu sparen. So muss das Separation-Teilverhalten beispielsweise nur Boids berücksichtigen, die sich in der unmittelbaren Umgebung befinden, und kann alle Boids außerhalb ignorieren. Cohesion sollte dagegen über einen weitaus größeren Radius verfügen, um einem Zerfasern des Schwarms in kleinere Gruppen entgegen zu wirken. Ein eingeschränkter Sichtradius in *Cohesion* führt dazu, dass die Fische eher im „Gänsemarsch“ hintereinander her schwimmen, und so eher Ketten statt Schwärme zu bilden. Die

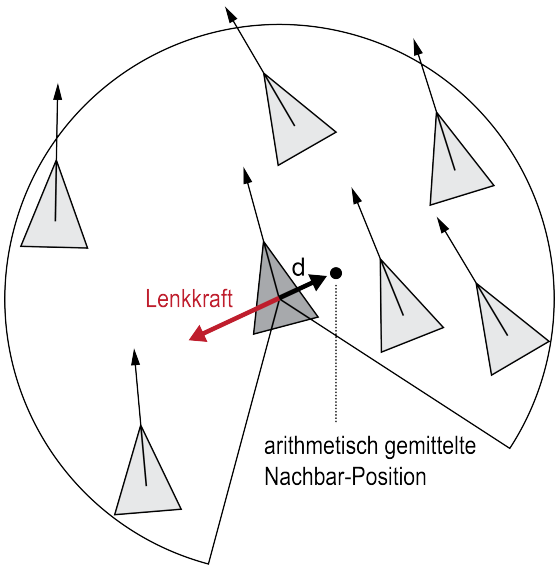


Abbildung 5.12: Berechnung der Separation-Kraft im Flocking-Algorithmus

Parameter zu *Alignment* sollten insbesondere auch im Hinblick auf die Größe des Aquariums abgestimmt werden. Zu große Sichtfelder können hier dazu führen, dass Fische, die sich durch *Containment* an einer Aquarienwand entlang bewegen, durch ihre abweichende Rotation weit entfernte Nachbarn zu stark beeinflussen. Dadurch werden Situationen begünstigt, in denen die individuellen Rotationen der Boids zu stark voneinander abweichen, als dass sich eine gemeinsame Schwimmbewegung abzeichnen könnte. Stattdessen entsteht ein unstrukturiertes Gesamtbild, bei dem die Fische ständig die Richtung wechseln und durcheinander schwimmen.

5.3.2 Swarming

Einen alternativen Ansatz zur Implementierung von Schwarmverhalten stellt das sogenannte *Swarming* dar, das im Buch *Ai for Game Developers* von David M. Bourg und Glenn Seemann beschrieben wird [BS04]. Der Swarming-Ansatz basiert auf dem sogenannten Leonard-Jones Potential. Dabei handelt es sich um eine Formel, die in der Molekularphysik angewendet wird, um die Wechselwirkung zwischen ungeladenen, nicht chemisch aneinander gebundenen Atomen anzunähern. Die Formel modelliert die Eigenschaft von Körpern, die sich in Abhängigkeit ihres Abstandes zueinander sowohl anziehen als auch abstoßen können. Im Bereich der Künstlichen Intelligenz lässt sich die gleiche Formel zweckentfremden, um sehr schnell ein einfaches Gruppenverhalten zu implementieren.

$$U = -\frac{A}{r^n} + \frac{B}{r^m}$$

In der Mechanik bezeichnet U dabei die interatomare potentielle Energie, die umgekehrt proportional zum Abstand r der Moleküle ist. Um die interato-

mare Kraft zwischen zwei Molekülen zu erhalten, muss diese Potentialfunktion abgeleitet werden, wodurch folgender Ausdruck entsteht:

$$F = -\frac{nA}{r^{n+1}} + \frac{mB}{r^{m+1}}$$

Bei den Werten A und B sowie den Exponenten n und m handelt es sich um Parameter, die frei gewählt werden können, um das gewünschte Verhalten zu modellieren. Im wissenschaftlichen Bereich würden diese Parameter in Abhängigkeit des zu modellierenden Materials angeglichen, um beispielsweise das Verhalten von Wasser oder Metall abzubilden. Die Formel besteht aus zwei Teilausdrücken, wobei der erste Term $-\frac{nA}{r^{n+1}}$ die Anziehungskomponente der Gesamtkraft modelliert und der Term $\frac{mB}{r^{m+1}}$ die abstoßende Komponente. Die letztere wirkt dementsprechend nur über relativ kleine Distanzen r, und wird für sehr kleine Werte von r sehr groß. Der negative Teil der Kurve repräsentiert die anziehende Komponente. Diese ist vergleichsweise gering, aber wirkt dafür über einen viel größeren Bereich von r. In unserem Anwendungsbereich des Schwarmverhaltens lassen sich die Parameter n und m dazu benutzen, die Flankensteilheit der Kurve einzustellen. Dadurch ist es möglich den Bereich einzustellen, in dem Anziehung oder Abstoßung dominieren. A und B kann man als Parameter für die Stärke der beiden Kraftkomponenten ansehen, n und m als Dämpfer.

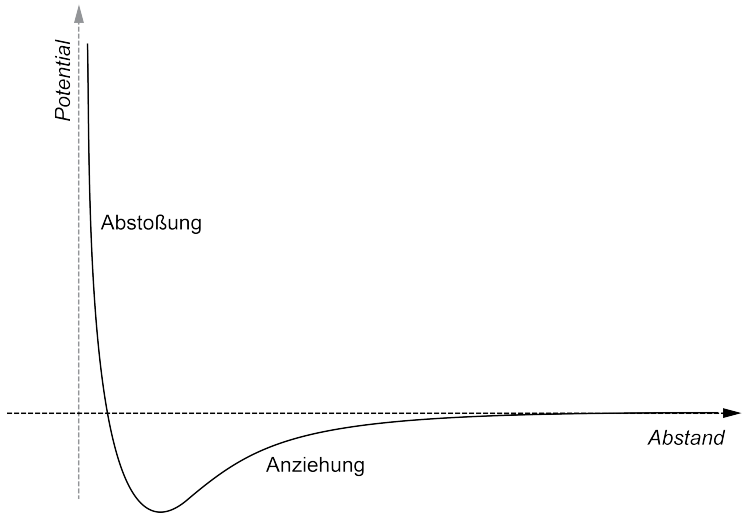


Abbildung 5.13: Lenard-Jones Potentialfunktion zur Schwarmsimulation

Ein Nachteil beim Swarming-Ansatz ist, dass die Berechnung bei einer hohen Anzahl von Objekten schnell sehr teuer werden kann. Ein weiterer Nachteil entsteht dadurch, dass nur die Positionen, aber nicht die Bewegungsrichtungen der benachbarten Objekte berücksichtigt wird. Dies führt zu einem hektisch anmutenden Gesamtbild mit permanenten starken Richtungswechseln der einzelnen Agenten. Swarming eignet sich daher gut zum Modellieren von Bienenschwärmen, aber weniger gut für die Fische unseres virtuellen Aquariums.

5.4 Kombination der Bewegungsmuster

Je nach Komplexität des simulierten Verhaltens können viele verschiedene Bewegungsmuster gleichzeitig auf einen Agenten einwirken. Hier stellt sich die Frage, nach welchem Schema all diese Teilkräfte kombiniert werden sollen, um die endgültige Steuerkraft zu erhalten. Hierzu haben sich drei verschiedene Ansätze etabliert.

5.4.1 Weighted Truncated Sum

Der einfachste Ansatz zur Kombination der Bewegungsmuster ist das einfache Aufaddieren aller Kräfte, wobei jede Teilkraft zuvor mit einer individuellen Gewichtung skaliert wird. Am Ende wird der Summenvektor auf den maximalen Betrag begrenzt, der auf den Agenten im Bewegungsmodell wirken darf (siehe Kapitel 5.1). Der Ansatz von *Weighted Truncated Sum* ist naheliegend und leicht zu implementieren, birgt aber diverse Nachteile. So kann die Berechnung bei einer hohen Anzahl an verwendeten Bewegungsmustern relativ kostspielig werden, da durch das Fehlen einer Priorisierung jede Behavior in jedem Frame komplett berechnet werden muss. Zudem gestaltet sich die Gewichtung der Bewegungsmuster als schwierig. Intuitiv neigt man dazu, wichtigeren Teilkräften wie z.B. dem Ausweichen von Hindernissen eine stärkere Gewichtung zuzuordnen, da die Vermeidung von Kollisionen in jedem Fall wichtiger sein muss, als beispielsweise das Anpeilen eines Beuteobjekts. Je niedriger hierbei die relativen Gewichtungsunterschiede sind, desto häufiger treten Situationen auf, in denen weniger wichtige Bewegungsmuster zu starken Einfluss haben. Beispielsweise kann es passieren, dass die Summe zweier niedrig gewichteter Teilkräfte wie *Wander* und *Separation* den Fisch in ein Hindernis leiten, und die Gegenkraft der *Spherical Obstacle Avoidance* nicht ausreicht, um einer Kollision entgegenzuwirken. Wählt man die Gewichtungen stattdessen zu groß, entstehen auch in unkritischen Situationen übertriebene Lenkbewegungen. Es bietet sich also eine Priorisierung der einzelnen Teilkräfte an, die ich im nächsten Ansatz vorstelle.

5.4.2 Weighted Truncated Running Sum with Prioritization

Im Gegensatz zum einfachen gewichteten Aufaddieren der Kräfte verfolgt die Technik *Weighted Truncated Running Sum* das Prinzip, dass die einzelnen Bewegungsmuster zunächst priorisiert werden. *Containment* und *Obstacle Avoidance* sollten hierbei die höchste Priorität haben, erst danach folgen Bewegungsmuster wie z.B. *Flee*. Der Algorithmus beginnt nun mit der Berechnung der am höchsten priorisierten Teilkraft und vergleicht den Betrag dieses Vektors mit der maximal möglichen Lenkkraft des Vehikels. Ist die maximale Lenkkraft noch nicht erreicht, fährt der Algorithmus mit der Berechnung des nächst-unwichtigeren Bewegungsmusters fort und addiert dessen Lenkkraft zur vorherigen. Solange die maximale Lenkkraft noch nicht erreicht ist, wird nach dem gleichen Schema die gesamte Liste nach absteigender Priorität addiert. Diese Technik hat den Vorteil, dass unwichtige Bewegungsmuster wie z.B. *Wander* in kritischen Situationen vollständig ignoriert werden. So werden Situationen vermieden, bei der sich Teilkräfte gegenseitig aufheben. Der Ansatz ist im Mittel in etwa so performant wie *Weighted Truncated Sum*.

5.4.3 Prioritized Dithering

Reynolds schlägt in seiner Veröffentlichung eine weitere Alternative zur Kombination der Bewegungsmuster vor, die sich durch eine besonders geringe CPU-Belastung auszeichnet. Ähnlich wie in *Weighted Truncated Running Sum* werden die einzelnen Bewegungsmuster zunächst nach ihrer Wichtigkeit geordnet. Allerdings wird pro Frame nur ein einziges Bewegungsmuster zur Berechnung ausgewählt. Dies geschieht sozusagen durch einen virtuellen Münzwurf, der zunächst für das am höchsten priorisierte Bewegungsmuster durchgeführt wird. War dieser Zufallswurf erfolgreich, wird die entsprechende Teilkraft berechnet und als finale Lenkkraft für dieses Frame übernommen. Schlug der Münzwurf fehl, wird nach dem gleichen Prinzip mit dem nächst-unwichtigeren Bewegungsmuster fortgefahren.

5.5 Evaluierung und Weiterentwicklung der Bewegungsmuster

Die vorgestellten klassischen Bewegungsmuster decken bereits einen großen Teil der Algorithmen ab, die zur Realisierung der in Sektion 2.2 aufgeführten Verhaltensweisen notwendig sind. Um alle Anforderungen an unsere Fisch-KI erfüllen zu können, muss das Modell jedoch an einigen Stellen angepasst und erweitert werden. So ist das von Reynolds vorgestellte *Containment* für unsere Anwendung nicht ausreichend. Meine Weiterentwicklung dazu stelle ich in Sektion 5.5.2 vor. Neue Bewegungsmuster sind zudem nötig, um die jeweils bevorzugte Aufenthalts-Tiefe einer Fischgattung einhalten zu können sowie zur horizontalen Ausrichtung der Fische in Ruhephasen. Zudem wurden kleine Anpassungen am physikalischen Bewegungsmodell vorgenommen, die ich in den folgenden Abschnitten vorstelle.

