Universität Koblenz, Akademisches Jahr 2003/2004

Simulation von Schwarmverhalten autonomer Objekte

Entwicklung einer Programmbibliothek

Studienarbeit

Nils Hornung
Fachbereich 4
Universität Koblenz-Landau,
Abteilung Koblenz

Ziel der vorliegenden Studienarbeit ist es, eine Programmbibliothek zur Entwicklung von Schwarmverhalten zu konzipieren und zu entwickeln. Dabei steht die Erweiterbarkeit und Nutzbarkeit dieses APIs für andere Anwendungen im Vordergrund. Eine einfache Beispielapplikation zeigt prototypisch die Möglichkeiten der Programmbibliothek.
Schwerpunkte der Arbeit sind eine Diskussion der Literatur zum Thema Schwarmverhalten bzw. *Flocking, Herding, Schooling* sowie eine Besprechung des Konzeptes der entworfenen Programmbibliothek. Dabei wird auf relevante Datenstrukturen, Schnittstellen und Simulationsmethoden eingegangen. Desweiteren werden Aspekte der Implementierung und Benutzung des APIs besprochen. Zusammenfassend werden Erweiterungsmöglichkeiten behandelt.

Endversion Abgabe.

Inhaltsverzeichnis

Institut für Computervisualistik
AG Computergraphik
Universität Koblenz-Landau,
Abteilung Koblenz

Betreuer:
Prof. Dr. Stefan Müller
Tel.: 0261–287–2727
stefanm@uni-koblenz.de
www.uni-koblenz.de/cg

1 Überblick über das Themengebiet

In den vergangenen Jahren lag das Augenmerk bei der Produktion digitaler visueller Effekte in Spielfilmen darauf, photorealistische Bilder zu erzeugen [Apodaca und Gritz 2000, S. 3 ff]. Darunter verstand man in erster Linie den Einsatz physikalisch begründeter Beleuchtungsmodelle in komplexen Szenen. Ähnlich beschränkten sich virtuelle Umgebungen auf statische *Walk-Through*-Welten. Aufwendige Beleuchtungssituationen wurden vorberechnet, eine dynamische Änderung der Szene war nicht möglich.

Verbesserte Computertechnologien eröffnen nun neue Möglichkeiten, sowohl für den *Offline*-Einsatz der Computergraphik in der Filmindustrie, als auch für die echtzeitfähige Darstellung hochdynamischer Szenarien. Immer häufiger finden Simulationstechniken Anwendung in Computergraphik-Software. Dabei werden Methoden aus der Physik — etwa bei der Bewegungssimulation — sowie aus der Künstlichen Intelligenz — etwa bei der Verhaltensgenerierung — an die Bedürfnisse der Computergraphik angepasst.

Da verbesserte Prozessoren nicht nur eine höhere Dynamik einer Szene erlauben, sondern auch eine höhere Komplexität, ergeben sich hier neue Herausforderungen. Neben der Simulation einzelner Objekte ist die autonome und koordinierte Bewegung ganzer Objektgruppen wie etwa bei Tierherden und -schwärmen oder bei Menschenmassen zu einer zentralen Fragestellung geworden. Es ist nicht verwunderlich, dass fast jede große internationale Kinoproduktion mit Massenszenen aufwartet (vgl. die „Herr der Ringe“-Trilogie, „Troja“) oder mit Vogel-, Fisch, und sogar mit Vampirschwärmen (vgl. „The Day after Tomorrow“, „Findet Nemo“, „Van Helsing“). Computerspiele bedienen sich schon länger Strategien der Künstlichen Intelligenz für zentrale Spielkomponenten. Inzwischen werden Vogel- und Fischschwärme auch zur reinen Aufwertung der Spieleszenerie eingesetzt (vgl. „Far Cry“).

Ziel dieser Studienarbeit ist die Entwicklung und Umsetzung einer Programmbibliothek zur Simulation von Schwarmverhalten. Zentraler Schwerpunkt ist dabei die Erweiterbarkeit und Nutzbarkeit des APIs für andere Anwendungen. Eine Beispielapplikation dient als Leitfaden für die Benutzung der Schnittstelle zur Programmbibliothek. Die Arbeit ist folgendermaßen gegliedert. Zunächst wird ein Überblick über das Themengebiet gegeben. Dabei wird auf bisherige Arbeiten und Veröffentlichungen zum Thema „Schwarmverhalten“ eingegangen. Im weiteren Verlauf werden Konzept, Implementierung und Benutzung der entwickelten Programmbibliothek vorgestellt. Die Besprechung umfasst die relevanten Datenstrukturen, Simulationsmethoden und Schnittstellen. Zum Schluss werden eine Reihe von Anregungen festgehalten, wie die Programmbibliothek erweitert und eingesetzt werden könnte.

1.1 Begriffsklärung

Zunächst folgt allerdings eine Erläuterung der verwendeten Begrifflichkeiten. Ein Objekt ist autonom, wenn es sich zum Teil selbständig verhält. Das Verhalten umfasst dabei insbesondere die eigene Positionierung und Bewegung in seinem statischen und dynamischen Umfeld. Die Bewegungen sind Beschränkungen unterlegen und gehorchen Gesetzmäßigkeiten. So ist die Bewegung eines Fahrzeugs i. A. auf die Erdoberfläche beschränkt. Es kann seine Orientierung nur ändern, d. h. lenken, solange seine lineare Geschwindigkeit ungleich Null ist, wenn es also nicht stillsteht. Das Verhalten eines autonomen Objektes geht auf äußere und innere Zustände ein, ist also *reaktiv*. Von außen kann das Verhalten eines Beutetieres z. B. durch die Anwesenheit eines Räubers beeinflusst werden. Das Verhalten des Räubers kann von dessen innerem Zustand abhängen. Ist er hungrig, wird er auf Jagd gehen.

Bedingt das reaktive Verhalten mehrere autonomer Objekte ein homogenes Verhalten als Gruppe, spricht man von kollektivem Verhalten (engl. *collective* oder *aggregate behaviour*) und speziell von Schwarmverhalten (engl. *flocking*, *herding* oder *schooling*). Anstelle von Objekten ist oft von Agenten oder Charakteren die Rede.[1] Beispiele sind Fische, Vögel und Insekten, aber im allgemeinen Sinne auch Herden- bzw. Rudeltiere oder Menschen.

1.2 Anwendungsgebiete

In den einleitenden Bemerkungen wurden zwei traditionelle Anwendungsfelder der Computergraphik erwähnt, die Film- und Fernsehindustrie einerseits und die Computerspiele-Industrie andererseits [Hägele 2003; Tu 1999; Reynolds 1999; Blumberg und Isla 2002]. Das Verhalten bewegter Objekte obliegt in der Filmindustrie traditionell dem Animator. Solche Objekte sind nicht autonom. Als Beispiel soll hier das *Keyframing*-Verfahren genannt werden. Es entspricht im Wesentlichen dem Vorgehen in der klassischen Zeichentrickproduktion. Auch das *Motion Capturing* gehört in diese Kategorie.

Auf ähnliche Weise gesteuerte Charaktere gibt es auch in Computerspielen. Reynolds [1999] weist in diesem Zusammenhang auf den *Avatar* hin, ein vom Spieler in Echtzeit gesteuertes Objekt, etwa ein Rennwagen. Typisch für Computerspiele sind aber besonders Charaktere, die ihr eigenes Verhalten selbständig improvisieren. Solche Charaktere sind autonom. Als frühes Beispiel gelten die „Sims". Heute finden sich in fast jedem Spiel autonome Charaktere, etwa als Gegenspieler.

Übernommen werden in der Spieleprogrammierung zum Teil Erkenntisse aus der künstlichen Intelligenz und der Roboterentwicklung. Zum Einsatz kommen u. a. evolutionäre Algorithmen, neurale Netze oder Zustandsautomaten. Zugrunde liegen oft Ergebnisse der Biologie und der Verhaltensforschung.

Da, wie oben gezeigt, immer mehr computeranimierte Charaktere und Objekte als sogenannte *Special Effects* in Spielfilmproduktionen eingesetzt werden, übernimmt man auch dort — v. a. für Massenszenen — das Konzept autonomer Agenten aus den genannten Disziplinen. Vorreiter waren die Pinguine in „Batmans Rückkehr" und die Stampede einer Herde Gnus im „König der Löwen". S. [Reynolds 2004]. Aktuell finden die Visualisierungen von Menschenmassen im „Herrn der Ringe" in den Medien Resonanz.

Außer in der Unterhaltungsindustrie wird autonomes Verhalten für die Vorhersage von Gruppenphänomenen simuliert. Z. B. spielt man Szenarien von Panik bei einem Rockkonzert durch. In der Biologie werden Simulationen genutzt, um theoretische Verhaltensmodelle zu überprüfen.

2 Konzepte von Schwarmverhalten in der Literatur

Die erste wichtige Arbeit zur Visualisierung und Simulation von Schwarmverhalten autonomer Agenten stammt von Craig Reynolds [1987]. Sie wird von fast jeder Computergraphik-Veröffentlichung zu Schwarmverhalten zitiert und legt damit eine wichtige Basis für die-

[1] Die Arbeit unterscheidet diese Begriffe nicht wesentlich.

ses Forschungsgebiet. Reynolds entwickelte Software zur Verhaltenssimulation u. a. für *Electronic Arts*, *Silicon Studio*, *DreamWorks Animation* und *Sony Computer Entertainment* [Reynolds 2004]. In Reynolds' Arbeit werden grundlegende Bestandteile eines Systems beschrieben, das Schwarmverhalten anhand autonomer Agenten (*bird-like objects*, *boids*) simuliert. Sein Modell besteht aus den folgenden vier Modulen.

Wahrnehmung. Das Wahrnehmungsmodul macht dem boid i Informationen über dessen nahe Umgebung $S(i)$ verfügbar. Bei Reynolds werden die Grenzen der Wahrnehmung primär durch einen Wahrnehmungsradius beschrieben. In $S(i)$ liegen diejenigen Objekte, die im Bereich des angegebenen Radius um den boid i liegen. Zusätzlich schlägt Reynolds [1987] vor, Objekte im Bereich vor dem boid stärker zu berücksichtigen.[2]

Verhalten. Heuristische Verhaltensregeln bedingen für jeden boid einen Kraft- oder Beschleunigungsvektor, den sogenannten Steuerungsvektor (engl. *steering*). Dieser 3D-Vektor ist abhängig von der vom boid wahrgenommenen Umgebung. Der Steuerungsvektor bewirkt ein Fortbewegungsverhalten konform der Verhaltensregeln.

Kinematik. Ein vereinfachtes physikalisches Modell des boid berechnet seine lineare Bewegung und seine Orientierung im Raum auf Basis der Beschleunigung oder Steuerungskraft des Verhaltensmoduls. Für die lineare Komponente verwendet Reynolds die NEWTONschen Bewegungsgleichungen mit expliziter numerischer EULER-Integration. Für die Berechnung der Orientierung über die Zeit verwendet er eine reine Heuristik.

Visualisierung. Das Visualisierungsmodul stellt den boid graphisch dar. Es wählt gegebenenfalls Animationszyklen passend zur Bewegung, die das Kinematikmodul vorgibt. Der Animationszyklus muss synchronisiert werden zur berechneten Bewegung.

Der Schwerpunkt von Reynolds' Arbeit liegt auf dem Verhaltensmodul, das durch Regeln, nach denen sich jeder einzelne boid richtet, ein konzertiertes Verhalten der Gesamtzahl der boids bewirkt. Es gibt nach dem Modell drei grundsätzliche Verhaltensziele, die das kollektive Verhalten erzeugen — Separation, Kohäsion und Ausrichtung.

> *Separation*: Jeder boid i hält Abstand zu den anderen boids in $S(i)$.
> *Kohäsion*: Jeder boid i sucht die Nähe zur Gruppe der boids in $S(i)$.
> *Ausrichtung*: Jeder boid i passt seine Geschwindigkeit den boids in $S(i)$ an.

Eine vierte Verhaltensweise verbindet die genannten Einzelverhalten zu einem Gesamtverhalten.

Reynolds' Algoritmus kann dann etwa wie folgt beschrieben werden. Vom Wahrnehmungsmodul erhält das Verhaltensmodul gefilterte Informationen aus dem Kinematikmodul über Position, Geschwindigkeit und Beschleunigung des boid. Jedes der drei Verhaltensziele erzeugt eine Steuerungskraft für den boid, die zu einem einzigen Steuerungsvektor kombiniert wird. Der wird vom Kinematikmodul zur Simulation der aktuellen Position des

[2] In späteren Veröffentlichungen wird ein Sichtwinkel eingeführt, innerhalb dessen alle Objekte aus $S(i)$ liegen müssen.

boid herangezogen. Das Visualisierungsmodul nutzt die Informationen der Kinematik zur visuellen Darstellung.

Spätere Arbeiten übernehmen i. A. Reynolds Vorschlag und ergänzen dabei die einzelnen Bestandteile. Zunächst werden Erweiterungen des Verhaltensmodules betrachtet.

