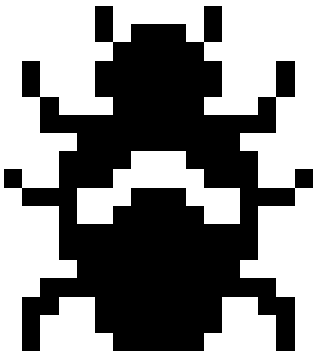
Simulation von Gruppenbewegungen durch Verhaltensgesteuerte Animation

Erstellung eines Multiagentensystems zur Generierung von Gruppenbewegungen mit Schnittstelle zu Maya und Lightwave, realisiert bei der I-D Media AG, Geschäftsbereich ID-TV



An der Fachhochschule Dortmund
im Fachbereich Informatik
erstellte Diplomarbeit
im Studiengang Allgemeine Informatik
zur Erlangung des Grades Diplom Informatiker (FH) von
Carsten Kolve, geboren am 03.02.1976

1. Betreuer: Prof. Dr. Burkhard Lenze
2. Betreuer: Prof. Dr. Paul Rietmann

Dortmund, 28. April 2000

Meinen Eltern

Kurzdarstellung

Die Erstellung von Massenszenen in der Computeranimation für den Einsatz in Film, Fernsehen und sonstigen Medien stellt einen Bereich dar, der mit den herkömmlichen Techniken der Computeranimation nur schwer oder überhaupt nicht zu realisieren ist. Die vorliegende Arbeit beschäftigt sich mit einem möglichen Ansatz, um solche Szenen dennoch mit relativ geringem Aufwand erstellen zu können: Verhaltensgesteuerte Animation. Es wird ein mögliches Modell zur Realisierung, basierend auf sogenannten autonomen Agenten mit festgelegten physikalischen Eigenschaften und Steuerverhalten, vorgestellt, welches beispielhaft in der Simulationssoftware *bugz* implementiert wurde, die eine Schnittstelle zur Standardsoftware Lightwave 6.0 sowie Maya 2.5 beinhaltet.

Schlüsselwörter

Animationstechniken, verhaltensgesteuerte Animation, Computergrafik, Computeranimation, Simulation, autonome Agenten, Multiagentensysteme, Steuerverhalten, Gruppenverhalten, Gruppenbewegung, Artificial Life, Delphi, Lightwave, LScript, Maya, MEL

Abstract

Creating computer animated crowd scences for use in film, television and other media is an area, where traditional techniques used in computer animation can only be used with great difficulties. This paper deals with one possible approach for creating these kinds of scenes at relativly low expense: Behavioural Animation. A implementation model is presented, based on so called autonomous agents, their physical properties and steering behaviours. It is examplary implemented in the simulation software *bugz*, including an interface to the animation packagese Lightwave 6.0 and Maya 2.5.

Keywords

animation techniques, behavioural animation, computer graphics, computer animation, simulation, autonomous agents, mult-agent systems, steering behaviours, group behaviour, crowd motion, artificial life, Delphi, Lightwave, LScript, Maya, MEL

Inhaltsverzeichnis

Vorwort

Diese Arbeit wurde in enger Zusammenarbeit mit der *I-D Media AG*, Geschäftsbereich *ID-TV* entwickelt.

Bei der I-D Media AG handelt es sich um ein stark wachsendes New-Media Unternehmen mit Hauptsitz in Berlin und Niederlassungen in Potsdam, Stuttgart, Hamburg und London. Die Full-Service Agentur im Bereich der digitalen Medien wurde 1988 von Bernd Kolb gegründet und ging 1999 unter dem Aufsichtsratsvorsitzenden Dr. h. c. Lothar Späth an den Neuen Markt. Die Firma gehört zu den drei erfolgreichsten Agenturen im New-Media Bereich in Deutschland und zählt namhafte Großunternehmen zu seinen Kunden.

Die I-D Media AG ist aufgeteilt in vier Geschäftsbereiche:

I-D Services
: entwickelt Internet-Auftritte und E-Commerce Lösungen.