5.5.1 Anpassen des Bewegungsmodells

Das in Sektion 5.1 vorgestellte physikalische Bewegungsmodell bildet die Grundlage für meine eigene Implementierung, wurde jedoch wie folgt an die Anforderungen des Projekts angepasst:

5.5.1.1 Reibungskraft des Wassers

In öffentlichen Implementierungen zu Reynolds Modell wird die Geschwindigkeit der Agenten durch einen festen Maximalwert begrenzt, auf den der Betrag des Geschwindigkeitsvektors im Falle einer Überschreitung gekappt wird. Das hat den Vorteil, dass sich das Bewegungsverhalten von computergesteuerten Objekten sehr genau kontrollieren lässt, führt aber bei suboptimaler Einstellung der übrigen Steering-Parameter zu einem unnatürlich abrupten Abstoppen der Objekte, sobald die maximale Geschwindigkeit erreicht wurde. Da in unserem Projekt die Spielbalance keine exakte Begrenzung der Geschwindigkeit notwendig macht und wir stattdessen ein möglichst natürliches Bewegungsverhalten anstreben, steuere ich die maximale Geschwindigkeit indirekt durch die simulierte Reibungskraft des Wassers. Die maximale Geschwindigkeit wird in meinem Ansatz weiterhin als Fließkommazahl angegeben und bildet nun die

Grundlage zur Berechnung einer Bremskraft, die immer entgegen des aktuellen Bewegungsvektors wirkt und dabei proportional zur Geschwindigkeit skaliert.

```
Vector3 breakForce = - this.Velocity * ( 1.0f / this.MaxSpeed);
```

Wie ein fallender Regentropfen, der ab einer gewissen Geschwindigkeit nicht weiter beschleunigt, da die Luftreibungskräfte die Erdbeschleunigung aufheben, so wird auch die Beschleunigung unseres Fisches bis zu dem eingestellten Grenzwert vollständig aufgehoben. Im Gegensatz zu Reynolds ursprünglichem Modell wirkt die Bremskraft jedoch auch schon bei geringen Geschwindigkeiten, sodass bis zum Erreichen der Maximalgeschwindigkeit kein abrupter Eingriff in den Bewegungsverlauf stattfindet.

5.5.1.2 Akkumulator für Steuerungskräfte

Um abrupte Beschleunigungen abzudämpfen und Oszillationen zwischen Steuerungskräften zu verhindern, benutzt die OpenSteer-Implementierung einen Akkumulations-Algorithmus, der die berechnete Beschleunigung pro Frame nicht vollständig an das Modell weitergibt, sondern stattdessen nur anteilig in einen akkumulierten Beschleunigungsvektor einberechnet. Dazu kann pro Fisch eine Fließkommazahl zwischen 0.0 und 1.0 angegeben werden, welche die Gewichtung der aktuellen Kraft angibt. Bei dem Maximalwert von 1.0 wird die Kraft ungefiltert an das Modell übertragen, was vor allem bei *Flocking* und *Wander* zu einem hektischen Bewegungsverlauf führt, bei dem die Fische bei ausreichend geringer Masse wie Kompassnadeln im Magnetfeld zittern und rotieren. Niedrige Akkumulatorwerte filtern umgekehrt Beschleunigungsänderungen sehr stark und führen stattdessen zu einem ruhigeren bis trägen Bewegungsbild. Das Prinzip der akkumulierten Beschleunigung ist hilfreich, aber in unserem Fall nicht ausreichend, da es Situationen gibt, in denen unsere Fische extrem beschleunigen sollen, wie beispielsweise bei der Flucht oder beim Kampf. Aus diesem Grund wird in meinem Modell nicht die Gesamtbeschleunigung des Fisches, sondern jede einzelne Steuerungskraft mit Hilfe von linearer Interpolation akkumuliert. Dazu wurde die gemeinsame Oberklasse aller Steerings durch den Fließkomma-Parameter *forceAccumulator* erweitert. Für Bewegungsmuster, in denen die Steuerungseingriffe eher sanft erfolgen sollen (z.B. *Flocking* und *Containment*), können spontane Schwankungen geglättet werden, während Flucht und Angriff ungefiltert abgebildet werden.

5.5.2 Optimiertes Containment für Aquarien

Das von Reynolds vorgestellte *Containment* (siehe Sektion 5.2.7) ist in seiner ursprünglichen Variante mit dem einzelnen Fühler-Vektor grundsätzlich für unseren Anwendungsfall gut genug geeignet, da die glatte rechteckige Aquarienform keine feinere Tast-Auflösung erfordert. Bei flachen Näherungswinkeln treten zwar leichte Kollisionen mit der Wand auf, die in der Praxis jedoch nicht störend auffallen. Ein größeres Problem bei der klassischen Herangehensweise ist hingegen das Hängenbleiben des Fisches in lokalen Minima, d.h. den Ecken des Aquariums. Die von mir untersuchten Implementierungen behalfen sich damit, dass Ecken stets durch kleine Kanten abgeschrägt wurden, was ich in meinem Ansatz vermeiden wollte. Außerdem sollte ein Fisch seine bevorzugte Schwimmrichtung beibehalten, da ein Fisch auf Nahrungssuche ungern in die Richtung

zurückschwimmt, aus der er gerade gekommen ist. In Reynolds ursprünglichem Ansatz hat hingegen nur der Richtungswinkel Einfluss auf die Drehrichtung. Die Anforderungen für meinen Anwendungsfall waren also die folgenden:

1. kein Feststecken in Ecken
2. Beibehalten der bisherigen Schwimmbewegung (Uhrzeigersinn / Gegenuhrzeigersinn)
3. möglichst geringe CPU-Last

5.5.2.1 Position Provenance Vector

Zur Entwicklung des optimierten Containment-Algorithmus ist es nötig zu wissen, aus welcher ungefähren Richtung sich der Fisch zuletzt angenähert hat. Ein einfaches Betrachten des letzten Geschwindigkeitsvektors reicht dabei oft nicht aus, da der Fisch möglicherweise kürzlich gestoppt oder eine kurze Ausweichbewegung vollzogen hat. Stattdessen brauchen wir einen gemittelten Vektor, der die Bewegungsrichtung des Fisches in den letzten Sekunden repräsentiert, um Rückschlüsse auf seine momentan bevorzugte Schwimmrichtung ziehen zu können. Dazu verwalten wir zu jedem Fisch einen sogenannten *Position Provenance Vekor*, wie er in [Cha03] zur Implementierung von Explorations-Verhalten vorgestellt wird. Dieser Vektor $\vec{Prov}_k$ wird wie folgt berechnet und wird dabei mit jeder Positionsänderung fortlaufend aktualisiert.

$$\vec{Prov}_k = \vec{D} \cdot a + \vec{Prov}_{k-1} \cdot b$$

Bei den beiden Koeffizienten a und b handelt es sich um Parameter, mit deren Hilfe wir kontrollieren können, über welches Zeitfenster wir die Bewegungsrichtung mitteln wollen. Beide Werte können beliebig gewählt werden, solange $a + b$ den Wert 1 ergibt. Kleine Werte für a bedeuten, dass jüngste Positionsänderungen im Vergleich zur betrachteten Gesamtdauer relativ wenig Einfluss haben, wohingegen große Werte für a den jüngsten Änderungen mehr Gewicht beimessen. Der Provenance-Vektor kann auch bei anderen Bewegungsmustern hilfreich sein. Beispielsweise kann die klassische Wander-Behavior damit so erweitert werden, dass bei der Wahl der Zufallsbewegungen Bereiche im Aquarium vermieden werden, in denen sich der Fisch kürzlich bereits aufgehalten hat.

5.5.2.2 Implementierung

Ein gesondertes Eingreifen ist im erweiterten Containment nur notwendig, falls der Fisch in einer Ecke festzustecken droht. Dies kann durch einfaches Vergleichen der Koordinaten und das Setzen entsprechender Flags festgestellt werden. Auf Basis dieser Flags können nun weitere gesetzt werden, die das Erreichen der jeweiligen Ecken beschreiben:

```
stuckInCornerFrontLeft = reachedMinZ && reachedMinX;
stuckInCornerFrontRight = reachedMinZ && reachedMaxX;
stuckInCornerBackLeft = reachedMaxZ && reachedMinX;
stuckInCornerBackRight = reachedMaxZ && reachedMaxX;
```

Mit Hilfe des Position-Provenance-Vektors kann nun entschieden werden, ob der Fisch eher eine Links- oder eine Rechtsdrehung bevorzugt:

```
turnRight = false;
turnLeft = false;
if (stuckInCornerFrontLeft)
{
if (Mathf.Abs(_fish.PositionProv.x) > Mathf.Abs(_fish.PositionProv.z))
       turnRight = true;
   else
       turnLeft = true;
}
else if (stuckInCornerFrontRight)
{
    if (Mathf.Abs(_fish.PositionProv.z) > Mathf.Abs(_fish.PositionProv.x))
       turnRight = true;
    else
       turnLeft = true;
}
else if (stuckInCornerBackLeft)
{
   if (Mathf.Abs(_fish.PositionProv.z) > Mathf.Abs(_fish.PositionProv.x))
       turnRight = true;
   else
       turnLeft = true;
}
else if (stuckInCornerBackRight)
{
   if (Mathf.Abs(_fish.PositionProv.x) > Mathf.Abs(_fish.PositionProv.z))
       turnRight = true;
   else
       turnLeft = true;
}
```

Ist eines der beiden turn-Flags gesetzt, so wird die Drehrichtung im nächsten Frame ohne weitere Berücksichtigung des Provenance-Vektors beibehalten. Im Gegensatz zur *Wall Avoidance* wirkt unsere Lenkkraft nicht entlang der Oberflächennormalen, da unsere Entscheidung zur Drehung auf den ermittelten Flags basiert und nicht auf dem Winkel des Agenten zur Wand. Stattdessen wirkt unsere Lenkkraft immer orthogonal zur momentanen Ausrichtung des Fisches, solange bis die Drehung komplett vollzogen ist. Die Stärke der Lenkkraft wird aber, wie bei der *Wall Avoidance*, anhand der Durchdringungstiefe des Fühler-Vektors skaliert, um unnötig hektische Kurskorrekturen zu vermeiden.

5.5.3 Einhalten des bevorzugten Schwimmbereichs

Eine weitere Anforderung zur naturgetreueren Simulation des Fischverhaltens ist das Einhalten bestimmter vertikaler Schwimmbereiche. So hat jede Fisch-Spezies einen mehr oder weniger eingeschränkten Höhenbereich im Aquarium, in dem sich die Fische dieser Gattung am wohlsten fühlen. Während sich beispielsweise Welse und Schmerlen praktisch ausschließlich am Grund aufhalten, gehen Salmler eher in den oberen und mittleren Regionen auf Nahrungssuche. Dies muss ein Aquarien-Besitzer auch durch die Gabe unterschiedlicher Futtersorten

berücksichtigen. Um bevorzugte Schwimmbereiche unterstützen zu können, habe ich ein einfaches Bewegungsmuster namens *SteerForLevel* implementiert, das an die bereits vorgestellte *Wall Avoidance* aus Sektion 5.2.7 angelehnt ist. Zu jeder Fischgattung wird dabei je eine prozentuale Höhenangabe für den oberen und den unteren Wohlfühlbereich definiert. Ein Wert von 0% entspricht dabei dem Grund des Aquariums, 100% der Wasseroberfläche. Das untere Limit eines Salmlers liegt dementsprechend bei etwa 20%. Unterschreitet der Fisch diesen y-Wert, so wirkt eine vertikale Lenkkraft, die ihn zurück in seinen Wohlfühlbereich zieht. Die Lenkkraft skaliert dabei proportional zur Größe der vertikalen Differenz.

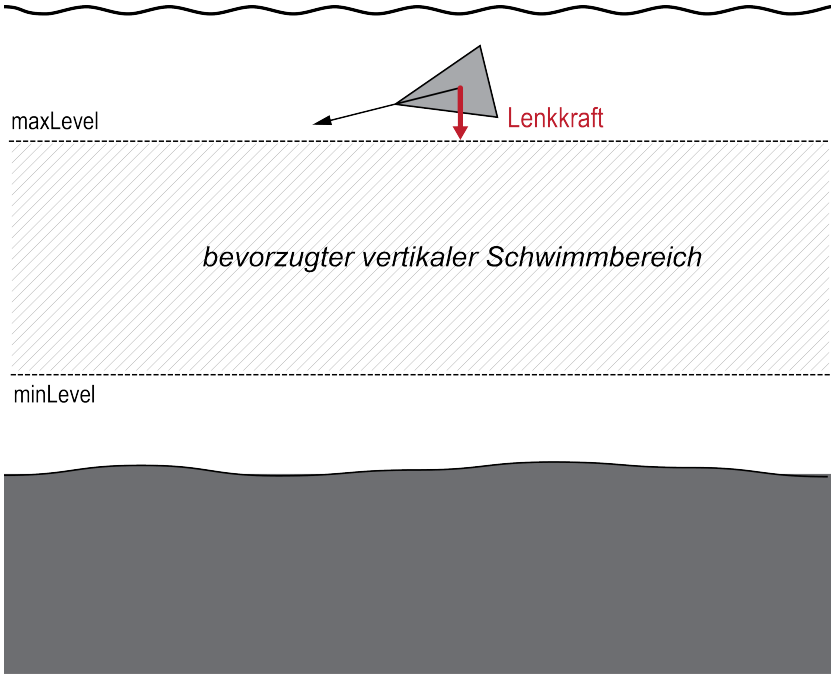


Abbildung 5.14: Lenkkraft zum Einhalten des bevorzugten Schwimmbereichs

5.5.4 Horizontale Ausrichtung des Fischkörpers

Die Ruhelage der meisten Fische ist entsprechend der Ausrichtung ihrer Schwimmblase horizontal zum Boden. Für ein stimmiges Gesamtbild muss die Ausrichtung des Fisches also spätestens zu Beginn einer Ruhephase korrigiert werden. Dazu wurde ein einfaches Steering namens *SteerToLevel* implementiert, dass eine Steuerkraft auf Basis der aktuellen Ausrichtung des Fisches vornimmt, ähnlich des im Flocking verwendeten *SteerForAlignment* (siehe Sektion 5.3.1.3). Statt der durchschnittlichen Ausrichtung der Nachbarn erfolgt die Ausrichtung des Fisches hier anhand eines Vektors, der die bevorzugte Ruhelage des Fisches beschreibt. Da die meisten Fische eine horizontale Ruhelage bevorzugen, muss hier die Differenz des normalisierten Ausrichtungsvektors des Fisches zur XZ-Ebene gewählt werden, was einfach dem y-Wert des Ausrichtungsvektors entspricht (siehe Abbildung 5.15).