2.1 Erzeugung von Verhalten

Tu [1999] führt statische Vorlieben ein. Das sind bei ihr allgemeine Präferenzen wie z. B. für Helligkeit oder Wärme. Außerdem fügt sie mentale Zustände hinzu. Das sind veränderliche „Wünsche“ wie Hunger oder Furcht. Ein einfaches Gedächtnis sorgt für eine gewisse Persistenz im Handeln. Von diesen neuen, internen Komponenten hängt das berechnete Verhalten von den von Tu modellierten Fischen zusätzlich ab. Tu erweitert das Verhaltensmodell außerdem um eine abstraktere Ebene, die sogenannte Intentionserzeugung (oder Kognition). Um auch biomechanisch detaillierter modellierte Objekte mit einer Vielzahl von Kontrollparametern steuern zu können, berechnet ihr Verhaltensmodul keine Steuerungskraft sondern eine mehrdimensionale Kontrollanweisung zur Motor- bzw. Bewegungskontrolle.

Reynolds selbst unterscheidet [1999] analog zwischen Handlungswahl, Steuerung und Fortbewegung. In [Reynolds 2000] stellt er ein Fallbeispiel vor. Hier wird der Begriff einer externen Umgebung, von der das Verhalten eines boid abhängt, auf die interne Umgebung ausgeweitet. Diese interne Umgebung entspricht den mentalen Zuständen und statischen Vorlieben bei Tu.

Verhalten wird also grundsätzlich als Funktion gesehen, die externe Sensorinformation und internen Zustand auf Anweisungen zur Bewegungskontrolle abbildet. Um die Komplexität der zu modellierenden Verhaltensweisen zu beherrschen, werden meist eine Reihe von Abstraktionsgraden eingeführt. Das bedeutet, dass die Abbildung nicht in einem Schritt erfolgt sondern sukzessive. Teilfunktionen setzen jeweils eine Untermenge der Sensor- und Zustandsinformation auf Steuerungsparameter um. Diese Steuerungsparameter werden, wieder unter Einbeziehung von Sensoren und Zustand, auf weitere Steuerungsparameter und schließlich auf die endgültigen Anweisungen zur Bewegungskontrolle abgebildet. S. dazu auch [Grzeszczuk und Terzopoulos 1995].

Vorstellen kann man sich dieses Verfahren konkret als Hierarchie. So werden häufig Funktionen für primitive Verhaltensweisen definiert, die dann von arbitrierenden Funktionen zu komplexem Verhalten zusammengesetzt werden.

In [Doncieux et al. 2003, S. 10 f] sind die einzelnen Module neuronale Netze, die untereinander komplexe Verknüpfungen aufweisen. In [Blumberg et al. 2001] kommunizieren mentale Module verschiedener Abstraktionsebenen über einen globalen mentalen Speicher. Dagegen wird die Kommunikation zwischen Modulen bei [Motta et al. 2002] von einem besonderen Kommunikationsmodul (*Main-Module*) übernommen.

In [Reynolds 1999; Farkas et al. 2000] werden von Hand modellierte Verhaltensweisen vorgestellt, in [Grzeszczuk und Terzopoulos 1995] werden dagegen mit Hilfe von Lernprozessen sowohl Motorkontrolle, als auch Kontrolle einfachen Verhaltens automatisch erzeugt. In [Doncieux et al. 2003] wird ein Prozess der evolutionären Erzeugung einfacher Verhaltensmodule beschrieben. Auch Reynolds selbst beschreibt in einigen seiner Publikationen evolutionäre Ansätze zur Generierung von Verhaltensmodulen. Für einen Überblick vgl. [Reynolds 2004].

Eine Übersicht über Entwicklungen, die relativ einfache mentale Modelle wie Zustandsautomaten zu integrativen Modellen mit umfassender Gehirnfunktionalität erweitern, liefern [Blumberg und Isla 2002]. Entwickelt werden Konzepte für autonome Charaktere in einer virtuellen Umgebung mit weiteren Charakteren. Darunter fällt die Erkennung von Verhaltensmustern bei den Charakteren in der Umgebung, um deren zukünftiges Verhalten vorherzusagen. Die autonomen Agenten sollen ihre eigenen Verhaltensziele planen und umsetzen können. Zudem stellen die Autoren bestehende Ansätze vor, Gefühle als Teil des internen Zustandes zu modellieren. Vgl. dazu auch [Mariske 2004]. Blumberg und Isla fassen in ihrer Arbeit außerdem verschiedene Konzepte von Gedächtnis zusammen. [Funge et al. 1999] entwickeln eine formale Sprache zur kognitiven Modellierung.

Toner und Tu untersuchen in verschiedenen Publikationen mit analytischen Methoden die Auswirkungen gleichartigen Verhaltens von Individuen auf die Gesamtheit eines Systems. Ihr Ansatz lehnt sich an die NAVIER-STOKES-Gleichungen der Ströhmungsmechanik an. Eine Reihe weiterer Veröffentlichungen hat sich seither mit der Analyse von kollektivem Verhalten befasst. Schechter [1999] gibt einen Überblick über diesen Aspekt von Schwarmverhalten. [Jadbabaie et al. 2003] untersuchen die Stabilitätseigenschaften von Reynolds' Modell. Es werden Separation, Kohäsion und Ausrichtung durch ein Steuerungsgesetz ausgedrückt, das mit einer allgemein formulierten Potentialfunktion arbeitet. Eine Übersicht über verschiedene andere Methoden und über Interaktionsmodelle, die solchen Potentialfunktionen zugrunde liegen können, geben [Bent et al. 2003, S. 357].

2.2 Biomechanik, Wahrnehmung und Visualisierung

Bei echtzeitkritischen Anwendungen wird die kinematische Simulation der Objektbewegungen fast immer auf ein einfaches Teilchensystem reduziert. Jedem Objekt wird eine punktförmige Masse zugewiesen. Mit deren Hilfe und der vom Steuerungsmodul übergebenen Antriebskraft oder -beschleunigung berechnet man nach den Grundlagen der NEWTON-Mechanik pro Zeitpunkt eine neue Position für das Objekt. Diese Position und einige weitere Verhaltensparameter wie Geschwindigkeit oder Steigungswinkel etc. dienen dann zur Visualisierung des autonomen Agenten. Dabei wird auf voranimierte Bewegungszyklen zurückgegriffen, die u. a. mit Keyframing oder Motion Capturing erstellt werden. Beispiele findet man in [Reynolds 2000; Pham et al. 2003; Fröhlich 2000].

[Tu 1999; Grzeszczuk und Terzopoulos 1995] verwenden zur realistischeren Simulation verschiedener Meereslebewesen ein biomechanisches Modell. Die Fische von Tu bestehen aus diskreten Punktmassen, die durch viskoelastische Einheiten (gedämpfte HOOKE-Federn) verbunden sind. Für jeden Punktmassenknoten wird die LAGRANGE-Funktion aufgestellt zur Ermittlung seiner Bewegung unter Berücksichtigung innerer viskoelastischer und äußerer hydrodynamischer Kräfte. Einige viskoelastische Einheiten werden als Muskeln gesteuert, indem die Ruhelänge der ihnen zugrundeliegenden Federn verändert wird. Diesen Federn werden Auslöserfunktionen zugewiesen. Eine ähnliche, wenn auch komplexere Methode stellen [Popović und Wu 2003] vor. Sie dient zur realistischen mechanischen Simulation von Vogelflug entlang eines vom Benutzer vorgegebenen Pfades. In beiden Fällen wird an das physikalische Modell ein geometrisches gekoppelt, das zur entgültigen Visualisierung dient.

Geht die Kinematik von Punktmassen aus, wird auch bei der Modellierung von Wahrnehmung i. A. stark vereinfacht. Man erhält die Information der Szene durch direkten, allmächtigen Zugriff auf den Szenengraph [Reynolds 2000]. Erst dann filtert man sensorty-

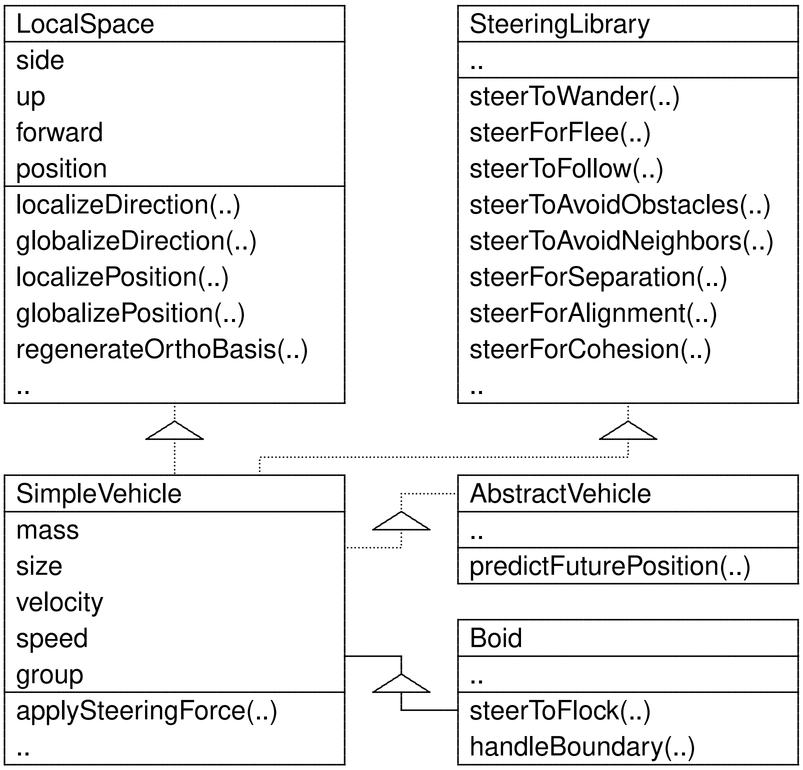

Abbildung 1: OpenSteer von C. Reynolds [2003]

pische Informationen $S(i)$ heraus (*principle of sensory honesty* [Blumberg und Isla 2002]). So werden z. B. weit entfernte Objekte nicht mehr wahrgenommen. Bei dieser Berechnung handelt es sich mit $\mathcal{O}(n^2)$ bei einer Szene mit n Elementen um eine der teuersten in der gesamten Simulation. Daher kommen in der Regel besondere Datenstrukturen zum Einsatz, die die lokale Suche von Objekten beschleunigen. Es gibt zwei grundsätzliche Strategien, eine Suche auf Objekten zu verbessern. Zum einen können *Bounding Volumes* eingesetzt werden. Eine zweite Möglichkeit ist *Space Partitioning*, die Unterteilung des Raumes in Regionen oder Voxel. Das kann z. B. eine Unterteilung in gleichgroße Voxel (*grid*-Unterteilung) sein. Eine Alternative ist ein BSP-Baum. Reynolds [2000] benutzt ein Voxelgrid zusammen mit kugelförmigen Bounding Volumes.

Ein anderer Ansatz für das Wahrnehmungsmodul wird von [Rabie und Terzopoulos 1997] verfolgt. Für jeden Agenten erzeugt man ein von ihm wahrgenommenes 2D-Retinabild. Im Anschluss werden daraus mit Methoden der Bildverarbeitung Informationen über das Umfeld des Agenten gewonnen.

2.3 Bestehende Implementierungen

Es existieren eine Reihe von Implementierungen zur Visualisierung von Verhalten. An dieser Stelle wird nur die Programmbibliothek *OpenSteer* von Reynolds [2003] betrachtet. Abb. 1 gibt eine Übersicht über einige wichtige Komponenten. LocalSpace trägt Eigenschaften zur Verwaltung eines lokalen Koordinatensystems für jedes Objekt bei. SteeringLibrary stellt die in [Reynolds 1999] entwickelten Verhaltensweisen zur Verfügung. Dabei wird vorausgesetzt, dass ein Agent (z. B. SimpleVehicle) die Schnittstelle AbstractVehicle implementiert.

Von einem solchen Agenten werden dann mehrere Klassen (u. a. Boid) abgeleitet. Diese fügen primitive Verhaltensweisen zu komplexen zusammen, so beispielsweise Schwarmverhalten (steerToFlock) aus Separation (steerForSeparation), Kohäsion (steerForCohesion) und Ausrichtung (steerForAlignment).

OpenSteer implementiert alle wesentlichen Aspekte aus Reynolds' Veröffentlichungen. Auffällig ist, dass die berechneten Steuerungskräfte vor der tatsächlichen Anwendung eine aufwendige Vorbearbeitung erfahren, die in der zitierten Literatur nicht auftaucht. Eigene Tests haben gezeigt, dass ein Verzicht auf diese Vorverarbeitung ungenügende Ergebnisse liefert. Es treten dann z. B. Kräfte auf mit extremen Längen- und Richtungsänderungen. Das hat einen unstetigen visuellen Eindruck zur Folge; die Objekte zittern. Ziel einer Vorbehandlung der Steuerungskraft muss daher eine zeitliche Persistenz der berechneten Steuerungskraft sein.

3 Architekturkonzept

Die folgenden Teilabschnitte befassen sich mit der Konzeption einer Architektur zur Simulation von Schwarmverhalten autonomer Objekte. Zunächst wird der Funktionsumfang der Bibliothek umrissen. Es werden Module und Komponenten entwickelt und dann anhand von Diagrammen die statischen Beziehungen zwischen den Bibliothekskomponenten beschrieben. Es wird ein grober Überblick über den Datenfluss zwischen den Modulen gegeben. Wichtige Funktionalitäten werden detailliert spezifiziert.