Cycosmos
: ist eine Community-Platform im Internet, die in Zukunft Kontakte von Firmen zum potentiellen Kunden für deren Artikel herstellen soll.

Living Screen
: ist im Bereich Software tätig und entwickelt vernetzte multimediale Kommunikations- und Präsentationstools.

ID-TV
: produziert 3D-Animationen (speziell computergenerierte Charaktere) und bietet digitale Video-Post-Produktion, DVD-Authoring, sowie Video-Streaming über das Internet. In Zukunft ist eine starke Expansion im Bereich des Interaktiven Digitalen Fernsehens geplant.

ID-TV beschäftigt zur Zeit rund 50 Mitarbeiter und ist damit eins der größten Animationshäuser in Deutschland. Neben den im künstlerisch / kreativen Bereich tätigen Abteilungen gibt es noch den für die Software-Entwicklung zuständigen Bereich *Research & Development*. Hier werden Tools entwickelt, die den Funktionsumfang der benutzten Animations- und Modellierungssoftware erweitern oder Arbeitsschritte automatisieren, Schnittstellen zwischen verschiedenen Animationssystemen geschaffen, Lösungen für Probleme aus dem Produktionsbereich erarbeitet.

1 Einführung in die Problematik

1.1 Aufgabenstellung

Die Diplomarbeit hat das Thema: *Simulation von Gruppenbewegungen durch Verhaltensgesteuerte Animation*. Es ist darzustellen, wie 2D-Gruppenbewegungen mit Hilfe der Animationstechnik der Verhaltensgesteuerten Animation rechnergestützt erzeugt werden können und eine entsprechende Software zu implementieren.

1.2 Motivation und Überblick

In der Natur kann an vielen Orten das koordinierte Verhalten einer Gruppe von Lebewesen beobachtet werden, so z.B. beim Flug eines Vogelschwarms in der Luft, der Bewegung einer Herde von Landsäugetieren, dem Treiben von Insekten oder auch im Trubel der Menschenmengen auf Grossveranstaltungen und in Einkaufspassagen.

Sollen diese alltäglichen Beobachtungen nun in Computeranimationen wiedergegeben werden, so wirft diese Arbeitsaufgabe Probleme auf: Bei einer solchen Gruppenbewegung sind häufig eine grosse Menge von Akteuren beteiligt, in der klassischen Computeranimation müßten die zugehörigen Objekte alle einzeln animiert werden, sehr viel Zeit und damit Kapitalaufwand würde für die Erstellung eines realistischen, komplexen Bewegungs- und Interaktionsmusters notwendig sein. Schon kleine Änderungen an einem auf diese Weise erstellten Bewegungsmuster können unverhältnismäßig großen Arbeitsaufwand nach sich ziehen, da evtl. die Bewegung aller beteiligten Akteure sich entsprechend ändern muß. Schon diese zwei Gründe zeigen, daß die Erstellung von Gruppenbewegungen von Hand nicht wirtschaftlich ist. Im folgenden wird gezeigt werden, wie alternativ mit Hilfe von verhaltensgesteuerter Animation, basierend auf einfachen Verhaltensregeln, komplexe Bewegungsmuster erzeugt werden können. Die Ergebnisse werden anhand einer Applikation verdeutlicht, die mit dem professionellen Animationswerkzeug Lightwave 6.0 zusammenarbeitet.

1.3 Begriffsdefinitionen

Wichtige Begriffe werden genau definiert, damit ihre Bedeutung im Zusammenhang des Textes zweifelsfrei geklärt ist.

Gruppe

Der Begriff der Gruppe kann je nach Bereich, in dem er verwendet wird, eine sehr weite Bedeutung haben. Da in dieser Arbeit die Bewegung von Lebewesem simuliert werden soll, wird die Bedeutung des Gruppenbegriffs aus der Verhaltensforschung abgeleitet. Dort wird unterschieden zwischen Aggerationen und Gesellschaften. Bei der Aggregation erfolgt eine Ansammlung von Lebewesen an einem bestimmten Ort allein aus dem Bestreben der einzelnen Individuen, diesen Ort aufzusuchen. Als Beispiel wäre z.B. ein Wasserloch zu nennen, an dem sich Tiere unterschiedlichster Arten zusammenfinden. Gesellschaften zeichnen sich dadurch aus, daß sich ein soziales Netz zwischen den Mitgliedern befindet. Offene Gesellschaften werden in ihrem Verhalten durch den Verlust oder das Hinzukommen von Mitgliedern nicht beeinträchtigt (als Beispiel wäre hier ein großer Fischschwarm zu nennen), geschlossene Gesellschaften reagieren auf die Veränderung der Gruppenanzahl oder das Austauschen von Mitgliedern mit der Änderung ihres Verhaltens [23].