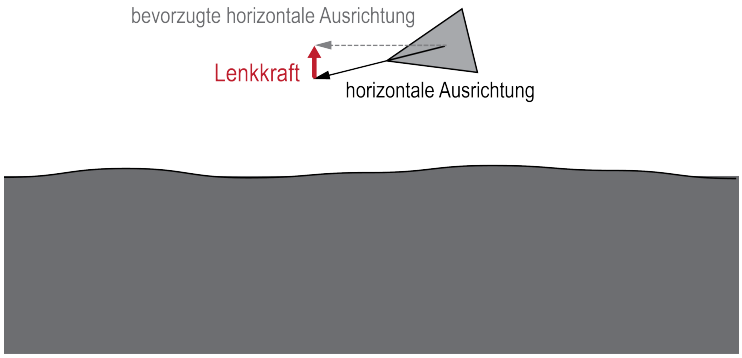


Abbildung 5.15: Lenkkraft zur Wiederherstellung der bevorzugten horizontalen Ruhelage

5.5.5 Wahl des Kombinationsalgorithmus

Nachdem wir nun über alle notwendigen Bewegungsmuster verfügen, stellt sich die Frage, welcher der in Sektion 5.4 vorgestellten Kombinationsalgorithmen für unseren Anwendungsfall am besten geeignet ist. *Prioritized Dithering* zeichnet sich durch die geringste CPU-Last aus, was jedoch durch ein deutlich unzuverlässigeres Verhalten erkauft wird. Gerade bei wichtigen Behaviors wie *Containment* oder *Obstacle Avoidance* kann es dabei schnell passieren, dass der Fisch in Hindernisse gedrückt wird, wenn korrigierende Teilkräfte zu kritischen Zeitpunkten nicht evaluiert wurden. In unserem Anwendungsfall könnte *Prioritized Dithering* deshalb nur in Kombination mit zusätzlichen Algorithmen zur Kollisionserkennung verwendet werden, wodurch wieder zusätzliche CPU-Last anfällt. *Weighted Truncated Sum* verhindert Kollisionen wesentlich häufiger, bringt aber auch entsprechend höhere Kosten mit sich, da in jeder Evaluation sämtliche Teilkräfte berechnet werden. Wir entscheiden uns für unser Aquarium deshalb für *Weighted Truncated Running Sum with Prioritization*, da diese im Mittel nicht nur sparsamer ist als die Variante ohne Priorisierung, sondern durch die feste Hierarchie im Hinblick auf ein natürliches Schwimmverhalten auch leichter zu parametrisieren ist. Eine hohe Priorisierung von *Obstacle Avoidance* macht es uns außerdem möglich, auf zusätzliche Algorithmen zur Kollisionserkennung verzichten zu können, um eine möglichst hohe Performanz erreichen zu können. Bei der Priorisierung der Bewegungsmuster liegt nahe, dass *Containment* und *Obstacle Avoidance* in der Hierarchie am höchsten anzusiedeln sind, da auch bei panischem Fluchtverhalten kein Fisch in Hindernisse oder gegen Aquarienwände schwimmen soll. Die Flee- und Evade-Behaviors folgen in der Prioritätenliste direkt dahinter, da der Fluchtreaktion auch in der Natur eine hohe Priorität zukommt. Im hinteren Drittel der Hierarchie ist das ziellose Umherstreifen anzuordnen, dass durch die Wander-Behavior realisiert wird. Dieser vorangestellt ist das Einhalten des bevorzugten Schwimmbereichs. An letzter Stelle folgt das Auspendeln des Körpers durch die Balance-Behavior, da dies nur in Ruhephasen

des Fisches notwendig ist.

1. Containment
2. Spherical Obstacle Avoidance
3. Evade, Flee
4. Seek, Pursuit
5. Separation
6. Cohesion
7. Alignment
8. Level (Einhalten des bevorzugten vertikalen Schwimmbereichs)
9. Wander
10. Balance (Einhalten der bevorzugten horizontalen Ruhelage)

Bei der Hierarchie der Bewegungsmuster ist zu beachten, dass sich die Gewichtungen und Prioritäten in Abhängigkeit der Situation und der gewählten Verhaltensweise des Fisches ändern können. Ob beispielsweise die Nahrungsaufnahme als wichtiger einzuschätzen ist als das Schwarmverhalten, hängt u.a. von der Distanz des Fisches zum Nahrungsobjekt ab. Kaum ein Fisch wird in Gefahrensituationen den Schutz des Schwarmes komplett verlassen, um ein weit entferntes Futterobjekt anzusteuern. Hier zeichnet sich bereits ab, dass in unserer Fisch-KI eine übergeordnete Instanz nötig wird, die solche Entscheidungen in Abhängigkeit von verschiedenen Faktoren trifft und die Bewegungsmuster dynamisch parametrisiert. Mit diesem Teil der KI beschäftigen wir uns in Kapitel 6.

5.5.6 Implementierung und Klassenarchitektur

Als Grundlage zur Implementierung der vorgestellten Bewegungsmuster diente die zu Anfang des Kapitels erwähnte Unity3D-Portierung von OpenSteer [Ame03]. Die Klassenstruktur meiner Implementierung (siehe Abbildung 5.16) wurde dabei grundsätzlich beibehalten und nur durch konkrete Unterklassen ergänzt. Alle Bewegungsmuster sind als konkrete Klassen realisiert, die von der gemeinsamen Oberklasse *Steerings* erben. In dieser Oberklasse ist in erster Linie das Methodeninterface zur Berechnung der Lenkkraft definiert. Jeder Agent verfügt über eine Instanz der Klasse *Radar*, die dafür zuständig ist, innerhalb eines vorgegebenen Radius die Umgebung des Agenten auf andere Fische, Hindernisse und Futterobjekte zu scannen, die dann jeweils in einer eigenen Container-Struktur verwaltet werden. Das Scannen der Umgebung ist somit die teuerste Aktion, da ihre Laufzeit $O(n^2)$ beträgt. Da diese Aktion aufgrund der eingeschränkten maximalen Fisch-Geschwindigkeit nicht in jedem Spielframe durchgeführt werden muss, kann diese Last auf mehrere Frames verteilt werden. Bei sehr langsamen Fischen reicht hier eine Aktualisierung in einem Intervall von etwa einer Sekunde, bei ruhenden Fischen sogar weniger. Die neue Unterklasse *AutonomousFish* erbt von der Oberklasse *AutonomousVehicle*, die in erster Linie das physikalische Bewegungsmodell der Agenten verwaltet. AutonomousFish wurde dabei durch die projektspezifischen Eigenschaften ergänzt, wie beispielsweise Hunger- und Stresspegel der Fische. Außerdem enthält die Klasse den KI-Code zur Wahl der auszuführenden Aktion, mit dem wir uns in Kapitel 6 ausführlich beschäftigen.

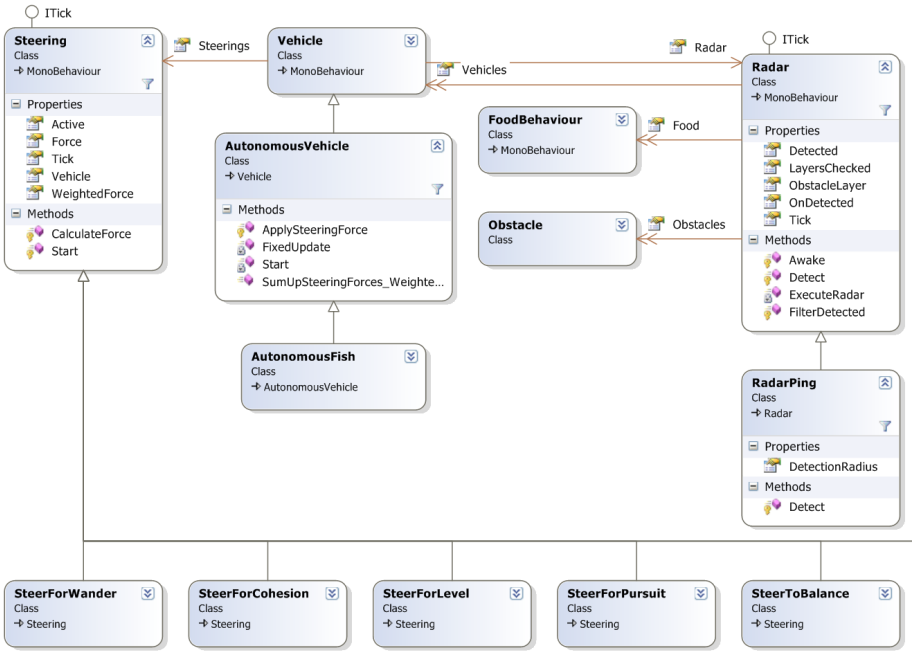


Abbildung 5.16: Klassendiagramm zur Implementierung der Bewegungsmuster

5.5.7 Laufzeit und Optimierungsmöglichkeiten

Weitere Performance-Optimierungen müssen zu einem späteren Zeitpunkt ergänzt werden, falls sich die Rechenleistung auf einer der Zielplattformen als zu gering erweist. Problematisch ist trotz der im letzten Abschnitt erwähnten Lastverteilung grundsätzlich das wiederholte Testen aller Objekte des Aquariums auf den Schnitt mit dem Radar-Bereich jedes einzelnen Fisches, woraus sich die Laufzeit von $O(n^2)$ ergibt. Auch wenn n dabei nicht die kugelförmigen Hindernisse umfasst (zumal diese keinen eigenen Radar verwalten), so kann sich eine hohe Anzahl dieser Hindernisse trotzdem stark negativ auf die Performance auswirken. Dies wird insbesondere kritisch, wenn die Form komplexerer Hindernis-Objekte aus vielen kleinen Kollisions-Kugeln nachmodelliert wird. Eine naheliegende Optimierungsmöglichkeit wäre deshalb ein geeignetes Interest-Management-System, um das Aquarium in Sektoren zu unterteilen (vgl. [BKV06]). Dadurch muss nicht der Radar eines jeden Fisches sämtliche Objekte des Aquariums wiederholt scannen, sondern kann sich auf diejenigen Objekte beschränken, die sich innerhalb der angrenzenden Sektoren seiner aktuellen Position befinden. Es bleibt die Notwendigkeit für eine einzelne zentrale Instanz, die regelmäßig die Positionen aller Objekte scannt, den jeweiligen Sektoren zuordnet und die Positionen für die Radar-Instanzen aller Fische bereit stellt. Da die Spielwelt in unserem Projekt sehr überschaubar und das Scannen eines einzelnen Objekts pro Radar auf eine einfache Vektor-Subtraktion hinausläuft, bleibt zu untersuchen, ab welcher Aquariengröße und Objektdichte eine solche Erweiterung sinnvoll wäre.

Kapitel 6

Aktionswahl

Mit einer Kombination der bisher vorgestellten Bewegungsmuster können prinzipiell alle Verhaltensweisen abgebildet werden, die in Abschnitt 2.2 gefordert wurden. Zur Futteraufnahme muss beispielsweise das Bewegungsmuster *Seek* bzw. *Pursuit* vorübergehend hoch priorisiert und auf das jeweilige Ziel eingestellt werden. Die Priorisierung der Seek-Behavior muss in diesem Fall solange beibehalten werden, bis entweder das Nahrungsobjekt erreicht ist oder vorzeitig von anderen Fischen aufgefressen wurde. Außerdem wäre denkbar, dass der Fisch während der Nahrungsaufnahme von einem übermächtigen Rivalen attackiert wird, sodass der attackierte Fisch vor Erreichen des Ziels die Flucht ergreift. Wir brauchen innerhalb unserer KI-Architekur also eine übergeordnete Ebene, die anhand der momentanen Situation solche temporären Ziele definiert, danach die Parametrisierung der Bewegungsmuster entsprechend vornimmt, und Abbruchbedingungen zur jeweils ausgeführten Aktion überprüft und behandelt. Wir untersuchen im Folgenden die gängigsten Techniken, um eine solche Logik zu realisieren.

6.1 KI-Architekturen

Eine KI-Architektur beschreibt die Art und Weise, wie die verschiedenen Teilkomponenten der künstlichen Intelligenz (z.B. Bewegungsmuster und Zieldefinitionen) organisiert und strukturiert sind, um zielgerichtetes Handeln zu ermöglichen. Dabei haben sich im Laufe der Zeit verschiedene Ansätze etabliert, die sich stark in Komplexität und geeignetem Anwendungsbereich unterscheiden.

6.1.1 Zustandsautomaten (Finite State Machines, FSM)

Ein einfacher Ansatz zur Realisierung zielorientierten Handelns innerhalb einer KI ist die Definition eines Zustandsautomaten. In unserem Beispiel befindet sich der Fisch dabei zu jedem Zeitpunkt in jeweils einem von mehreren Zuständen, die sich in ihrer Zieldefinition unterscheiden. Im wachen und leicht hungrigen Zustand könnte der Fisch beispielsweise in moderatem Tempo auf der Suche nach Nahrung oder Partnern das Aquarium durchqueren. In diesem Zustand würde dazu die Wander-Behavior relativ hoch priorisiert werden, in der Hierarchie beispielsweise direkt nach dem Ausweichen von Hindernissen und Gefahrenquellen.