3.1 Funktionsumfang der Bibliothek

Die zu entwickelnde Programmbibliothek kann nicht alle Anwendungsfälle und Techniken implementieren. Das Gebiet der Verhaltensgenerierung und -visualisierung ist einfach zu umfassend. Der Umfang der Bibliothek wird sich also auf wenige Funktionen einschränken müssen. Um eine Erweiterbarkeit zu gewährleisten, wird die Konzeptbeschreibung allerdings Funktionalitäten umfassen, die in der späteren Implementierung unberücksichtigt bleiben.

Grundsätzlich wird mit der Bibliothek eine Erzeugung von Verhalten möglich sein, das zur Visualisierung von Vogel- und Fischschwärmen notwendig ist. Vorerst werden also Charaktere, deren Bewegungsfreiheit sich auf die Erdoberfläche beschränkt, nicht unterstützt. Die Bibliothek wird die Simulation kollektiver Verhaltensmuster erlauben. Es soll zwischen verschiedenen Gruppen von Agenten unterschieden werden können, so dass die Modellierung von Räuber-Beute-Szenarien möglich wird. Zu implementieren ist primär also Schwarmverhalten in Zusammenhang mit Räuberverhalten und Flucht vor einem Räuber. Außerdem steht die Erweiterbarkeit um neue Verhaltensweisen im Vordergrund.

Die Visualisierung des Agenten ist nicht Sache der Verhaltensbibliothek. Die physikalisch-kinematische Simulation des Agenten dagegen gehört zum Funktionsumfang der Bibliothek. Sie wird umgesetzt wie in den Veröffentlichungen von Reynolds [1987; 1999; 2000] beschrieben. Die Filterung von Umgebungsinformation, d. h. die Wahrnehmung, ist ebenfalls Teil der Bibliothek.

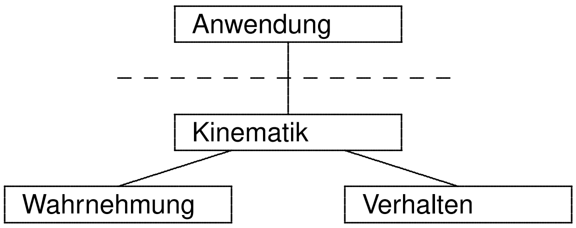

Abbildung 2: Grobes Konzept der Programmbibliothek

Es wird der innere Zustand eines Agenten modelliert, wie auch seine externe Umgebung. Durch den Schwerpunkt auf Räuber-Beute-Verhalten ergeben sich als innere Zustände in erster Linie Hunger und Furcht. Die Modellierung der äußeren Umgebung muss dynamische und statische Objekte umfassen. Damit die Bibliothek nutzbar bleibt, muss es eine Umgebungsgrenze geben, innerhalb der sich ein Agent aufhält. Diese Grenze soll an die Anwendung angepasst werden können. Weitestgehend verzichtet wird darauf, explizit das Gedächtnis des Agenten zu modellieren. Wird die Funktionalität von Gedächtnis benötigt, muss sie als innerer Zustand berücksichtigt werden.

Unterstützen soll das Konzept eine hierarchische Struktur der Verhaltenserzeugung. Primitive Kernverhaltensweisen wie eine Auswahl aus [Reynolds 1999] werden feste Bestandteile des Systems sein. Ebenso soll die Bibliothek dem Anwender die klassischen höheren Verhaltensweisen wie Schwarmverhalten, Räuberverhalten, Flucht vor einem Räuber und evtl. Formations-Flug oder zielloses Umherwandern zur Verfügung stellen. Darüber hinaus wird sie eine Erweiterung um neue Verhaltensweisen und die Änderung der hierarchischen Struktur auf Programmierebene zulassen.

Eine Intentionserzeugung teilt jeder höheren Verhaltensweise abhängig von äußeren und inneren Stimuli eine Priorität zwischen 0 und 1 zu. Das Verhalten mit höchster Priorität wird angewandt. Verhaltensweisen können andere Verhaltensweisen aufrufen, wodurch eine Hierarchie entsteht. Es liegt in der Verantwortung des Programmierers, der die Bibliothek erweitern möchte, Aufrufszyklen zu vermeiden. Jedes Verhaltensprimitiv und jede höhere Verhaltensfunktion erzeugt einzig eine dreidimensionale Steuerungsbeschleunigung. Gewünschte Zielpositionen, Geschwindigkeiten und Kräfte müssen daher adequat in einen solchen Beschleunigungsvektor umgerechnet werden.

Im einfachen Fall modelliert ein Anwender nur Vorlieben und Gewohnheiten für jeden Agenten, indem er dessen Spezies angibt. Die Funktionen der Bibliothek erzeugen dann automatisch den Beschleunigungsvektor für die weitere Simulation. Die Kinematik sorgt für die Bewegungssimulation des Agenten. Ein Benutzer kann Position, Geschwindigkeit, Orientierung und Beschleunigung des Agenten abfragen.

Desweiteren sollte die Programmbibliothek die Möglichkeit bieten, die Bewegungssimulation für einen Agenten auszusetzen oder von außen zu steuern. So wäre es für den Anwender ohne Erweiterung der Bibliothek möglich, statische Objekte in die Simulation einzubinden oder komplexere Bewegungsmodelle anzuwenden.

3.2 Komponentenbeschreibung

Folgender Abschnitt spezifiziert die oben erläuterte Funktionalität der Programmbibliothek genauer. Es wird auf die Teilkonzepte Wahrnehmung, Verhaltenserzeugung und Steuerungs-

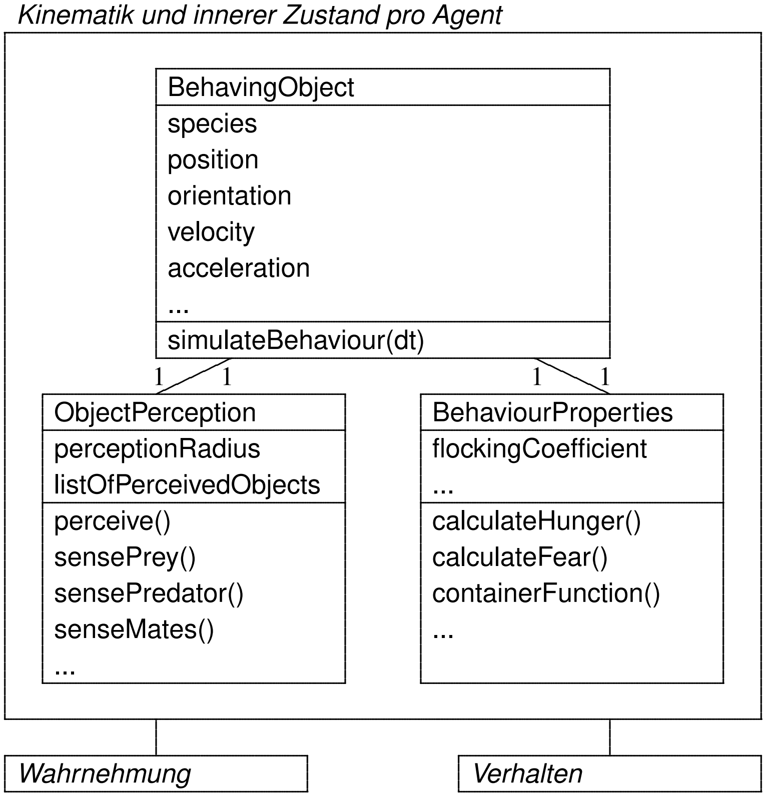

Abbildung 3: Komponente „Kinematik und innerer Zustand“

kontrolle eingegangen. Dabei werden die Teilkomponenten der Bibliotheksarchitektur beschrieben. Abb. 2 zeigt die grobe Struktur der Architektur. Eine Applikation greift auf die zentrale Kinematikkomponente zu. An die Kinematik ist die Wahrnehmungs- sowie die Verhaltenskomponente angeschlossen. Wahrnehmung und Verhaltensfunktionen ruft der Benutzer nicht direkt auf. Ziel des Konzeptes ist es, mit der Kinematikkomponente dem Benutzer alle für ihn interessanten Methoden anzubieten.

Kinematik und innerer Zustand. Die Kinematikkomponente ist für die Berechnung der Position eines Agenten zuständig, s. Abb. 3. Sie ist das architektonische Herzstück der Bibliothek und bietet die Schnittstelle zum Anwender. Von hier aus werden alle anderen Komponenten aufgerufen. Über diese Komponente läuft der Informationsfluss. An sie werden die berechneten Größen von den weiteren Komponenten zurückgeliefert. In dieser Komponente werden zentrale Daten des Agenten verwaltet. Solche Daten sind physikalischer Natur (Position, Orientierung, Geschwindigkeit und Beschleunigung in BehavingObject) oder beschreiben den inneren Zustand des autonomen Charakters (in ObjectPerception und BehaviourProperties).

Bei der Kinematikkomponente handelt es sich um diejenige, die einen Agenten individualisiert. Alle weiteren Komponenten sind systemweit vom einzelnen autonomen Charakter unabhängig. Die Wahrnehmungsdatenbank ist für alle Charaktere in der Regel diesselbe. Genauso werden für jeden Agenten diesselben Verhaltensregeln angewandt, es unter-

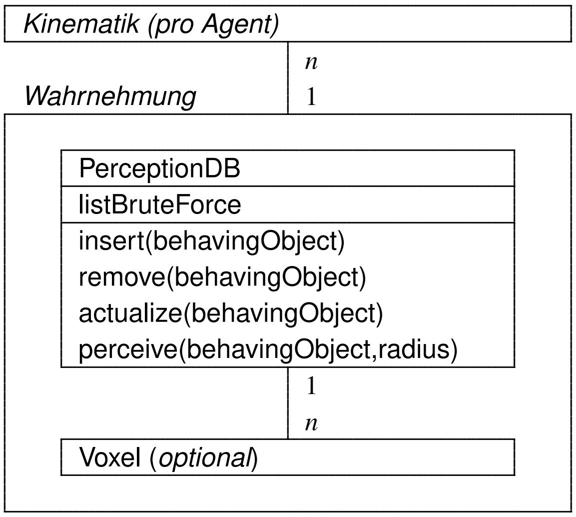

Abbildung 4: Komponente „Wahrnehmung“

scheiden sich nur die Aufrufsparameter. In der Kinematik werden zwei bibliotheksinterne Schnittstellen unterschieden. ObjectPerception übernimmt die Kommunikation mit der Wahrnehmungsdatenbank. Hier werden die Daten der Wahrnehmung individualisiert für jeden Agenten und verwaltet. Konkret kann man sich vorstellen, dass hier in Listen relevante Informationen gespeichert werden. Die Listen beinhalten z. B. alle solchen Objekte, die ein Agent wahrnimmt (listOfPerceivedObjects).

Eine weitere Schnittstelle gibt es zwischen Kinematik und Verhaltenskomponente. BehaviourProperties kontrolliert den Kommunikationsfluss bzgl. der Verhaltenssimulation und verwaltet entsprechende Daten. Konkret sind das an dieser Stelle die Informationen, die den inneren Zustand eines Agenten ausmachen, etwa die individuellen Koeffizienten des Schwarmverhaltens. Hier sollten solche Berechnungen durchgeführt werden können, die zur Unterstützung von Intentionserzeugung und von den einzelnen Verhaltensroutinen notwendig sind. Beispielsweise könnte in Abhängigkeit der gespeicherten Daten ermittelt werden, wieviel Hunger oder wieviel Furcht ein Agent im Augenblick hat (calculateHunger und calculateFear). Hier kann auch bestimmt werden, in welcher Region der Charakter sich aufhält (containerFunction).

Wahrnehmung. Die Komponente zur Modellierung der Wahrnehmung hat als Hauptaufgabe, alle vom Agenten i wahrgenommenen Objekte $j \in S(i)$ zu ermitteln, s. Abb. 4. Es wird eine Liste derjenigen Objekte j erzeugt, die i sensorisch erfasst. Dazu soll zunächst eine *Brute-Force*-Methode angewandt werden, die jeden Agent mit jedem anderen vergleicht (perceive). Jedes Objekt, das an der Simulation teilnimmt, wird gespeichert (listBruteForce) und verwaltet (insert, remove und actualize). Aufwand ist daher $\mathcal{O}(n^2)$ für eine Gesamtzahl von n simulierten Agenten. Im Konzept ist eine Erweiterung auf eine Struktur zur Verbesserung der Komplexität vorgesehen. Die Objekte würden dann in einem uniformen grid aus Voxel-Elementen gespeichert wie von [Reynolds 2000] vorgeschlagen. Die Wahrnehmungskomponente bietet i den vollen Zugriff auf alle Objekte $j \in S(i)$. So ist es i möglich, die ensprechenden Sensorinformationen zu ermitteln. Desweiteren unterstützt die

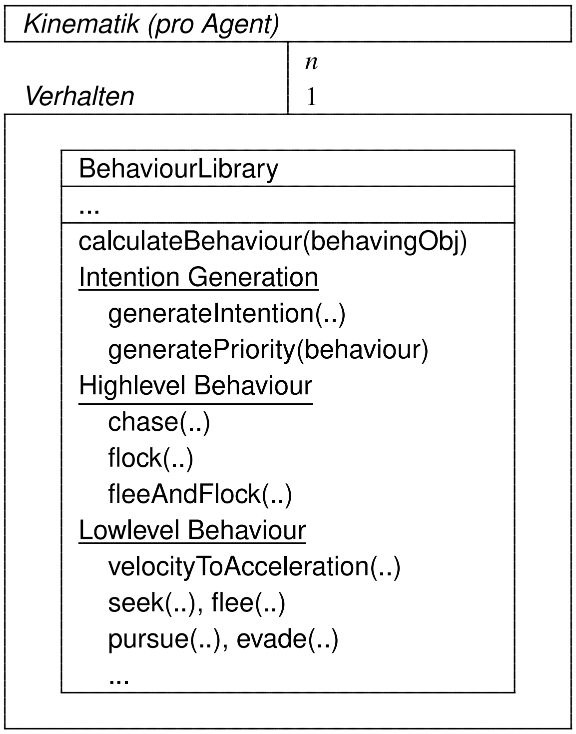

Abbildung 5: Komponente „Verhalten“

Wahrnehmung den Agenten *i* bei der Suche nach Artgenossen, Beute oder Jägern, etwa indem sie entsprechende Listen generiert. Ist die Unterscheidung verschiedener Sinne zur Wahrnehmung gewünscht, z. B. Sehen und Hören, soll die Komponente durch den Programmierer erweiterbar sein.