Um welche Art der Gruppe es sich jeweils handelt, ist dem spezifischen Kontext zu entnehmen, der Fokus liegt aber vor allem bei Aggregationen und offenen Gesellschaften - individuelle Kenntnis voneinander, wie sie für eine geschlossene Gesellschaft notwenig ist, wird nicht nachgebildet.

Verhalten

Die allgemeine Definition von Verhalten wird im spezifischen Kontext der verhaltensgesteuerten Animation auf das sogenannte autochthone Verhalten beschränkt. Dieses ist als „Gesamtheit der Reaktionen, die auf einem spezifischem Antrieb beruhen und durch einen passenden Schlüsselreiz ausgelöst werden“ [4] definiert. Individuelle Lernvorgänge, wie sie beim allochthonen Verhalten beobachtet werden, finden keine Berücksichtigung.

Autonomer Agent / Akteur / Charakter / Bug

Mit diesen Begriffen wird die rechnerinterne Repräsentation eines beliebigen, sich bewegenden, mit seiner Umwelt interagierenden und autonom handelnen Lebewesens verstanden [8]. Bei der Darstellung der theoretischen Grundlagen wird in diesem Text der neutralere Begriff *autonomer Agent/Akteur/Charakter* gewählt, bei der Darstellung der Implementierung eher der (nur speziell in dieser Arbeit verwendete) Begriff *Bug* (amerik. Insekt / Käfer). In anderen Arbeiten zu diesem Thema werden Bugs auch als *„Boids“* (Bird-like Objects) [9] oder auch *„Animats“* [12] bezeichnet.

Bewegung
Wenn im folgenden von der Bewegung eines autonomen Agenten gesprochen wird, so ist damit im allgemeinen nicht die Eigenbewegung des durch den Agenten repräsentierten Lebewesens gemeint (z.B. die Bewegung der Beine, das Schlagen der Flügel etc.), sondern nur die Veränderung der Position des ganzen Lebewesens im Raum.

Bewegungsmuster
Die meist charakteristische Gesamtheit der Bewegungen einer Gruppe von autonomen Agenten wird als Bewegungsmuster verstanden.

1.4 Vereinbarungen

Koordinatensystem
Der Ursprung des hier verwendeten Koordinatensystems befindet sich in der linken oberen Ecke des angezeigten Fensters. Der positive Y-Achsenabschnitt nimmt nach unten hin größere Werte an.

Winkel
Aus den Eigentümlichkeiten des Koordinatensystems folgt, daß die Gradzahl eines Winkels nicht (wie üblich) gegen den Uhrzeigersinn zunimmt, sondern mit dem Uhrzeigersinn.

1.5 Hinweise zum Lesen

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

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

Quelltext und Pseudocode
Quelltext und Pseudocode wird in dieser Arbeit in `Courier New` dargestellt. Variablen, Konstanten, Prozedur- und Funktionsbezeichnungen sind durchgängig in englisch gehalten. Beispiel:

```
procedure WelcomeMessage;
  begin
    writeln('Welcome to');
    writeln('bugz!');
  end;
```

Simulationen und Filme

An einigen Stellen werden die theoretischen Ergebnisse anhand von Simulationen und Animationsfilmen verdeutlicht.

Bei den Simulationen handelt es sich um .bgz-Dateien, die mit der im Rahmen der Diplomarbeit erstellten Software *bugz* angezeigt werden können. Die Dateien befinden sich auf dem beigefügten Datenträger, dem Ordner für die Software *bugz* untergeordnet.