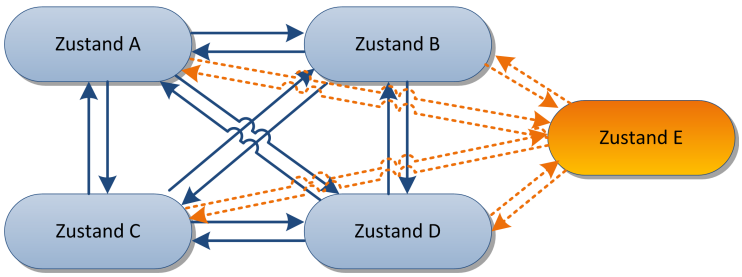


Abbildung 6.1: endlicher Zustandsautomat nach Einfügen eines neuen Zustandes

Sobald der Fisch ein Nahrungsobjekt entdeckt hat, wechseln wir in den Zustand „Nahrung ansteuern“, bei dem das Wander-Bewegungsmuster deaktiviert und Seek / Pursuit aktiviert wird. Um aus diesem Zustand in einen anderen wechseln zu können, müssen ständig mehrere Abbruch-Bedingungen überprüft werden:

- Ist das Ziel erreicht? Dann zurück zu *Wander*
- Wurde das Ziel mittlerweile von einem anderen Fisch gefressen? Dann zu *Wander*.
- Ist mittlerweile ein anderes Ziel in Sichtweite, das möglicherweise leichter zu erreichen ist? Dann dieses als neues Ziel wählen.
- Wird der Fisch angegriffen, sodass die Zielverfolgung abgebrochen werden muss? Dann Flucht oder Kampf.

Der Vorteil der Zustandsautomaten ist, dass sie sich leicht und schnell implementieren lassen. Sie eignen sich damit gut für einfache Anwendungsfälle. Das grundsätzliche Problem der Zustandsautomaten ist aber, dass die Logik zum Wechseln der Zustände direkt mit der eigentlichen Handlung des jeweiligen Zustandes verknüpft ist. Unser obiges Beispiel zum Ansteuern der Nahrung bietet dabei mindestens drei potentielle Übergänge zu anderen Zuständen. Die jeweilige diesbezügliche Entscheidungslogik wiederholt sich zu großen Teilen in anderen Zuständen. Beispielsweise wird der Angriff eines Fisches auch fast alle anderen Zustände unterbrechen. Das Hauptproblem ist dabei, dass wir beim Einführen eines neues Zustandes alle anderen Zustände überprüfen und ggf. verändern müssen (siehe Abbildung 6.1), um neue Verbindungen und Wechselbedingungen definieren zu können. Dadurch wird der Zustandsautomat sehr schnell unflexibel und fehleranfällig.

Vor- und Nachteile

Vorteile:

- Sehr leicht zu implementieren
- Verhalten exakt definierbar

Nachteile:

- Keine Trennung von Bedingung und Aktion: Redundanz, Fehlanfällig, wird schnell zu komplex

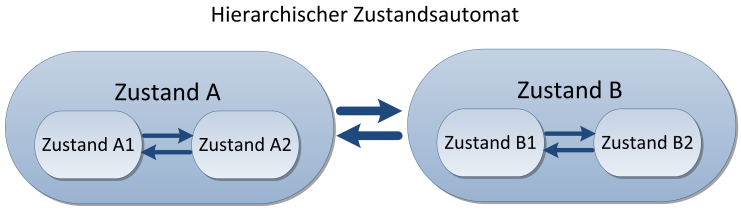


Abbildung 6.2: Hierarchischer Zustandsautomat

6.1.2 Hierarchische Zustandsautomaten (HFSM)

Um das Problem der steigenden Komplexität der Zustandsautomaten etwas besser in den Griff zu bekommen, kann man versuchen, mehrere verwandte Zustände zusammenzufassen, die sich dann eine gemeinsame Logik zum Wechseln der Zustände teilen. Bezogen auf unser Fress-Beispiel könnten beispielsweise die Teilzustände des Anpeilens der Nahrung sowie die anschließende Phase des Verspeisens unter einem gemeinsamen Oberzustand zusammengefasst werden, der dann einen Teil der gemeinsamen Abbruch-Bedingungen (Attacke durch Fressfeind) vorgibt. Da sich jeweils nur bestimmte Zustände für diese Zusammenfassung eignen, bleibt das Problem der drohenden Unüberschaubarkeit von Zustandsautomaten aber grundsätzlich bestehen.

Vor- und Nachteile

Vorteile:

- Verhalten exakt definierbar
- Etwas größere Komplexität möglich im Vergleich zu einfachen FSM
- Implementierung etwas aufwändiger als bei FSM

Nachteile:

- Noch keine konsequente Trennung von Bedingung und Aktion

6.1.3 Behavior Trees

In den im letzten Kapitel vorgestellten Hierarchischen Zustandsautomaten ließ sich bereits ansatzweise eine Trennung von Entscheidungslogik („Condition“) und Handlungsalgorithmus („Action“) erkennen. Mit den *BehaviorTrees* wird dieser Ansatz konsequent und vollständig weiterverfolgt. Erste Konzepte zu Behavior Trees wurden im Jahr 2001 im Bereich der mathematischen Softwareentwicklung von R.G. Dromey vorgestellt [R.G01] [GPHSD05]. In der Spieleentwicklung wurden erste Varianten ab 2004 in Triple-A-Spielen wie *Halo2* eingesetzt [Dyc07] und seitdem ständig weiterentwickelt. Der KI-Experte Alex J. Champandard setzte Behavior Trees u.a. für die *R.A.G.E*-Engine der Serie *Grand Theft Auto* und *Red Dead Redemption* ein und veröffentlischte mehrere Bücher und Artikel zu dem Thema [Cha07a] [Cha03] [Cha07b]. Als zusätzliche Referenzen zum Thema dienten [Kna11] und [Mar12].

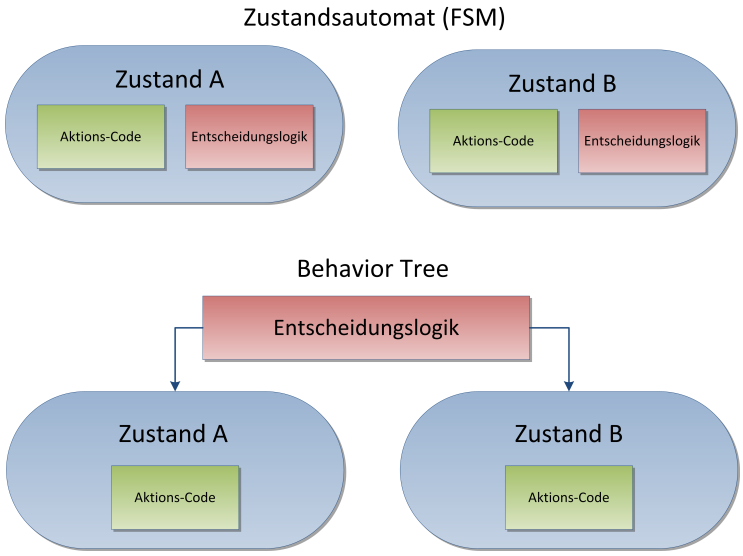


Abbildung 6.3: Entscheidungslogik in Zustandsautomat und Behavior Tree

Behavior Trees ersetzen den zu zunehmender Unübersichtlichkeit tendierenden Wust von Zuständen der FSMs durch einen restriktiveren und strukturierteren Ansatz. Die Entscheidungslogik zum Wechseln der Zustände ist komplett vom eigentlichen Aktionscode getrennt und wird in einer zentralen Baumstruktur organisiert. Durch das Entfernen der Zustandsübergänge handelt es sich bei den Verhaltensweisen somit um keine eigentlichen Zustände mehr, sondern um eine Sammlung von Aktionen, die dynamisch ausgeführt und beendet werden. Technisch gesehen handelt es sich bei Behavior Trees um gerichtete azyklische Graphen mit genau einer Wurzel. Behavior Trees werden aufgebaut indem die Teil-Verhaltensweisen einer KI hierarchisch organisiert werden. Diese Teil-Verhaltensweisen sind einzelne Unterbäume des Graphen, die ihrerseits aus verschiedenen Arten von Knoten bestehen. Diese verschiedenen Knotentypen bilden die Komponenten, mit deren Hilfe man die Traversierung des Baumes kontrolliert, um eine spezielle Verhaltenslogik zu erzielen. Bei einer Traversierung des Baumes werden die einzelnen Knoten (bzw. ihre Unterbäume) aufgerufen und entsprechend ihrer jeweiligen Semantik ausgewertet. Am Ende eines solchen Aufrufes steht pro Knoten einer der folgenden Rückgabe-Zustände:

- SUCCESS - erfolgreich beendet
- RUNNING - Die zugehörige Aktion wurde bereits gestartet und ist noch nicht beendet, und sollte im nächsten Simulationsschritt nochmals aufgerufen werden.
- FAILED - sauberer Fehlschlag ohne Nebenwirkungen
- ERROR - Es trat ein Fehler auf, dessen Nebenwirkungen explizit behandelt werden müssen.

Eine Traversierung des Baumes erfolgt immer ausgehend vom Wurzelknoten und endet jeweils in einem Blattknoten, der die aktuell auszuführende Aktion

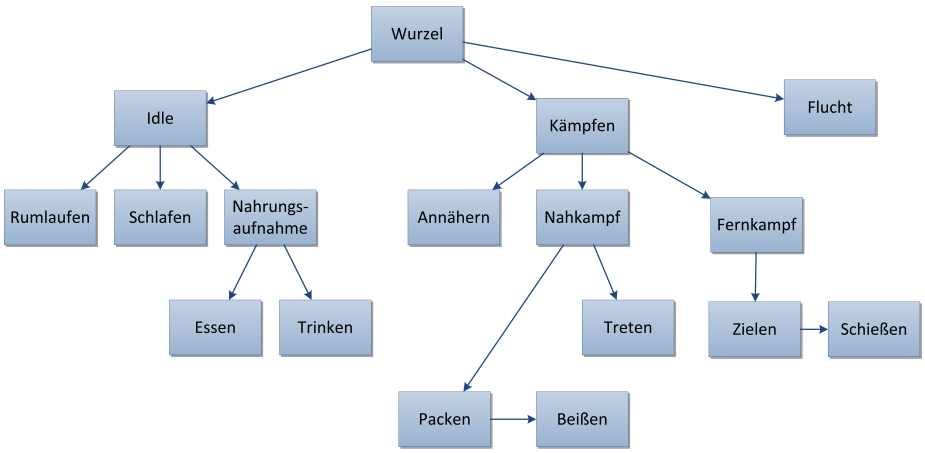


Abbildung 6.4: Beispiel für Behavior Tree

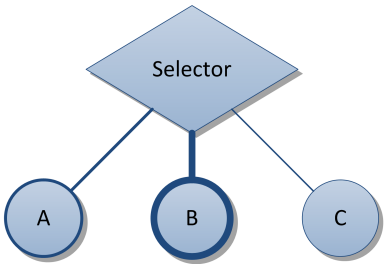


Abbildung 6.5: Behavior Tree: Selector-Knoten

repräsentiert. Benachbarte Blattknoten einer Aktion können auch Bedingungen enthalten, die zu einem Abbruch der Traversierung des jeweiligen Sub-Baums führen können. Diese Abbruchmeldung wird als *FAILED* an den Elternknoten weitergereicht, der daraufhin - falls vorhanden - mit der Traversierung des nächsten Kindknoten fortfährt, oder das *FAILED* - Ergebnis an seinen eigenen Elternknoten zurückgibt. Innerhalb von Behavior Trees existieren folgende Arten von Knoten:

6.1.3.1 Priority Selectors

In jeder Traversierung rufen Priority Selectors ihre Kindknoten nacheinander anhand einer vorgegebenen Reihenfolge auf, solange bis der erste entweder *SUCCESS* oder *RUNNING* zurückgibt. Eine von zwei Möglichkeiten dabei ist, den letzten Knoten, der *RUNNING* gemeldet hat, im nächsten Traversierungsaufruf nochmals zu besuchen. Die andere Variante ist, wieder den Kindknoten mit der höchsten Priorität aufzurufen und die letzte als *RUNNING* gemeldete Aktion implizit abzubrechen, falls sie im aktuellen Durchgang nicht erneut *RUNNING* meldet.

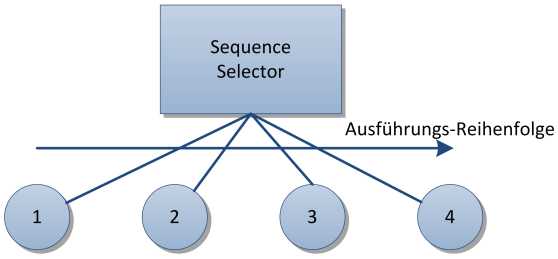


Abbildung 6.6: Behavior Tree: Sequence-Knoten zur seriellen Abarbeitung von Teilverhalten

6.1.3.2 Sequences

Diese Art von Knoten ist für die Ausführung von aufeinander aufbauenden Aktionen konzipiert. Die Sequences entsprechen einer fest definierten Liste von Anweisungen, die nacheinander abgearbeitet werden. Bei der Traversierung ruft eine Sequence zunächst den ersten Kindknoten in der Liste auf. Meldet dieser den Status *RUNNING*, so wird dieser Status an den Elternknoten der Sequence zurückgegeben ohne die übrigen Kindknoten zu besuchen. In der nächsten Traversierung wird direkt der Kindknoten besucht, der *RUNNING* gemeldet hatte. Nach einem *SUCCESS* dieses Knotens wird mit dem nächsten Kindknoten in der Liste fortgefahren. Auf diese Weise setzt sich die Ausführung fort, bis sämtliche Kindknoten erfolgreich abgearbeitet wurden. Falls mindestens einer davon *FAILED* meldet, schlägt die gesamte Sequenz fehl, und in der nächsten Traversierung wird wieder mit der ersten Kindknoten der Sequence begonnen. Im Unterschied zu Priority Selectors wird dabei also immer sofort der letzte Knoten zurückgegeben, der *RUNNING* gemeldet hat – solange bis sämtliche Knoten besucht worden sind.