Verhalten. Die Verhaltenskomponente ist das funktionale Herzstück der Bibliothek, s. Abb. 5. Sie stellt nach [Reynolds 1999] primitive Verhaltensweisen wie seek, flee, pusue, evade, arrive, separate, cohere, align und contain zur Verfügung. Diese können von höheren Verhaltensweisen genutzt werden, wie von chase, flock und fleeAndFlock. Für jede höhere Verhaltensart gibt es eine Funktion, die eine Priorität zwischen 0 und 1 für das Verhalten ermittelt (generatePriority). Eine Intentionsgenerierung ermittelt aus diesen Prioritäten, welches Verhalten angewandt wird (generateIntention). Die genaue Funktionalität der Verhaltensweisen wird in Abschnitt 3.3 definiert.

Eine Anwendung, die die entworfene Bibliothek benutzen will, muss also eine Reihe Objekte anlegen vom Typ BehavingObject aus der Kinematikkomponente, s. Abb. 6. Jeder solche Agent macht sich über die Teilkomponente ObjectPerception initial der Wahrnehmungsdatenbank PerceptionDB bekannt und übergibt ihr dann in jedem Simulationsschritt die eigene Identität. Aus diesen Informationen stellt die Wahrnehmungskomponente nun im Wesentlichen eine individuelle Liste der wahrgenommenen Objekte für den Agenten zusammen. Diese Liste bezieht BehavingObject in jedem Schritt von der Wahrnehmung und stellt sie explizit mit der eigenen Identität der Verhaltenskomponente BehaviourLibrary zur Verfügung. Die berechnet dann den individuellen Steuerungsvektor für den Agenten.

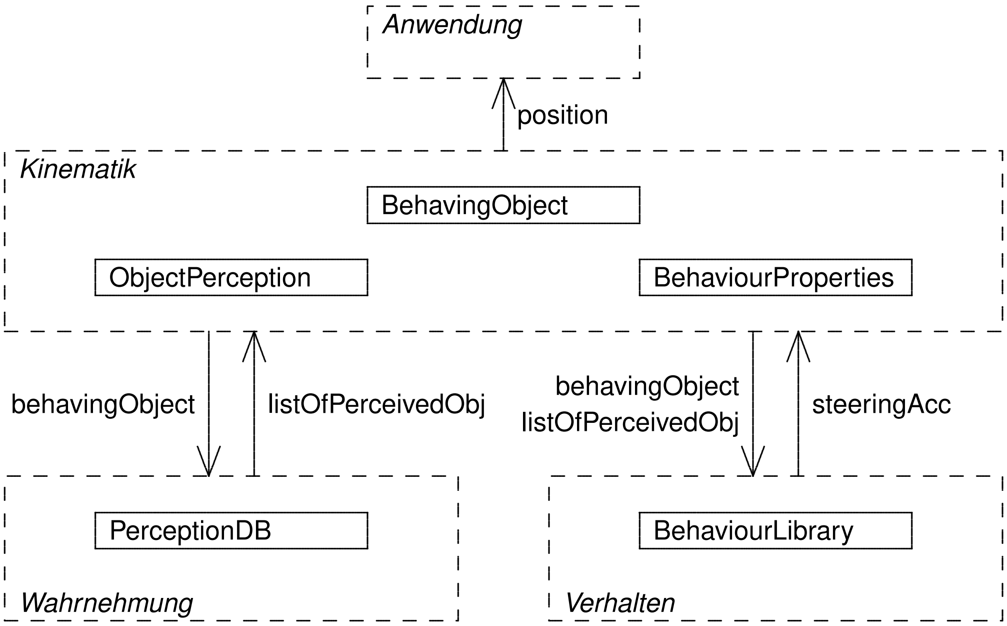

Abbildung 6: Datenfluss zwischen den Komponenten

Daraus ergibt sich eine neue Position im dreidimensionalen Raum. Pro Zeitpunkt in der Simulation liefert also jedes Objekt vom Typ BehavingObject im einfachsten Fall diese neue Position an die Applikation zurück.

3.3 Wichtige Funktionen

Einige der Funktionen aus der Verhaltenskomponente sowie aus der Kinematikkomponente werden im Folgenden genauer behandelt. Zunächst werden Verhaltensprimitive betrachtet, dann die höheren Verhaltensweisen. Im Anschluss wird kurz auf Reynolds' Heuristik zur Neuberechnung der Orientierung eines Agenten eingegangen.

velocityToAccleration, seek. Unter den Lowlevel-Verhalten kann man zwei Funktionen als grundlegend für die meisten anderen ansehen. Es handelt sich um velocityToAcceleration zur Bestimmung eines Beschleunigungsvektors, wenn eine gewünschte Geschwindigkeit gegeben ist. Sehr ähnlich arbeitet die Funktion seek, die den entsprechenden Beschleunigungsvektor für eine Zielposition ermittelt. In Anlehnung an [Reynolds 1999; Farkas et al. 2000] werden diese Umrechnungen wie folgt definiert.

Sei $\mathbf{v}_i^{(\Delta t)}$ die zum Zeitpunkt $t+\Delta t$ gewünschte Geschwindigkeit, $\mathbf{v}_i$ die Geschwindigkeit, $\mathbf{r}_i$ die Position des Agenten i zur Zeit t, so gelte für die Steuerungsbeschleunigung $\mathbf{a}_i$

$$\mathbf{a}_i := \frac{\mathbf{v}_i^{(\Delta t)} - \mathbf{v}_i}{\Delta t} \qquad \text{(velocityToAcceleration)}$$

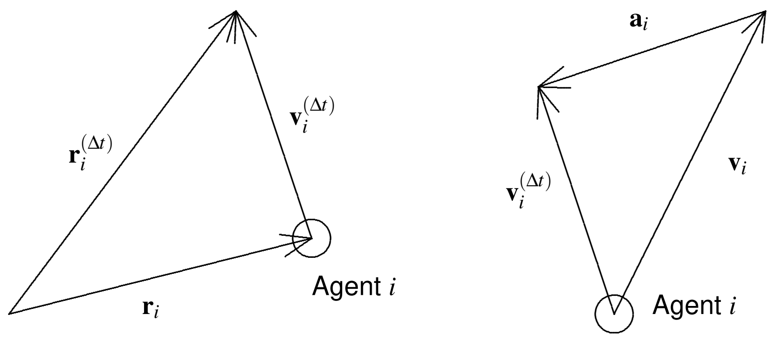

Abbildung 7: seek (links) und velocityToAcceleration (rechts)

und mit $\mathbf{r}_i^{(\Delta t)}$ als zum Zeitpunkt $t + \Delta t$ gewünschter Position

$$\mathbf{a}_i := \frac{1}{\Delta t} \cdot \left(\left(\frac{\mathbf{r}_i^{(\Delta t)} - \mathbf{r}_i}{\Delta t} \right) - \mathbf{v}_i \right) \tag{seek}$$

In der tatsächlichen Implementierung wird auf die Skalierung mit $\frac{1}{\Delta t}$ verzichtet. Die Kinematik übernimmt diese Aufgabe durch Skalierung und Trunkierung der Beschleunigung und Geschwindigkeit während der Integration gemäß minimaler und maximaler Beschleunigung und Geschwindigkeit eines Agenten.

Wenn man den Parameter „gewünschte Geschwindigkeit" von velocityToAcceleration $\mathbf{v}_i^{(\Delta t)} = \frac{\mathbf{r}_i^{(\Delta t)} - \mathbf{r}_i}{\Delta t}$ setzt, kann man erkennen, dass — in Termini der Programmierung — seek das Verhaltensprimitiv velocityToAcceleration aufruft.

pursue, arrive. Wie seek oben definiert wurde, ist es nicht uneingeschränkt zur Verfolgung eines mobilen Zieles einzusetzen. Wählt man nämlich die aktuelle Position des Zielobjektes als Eingabe, kann der Agent sein Ziel leicht verfehlen, eben weil es sich bewegt. Zudem wirkt ein Agent u. a. dadurch intelligent, dass er die Position des bewegten Zieles vorausahnt. Nach Reynolds [1999] wird ein neues Verhalten pursue als seek definiert mit der vorhergesagten Position des verfolgten Objektes als Zielort. Da diese Vorhersage extrem abhängig von den Bewegungsmöglichkeiten ist, die ein Zielobjekt hat, soll sie nicht als Verhaltensprimitiv Bestandteil der Verhaltenskomponente sein. Vielmehr wird sie der Kinematikkomponente zugeordnet. I. A. wird man davon ausgehen, dass der Positionsvorhersage die Annahme linearer Bewegung des Zielobjektes zugrunde liegt. Es wird also davon ausgegangen, dass ein verfolgter Vogel oder Fisch keine Kurve fliegt oder schwimmt. Die vorhergesagte Position hängt dann einzig davon ab, für welchen (zukünftigen) Zeitpunkt $t + \Delta t$ man sie ermitteln will. Reynolds [dito] schlägt vor, dass dieser Zeitpunkt weiter in der Zukunft liegen soll, je weiter Ziel und Verfolger voneinander entfernt sind. Auch hier vereinfacht er die Annahme wieder, indem er einen linearen Zusammenhang zwischen Δt und dem Abstand d zwischen Agent und Zielobjekt wählt, also $\Delta t \sim d$ oder $\Delta t = c \cdot d$, c const.[3] Außer vom Zielobjekt selbst ist pursue damit von einer Konstanten c abhängig.

[3] Man spreche $\sim$ „ist proportional zu".

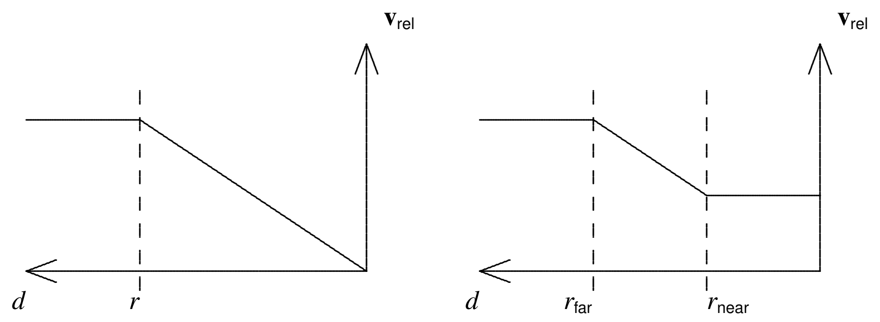

Abbildung 8: Das arrive-Verhalten nach Reynolds (links) und erweitert (rechts)

Reynolds unterscheidet neben seek und pursue noch ein weiteres Verhalten, das mit Zielverfolgung im Allgemeinen zusammenhängt. Die bisher beschriebenen Verhaltensweisen haben — je nach Implementierung — keine Auswirkung auf die Geschwindigkeit des verfolgenden Agenten oder wählen die größtmögliche. Bei arrive definiert Reynolds nun einen Radius um das Zielobjekt, innerhalb dessen der Verfolger seine Geschwindigkeit relativ zum Ziel abbremst. Innerhalb von diesem Radius ist die gewünschte Geschwindigkeit direkt proportional zum Abstand Agent–Objekt, also $\mathbf{v}_{\text{rel}}^{(\Delta t)} \sim d$. D. h. je näher der Agent am Ziel ist, desto langsamer wird er.