Simulation 1-1: simulation.bgz

Animationsfilme finden sich gleichfalls auf dem beigefügten Datenträger, im dafür vorgesehenen Ordner `Animationen`.

Film 1-1: film.mov

2 Verhaltensgesteuerte Animation

In diesem Kapitel wird der Begriff der verhaltensgesteuerten Animation geklärt und ein System vorgestellt, das sich zur Simulation von Gruppenbewegungen eignet.

2.1 Gebräuchliche Animationstechniken

Im folgenden werden in der Computeranimation gebräuchliche Animationstechniken vorgestellt und auf ihre Verwendbarkeit in Bezug auf die Generierung von Gruppenbewegungen hin überprüft.

2.1.1 Schlüsselbild-Technik

Bei der *Schlüsselbild-Technik* (auch *Keyframe-Methode*) handelt es sich wohl um die gebräuchliste aller Animationsarten. Hierbei definiert der Animator Eigenschaften eines Körpers (wie Position im Raum, Neigungswinkel etc.) zu bestimmten Zeitpunkten. Das Animationssystem interpoliert (z.B. linear) zwischen den entsprechenden Eigenschaftswerten an diesen Schlüsselbildern. Diese Methode ist sehr gut dafür geeignet, einzelne Gegenstände oder Charaktere zu animieren. Will man allerdings viele Akteure auf einmal animieren, kommt sehr viel Arbeit auf den Animator zu - er muß für jedes einzelne Objekt die Bewegung zu jedem Zeitpunkt animieren. Dies stellt bei großen Mengen von zu animierenden Akteuren einen nicht vertretbaren Arbeitsaufwand dar - ausserdem ist nicht sichergestellt, daß überhaupt ein organisch anmutendes Gesamtverhalten zu erwarten ist.

Film 2-1: KeyframeAnimation.mov

2.1.2 Motion Capturing

Unter Motion Capturing versteht man die Aufzeichnung von Bewegungen eines realen Akteurs mittels an dessen Körper angebrachter Sensoren, deren Veränderung registriert wird. Die so gewonnen Daten können dann auf einen virtuellen Charakter transferiert werden. Der Vorteil dieser Animationsmethode liegt darin, daß man in relativ kurzer Zeit sehr gute und sehr realistische Bewegungsabläufe für die Computeranimation erhält. Im allgemeinen wird Motion Capturing dazu verwendet um die Bewegung eines menschlichen Charakters aufzuzeichnen, es ist aber theoretisch auch durchaus möglich, die Bewegungen von Individuen in einer Gruppe aufzuzeichnen. Dies würde mit Sicherheit zu sehr guten Ergebnissen führen, hat aber zwei entscheidenen Nachteile: Zum einen ist ein entsprechendes Motion Capture System sehr teuer (der Preis liegt zum Zeitpunkt der Erstellung dieser Arbeit bei ca. 600.000 DM

für eine Anlage), zum anderen müßten zur Gewinnung der Daten auch erst reale Akteure zur Aufzeichung bestellt werden - und dies würde den eigentlichen Sinn der durch Computer simulierten Gruppenbewegung, eben keine große Anzahl von realen Akteuren zu benötigen, in vielen Fällen zunichte machen.

Film 2-2: MotionCapturing.mov

2.1.3 Pfad-Animation

Die Technik der *Pfad-Animation* wird häufig benutzt, wenn ein Objekt sich entlang eines bestimmten Weges bewegen soll. Hierbei wird zuerst ein Pfad gezeichnet (zB. ein Splinepfad), und dann das entsprechende Objekt über einen bestimmten Zeitraum entlang diese Pfades bewegt. Um nachträglich die Animation zu ändern braucht dann nur der Pfad angepaßt werden.

Diese Technik eignet sich sehr gut dazu, um schon vorhandene Gruppenbewegungen im nachhinein noch zu bearbeiten. Da man einen Bewegungspfad auch lokal leicht verändern kann, benötigt man für die nachträgliche Korrektur nur sehr wenig Aufwand. Das Problem bei der Pfad-Animation in Bezug auf die Generierung von Gruppenbewegungen liegt darin, daß die Bewegungspfade für jedes einzelne Mitglied der Gruppe auch noch von Hand generiert werden muß, was den gleichen Arbeitsaufwand wie bei der Keyframe-Technik bedeutet.