6.1.3.3 Loops

Loops verhalten sich ähnlich wie Sequence Selectors, fahren aber nach Besuch des letzten Kindknotens wieder beim ersten Kindknoten fort, ohne den Rückgabewert an den Elternknoten durchzureichen.

6.1.3.4 Random Selectors

Diese Selector-Variante wählt unter seinen Kindknoten zufällig einen einzelnen aus, der besucht werden soll. Ein als *RUNNING* gemeldeter Knoten wird bei jeder erneuten Traversierung solange direkt besucht, bis er *SUCCESS* oder *FAILED* meldet.

6.1.3.5 Concurrent Selectors

Dieser Selector-Typ besucht bei jeder Traversierung sämtliche seiner Kindknoten. Eine vorher festgelegte Anzahl an Kindknoten muss einen Fehlschlag melden, damit der komplette Selector ebenfalls *FAILED* an den Elternknoten meldet. Concurrent Selectors eignen sich zur parallelen Ausführung mehrerer Ak-

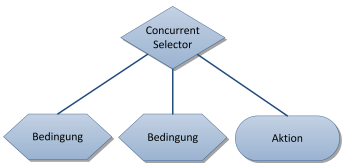


Abbildung 6.7: Behavior Tree: Concurrent-Selector

tionen, aber auch gut zur Überprüfung von Vorbedingungen für eine jeweilige Aktion. Diese Vorbedingungen werden als Blätter des Graphen repräsentiert (siehe Sektion 6.1.3.7), die in diesem Fall in beliebiger Anzahl an den Sequence Selector gehangen werden können. Letzter Kindknoten des Sequence Selectors ist dabei das Verhalten, dass im Erfolgsfall ausgeführt wird. Um bei der Implementierung Rechenleistung zu sparen, bietet sich dabei an, die Knoten anhand ihrer Fehlschlags-Wahrscheinlichkeit auszuwerten, um im Fall eines Fehlschlags des gesamten Knotens möglichst wenige Evaluierungen durchführen zu müssen.

6.1.3.6 Decorator

Diese spezielle Art von Knoten trifft im Gegensatz zu den bisher vorgestellten Knoten keine Auswahl, sondern hat normalerweise nur einen einzelnen Kindknoten. Decorator werden benutzt, um die Funktionalität von Kindknoten zu erweitern. Der Decorator benötigt dabei keine genauen Informationen über die Funktionalität seiner potentiellen Kindknoten, wodurch er modular im Graphen wiederverwendet werden kann.

Filter Diese Art von Decorator verhindert, dass sein Kind-Verhalten unter bestimmten Umständen aktiviert wird:

- Verhindere per Timer, dass das Verhalten zu oft innerhalb eines bestimmten Zeitraums ausgeführt wird
- Deaktiviere das Verhalten komplett für einen bestimmten Zeitraum
- Begrenze die Anzahl gleichzeitig laufender Verhaltensweisen

Manager & Handler Diese Decorator werden dazu benutzt, um die Ausführung von ganzen Subbäume zu kontrollieren, statt einzelner Aktionen:

- Überprüfe den Rückgabewert und starte ggf. die Ausführung des Kindknoten neu
- Speichere Informationen zu mehreren Kindknoten

Control Modifier Dieser Typ wird dazu benutzt, um den Rückgabestatus seines Subbaums zu verfälschen, um dem Elternknoten einen anderen Verlauf vorzutäuschen. Control Modifier werden in der Praxis vor allem genutzt, wenn ein Behavior Tree zur Laufzeit aufgebaut oder modifiziert wird.

- Erzwinge einen bestimmten Rückgabestatus, z.B. immer *SUCCESS*
- Täusche das Verhalten eines eigentlich fehlgeschlagenen Verhaltens als *RUNNING* vor

Meta Operations Diese Decorator werden während der Entwicklung dazu genutzt und können bei Bedarf automatisch in den Graphen eingefügt werden.

- Einfügen von Debug-Breakpoints, bei denen die Ausführung pausiert wird, um Benutzereingaben abzufragen
- Logging von Statusinformationen zur Ausführung des jeweiligen Knotens

6.1.3.7 Blätter des Behavior Trees

Die Blätter des Behavior Trees stellen die Schnittstelle zur eigentlichen Spielwelt dar. Ein Blatt gehört immer einer der beiden folgenden Kategorien an:

1. Conditions Die bereits im Absatz refConcurrentSelectors erwähnten *Conditions* repräsentieren die Überprüfung einer Bedingung, die innerhalb der Spielwelt erfüllt sein muss. Dabei kann z.B. der Status des Agenten und seiner Umgebung überprüft werden, wie z.B. das Testen auf Kollisionen.

2. Actions Actions sind diejenigen Blätter im Behavior Tree, die den Code zur Ausführung eines konkreten Teilverhaltens beinhalten, also z.B. das fortlaufende Abspielen einer Animation oder die Bewegung des Agenten in eine bestimmte Richtung.

Vor- und Nachteile

Vorteile:

- Vollständige Trennung von Entscheidungslogik und Aktionscode
- Hohe Komplexität möglich
- Übersichtlichkeit und Flexibilität dank modularem Aufbau
- relativ leicht zu implementieren und zu editieren

Nachteile:

- Die Prioritäten der Verhaltensweisen sind starr definiert

6.1.4 Planer

Als *Planer* bezeichnet man im Bereich der Künstlichen Intelligenz eine spezielle Architektur, die ähnlich wie die bisher betrachteten Architekturen pro Aufruf jeweils einen aktuellen Zielzustand liefert. Die Art und Weise, wie ein Planer zu diesem Ergebnis kommt, unterscheidet sich dabei jedoch deutlich. Ähnlich wie beim Behavior Tree wird auch beim Planer der Code, der das eigentliche Ausführen einer Aktion beinhaltet, von der Logik zum Wechseln der Zustände getrennt. Ein Planer arbeitet dabei zielgerichtet, d.h. er braucht als Vorgabe ein übergeordnetes Ziel, wie z.B. in einem Shooter das Töten des Gegners. Um dieses Ziel zu erreichen, analysiert der Planer den momentanen Zustand der Spielwelt und gleicht ihn mit seinem Repertoire an verfügbaren atomaren Aktionen ab. Über diese atomaren Aktionen kann ein Planer eine Sequenz von Teilzielen (der

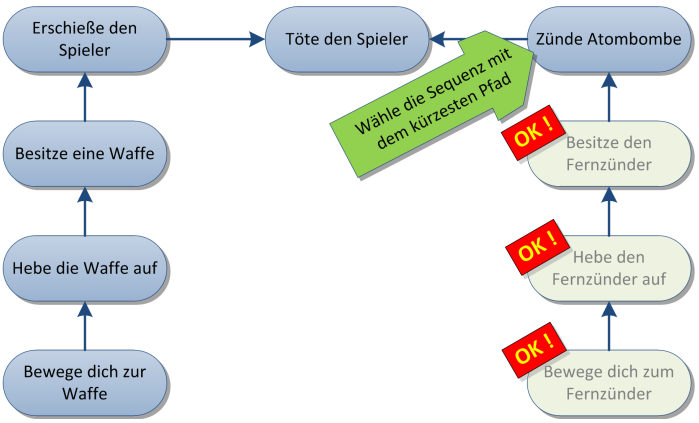


Abbildung 6.8: Planer-Architektur

„Plan“) errechnen, die letzten Endes zum Erreichen des übergeordneten Ziels führt. Von allen möglichen Sequenzen wählt der Planer dabei die kürzeste bzw. am besten geeignete. Im Gegensatz zu anderen Architekturen erfolgt die Evaluierung eines Planers also nicht ausgehend vom aktuellen Zustand des Agenten, sondern rückwärts ausgehend vom übergeordneten Ziel (siehe Abbildung 6.8). Lautet das Ziel beispielsweise „Töte den Spieler“, so könnte ein Planer beispielsweise erkennen, dass eine Möglichkeit zum Erreichen dieses Ziels „Erschieße den Spieler“ lautet. Die Voraussetzung dafür ist dementsprechend, dass sich eine Waffe im Inventar des Agenten befindet. Ist dies nicht der Fall, muss der Agent erst eine aufheben. Ist keine Waffe in der Nähe, muss der Agent sich dorthin bewegen, wo er eine vermutet. Hat er keine Information darüber, wo sich eine Waffe befindet, muss er nach einer suchen. Falls aber beispielsweise eine Aktion namens „Zünde Atombombe per Fernzünder“ existiert, die ebenfalls zum Erreichen der Vorgabe „Töte Spieler“ geeignet ist und der Agent sich bereits im Besitz des Fernzünders befindet, dann wird sich der Planer sehr wahrscheinlich für diese einfachere Sequenz entscheiden. Das Ergebnis dieser rückwärts-gerichteten Evaluierung ist ein Plan, der vorwärts ausgeführt werden kann.

Bei der Implementierung bietet die Planer-Architektur den großen Vorteil, dass neue Aktionen einfach dem Repertoire zugefügt werden können, und der Planer sie direkt verwenden kann. Dadurch verringert sich die Entwicklungszeit, und im Gegenzug steigt die Flexibilität und erreichbare Komplexität der ausführbaren Aktionen. Andererseits wird auch die Vorhersehbarkeit für bestimmte Situationen erschwert, und der Entwickler verliert mit wachsender Anzahl der Aktionen immer weiter die autoritative Kontrolle über den genauen Aktionsverlauf. In einem Zustandsautomaten oder einem Behavior Tree sind kreative Aktionen die Ausnahme. In einem Planer hingegen sind geskriptete, explizit definierte Aktionen die Ausnahme. Will man in einer bestimmten Situation immer eine fest-definierte Handlungssequenz, so muss man die dynamische Zielfindung des Planers per Skript umgehen. Die Planer-Architektur eignet sich für komplexere Aufgaben. In kommerziellen Spielen ist sie bisher nicht so weit verbreitet wie die Behavior Trees, aber wurde in sehr erfolgreichen Titeln wie z.B. *Killzone 2* [Str09]) oder *F.E.A.R* von Monolithic eingesetzt [Ork06], das in der Fachpresse insbesondere für das taktische Vorgehen der KI-gesteuerten Gegner

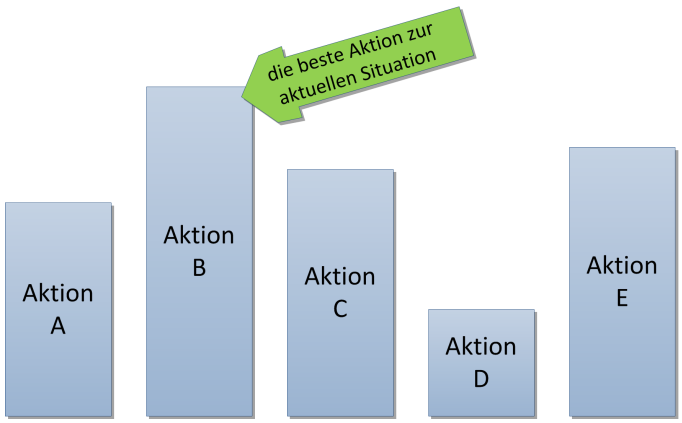


Abbildung 6.9: Utility-basierte KI-Architektur

gelobt wurde.

Vor- und Nachteile

Vorteile:

- kreativ beim Finden neuer Lösungen
- neue Aktionen lassen sich leicht einbinden
- relativ leicht zu implementieren und zu editieren

Nachteile:

- Verlust der exakten Kontrolle auf das KI-Design
- wiederholte Planungsprozesse können relativ hohe CPU-Last verursachen

6.1.5 Utility-Based AI

Eine weitere Architektur, die weniger strukturiert ist als Zustandsautomaten oder Behavior Trees, wird als „Utility-Based“ bezeichnet. Ähnlich wie ein Planer hat ein Utility-basiertes System keine fest vorgegebene Struktur der möglichen Aktionen. Stattdessen werden jeweils alle potentiellen Aktionen auf Basis der aktuellen Spielsituation betrachtet und ausgewertet. Zur Bewertung einer jeweiligen Aktion werden dazu eine Reihe von gewichteten Faktoren gegeneinander abgewogen. Gewählt wird schließlich die Aktion, die in diesem Prozess am höchsten bewertet wurde.

Utility-basierte Systeme können prinzipiell in vielen verschiedenen Spielegenres eingesetzt werden, eignen sich aber am besten in Bereichen, in denen es eine große Anzahl an Aktionen gibt, die in einer bestimmten Situation potentiell miteinander konkurrieren. In diesen Fällen ist der mathemaische Ansatz Utility-basierter Systeme nötig, um die beste Alternative evaluieren zu können. Dementsprechend kommt diese Architektur vor allem in Strategiespielen, Rollenspielen und Simulationen zum Einsatz. Das prominenteste Beispiel ist als

meistverkaufte Spieleserie aller Zeiten aber *The Sims* von Maxis. Die Entwicklung der Sims-AI ist dabei sehr gut dokumentiert (vgl. [WW01] [For02] [Eva09]). Hier war die Utility-basierte Architektur mitverantwortlich für die große Menge an Erweiterungen, die für jeden Serienteil veröffentlicht wurden. Ähnlich wie bei der Planer-Architektur können neue Aktionen und Gegenstände sehr einfach zum bestehenden Repertoire hinzugefügt werden, ohne andere Aktionen oder den grundlegenden Code zur Entscheidungsfindung verändern zu müssen.