Die Definition von Reynolds wird für diese Arbeit im Hinblick auf die höhere Verhaltensweise chase folgendermaßen erweitert. Ein Agent soll seine Geschwindigkeit zwar relativ dem Zielobjekt anpassen, nicht aber völlig auf Null drosseln. Es wird daher ein äußerer Radius r_{far} definiert, mit dessen Unterschreitung der Bremsvorgang wie bei Reynolds berechnet wird. Wird aber ein zweiter, innerer Radius r_{near} überschritten, fällt das arrive-Verhalten wieder auf ein normales pursue zurück, skaliert um das Verhältnis der beiden Radien. So wird vermieden, dass ein Räuber bei der Verfolgung seiner Beute unnötig beschleunigt wird, obwohl er sie schon fast erreicht hat. Das geschieht bei pursue. Die Verhaltensweise arrive wurde nach Tests um die beschriebene Eigenschaft erweitert. Sie erscheint dem Verfasser visuell plausibler als Reynolds' Definition und gleichzeitig etwas stabiler als ein einfaches pursue, bei dem ein Räuber nach dem Beutefang oft zu extremen Bremsmanövern gezwungen ist. Zudem ist Reynolds' Definition des arrive-Verhaltens ein Unterfall der beschriebenen Erweiterung. Man setze $r_{\text{far}} = 0$.

flee, evade. Die Verhaltensprimitiva flee und evade sind Gegenstücke zu seek und pursue. Es gilt z. B. $\mathbf{a}_i^{(\text{flee})} = -\mathbf{a}_i^{(\text{seek})}$. Auch hier wird der Unterschied zwischen statischen und dynamischen Zielen gemacht (evade). Ein Gegenstück zu arrive ist allerdings nicht definiert.

separate, cohere, align. Das separate-Verhalten zeichnet sich dadurch aus, dass der Charakter i versucht, sich von den Charakteren in seiner Umgebung zu entfernen. Der Drang ist umso stärker, je näher die anderen Agenten sind. Für jeden benachbarten Agenten j trägt $\mathbf{a}_{ij} = \frac{\mathbf{r}_i - \mathbf{r}_j}{d^2}$, $d = |\mathbf{r}_j - \mathbf{r}_i|$ zu separate bei. $\mathbf{a}_i$ ist somit die Summe der einzelnen abstoßenden Beschleunigungen.

$$\mathbf{a}_i = \sum_j \mathbf{a}_{ij} = \sum_j \frac{\mathbf{r}_i - \mathbf{r}_j}{d^2} \qquad \text{(separate)}$$

Das cohere-Verhalten verhindert dagegen, dass sich ein Agent zu weit von den Charakteren in seiner Umgebung entfernt. Es ist definiert als ein seek-Verhalten bzgl. des gewichteten Massenschwerpunktes $\frac{\sum_j m_j \cdot \mathbf{r}_j}{\sum_j m_j}$ aller benachbarten Agenten j als Zielposition von seek.

$$\mathbf{a}_i = \frac{1}{\Delta t} \cdot \left(\frac{\frac{\left(\sum_j m_j \cdot \mathbf{r}_j\right)}{\sum_j m_j} - \mathbf{r}_i}{\Delta t} - \mathbf{v}_i \right) \qquad \text{(cohere)}$$

Analog ist das align-Verhalten als velocityToAcceleration-Aufruf zu verstehen mit der Durchschnittsgeschwindigkeit der benachbarten Agenten als dem Parameter der gewünschten Geschwindigkeit.

$$\mathbf{a}_i = \frac{1}{\Delta t} \cdot \left(\frac{\sum_j \mathbf{v}_j}{\sum_j 1} - \mathbf{v}_i \right) \qquad \text{(align)}$$

contain. Das contain-Verhalten sorgt dafür, dass ein Agent sich nur in einer durch eine Funktion gegebenen Region aufhält. Die Funktion muss für eine gegebene Position bestimmen können, ob diese Position innerhalb der erlaubten Region liegt. Darüber hinaus muss sie die gegebene Position an ihrer Umgebungsgrenze spiegeln können. Im Kern ist contain dann ein seek-Verhalten nach der gespiegelten eigenen zukünftigen Position, wenn diese außerhalb der Region liegt. Ist sie im erlaubten Bereich, erzeugt contain die Nullbeschleunigung. Die beschriebene Heuristik ist problematisch bei speziellen Regionen, z. B. bei solchen, die lange und schmale Buchten umfassen. Da das contain-Verhalten aber primär das praktische Ziel besitzt, die Agenten einerseits davon abzuhalten, den Sichtbereich der virtuellen Kamera zu verlassen, und andererseits nicht durch ein virtuelles Terrain durchzubrechen, ist diese Einschränkung unbedenklich. Gibt es in einer Szene dennoch kompliziert gestaltete Aufenthaltsbereiche für die Charaktere, könnte man diese mit dem avoidObstacle-Verhalten modellieren, das in diesem Konzept nicht zur Implementierung vorgesehen ist, jedoch erweitert werden kann.

flock, fleeAndFlock. Das flock-Verhalten gehört zu den höheren Verhaltensweisen, die sich aus mehreren anderen Verhalten zusammensetzen. Der Beschleunigungsvektor, der ein flock-Verhalten, das ist ein Schwarmverhalten, bewirkt, setzt sich als lineare Interpolation aus den Steuerungsvektoren von separate, cohere und align zusammen. In der Implementierung gehen die Einzelkomponenten nach Reynolds' Empfehlung normalisiert ein.

$$\mathbf{a}_i = \lambda_1 \mathbf{a}_i^{(\text{separate})} + \lambda_2 \mathbf{a}_i^{(\text{cohere})} + \lambda_3 \mathbf{a}_i^{(\text{align})} \qquad \text{(flock)}$$

Schwarmverhalten ist somit von den Parametern $\boldsymbol{\lambda} = (\lambda_1\ \lambda_2\ \lambda_3)^{\mathrm{T}}$ abhängig. Reynolds [1987; 1999] unterscheidet außerdem verschiedene Parameter für die Wahrnehmung der Agenten in $S(i)$, so dass er einen neundimensionalen Parameterraum für Schwarmverhalten ausmacht. Im Konzept dieser Arbeit soll das aber unberücksichtigt bleiben.

fleeAndFlock schaltet, wenn ein Jäger wahrgenommen wird, auf ein Fluchtverhalten gegenüber diesem Jäger um. Ist kein Jäger „in der Nähe“, greift fleeAndFlock auf flock zurück. Im vorliegenden Konzept ist beiden Verhaltensweisen (flock, fleeAndFlock) gemein, dass der Steuerungsvektor von contain, ist er verschieden von Null, den berechneten Steuerungsvektor ersetzt. So entsteht bei fleeAndFlock eine Hierarchie, in der contain über evade, und evade über flock steht. Wenn die einzelnen Verhaltensweisen von fleeAndFlock im Falle, dass sie nicht „anschlagen“, einen Nullvektor zurückliefern, könnte man alternativ die Steuerungsbeschleunigungen aller Teilverhalten addieren. Man beachte, dass dann aber die hierarchisch höheren Verhaltensweisen negativ von niedrigeren beeinflusst und aufgehoben werden können.

chase. Das chase-Verhalten, das Reynolds in seinen Veröffentlichungen nicht beschreibt, basiert im vorliegenden Konzept auf der oben entwickelten Erweiterung des primitiven arrive-Verhaltens. Außerdem beruht es wesentlich auf der Fähigkeit der Wahrnehmungskomponente, genau einen Agenten zu ermitteln, der als Beute gewählt wird. Darunter soll man insbesondere verstehen, dass die ausgewählte Beute in einem zu bestimmenden Winkel vor dem Jäger liegt, wenn dieser sie das erste Mal wahrnimmt. Solange der Jäger seine einmal gewählte Beute nicht aus den Augen verliert, sucht er sich keinen neuen Beuteagenten für die Jagd. S. dazu [Tu 1999]. Die Aufgabe von chase beschränkt sich dann im Wesentlichen auf den arrive-Aufruf, sowie darauf, den Jäger abzubremsen, wenn er keine Beute wahrnimmt. Das erscheint grundlegend für die Plausibilität eines Jagdverhaltens und entspricht dem, was man informell als „Lauern“ bezeichnen könnte.

banking. Nachdem die wichtigsten Funktionen der Verhaltensbibliothek besprochen wurden, geht dieser Absatz kurz auf die Bewegungsberechnung ein. Ein Agent wird modelliert mit Position und Orientierung. Das entspricht einem Körper mit seinen sechs Raumfreiheitsgraden. Während die drei Freiheitsgrade der Position mit Hilfe der NEWTONschen Bewegungsgleichungen bestimmt werden, gibt Reynolds [1999] für die Veränderungen in der Orientierung eine heuristische Berechnungsvorschrift.

Auf die NEWTONschen Bewegungsgleichungen soll nicht weiter eingegangen werden. Der Leser sei auf die Basisliteratur zur klassischen Mechanik hingewiesen, z. B. auf [Landau 1997, Kap. 1]. Durch Angabe der Beschleunigung kann die neue Geschwindigkeit und Position des Agenten durch Integration bestimmt werden. Als numerisches Integrationsverfahren wird seiner Einfachkeit halber die explizite EULER-Methode verwendet.

Reynolds' Verfahren, die Orientierung eines Agenten zu Berechnen, wird zunächst informell beschrieben. Es gilt, die drei Achsen des lokalen Koordinatensystems resp. die Orientierung als Rotationsmatrix anhand der gegebenen Größen eindeutig festzulegen. Zentraler Gedanke Reynolds' ist, die $\mathbf{e}_1$-Achse (vorwärts) in Richtung der Geschwindigkeit des Agenten zu wählen, sowie die $\mathbf{e}_2$-Achse (oben) in Gegenrichtung der Erdbeschleunigung $\mathbf{g}$. Damit es keine großen Schwankungen der $\mathbf{e}_2$-Achse gibt, geht anteilig die alte $\mathbf{e}_2$-Achse ($\mathbf{e}_2^{\mathrm{old}}$) aus dem vergangenen Simulationsschritt in die Berechnung ein. Soll sich der Agent

— wie beispielsweise ein Flugzeug — in die Kurve legen (banking), geht zusätzlich der berechnete Beschleunigungsvektor ($\mathbf{a}_i$) in die Berechnung der $\mathbf{e}_2$-Achse ein. Durch Anwendung des Kreuzproduktes erhält man die neue $\mathbf{e}_3$-Achse. Das Koordinatensystem wird nun einfach orthogonalisiert, indem man die $\mathbf{e}_2$-Achse durch das Kreuzprodukt der bei den schon rechtwinkligen Achsen ersetzt. Es gilt also mit den Gewichtungskoeffizienten $\boldsymbol{\mu} = (\mu_1\ \mu_2\ \mu_3)^T$

$$\begin{aligned}
\mathbf{e}_1 &:= \mathbf{v}_i \\
\mathbf{e}_3 &:= \mathbf{e}_1 \times \mathbf{e}_2' \qquad\qquad \text{wobei} \quad \mathbf{e}_2' := \mu_1 \mathbf{e}_2^{\text{old}} + \mu_2 \mathbf{a}_i - \mu_3 \mathbf{g} \\
\mathbf{e}_2 &:= \mathbf{e}_1 \times \mathbf{e}_3
\end{aligned}$$

Durch die anschließende Normalisierung der Achsen erhält man ein orthonormales Koordinatensystem, dessen Achsen als Rotationsmatrix $\mathbf{R} = (\mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3)$ zur Bestimmung der Orientierung des Agenten benutzt werden kann.

4 Implementierungsdetails

Nachdem in den vorigen Abschnitten das Konzept der Programmbibliothek in der Literatur begründet und vorgestellt wurde, geht der folgende Abschnitt auf ausgewählte Details der Implementierung ein. Das können Besonderheiten wie die Wahl von Parametern oder Probleme bei der Umsetzung sein. Darüber hinaus wird die Benutzerschnittstelle der Bibliothek anhand einer Beispielapplikation eingeführt. Abschließend werden im Teilabschnitt 4.3 Ergebnisse präsentiert. Das umfasst einen Überblick über die Vollständigkeit der Implementierung und ein Beispiel der Bibliothek in Benutzung.

Einleitend ist festzustellen, dass die Implementierung in C++ [Stroustrup 2000] aufbauend auf obigem Konzept, relativ problemlos erfolgte. Entwickelt wurde unter dem Betriebssystem *Linux*. Die Implementierungsarbeit war „straight forward". Die einzelnen Komponenten BehavingObject, ObjectPerception, BehaviourProperties, PerceptionDB und BehaviourLibrary finden sich als Klassen wieder. Die einfache Implementierung ist nach Einschätzung des Verfassers im Besonderen der anwendungsnahen Aufbereitung des Stoffes bei Reynolds anzurechnen, der das wissenschaftliche Gebiet der Simulation und Visualisierung von Schwarmverhalten maßgeblich für die Computergraphik prägt.

Im Folgenden wird die Implementierung am Beispiel einer Kernverhaltensweise des Systems, des Schwarmverhaltens, gezeigt. Man rufe sich die mathematische Spezifikation von flock aus Teilabschnitt 3.3, S. 15, ins Gedächtnis. In der Implementierung bezeichne flockParam die Parameter $\boldsymbol{\lambda}$ für das Schwarmverhalten. An anderer Stelle wurde schon darauf hingewiesen, dass die Skalierung mit $\frac{1}{\Delta t}$ aus der Spezifikation in der Umsetzung wegfällt und durch entsprechende Maßnahmen in behavingObject ersetzt wird. Zudem wurde erwähnt, dass die Teilverhalten normalisiert in die Gewichtung eingehen, wie von Reynolds empfohlen. Listing 1 zeigt den Programmcode der Definition von flock.