Vor- und Nachteile

Vorteile:

- ermöglicht abwechslungsreiches Verhalten der KI
- reagiert stabil auf außergewöhnliche Situationen

Nachteile:

- Verlust der exakten Kontrolle auf das KI-Design
- schwierig zu designen, editieren und abzustimmen

6.1.6 Neuronale Netze

Einen sehr speziellen Ansatz zur Realisierung von Künstlicher Intelligenz stellen die *Neuronalen Netze* dar, die erstmals bereits in den 1960er Jahren kommerziell genutzt wurden (vgl. [BW60]). Nach Vorbild des biologischen Gehirns werden dabei Neuronen als Datenstruktur nachgebildet und zu einem Verbindungsnetzwerk verknüpft. Mit Hilfe dieses Netzwerks lassen sich diskrete, reell-wertige Funktionen berechnen. Um brauchbare Ergebnisse zu liefern, muss das Netz dazu wie beim biologischen Vorbild zunächst eine Lernphase durchlaufen. Dabei werden beispielhaft Datensätze der später zu berechnenden Art in das Netz eingespeist, zusammen mit dem jeweiligen Ergebnis. Die Informationsspeicherung verteilt sich dabei über das gesamte Netz. Neuronale Netze eignen sich gut für die Analyse verrauschter Daten und werden heute beispielsweise sehr erfolgreich bei der Erkennung von Spam-Mails oder im Bereich der Bildmuster-Erkennung eingesetzt [Bar98].
Das erste kommerzielle Computerspiel, bei dem Neuronale Netze in großem Stil zum Einsatz kamen, war *Creatures* von Steve Grand, das bei seinem Erscheinen Mitte der 1990er Jahre einen Durchbruch im Artificial Life – Segment der Künstlichen Intelligenz darstellte [SG97]. Grand kombinierte virtuelle DNS-Stränge, Biochemie und ein auf Neuronalen Netzen basierendes künstliches Gehirn. Diese Kombination ermöglichte es seinen künstlichen Kreaturen sogar, einfache englische Begriffe zu lernen, vergleichbar etwa mit dem IQ einer Hauskatze [ZED+09].
In konventionellen Spielegenres kommen Neuronale Netze dagegen vergleichsweise selten zum Einsatz, da die zu erwartenden Ergebnisse eines Anwendungsfalls oft schwer einschätzbar sind. In den übrigen vorgestellten Techniken sind die einzelnen Komponenten und Zustände leicht einsehbar und konfigurierbar, wenn auch u.U. unübersichtlich bei hoher Komplexität und schlecht gewählter Architektur. Die Neuronalen Netze hingegen sind ein geschlossenes System, das sich

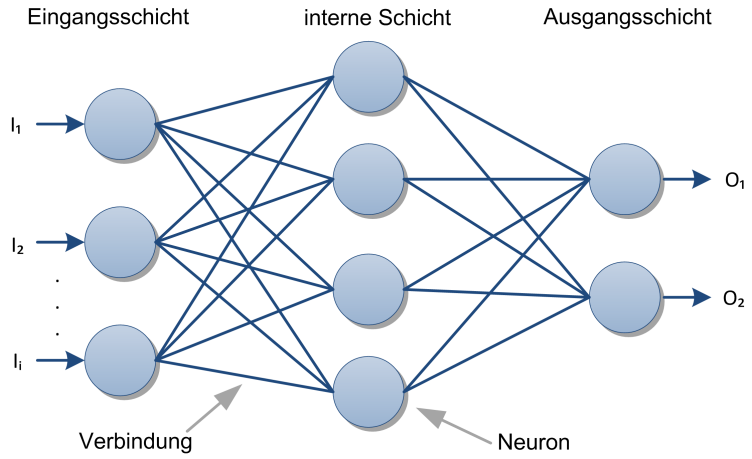


Abbildung 6.10: schematischer Aufbau künstlicher Neuronaler Netze

nur durch die Lernphase konfigurieren lässt. Diese Lernphase muss an irgendeinem Punkt als beendet erklärt werden. Danach bleibt dem Entwickler nur die Hoffnung auf ein zufriedenstellendes Ergebnis. Entsprechen dann einzelne Verhaltensweisen nicht den Erwartungen, so bleibt als einzige Option eine erneute Trainingsphase auf Basis von anderen Eingabedaten [Cha02]. Trotzdem werden Neuronale Netze erfolgreich in einigen aktuellen Spielen eingesetzt. So konnte der Entwickler Michael Robbins mit Hilfe Neuronaler Netze die taktische künstliche Intelligenz im Strategiespiel *Supreme Commander 2* (Gas Powered Games) verbessern. Der Vorteil beim Spiel gegen menschliche Spieler ist hierbei die - theoretische - nachträgliche Lernfähigkeit der KI, die aus ihren Fehlern lernt und sich damit auf neue Taktiken des Gegners einstellen kann. Durch die erwähnten Nachteile bringt dieser Ansatz in der Praxis jedoch oft so viele Probleme mit sich, dass sich Neuronale Netze oft nur für einen sehr eingeschränkten Teilbereich einer Spiele-KI eignen.

Vor- und Nachteile

Vorteile:

- ermöglicht dynamisches Lernen während des Spiels
- kann realtiv schnell eingerichtet werden

Nachteile:

- vollständiger Verlust der Kontrolle auf das KI-Design
- praktisch unmöglich zu editieren oder zu optimieren

6.2 Evaluierung der vorgestellten KI-Architekturen

Nachdem wir im letzten Abschnitt die verschiedenen Alternativen zur Architektur der Verhaltenslogik betrachtet haben, evaluieren wir diese nun im Hinblick

auf die in Kapitel 2.2 definierten Anforderungen. Neuronale Netze bieten für unseren Anwendungsfall keinen Mehrwert, da sich unsere Fische nicht grundsätzlich auf neue Situationen einstellen müssen. Umgekehrt ist die Gefahr groß, dass nach der Trainingsphase das gewünschte Verhalten nicht fehlerfrei erzielt werden kann. Aus ähnlichen Gründen scheidet auch die Planer-Architektur aus: Das Verhalten von echten Fischen basiert auf spontanen Bedürfnissen statt auf übergeordneten Zielen, die etappenweise evaluiert werden müssten. Dementsprechend schlecht bildet der Planer diesen natürlichen Entscheidungsprozess ab. Ähnlicher ungeeignet ist in unserem Fall auch der Einsatz einer Utility-basierten Architektur, da praktisch keinerlei Handlungsalternativen existieren, die gegeneinander abgewogen werden müssten. Eine Bewertung von Handlungsalternativen ist in unserem Fall nur an einigen wenigen Punkten nötig, wie beispielsweise bei der Entscheidung zwischen Flucht und Kampf. Entscheidungen wie diese können jeweils durch eine einfache Bewertungsfunktion getroffen werden, ohne die gesamte KI-Architektur an diesem Prinzip ausrichten zu müssen.

Die strukturierten KI-Architekturen der Zustandsautomaten und Behavior Trees bilden das Verhalten der Fische am übersichtlichsten und besten ab und bieten dabei den Vorteil, gezielter an einzelnen Punkten der Verhaltenslogik eingreifen zu können. Ein Zustandsautomat hat hierbei den Nachteil, dass sich bestimmte Bedingungen zu Zustandswechseln wiederholen, was prinzipiell die Gefahr von Redundanzen mit sich bringt. Zwar könnten diese durch die Implementierung von gemeinsam genutzten Bedingungs-Funktionen in überschaubaren Grenzen gehalten werden. Ich entscheide mich bei der Wahl der Architektur aber letzten Endes für den Behavior Tree, da er eine bessere Wartbarkeit mit sich bringt. Es ist nicht auszuschließen, dass die KI später noch durch zusätzliche Aktionen wie beispielsweise Brut- und Balzverhalten ergänzt wird. Diese Aktionen können dann entsprechend ihrer Priorität sehr einfach in die bestehende Entscheidungslogik eingefügt werden, ohne dass die übrigen Aktionen verändert werden müssen.

6.3 Architektur und Aufbau der Fisch-KI

Nachdem wir uns im letzten Abschnitt für den Behavior Tree als KI-Architektur entschieden haben, bauen wir den Graphen nun im Hinblick auf die in Kapitel 2.2 aufgeführten Verhaltensweisen anhand der in Sektion 6.1.3 vorgestellten Elemente auf. Es liegt nahe, als Wurzelknoten einen Selektor zu wählen, der eine grobe Auswahl der grundlegenden Verhaltensweisen vornimmt. Deren Unterbäume können später beliebig filigran über weitere Selektoren verzweigen, bis sie schließlich in einzelnen Aktionen oder Aktions-Sequenzen enden. Bei der Wahl des konkreten Selector-Typs entscheide ich mich für einen Priority Selector, was dem Graphen im Hinblick auf die Entscheidungs-Semantik eine übersichtliche Struktur verleiht.

Bei der Priorisierung der Unterbäume orientiere ich mich an der klassischen Maslowschen Bedürfnishierarchie (vgl. [JH10]), nach welcher der unmittelbare Schutz der köperlichen Unversehrtheit eines Lebewesens die wichtigste Grundlage für dessen Wohlbefinden bildet. Dementsprechend enthält der Unterbaum zur Kontrolle des Fluchtverhaltens die höchste Priorität. Die Auswertungsreihenfolge des Priority Selektors stellt dabei sicher, dass eine Panik sämtliche Aktionen

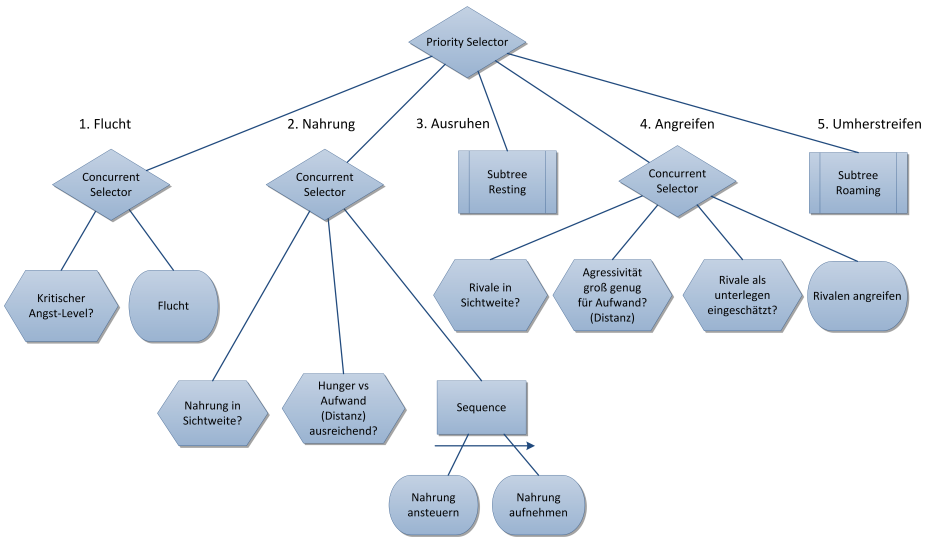


Abbildung 6.11: Hauptzweige des Behavior Trees

des Fisches unterbricht, wie in Kapitel 2.2 gefordert. Als nächstes Kind des Wurzelknotens folgt der Unterbaum zum Verhalten der Nahrungsaufnahme. Hierbei muss durch eine entsprechende Bedingung zunächst entschieden werden, ob die Aufnahme eines gesichteten Futter-Objektes den zu erwartenden Aufwand lohnt. Ein sehr hungriger Fisch wird dabei größere Wege in Kauf nehmen und ggf. den Schutz des Schwarmes verlassen. Entscheidet sich ein Fisch zur Aufnahme des Nahrungsobjekts, so wird durch einen entsprechenden Sequenz-Knoten das Objekt zunächst angeschwommen, dann verzehrt. Eine vorangestellte Bedingung überprüft dabei in jeder Traversierung, ob das angepeilte Objekt noch existiert, und/oder ob mittlerweile ein anderes Nahrungsobjekt in Sichtweite gekommen ist, das ggf. bevorzugt wird. Die Sequenz wird dementsprechend nicht weiter ausgeführt, sobald entweder keine Nahrung mehr existiert oder der Fisch ausreichend gesättigt ist.

Als weitere Kindknoten des Wurzel-Selektors folgen der Subbaum zur Realisierung von Ruhephasen, gefolgt vom Subbaum zum Angreifen von Rivalen. Als Handlungsalternative mit der niedrigsten Priorität folgt schließlich das ziellose entspannte Umherstreifen durch das Revier, das dementsprechend nur dann ausgeführt wird, wenn keines der zuvor genannten Bedürfnisse befriedigt werden kann oder muss.

6.4 Implementierung des Behavior Trees

Die gängigen Ansätze zur Implementierung von Behavior Trees haben sich in den letzten Jahren stark weiter entwickelt, wie von Alex J. Champandard ausführlich in [Cha12] dokumentiert. Mit ihrer zunehmenden Verbreitung in kommerziellen Großprojekten tendierten die Bäume dazu, immer komplexer zu werden. Die steigende Komplexität bezieht sich dabei sowohl auf die Tiefe (d.h. die Anzahl der aufeinander aufbauenden Entscheidungsebenen), als auch auf die Breite (d.h. Anzahl der Handlungsalternativen pro Subbaum) sowie die Anzahl der au-

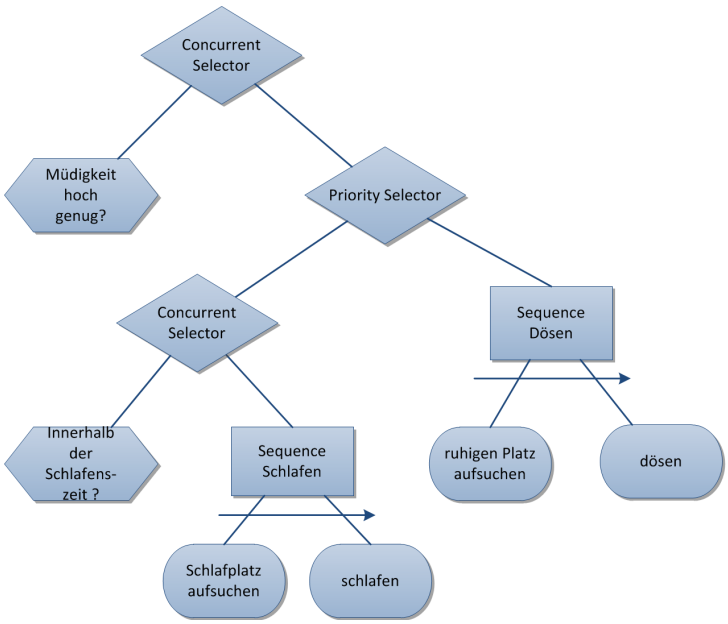


Abbildung 6.12: Unterbaum zum Regeln des Verhaltens bei Müdigkeit

tonomen Agenten im Spiel, die mit der Größe und Dichte der Spielwelt steigt. Mit der zunehmenden Komplexität der Graphen wurde vor allem ein möglichst geringer Speicherbedarf wichtig, zumal Behavior Trees vor allem auf Next-Gen Konsolen mit relativ begrenztem Arbeitsspeicher zum Einsatz kommen. Während in der ersten Generation der Behavior Trees noch sämtliche Spielobjekte eine eigene Kopie der Baumstruktur verwalteten, ist die aktuelle Generation im Hinblick auf die Vermeidung von redundanten Daten optimiert. Die Spielobjekte greifen dabei auf eine gemeinsame Baumstruktur zu und speichern nur ihre individuellen Zustandsdaten. Dieser Ansatz der Implementierung ist aufwändiger und langwieriger in der Entwicklung als der erstgenannte. Da in unserem Projekt nicht mehr als etwa 50 Fische gleichzeitig verwaltet werden müssen und die Gefahr für Speicher-Engpässe aufgrund der projektbedingt kleinen Spielwelt vergleichsweise gering ist, verzichte ich in meiner Implementierung auf den zusätzlichen Aufwand zur Speicheroptimierung. Sollte sich eine Plattform als zu ressourcenschwach erweisen, kann die Implementierung später bei Bedarf noch ausgetauscht werden, zumal sich die grundlegenden Komponenten der Architektur in den verschiedenen Generationen der Behavior Trees nicht unterscheiden. Meine Implementierung basiert dabei auf der ersten Variante der von Alex J. Champandard in [Cha12] vorgestellten Ansätze und wurde an mehreren Stellen erweitert und an den Bedarf meines Projektes angepasst. Während im Original mit Behavior, Composite, Selector und Sequence nur vier verschiedene Klassen existieren, erweitere ich in meinem Modell die Klassenstruktur um mehrere Unterklassen, wie in Abbildung 6.13 dargestellt.

In meiner Implementierung bildet die abstrakte Klasse *Behavior* die Basis für alle Verhaltensweisen, die wiederum aus beliebig komplexen Konstrukten aus Selektoren, Sequenzen, Conditions und Actions bestehen können, wie in Sektion 6.1.3 erläutert. Die Blätter unseres Graphen bilden entweder Bedingungen

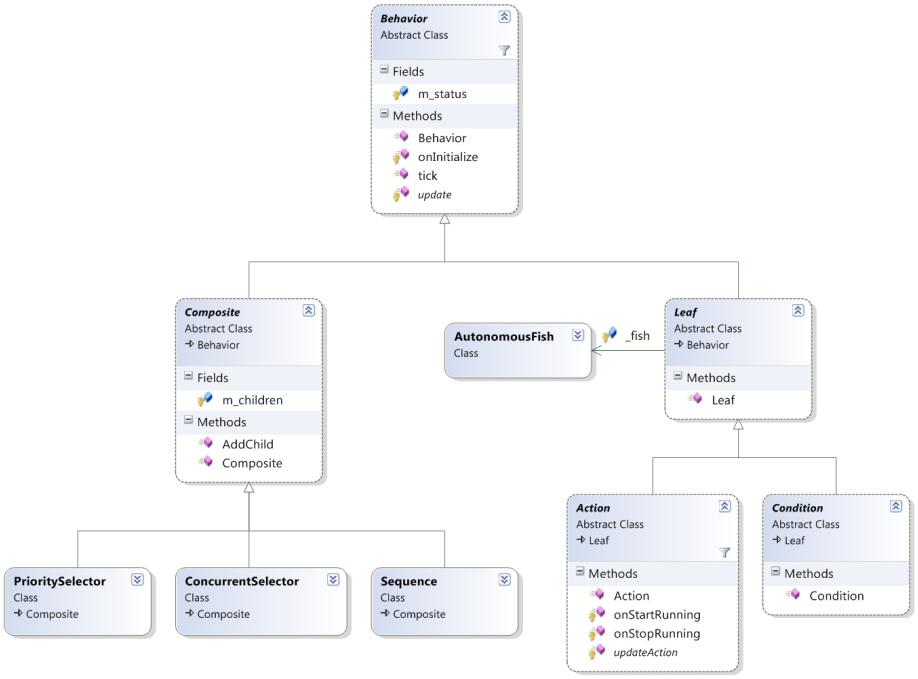


Abbildung 6.13: Klassenmodell zur Implementierung des Behavior Trees

oder Aktionen, die beide von der gemeinsamen abstrakten Oberklasse *Leaf* erben. Da die Blätter die Schnittstelle zur restlichen Spiellogik darstellen, erhält diese Klasse als einzige im Behavior Tree eine Referenz zu ihrem Agenten (repräsentiert durch die Klasse *AutonomousFish*). Da der Code zum eigentlichen Ausführen einer jeweiligen Verhaltensweise in der Klasse *Action* enthalten ist, wurde diese außerdem um virtuelle Methoden *OnStartRunning()* und *OnStopRunning()* ergänzt, die jeweils beim Beginn bzw. beim Beenden einer Aktion ausgeführt werden. Dadurch wird Code, der zur Initialisierung eines Verhaltens dient, nicht in jedem *RUNNING*-Durchgang erneut ausgeführt, wie beispielsweise das Ändern der Steering-Gewichtungen zu Beginn einer neuen Aktion.

6.5 Laufzeit und Optimierungsmöglichkeiten

Die Laufzeit meiner Implementierung unterscheidet sich mit $O(n)$ nicht von der Variante, die in aktuellen kommerziellen Spielen verwendet wird. Wie bereits im letzten Abschnitt erwähnt, lässt sich aber der Speicherbedarf des Graphen durch einen Wechsel auf die zweite Implementierungs-Generation der Behavior Trees verbessern. Eine weitere Optimierungsmöglichkeit stellt das Verteilen der Traversierungs-Last über mehrere Frames dar, ähnlich wie beim intervallbasierten Scannen der Umgebung, das in Kapitel 5.5.6 erläutert wurde. Der Nachteil ist dabei, dass die Fische verzögert auf Reize reagieren, was u.a. zu einem sichtbar verzögerten Fluchtverhalten führen kann. Eine denkbare Alternative wäre, einen Decorator-Knoten hinter den zeitkritischen Subbaum des Fluchtverhaltens zu platzieren, sodass zwar die Panik-Reaktion ständig über-

prüft wird, aber alle weniger zeitkritischen Subbäume nur in bestimmten Intervallen traversiert werden. In allen übrigen Frames wird stattdessen immer direkt diejenige Aktion aufgerufen, die zuletzt *RUNNING* gemeldet hat. Beim aktuellen Stand des Behavior Trees ist durch eine solche Optimierung allerdings nur ein verschwindend geringer Performancezuwachs zu erwarten. Erst bei einer wesentlich größeren Anzahl von Fischen oder weiteren Condition-Knoten mit teureren Berechnungen ist zu überprüfen, ab wann ein solcher Eingriff messbare Verbesserungen mit sich bringt.

Teil III

Zusammenfassung und Ausblick

Das Ziel des Projekts war, auf aktuellen handelsüblichen PCs und mobilen Endgeräten eine möglichst naturgetreue Simulation und Animation von Zierfischen zu erreichen. Zu jedem der drei Teilbereiche Animation, Navigation und Aktionswahl haben wir dazu aktuelle Techniken aus den Bereichen der Künstlichen Intelligenz / Artificial Life, Robotik und aus kommerziellen Videospielen evaluiert und adaptiert. Im Folgenden untersuchen wir, welche Teilziele im Laufe des Projekts erreicht worden sind, und welche Verbesserungen und Erweiterungen noch denkbar und wünschenswert wären.

Performance Der aktuelle, in dieser Arbeit beschriebene Stand des Projekts läuft in der nativen Version auf aktuellen gehobenen Mittelklasse-PCs mit etwa 150 Frames pro Sekunde, bei einer Darstellung von 30 Fischen, durchschnittlich dichter Aquarienbepflanzung, vollen Details unter einer Auflösung von 1920 x 1080 Bildpunkten. Eine ähnlich hohe Framerate kann auch im Webbrowser erreicht werden, falls beim Export des Projekts die native Schnittstelle von Google Chrome genutzt wird. Unter den übrigen Webbrowsern, die zur Darstellung das konventionelle Unity3D-Plugin verwenden, bricht die Framerate um etwa 40 % ein. Auf mobilen Endgeräten konnte das Projekt bisher nur sehr oberflächlich getestet werden. Es zeigte sich aber, dass der größte Teil des Performance-Einbruchs dabei auf die fehlende hardwareseitige Grafikbeschleunigung zurückzuführen ist. Mit reduzierten Grafik-Einstellungen lief das Projekt mit immerhin zweistelliger Framerate auch auf einem Android Tablet der ersten Generation (Samsung Galaxy Tab). Durch weitere Optimierungen oder notfalls einer Reduzierung der grafischen Auflösung ist eine Darstellung mit 25 Frames auch auf älteren Plattformen durchaus realistisch. Die grafische Qualität wird dann jedoch nicht mehr an die aktueller 2D-Aquarienspiele heranreichen.

Animation Das vereinfachte Feder/Masse-System in Kombination mit dem FishCurveBender-Algorithmus aus Kapitel 4.4 bietet bei geringer Prozessorlast einen zufriedenstellenden Grad an Realismus. Durch weitere Optimierungen könnte auf dieser Grundlage der optische Eindruck weiter verbessert werden. So wirkt das exakte Nachbeschreiben der Kurvenbahn momentan für Fischkörper zu flüssig und zu flexibel. Um deren Steifigkeit besser abzubilden, könnte beispielsweise der Winkel der Kurvenbiegung zwischen jedem Segment um einen noch zu bestimmenden Prozentwert oder durch eine dynamische Funktion abgeflacht werden. Ein maximal realistischer Eindruck der Lokomotion kann allerdings langfristig wohl nur mit einem komplexen physikalischen Modell nach Vorbild von Tu und Terzopoulos (siehe Kapitel 3.1) erzielt werden.

Navigation und Wegfindung Diverse Probleme ergeben sich momentan noch dadurch, dass die Navigation der Fische ausschließlich reaktiv erfolgt. Es findet also keinerlei erweiterte Wegfindung statt, die es dem Fisch ermöglichen würde, gezielt zu einem Unterschlupf oder aus einem Hindernisbereich herauszuschwimmen. Da der Spieler momentan uneingeschränkten Freiraum bei der Einrichtung des Aquariums hat, ist es deshalb leicht möglich, dass sich der Fisch beispielsweise durch Hindernisse nahe der Aquarienwand in eine Sackgasse manövriert, aus der er mit Hilfe der reaktiven Bewegungsmuster nur noch mit Glück herausfindet. Abhilfe könnten hierbei beispielsweise

Navigations-Kraftfelder schaffen, die den Fisch auf dem gewünschten Pfad an solchen Hindernissen herum führen. Diese Technik ist bereits unter dem Namen *Flow Field Following* als Bewegungsmuster in OpenSteer implementiert. Langfristig sollten sich Fische jedoch auch wie in der Realität an ihre Lieblingsplätze zurückziehen können. Dazu wird es nötig, vollwertige Wegfindungs-Algorithmen auf Basis von dreidimensionalen Navigations-Meshes zu implementieren (vgl. [Del00]). In jedem Fall wird eine Analyse-Phase des Aquariums nötig, die nach jeder Veränderung der Aquarieneinrichtung durchlaufen wird, um passend zum verwendetem Navigations-Algorithmus Kraftfelder bzw. Suchräume zu generieren.

Aktionswahl Die momentan implementierten Aktionen im Behavior Tree decken nur einen Teil dessen ab, was bis zur Veröffentlichung des Projekts unterstützt werden sollte. Aufgrund der zur Zeit noch mangelhaften Wegfindung kann der Resting-Unterbaum noch nicht vollständig implementiert werden, da u.a. ein fehlerfreies Aufsuchen von Ruheplätzen nicht garantiert werden kann. Der Graph sollte später außerdem durch zusätzliche Aktionen erweitert werden, wie beispielsweise das Balz- und Brutverhalten. Das setzt voraus, dass zuvor auch das Alter und Heranwachsen der Fische berücksichtigt wird. Der aktuelle Stand des Behavior Trees hat jedoch gezeigt, dass sich diese Architektur sehr gut für die Anforderungen unseres Projekts eignet.