Nach dem gleichen Schema werden auch die Verhaltensprimitiva separate, cohere und align definiert. Man betrachte z. B. die Spezifikation von separate, S. 15. Listing 2 ist eine direkte Umsetzung der mathematischen Definition dieses Basisverhaltens.

```
// Highlevel Behaviours: flock / Schwarmverhalten
Vector3D BehaviourLibrary::flock ()
{

  // kein Schwarmverhalten, wenn contain höchste Prioritaet hat
  if (generatePriority (CONTAIN) == 1.0f)
    return contain (thisObject.getBehaviourProperties ());

  // lies die flocking Parameter lambda des Agenten
  Vector3D flockParam =
    (thisObject.getBehaviourProperties ()).getFlockingCoefficient ();

  // ermittle die Steuerungsbeschleunigung fuer die Basisverhalten
  Vector3D sep = separate ();
  Vector3D coh = cohere ();
  Vector3D ali = align ();

  // normalisiere die Beschleunigungsvektoren
  if (sep.Length ())   // |sep| != 0
    sep.Normalize ();
  if (coh.Length ())   // |coh| != 0
    coh.Normalize ();
  if (ali.Length ())   // |ali| != 0
    ali.Normalize ();

  // addiere die mit lambda gewichteten Beschleunigungsvektoren
  Vector3D steering = (sep * flockParam [0]) +
                      (coh * flockParam [1]) +
                      (ali * flockParam [2]);

  // das Ergebnis ist die gesuchte Steuerungsbeschleunigung fuer flock
  return steering;

}
```

Listing 1: Definition von flock-Verhalten

4.1 Besonderheiten

Auch wenn die Implementierung des Konzeptes in der Programmiersprache C++ auf direktem Weg erfolgen kann, gibt es doch einige Details zu beachten. Darunter werden Alternativen und Verbesserungen verstanden, die — zumindest zeitweise — in die Bibliothek integriert wurden. Die ersten beiden Punkte beschäftigen sich mit verschiedenen Aspekten zur Verbesserung des berechneten Beschleunigungsvektors. Im dritten Punkt soll ein alternatives Verfahren zur Bestimmung einer orthonormalen Basis für das banking aufgezeigt werden. Zum Ende des Teilabschnitts wird kurz auf die Auswirkungen der Parameter $\boldsymbol{\lambda}$ des Schwarmverhaltens eingegangen.

Skalierungsmaßnahmen. Die Definition des Steuerungsvektors bzgl. einer Zielposition wurde in Abhängigkeit des Zeitpunktes $t + \Delta t$ gegeben, der besagt, wann das Ziel erreicht werden soll, vgl. S. 13, Abb. 7 (seek). An dieser Stelle wurde schon darauf hingewiesen,

```
// Lowlevel Behaviours: separate
Vector3D BehaviourLibrary::separate ()
{

  // default-Beschleunigung, wenn es keine Nachbarn gibt
  Vector3D steering = Vector3D (0, 0, 0);

  // Summe ueber alle Agenten j, die i wahrnimmt
  vector <BehavingObject*>::iterator nearbiesIterator;
  for (nearbiesIterator = nearbies.begin ();
       nearbiesIterator != nearbies.end ();
       nearbiesIterator++) {

    // Differenzvektor zwischen i und j
    Vector3D offsetVec = thisObject.getPosition () -
                         (*nearbiesIterator)->getPosition ();

    // quadratischer Abstand d zwischen i und j
    float offsetLength2 = offsetVec.ScalarProd (offsetVec);

    // verhindere Nulldivision,
    // der Wert von offsetLength2 ist dann unerheblich
    if (offsetLength2 == 0)
      offsetLength2 = 0.1;

    // addiere Beitrag von j zu Beschleunigungsvektor
    steering = steering + offsetVec * (1.0f / offsetLength2);
  }

  // das Ergebnis ist die gesuchte Steuerungsbeschleuigung fuer separate
  return steering;

}
```

Listing 2: Definition von separate-Verhalten

dass in der Implementierung auf eine solche Skalierung verzichtet wurde in Anlehnung an Reynolds [1999]. Grund ist, dass in einer konkreten Anwendung mit einem Zielobjekt selten ein Zeitpunkt modelliert wird, zu dem seine Position erreicht sein soll. Oft ist dieser Zeitpunkt ungenau gegeben, etwa durch die Anweisung „So schnell wie möglich“ oder „Unter Beibehaltung der aktuellen Geschwindigkeit“. Daher wird bei der Berechnung des Steuerungsvektors in der Verhaltenskomponente im Wesentlichen auf Skalierungen des Ergebnisses verzichtet und die weitere Behandlung der Kinematikkomponente, namentlich der Klasse BehavingObject anheim gestellt.

In dieser Klasse wird für jedes Objekt eine minimale und maximale Geschwindigkeit sowie eine maximale Kraft modelliert. Eine minimale Geschwindigkeit ist v. a. dann interessant, wenn es sich bei den Agenten um Flugobjekte handelt, die eine Grundgeschwindigkeit benötigen, um ihre Höhe zu halten.

```
  float maxSpeed;   // maximale Geschwindigkeit
  float minSpeed;   // minimale Geschwindigkeit
  float maxForce;   // maximale Steuerungskraft
```

Vor den entsprechenden Integrationsschritten werden die erhaltenen Werte trunkiert.

```
// Trunkierung an maximaler Steuerungskraft,
// veldot ist die Beschleunigung
if (veldot.Length () * mass > maxForce) {
  veldot.Normalize ();
  veldot = (veldot * maxForce) * (1.0f / mass);
}

// Trunkierung an maximaler Geschwindigkeit,
// posdot ist die Geschwindigkeit
if (posdot.Length () > maxSpeed) {
  posdot.Normalize ();
  posdot = posdot * maxSpeed;
}

// Trunkierung an minimaler Geschwindigkeit,
// posdot ist die Geschwindigkeit
if (posdot.Length () < minSpeed && posdot.Length () != 0) {
  posdot.Normalize ();
  posdot = posdot * minSpeed;
}
```

Nun ist z. B. beim seek-Verhalten der Steuerungsvektor — ohne Skalierung mit $\frac{1}{\Delta t}$ — so gewählt, dass das Ziel im nächsten Simulationsschritt erreicht werden könnte. Damit die Länge des Vektors trotzdem eine Auswirkung in der Simulation hat, wäre es denkbar, die Beschleunigung vor der Trunkierung mit einem konstanten Wert zu skalieren.

```
veldot = veldot * dt * 0.025;
```

Mit einer Skalierung lassen sich weitere interessante Effekte erzielen. Nimmt ein Agent einen Jäger wahr, werden in vorliegender Umsetzung Geschwindigkeit und Beschleunigung mit dem Faktor 5 multipliziert. Wenn also Gefahr droht und einem realen Beutetier das ausgeschüttete Adrenalin zu Höchstleistungen verhilft, übernimmt dies für den virtuellen Agenten ein Skalierungsfaktor.

```
if (objectPerception.sensePredator () && !(getSpecies () == PREDATOR)) {
   veldot = (veldot * 5.0f);
   posdot = (posdot * 5.0f);
 }
```

Bewegungsglättung. In Abschnitt 2.3 wurde bereits darauf hingewiesen, dass es problematisch ist, die von den Verhaltensweisen erzeugten Beschleunigungen direkt zur Bewegungsberechnung heranzuziehen. In die Heuristiken zur Berechnung dieser Vektoren gehen zuvor angewandte Steuerungsbeschleunigungen nicht ein. Das hat zur Folge, dass sich direkt aufeinanderfolgende Vektoren in Richtung und Länge eklatant voneinander unterscheiden. Solche Beschleunigungsmanöver wirken unplausibel. Auch die Geschwindigkeitsvektoren, die man durch Integration über die Beschleunigung erhält, erscheinen dann unstetig, so dass sich die Orientierung der Agenten stark ändern kann. Das ist für die Beispielapplikation mit einfacher Visualisierung der Agenten als Kugeln, wie sie unten vorgestellt wird, nicht gravierend. Wählt man jedoch realistischer Fischmodelle, fallen die Unstetigkeiten als Zittern negativ auf.

Abhilfe schafft eine Vorbehandlung des Beschleuningsvektors, bevor er in die numerische Integration eingeht. Ähnliche Methoden wären auch auf die Geschwindigkeit übertragbar, sollen hier aber nicht vorgestellt werden.

Da das Grundproblem starke Änderungen im Beschleunigungsverhalten sind, wird im Folgenden die zeitliche Ableitung der Beschleunigung $\dot{\mathbf{a}}_i$ betrachtet. Ein erster Ansatz zur Verbesserung wäre, $|\dot{\mathbf{a}}_i|$ nach oben zu begrenzen. Alternativ könnte man $\dot{\mathbf{a}}_i$ um einen konstanten Faktor c skalieren. Dieser Faktor wirkt nach Meinung des Verfassers dann ausreichend, wenn er kleiner $\frac{1}{2}$ gewählt wird. Diese Alternative wurde ihrer Einfachkeit halber in der vorliegenden Implementierung umgesetzt.[4] Da sie für einfache Visualisierungen vernünfige Verbesserungen liefert, wurde darauf verzichtet, alternative Ideen zu implementieren. Diese Ideen sollen hier trotzdem kurz angerissen werden.

Sowohl starke Änderungen in der Länge des Beschleunigungsvektors, als auch in seiner Orientierung könnten getrennt behandelt werden. Starke Längenänderungen könnte man durch Trunkierung oder Skalierung von $\frac{\mathrm{d}}{\mathrm{d}t}|\mathbf{a}_i|$ weiter einschränken. Für die Einschränkung von Änderungen in der Orientierung sieht der Verfasser eine Möglichkeit, die auf der Betrachtung der normalisierten Beschleunigung $\mathbf{a}_i^{(n)}$ und ihrer Ableitung nach der Zeit beruhen. So könnte man $\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{a}_i^{(n)}$ trunkieren oder skalieren, um die gewünschte Einschränkung der Orientierungsänderung angenähert zu erreichen.

Zu den Überlegungen gehört nicht zuletzt die numerische Umsetzung. Es wird zunächst der implementierte Ansatz beleuchtet. $\dot{\mathbf{a}}_i$ wird angenähert als Differenz zwischen aktueller Beschleunigung und der Beschleunigung im vorigen Simulationsschritt, also

```
Vector3D accdot = (acceleration - oldAcceleration)
```

Eine Skalierung kann dann umgesetzt werden als

```
accdot = accdot * 0.25;
```

und geht wie folgt in die Simulation mit ein

```
veldot = oldAcceleration + accdot;
```

Berechnet man nun aus alter und neuer normierter Beschleunigung das Skalarprodukt — das ist in diesem Fall der Kosinus des Winkels zwischen beiden Beschleunigungen — hätte man ein weiteres einfaches Maß für die Abweichung in der Orientierung der Steuerungsvektoren.

Orthogonalisierung nach GRAM-SCHMIDT. Im Teilabschnitt 3.3 wurde die Heuristik von Reynolds zur deterministischen Bestimmung eines orthonormalen Objektkoordinatensystemes vorgestellt. Als Alternative implementiert die Programmbibliothek die Orthogonalisierung nach GRAM-SCHMIDT [Barnett 1990, S. 210 f]. Hier können wie bei Reynolds

[4] mit dem Faktor $c = \frac{1}{4}$

initial z. B. folgende Werte gewählt werden

$$\begin{aligned}
\mathbf{e}'_1 &:= \frac{\mathbf{v}_i}{|\mathbf{v}_i|} \\
\mathbf{e}'_2 &:= \mu_1 \mathbf{e}_2^{\text{old}} + \mu_2 \mathbf{a}_i - \mu_3 \mathbf{g} \\
\mathbf{e}'_3 &:= \mathbf{e}_3^{\text{old}}
\end{aligned}$$

Der GRAM-SCHMIDT-Prozess ist dann gegeben als

$$\begin{aligned}
\mathbf{e}_1 &= \mathbf{e}'_1 \\
\mathbf{e}_2 &= \mathbf{e}'_2 - (\mathbf{e}_1 \cdot \mathbf{e}'_2)\mathbf{e}_1 \\
\mathbf{e}_3 &= \mathbf{e}'_3 - (\mathbf{e}_1 \cdot \mathbf{e}'_3)\mathbf{e}_1 - (\mathbf{e}_2 \cdot \mathbf{e}'_3)\mathbf{e}_2
\end{aligned}$$

Im Anschluss müssen die erhaltenen Vektoren noch normalisiert werden. Auf den ersten Blick scheint die alternative Methode weder bessere noch schlechtere Ergebnisse zu liefern als Reynolds' Vorschlag. Eine analytische Untersuchung wurde nicht vorgenommen.

Parameterwahl für flocking. Verschiedene Testläufe mit unterschiedlicher Belegung der Koeffizienten $\boldsymbol{\lambda}$ für das Schwarmverhalten ergaben für die Beispielapplikation gute Ergebnisse bei $\boldsymbol{\lambda} = (0{,}32\ \ 0{,}35\ \ 0{,}35)^T$. Wählt man den cohere-Koeffizienten deutlich größer als den separate-Koeffizienten, veringert sich der Abstand der Agenten im Schwarm stark bis hin zum Kollaps. Umgekehrt gilt analoges. Bei hohem align-Koeffizienten zeigt die Gruppe als Ganzes ein einheitliches Verhalten und legt gemeinsam weite Strecken zurück. Bei niedrigem align-Koeffizienten bleibt der Schwarm eher auf der Stelle und nimmt Kugelform an. Stellt sich kein Gleichgewicht ein, pulsiert diese Kugel aus Agenten leicht.

Besonders auffällig scheint, dass die ersten beiden Koeffizienten eher im Verhältnis gesehen eine intuitive Wirkung haben. Sie wirken sich auf den Abstand der Agenten zueinander aus. Es ist allerdings schwierig, den Abstand unter geeigneter Wahl des Verhältnisses so zu beeinflussen, dass er eine konkrete Größe einnimmt. Das ist oft nicht einmal nach langem Experimentieren zu erreichen. Es wäre mit Sicherheit benutzerfreundlicher, cohere und separate zu einem einzigen Verhaltensprimitiv zusammenzufassen, mit dem direkt der Abstand zwischen zwei Agenten modelliert werden kann.

[Bent et al. 2003, S. 357] geben einen Überblick über weitere Anziehungs- und Abstoßungsfunktionen, die in der Literatur zur Modellierung von Schwarmverhalten herangezogen wurden. Einige dieser Modelle lassen es zu, einen Abstand zu bestimmen, bei dem Anziehungs- und Abstoßungskräfte ein Gleichgewicht bilden. [Bent et al. 2003] zeigen allerdings, dass sich dieser Gleichgewichtsabstand nur bei dem seltenen Fall von genau zwei interagierenden Charakteren einstellt. Sie berechnen einige Eigenschaften des Abstandes zwischen Agenten für das von ihnen gewählte Modell analytisch für große Schwärme, etwa untersuchen sie Konditionen die zu stabilen oder zu kollabierenden Gruppen führen. Zur Stabilität von Schwarmverhalten sei noch einmal auf [Jadbabaie et al. 2003] verwiesen. Diese Untersuchungen für die Computergraphik konkret nutzbar zu machen, wäre eine Aufgabe für die Forschung in diesem Bereich.

4.2 Benutzerschnittstelle

Im Folgenden soll beispielhaft die Nutzung der entwickelten Programmbibliothek erklärt werden. Im Teilabschnitt 4.3 werden dann Ergebnisse der Beispielanwendung gezeigt. Die Beispielanwendung wird als Überblick vorgestellt. Zum einfacheren Verständnis werden viele Details, die keine zentrale Rolle spielen, bei der Besprechung weggelassen.

Die main-Routine der Anwendung ruft die in Graphics gekapselten GLUT- und GL-Funktionen auf, zunächst mit initGLUT die initialisierenden Befehle, dann die Berechnungsschleife mit runGLUT. Neben den GLUT-Routinen wird auch die Anwendung selbst initialisiert und in der Hauptschleife ausgeführt. Die Anwendung (Application) besitzt als sogenannte *member*-Variablen einen Vektor von Zeigern auf Objekte der Klasse Bird. Bird ist ein SceneObject, also ein Objekt der Szene. Außerdem besitzt Application ein member PerceptionDB.

Neben einer Funktion zur Initialisierung der Szenenobjekte, gibt es in Application die Funktionen simulate und draw, die über alle diese Szenenobjekte iterieren und deren eigene simulate- und draw-Funktionen aufrufen. Diese Funktionen sollen nun mit der Klasse Bird betrachtet werden.

Bird besitzt als member-Variable ein BehavingObject der Programmbibliothek. Es sind also für ein Bird-Objekt alle Daten und Funktionen von BehavingObject, und damit die gesamte Funktionalität der Programmbibliothek, instanziiert. Die Initialisierungsfunktion von Bird weist denn auch den Daten von BehavingObject erste Werte zu. Es werden z. B. Position und Geschwindigkeit zufällig gesetzt. Damit alle BehavingObjects auf der selben Wahrnehmungsdatenbank arbeiten, übergibt Bird ihnen die member-Datenbank von Application als Referenz bzw. Zeiger. In der vorliegenden Beispielapplikation werden eine Reihe von Beuteobjekten und genau ein Jäger initialisiert. Der entsprechende Aufruf kann folgendermaßen aussehen.

```
behavingObject.init (PREY, initialPosition, ee1, ee2, ee3,
                     initialVelocity, initialAcceleration, db);
```

Dabei wird das BehavingObject als Beute (PREY) mit einer Anfangsposition, einer x-, y- und z-Achse seiner lokalen Orthonormalbasis, einer Anfangsgeschwindigkeit und -beschleunigung sowie einem Zeiger auf die Wahrnehmungsdatenbank gesetzt.

Die simulate-Methode von Bird ruft dann pro Simulationsschritt die Funktion simulateBehaviour von BehavingObject auf, die alles Weitere automatisch übernimmt. Sie braucht als Parameter die Zeit in Millisekunden, die seit ihrem letzten Aufruf verstrichen ist.

```
behavingObject.simulateBehaviour (dt);
```

Die Funktion draw von Bird liest dann die simulierte Position und Orientierung von BehavingObject und ruft die entsprechenden Graphics-Funktionen auf, um den Agenten als graue Kugel mit seinen Koordinatenachsen zu zeichnen.

```
Vector3D pos = behavingObject.getPosition ();
Vector3D gray = Vector3D (0.5, 0.5, 0.5);

drawSphere (pos [0], pos [1], pos [2], 2.0f, gray);
drawFrame (pos [0], pos [1], pos [2],
```

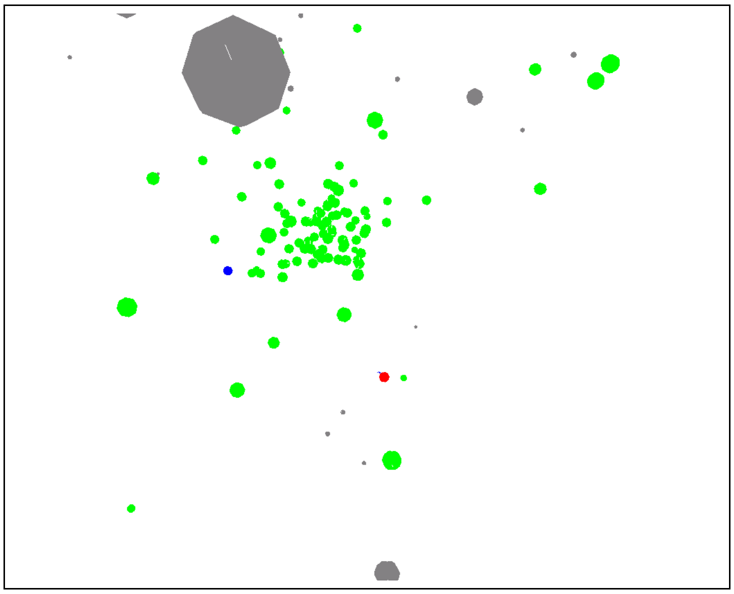

Abbildung 9: Simuliertes Schwarmverhalten und Jäger. Der Jäger ist rot gekennzeichnet, alle von ihm wahrgenommenen Objekte grün, sein Beuteziel blau.

```
behavingObject.getForward (),
behavingObject.getUp (),
behavingObject.getSide (), 4.0f);
```

4.3 Ergebnisse

Die Anwendung zeigt, dass die Implementierung einer Programmbibliothek zur Simulation von Schwarmverhalten erfolgreich war und mit wenigen einfachen Benutzeraufrufen überzeugende Ergebnisse liefert.

Bibliothek und Beispielapplikation sind plattformunabhängig gehalten und können mit den Betriebssystemen *Linux* und *Windows* benutzt werden.

Die Voreinstellungen der Parameter sind sinnvoll gewählt. Man sieht einen Jäger, der einen Schwarm Beutetiere jagd, symbolisiert durch Kugeln, die sich bewegen. Nimmt der Jäger ein Beuteobjekt wahr, dass sich günstig für einen Angriff in seiner Bewegungsrichtung befindet, verfolgt er es und lässt dabei nicht locker, bis es wieder aus seinem Blickfeld verschwindet, s. Abb. 9. Ist keine Beute in seinem Umfeld, bremst er ab und lauert auf die nächste Gelegenheit.

Die Beutetiere werden zufällig positioniert und finden sich selbständig zu dynamischen Schwärmen, s. Abb. 11. Man sieht attraktive Wendemanöver. Nähert sich der Jäger, stäuben die Beuteobjekte reflexartig auseinander und flüchten, jeder nach seinen Möglichkeiten, s. Abb. 10.

Die Programmbibliothek ist so angelegt, dass sie leicht erweitert werden kann. Mit Erweiterungsvorschlägen beschäftigt sich etwas ausführlicher der nächste Abschnitt. Es fehlen die Verhaltensweisen Formationsflug und Umherwandern.

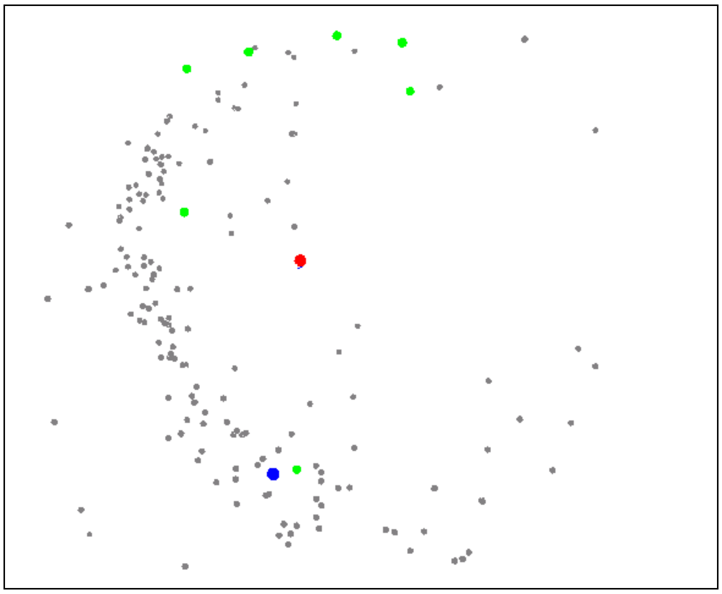

Abbildung 10: Simuliertes Schwarmverhalten und Flucht. Beuteobjekte stäuben auseinander, ist ein Jägers in der Nähe.

Bislang nur dürftig umgesetzt ist insbesondere die Modellierung des inneren Zustandes eines Agenten. Die Einbindung statischer Objekte in die Verhaltenssimulation ist ebenfalls noch nicht möglich. Es fehlt außerdem eine konsistente Intentionsgenerierung. So übernimmt es das fleeAndFlock-Verhalten selbst, zwischen contain, flock und flee zu arbitrieren, obwohl genau das Aufgabe von Prioritätenermittlung und Intentionserzeugung ist. Nicht umgesetzt ist auch die komplexitätsverbessernde Verwaltung der Agenten und Objekte in der Wahrnehmungsdatenbank. Hier sollte ein Raumpartitionierungsmodell zum Tragen kommen.

Ein weiterer deutlicher Schwachpunkt der bisherigen Umsetzung ist die Anpassungsfähigkeit an den Benutzer. Eine Parametrisierung verschieden von den voreingestellten Werten ist schwierig, wenn nicht unmöglich, ohne den Code der Bibliothek selbst zu ändern. Es wäre aber leicht Abhilfe möglich, wenn die Parameter grundsätzlich aus Konfigurationsdateien eingelesen würden. So könnte man dem Benutzer verschiedene Vorkonfigurationen zur Verfügung stellen, die er Schritt für Schritt abwandeln und an seine eigenen Bedürfnisse anpassen kann.

5 Erweiterungsmöglichkeiten

Als Ausblick werden nun noch einige Anregungen gegeben, wie die Bibliothek um Funktionalität erweitert werden kann.

Im vorigen Abschnitt wurden dazu schon einige Gedanken vorgestellt. So könnte eine konsistente Intentionsgenerierung die Erweiterung der Programmbibliothek um neue Verhaltensweisen deutlich erleichtern. Eine Konfigurationsdatei, aus der die Parameter der Verhaltensgenerierung eingelesen werden, ließe dagegen die Nutzungsmöglichkeiten der

Bibliothek erhöhen. Mit einem genügend hohen Forschungsaufwand könnte man die Basisfunktionen von Schwarmverhalten unter Einbeziehung neuester Erkenntnisse aus der biologischen Mathematik evtl. so umstrukturieren, dass ein Endbenutzer die gewünschte räumliche Durchschnittsverteilung — etwa als durchschnittlichen Abstand zwischen den Agenten — direkt angeben kann. In der Wahrnehmung wäre eine Verringerung der Komplexität wünschenswert. Die Kinematik könnte alternative Bewegungsberechnungen anbieten, die sich an konkreten Agenten, beispielsweise an Flugzeugen, orientieren und diese realistisch modellieren. Insbesondere die Einschränkung der Bewegungsfreiheit auf die Erdoberfläche wäre eine Herausforderung. Statische Objekte sollten in die Simulation mit einbezogen werden können, wie im Konzept kurz dargelegt.