Anhang A

Inhalt der Daten-CD

A.1 Sourcecode

Dieser Ordner enthält das vollständige Projekt für Unity Version 4.x. Die in der Arbeit beschriebenen Techniken befinden sich im Unterordner *Assets//Scripts//_KI*, innerhalb der folgenden Dateien bzw. Unterverzeichnisse:

AutonomousFish.cs Die zentrale Klasse zur Verwaltung der Fisch-Entitäten, mit Referenzen zu allen Bewegungsmustern, dem Behavior Tree, sowie dem in Sektion 5.4.2 beschriebenen Algorithmus *SumUpSteeringForces_WeightedTruncatedSumWithPriorization()* zum Aufsummieren der Steuerkräfte.

FishCurveBenderPositional.cs Erweiterte Version des FishCurveBender-Algorithmus zum Verbiegen des Fischkörpers anhand des Bewegungspfades, basierend auf nicht-hierarchischen Bone-Strukturen.

FishLocomotionSpring.cs Enthält den Algorithmus zur Animation der Fische nach Vorbild der BCF-Lokomotion, basierend auf dem in Kapitel 4.3.2 beschriebenen Feder-/Masse-System.

SteeringBehaviors Unterordner mit sämtlichen Bewegungsmustern, inkl. der neuen Klassen *SteerForLevel* und *SteerToBalance*

BehaviorTreePnx.cs Enthält die Implementierung des Behavior Trees der Fische mit allen konkreten Condition- und Action-Subklassen.

A.2 Demo-Builds zur Präsentation

Zur Demonstration der von mir implementierten Techniken habe ich in Unity drei verschiedene Test-Szenen erstellt und als Standalone-Builds für Windows exportiert.

A.2.1 Demo BehaviorTree

Dieser Build enthält eine Szene mit einem vollständig eingerichteten Aquarium und mehreren Fischen. Diese Szene eignet sich gut, um die Bewegungsmuster-Klassen und den Behavior Tree in Aktion zu beobachten:

- Die kleinen blauen Neon-Tetras bleiben in kleinen Gruppen zusammen oder bilden einen großen Schwarm (Bewegungsmuster *SteerForAlignment, SteerForCohesion, SteerForSeparation*).
- Scharmfische, die ihre Nachbarn aus dem Sichtfeld verloren haben, bleiben stehen bis sie wieder 'abgeholt' werden.
- Die Fische der übrigen Gattungen durchstreifen zufallsbasiert das Aquarium (*SteerForWander*).
- Alle Fische weichen Hindernissen und den Aquarienwänden aus (*SphericalObstacleAvoidance, SteerForContainment*).
- Die gelb-schwarze Schmerle bevorzugt die tieferen Regionen im Aquarium (*SteerForLevel*).
- Nach Anwählen des Fütterungstools (Taste '1') können mit der linken Maustaste einzelne Futter-Objekte ins Aquarium geworfen werden, die von hungrigen Fischen sehr schnell angeschwommen werden (*SteerToSeek*). Für relativ satte Fische reduziert sich der Aktionsradius zum Anschwimmen von Futterobjekten (BehaviorTree: *Condition_IsFoodInterestingEnough*).
- Fische, die durch Klopfen aufgeschreckt werden (per Hand-Tool, Taste '3'), unterbrechen alle anderen Aktionen und fliehen vor der Klopf-Position (Bewegungsmuster: *SteerToFlee*, BehaviorTree: Flucht-Subbaum mit *Condition_CheckForPanicReaction*).
- Alle Fische benutzen den in Kapitel 4.3.2 beschriebenen Feder-/Masse-basierten Animations-Algorithmus, der in der Klasse *FishLocomotionSpring* implementiert ist. Der Algorithmus zum Verbiegen der Fischkörper in Kurven wird hingegen von den verwendeten Modellen noch nicht unterstützt. Für ihn existiert eine eigene Szene mit einem speziell angepassten Modell (siehe Sektion A.2.3).

Tastenbelegung

C	Kameraperspektive wechseln
X	reinzommen
Y	rauszoomen
Return	Photo-Modus beenden

A.2.2 Demo FishLocomotionSpring

Diese Szene enthält einen einzelnen Fisch, für den alle Bewegungen deaktiviert sind, um die Klasse *FishLocomotionSpring* zu testen, in der das Feder-/Masse-System aus Kapitel 4.3.2 implementiert ist. Durch Drücken (und Halten) der Taste 'A' bzw. 'D' wird ein Kontraktions-Impuls an die linke bzw. rechte Zentrierungsfeder des Kopf- Segments geschickt. Bei abwechselndem Drücken in der Eigenfrequenz des Feder-/Masse-Systems kann so eine Sinuswelle durch den Fischkörper geschickt werden, die achtern über die Schwanzflosse auswandert.

Tastenbelegung

A	Kontraktion links
D	Kontraktion rechts

A.2.3 Demo FishCurveBender

In dieser Szene bewegt sich ein einzelnes Test-Modell per Wander-Bewegungsmuster durch das Aquarium, wobei der FishCurveBender-Algorithmus die einzelnen Segmente der Wirbelsäule anhand des zuletzt beschriebenen Bewegungspfades ausrichtet, wie in Kapitel 4.4 beschrieben. Die Berechnung erfolgt dabei grundsätzlich unter Einberechnung der horizonalen Auslenkung des FishLocomotionSpring-Algorithmus, wie in Abbildung 4.9 gezeigt. Im Gegensatz zum letztgenannten Demo erfolgt der Kontraktionsimpuls hier jedoch automatisch. Der Anteil des Lokomotions-Effekts liegt nach dem Laden der Szene jedoch noch bei 0 %. Durch Drücken und Halten der Taste 'E' kann der Einfluss des FishLocomotionSpring-Anteils erhöht werden. Zum Erreichen des vollständigen Effektanteils muss die Taste etwa drei Sekunden lang gehalten werden. Durch Drücken und Halten der Taste 'Q' lässt sich der Effekt wieder reduzieren.

Tastenbelegung

Q	Effekt des Feder-/Masse-Systems verringern
E	Effekt des Feder-/Masse-Systems erhöhen
F	Kamera folgt dem Fisch (an/aus)
X	reinzoomen
Y	rauszoomen
C	Kamera-Perspektive wechseln

Literaturverzeichnis

[Ame03] Sony Computer Entertainment America. Open steer. http://opensteer.sourceforge.net, 2003.

[Bar98] Marian Stewart Bartlett. *Face image analysis by unsupervised learning and redundancy reduction.* PhD thesis, University of California, San Diego, 1998. AAI9907603.

[BKV06] Jean-Sébastien Boulanger, Jörg Kienzle, and Clark Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, New York, NY, USA, 2006. ACM.

[BS04] David M. Bourg and Glenn Seemann. *AI for Game Developers*, pages 83–86. O'Reilly Media, Inc., 2004.

[Buc05] Matt Buckland. *Programming Game AI by Example*, pages 104–106. Wordware Publishing, Inc., 2005.

[BW60] Marcian Hoff Bernhard Widrow. Adaptive switching circuits. *Proceedings WESCON. ZDB-ID 267416-6, p. 96 - 104*, 1960.

[Cha02] Alex J. Champandard. *The Dark Art of Neural Networks*, pages 640–651. Charles River Media, 2002.

[Cha03] Alex J. Champandard. *Ai Game Development: Synthetic Creatures With Learning and Reactive Behaviors*, pages 116 – 118. New Riders, 2003.

[Cha07a] Alex J. Champanard. Assaulting f.e.a.r.'s ai: 29 tricks to arm your game,. *aigamedev.com/reviews/fear-ai*, 2007.

[Cha07b] Alex J. Champandard. Understanding behavior trees. http://aigamedev.com/open/article/bt-overview/, 2007.

[Cha12] Alex J. Champandard. Understanding the second-generation of behavior trees. http://aigamedev.com/insider/tutorial/second-generation-bt/, 2012.

[Del00] Mark Deloura. *Simplified 3D Movement and Pathfinding Using Navigation Meshes.* Charles River Media, 2000.

[Dyc07] Max Dyckhoff. Evolving halo3's behavior tree. Technical report, Game Developer's Conference, 2007.

[Eva09] Richard Evans. Ai challenges in sims 3. *AIIDE09*, 2009.

[For02] Kenneth Forbus. Simulation and modeling: Under the hood of the sims. *http ://www.cs.northwestern.edu/forbus/c95-gd/lectures/TheSims Under the Hood files/v3 document.htm*, 2002.

[Frö97] Torsten Fröhlich. Das virtuelle ozeanarium, thema forschung 2/97. Technical report, Technische Universität Darmstadt, 1997.

[Frö00] Torsten Fröhlich. The virtual oceanarium - a visual computer simulation of europe's largest aquarium. Technical report, Communications Of ACM, 2000.

[GPHSD05] Cesar Gonzalez-Perez, Brian Henderson-Sellers, and R. Geoff Dromey. A metamodel for the behavior trees modelling technique. In *ICITA (1)*, pages 35–39, 2005.

[GVL96] Bruce C. Jayne George V. Lauder. Pectoral fin locomotion in fishes: Testing drag-based models using three-dimensional kinematics. In *Amer. Zool. 36*, pages 567–581, 1996.

[HG06] Rüdiger Riehl Hartmut Greven. *Verhalten der Aquarienfische 1.* Birgit Schmettkamp Verlag, 2006.

[J.G32] J.Gray. Studies in animal locomotion i. the movement of fish with special reference to the eel. *Laboratory of Experimental Zoology, Cambridge*, 1932.

[JH10] Heinz Heckhausen Jutta Heckhausen. *Motivation und Handeln*, chapter Kapitel 3.3.3: Das Hierarchie-Modell von Maslow. Springer Verlag, 2010.

[Kla97] Roger Klambauer. Verhalten künstlicher lebewesen im virtuellen ozeanarium. Technical report, Fachhochschule Darmstadt, Fachbereich Informatik, 1997.

[Kna11] Bjoern Knafla. Introduction to behavior trees. http://bjoernknafla.com/introduction-to-behavior-trees, 2011.

[Lin78] C. C. Lindsey. *Fish Physiology Vol. VII - Locomotion. Form, function and locomotory habits.* Academic Press, Inc. (London) LTD., 24/28 Oval Road, London NW1 7DX, 1978.

[Mar12] Dave Mark. Ai architectures - what's on the menu? In *Game Developer Magazine August 2012*, pages 7 – 12, 2012.

[M.B26] M.Breder. The locomotion of fishes. In *Zoologica, vol. 4*, pages 159 – 256, 1926.

[Mül97] S. Müller. Das virtuelle ozeanarium für die expo '98 in lissabon. In *Berichtsband zum Symposium Hannover: EXPO 2000*, pages 185–193, 1997.

[MS99] J. Bruce C. Davies Michael Sfakiotakis, David M. Lane. Review of fish swimming modes for aquatic locomotion. In *IEEE Journal of oceanic engineering, Vol. 24, No. 2*, pages 237 – 252, 1999.

[Nür89] Günther Nürnberger. *Approximation by Spline Functions.* Springer Verlag, 1989.

[Ork06] Jeff Orkin. Three states and a plan: The a.i. of f.e.a.r. *Game Developer's Conference*, 2006.

[Rey87] C. W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. In *Computer Graphics (SIGGRAPH '87 Conference Proceedings)*, volume 21, pages 25–34, 1987.

[Rey99] Craig W. Reynolds. Steering behaviors for autonomous characters. In *Proceedings of GDC 1999*, pages 763–782, 1999.

[R.G01] R.G.Dromey. Genetic software engineering - simplifying design using requirements integration. Technical report, IEEE Working Conference on Complex and Dynamic Systems Architecture, 2001.

[SG97] Anil Mahotra Stephen Grand, Dave Cliff. Creatures: Artificial life autonomous software agents for home entertainment. Technical report, Agents '97, Marina Del Ray, California, 1997.

[Str09] Remco Straatman. The ai in killzone 2's bots: Architecture and htn planning.
http://aigamedev.com/premium/presentations/killzone2-planning/,
2009.

[TT94] Xiaoyuan Tu and Demetri Terzopoulos. Artificial fishes: physics, locomotion, perception, behavior. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pages 43–50, New York, NY, USA, 1994. ACM.

[Vie94] Dr. Jörg Vierke. *Fischverhalten beobachten und verstehen.* Franck Kosmos, 1994.

[Web73] P.W. Webb. Kinematics of pectoral fin propulsion in cymalogaster aggregate. In *J. Exp. Biol. vol. 59*, pages 697 – 710, 1973.

[WW83] P. W. Webb and D. Weihs. *Fish Biomechanics*, chapter Optimization of locomotion, pp. 339/371. New York: Praeger, 1983.

[WW01] Kenneth Forbus Will Wright. Some notes on programming objects in the sims. *http ://www.qrg.cs.northwestern.edu/papers/Files/Programming Objects in The Sims.pdf*, 2001.

[ZED+09] Marjorie A. Zielke, Monica J. Evans, Frank Dufour, Timothy V. Christopher, Jumanne K. Donahue, Phillip Johnson, Erin B. Jennings, Brent S. Friedman, Phonesury L. Ounekeo, and Ricardo Flores. Serious games for immersive cultural training: Creating a living world. *IEEE Computer Graphics and Applications*, 29(2):49–60, March/April 2009.