5.1 Erweiterte Funktionalität

Innerhalb der Programmbibliothek bietet der Bereich der Intentionsgenerierung und Auswahl an Verhaltensweisen großes Erweiterungspotential. Hier könnten eine Reihe von Konzepten aus der Künstlichen Intelligenz integriert werden. Darunter fällt u. a. die kognitive Modellierung. Greift die Bibliothek bislang noch allmächtig auf die Eigenschaften der wahrgenommenen Agenten zu, wäre es denkbar, dass in einer erweiterten Version mehr Wert auf die Simulation und Verwaltung der Sensorinformation aus der Umwelt gelegt wird. Ein Agent könnte sich den vergangenen Zustand seiner Umgebung merken und daraus Schlussfolgerungen über die Gegenwart oder Zukunft ziehen. Er könnte das Verhalten anderer Charaktere analysieren und daraus lernen. Nicht zuletzt könnte er seine Ziele strategisch planen und verschiedene Verhaltensweisen gezielt nacheinander ausführen, um höhere Ziele zu erreichen.

Nicht nur innerhalb der Programmbibliothek gibt es Erweiterungsmöglichkeiten. Auch die Visualisierung bedarf einer konzeptionell überlegten Anbindung. Vorvisualisierte Verhaltensdarstellungen müssten anhand der Daten aus der Programmbibliothek passend ausgewählt und synchronisiert werden. Übergänge zwischen den vorgefertigten Visualisierungen sollten automatisch generiert werden. Eine Anbindung an eine Motion-Capturing-Datenbank wäre in diesem Zusammenhang ein ehrgeiziges Ziel.

5.2 Wissenschaftlich fundierte Benutzung

Zum Schluss wird noch darauf hingewiesen, dass nicht nur technische Verbesserungen um neue Funktionen möglich sind, sondern die konkrete Anwendung und Benutzung Betätigungsfelder beinhaltet, die eine breite Vielfalt bieten. Das Anwendungsgebiet in der Produktion von Unterhaltungsmedien ist nicht das einzige. Verhaltenssimulationen und -visualisierungen sind interessant für die Biologie, um bestehende Theorien zu überprüfen. Es gibt ähnlich gelagerte Anwendungen in der Ökologie wie in der Ökonomie. In der Architektur spielt man Massenszenarien vor dem Bau von Gebäuden für die öffentliche Benutzung durch. Eine überzeugende Visualisierung ist da von großem Wert und mag helfen, erfolgsversprechende Konzepte durchzusetzen. Für alle diese Anwendungen, aber auch für die Unterhaltungsindustrie selbst, ist es grundlegend, die Verhaltenssimulation und -visualisierung auf realistischen Daten durchzuführen, die eine wissenschaftliche Grundlage besitzen.

Eine Vielzahl von Veröffentlichungen untersucht beispielsweise das Sozialverhalten von Vogelgruppen. Es sei hier auf [Hilton et al. 1999] verwiesen, die Reaktionsgeschwindigkeiten von Rotschenkel-Kolonien in Schottland auf Angriffe durch Sperber dokumentieren. [Fernández-Juricic 2001] berichtet über die Auswirkungen der Anwesenheit des Menschen auf den durchschnittlichen Abstand der einzelnen Mitglieder in Vogelgruppen, untersucht in verschiedenen Madrider Parks. Direkt relevant für den Schwarmalgorithmus sind hingegen die Flugdistanzen. Umfangreiche Tabellen für die verschiedensten japanischen Wasservogelarten enthält [Mori et al. 2001].

Als Ausblick auf zukünftige Aufgaben in der Simulation von Schwarmverhalten bleibt somit festzuhalten, dass die durchaus wissenschaftlich begründeten Verhaltensmodelle, die in der Computergraphik verwendet werden, so anzupassen sind, dass solche konkreten Ergebnisse aus anderen Disziplinen direkt und möglichst leicht integriert werden können.

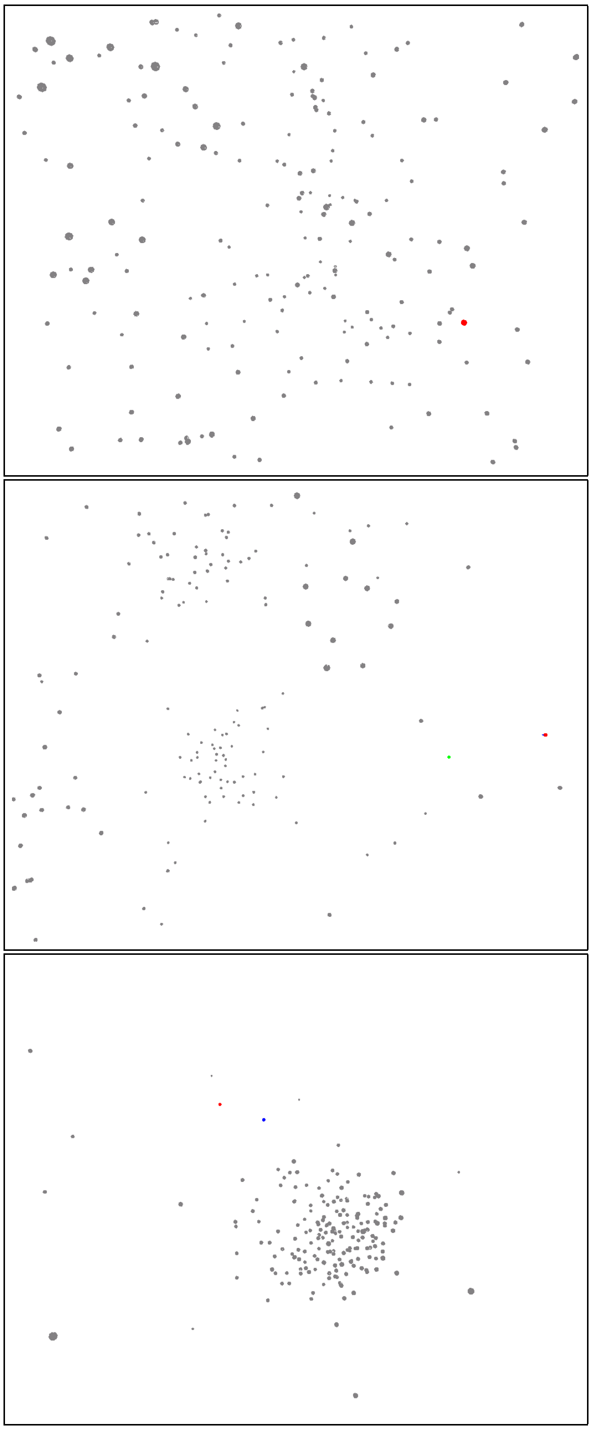

Abbildung 11: Simuliertes Schwarmverhalten. Zufällig positionierte Objekte (oben) finden sich nach und nach zu einem Schwarm (mitte, unten).

Literatur

APODACA, A., UND GRITZ, L. 2000. *Advanced RenderMan: Creating CGI for Motion Pictures*. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann Publishers / Academic Press, San Francisco.

BARNETT, S. 1990. *Matrices: Methods and Applications*. Oxford Applied Mathematics and Computing Science Series. Clarenden Press / Oxford University Press, Oxford. Nachdruck 1994.

BENT, L., EDELSTEIN-KESHET, L., MOGILNER, A., UND SPIROS, A. 2003. Mutual interactions, potentials, and individual distance in a social aggregation. *Journal of Mathematical Biology 47* (Mai), Seiten 353–389.

BLUMBERG, B., BURKE, R., DOWNIE, M., UND ISLA, D. 2001. A layered brain architecture for synthetic creatures. In *Proceedings of the IJCAI*.

BLUMBERG, B., UND ISLA, D. 2002. New challenges for character-based AI for games. In *Proceedings of the AAAI spring symposium on AI and Interactive Entertainment*.

DAVIS, T., NEIDER, J., SHREINER, D., UND WOO, M., Hrsg. 2003. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.4*, 4. Aufl. Addison-Wesley, Boston, Nov.

DONCIEUX, S., FILLIAT, D., GUILLOT, A., UND MEYER, J.-A. 2003. Evolutionary approaches to neural control of rolling, walking, swimming and flying animats or robots. In *Biologically Inspired Robot Behavior Engineering*, R. Duro et al., Hrsg., Bd. 109 der Reihe *Studies in Fuzziness and Soft Computing*. Physica-Verlag, Heidelberg, Seiten 1–43.

FARKAS, I., HELBING, D., UND VICSEK, T. 2000. Simulating dynamical features of escape panic. *Nature 407* (Sep), Seiten 487–490.

FERNÁNDEZ-JURICIC, E. 2001. Avian spatial segregation at edges and interiors of urban parks in madrid, Spain. *Biodiversity and Conservation 10*, Seiten 1303–1316.

FRÖHLICH, T. 2000. The virtual oceanarium. *Communications of the ACM 43*, 7 (Jul), Seiten 95–101.

FUNGE, J., TERZOPOULOS, D., UND TU, X. 1999. Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. In *Proceedings of SIGGRAPH 99*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, Seiten 29–38.

GRZESZCZUK, R., UND TERZOPOULOS, D. 1995. Automated learning of muscle-actuated locomotion through control abstraction. In *Proceedings of SIGGRAPH 95*, ACM Press / ACM SIGGRAPH, Los Angeles, Computer Graphics Proceedings, Annual Conference Series, ACM, Seiten 63–70.

HÄGELE, T. 2003. Animation überschreitet Genre-Grenzen. *Digital Production 7*, 6 (Nov), Seiten 91–97.

HILTON, G., CRESSWELL, W., UND RUXTON, G. 1999. Intraflock variation in the speed of escape-flight response on attack by an avian predator. *Behavioral ecology 10*, 4, Seiten 391–395.

JADBABAIE, A., PAPPAS, G., UND TANNER, H. 2003. Stable flocking of mobile agents, part I and II. In *42nd IEEE Conference on Decision and Control*, IEEE Press, Maui, Hawaii, Seiten 2010–2021.

LANDAU, L. 1997. *Lehrbuch der theoretischen Physik: Mechanik*, 14. Aufl., Bd. 1. Verlag Harri Deutsch, Thun und Frankfurt am Main. In dt. Sprache hrsg. von P. Ziesche.

MARISKE, H.-A. 2004. Hunger, Wut und Müdigkeit: Softwareagenten mit Emotionen. *C't Magazin für Computertechnik*, 2 (Jan), Seiten 82–83.

MORI, Y., SODHI, N., KAWANASHI, S., UND YAMAGISHI, S. 2001. The effect of human disturbance and flock composition on the flight distances of waterfowl species. *Journal of Ethology 19*, 2, Seiten 115–119.

MOTTA, L., FIGUIEREDO, T., UND RAUPP, S. 2002. A framework to investigate behavioural models. In *Proceedings of the WSCG 02*.

PHAM, B., STEPHENS, K., UND WARDHANI, A. 2003. Modelling fish behaviour. In *Proceedings of GRAPHITE 03*.

POPOVIĆ, Z., UND WU, J.-C. 2003. Realistic modeling of bird flight animations. *ACM Transactions on Graphics 22*, 1 (Jan), Seiten 888–895.

RABIE, T., UND TERZOPOULOS, D. 1997. Animat vision: Active vision in artificial animals. *Videre 1*, 1 (Herbst).

REYNOLDS, C. 2000. Interaction with groups of autonomous characters. In *Proceedings of the 2000 Game Developers Conference*, CMP Game Media Group, Seiten 449–460.

REYNOLDS, C. 1987. Flocks, herds, and schools: A distributed behavioral model. In *Proceedings of SIGGRAPH 87*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, Seiten 25–34.

REYNOLDS, C. 1999. Steering behaviors for autonomous characters. In *Proceedings of the 1999 Game Developers Conference*, Seiten 763–782.

REYNOLDS, C., 2003. OpenSteer: Steering behaviors for autonomous characters, vers. 0.8, Okt. http://www.red3d.com/cwr/opensteer.

REYNOLDS, C., 2004. Craig W. Reynolds (homepage), Feb. www.red3d.com/cwr.

SCHECHTER, B. 1999. Birds of a feather. *New Scientist* (23. Jan), Seiten 30–33.

STROUSTRUP, B. 2000. *The C++ Programming Language*, 4. Aufl. Addison-Wesley Verlag / Pearson Education, München. Dt. Übersetzung der Special Edition von N. Josuttis und A. Lörke.

TU, X. 1999. *Artificial Animals for Computer Animation: Biomechanics, Locomotion, Perception, and Behavior*, Bd. 1635 der Reihe *Lecture Notes on Computer Science*. Springer Verlag, Berlin.

Danksagungen

Vielen Dank an Torben Fugger für alle seine Anregungen und, dass er mir seine Spieleengine zum Testen der Programmbibliothek in einer vorläufigen Version zur Verfügung gestellt hat. Vielen Dank an alle Studierenden im *Mixed-Reality*-Labor für ihre Einschätzungen und Kommentare. Besonderen Dank an Prof. Dr. Stefan Müller für die Betreuung der Arbeit.

Studienarbeit

im Studiengang Computervisualistik an der Universität Koblenz-Landau, Abteilung Koblenz.

Elektronische Kopien der Arbeit und der Source-Code von Programmbibliothek und Beispielanwendung können beim Verfasser angefordert werden (`hornung` an der Universität Koblenz).

UNIVERSITÄT KOBLENZ · LANDAU