Film 2-3: PathAnimation.mov

2.1.4 Skript-Technik

Bei dieser Form der Animation wird die Bewegung durch mathematische Formeln und Algorithmen realisiert. Häufig findet die *Skript-Technik* (auch *Prozedurale Animation*) bei Problemen ihr Einsatzgebiet, wo einfache mathematische Formeln schwierig von Hand durch die Keyframe-Technik zu realisierenden Lösungen gegenüberstehen. Ein einfaches Beispiel dafür wäre zB. ein Rad, das sich über eine Ebene bewegt. Die Rotation einer Kugel zu berechnen, die notwendig ist, damit der abgerollte Teil des Umfangs der Translation der Kugel entspricht ist wesentlich einfacher und genauer, als zu versuchen, die Rotation (evtl. sogar bei wechselnden Geschwindigkeiten) durch Keyframes per Augenmaß zu realisieren. Häufig kommt die Skript-Technik in gebräuchlichen Animationspakten in Form sogenannter *Expressions* zum Einsatz, die Attribute eines Objektes aufgrund der Verarbeitung anderer Attribute berechnen.

Film 2-4: ScriptAnimation.mov

2.1.5 Partikelsysteme

Partikelsysteme bestehen aus einer Menge von Massepunkten im Raum, an die Objektgeometrien gebunden sein können. Auf diese Massepunkte werden simulierte physikalische Kräfte angewandt, wie zB. Gravitation oder Wind. In einem Simulationsprozess wird die Bewegung eines jeden Partikels aufgrund der Einwirkung solcher Kräfte berechnet. Ein Partikel hat neben der Position im Raum, der Geschwindigkeit und der Masse auch noch eine Zeitspanne, in der es existiert und folglich auch ein Alter. Partikel interagieren untereinander nicht. Der Hauptanwendungsbereich für Partikelsysteme ist die Generierung von Effekten wie Feuer, Rauch, aufgewirbeltem Staub oder Regen. [13] In begrenztem Umfang lassen sich aber auch Gruppenbewegungen sehr gut mit dieser Technik realisieren - dann nämlich, wenn Interaktion unter den einzelnen Mitgliedern nicht mehr erkennbar ist, zB. weil die Menge der Mitglieder sehr groß ist (Fliegenschwarm etc.). Bei Gruppenbewegungen jedoch, wo es dringend notwendig ist, daß Akteure miteinander agieren (zB. bei der Simulation eines Marathonlaufs), erweist sich die Animationstechnik der Partikelsysteme als untauglich.

Film 2-5: ParticleSystem.mov

2.2 Klärung des Begriffes

Der Ansatz bei *Verhaltensgesteuerter Animation* ist genau umgekehrt zum Partikelsystem: Der Akteur entscheidet zu jedem Zeitpunkt der Simulation, aufgrund der ihm vorliegenden Daten über seine Umgebung und Kenntnis seiner Ziele, in welche Richtung er sich mit welcher Kraft bewegt. Das bewegte Objekt wird zum sich bewegenden und autonom handelnen Agenten. Diese autonomen Agenten haben keine Lebensdauer, sondern sind im Rahmen der Simulation sozusagen unsterblich. Da dieses Modell dem Wesen einer tatsächlichen Gruppe, die ja auch aus autonom handelnen Lebewesen besteht, sehr nahe kommt, stellt es sich als geeignetes Konzept dar, um Gruppenbewegungen zu simulieren.

2.3 Ursprung

Das Konzept der Verhaltensgesteuerten Animation enstammt aus einem Unterbereich der *Künstlichen Intelligenz (KI)*, der *Verteilten künstlichen Intelligenz (VKI)*. Der Großteil der Forschung im Bereich der Künstlichen Intelligenz beschäftigt sich mit dem Problem, einem einzigen Agenten intelligentes Verhalten zu verleihen. Dieses intelligente Verhalten zeigt sich in der Lösung von Problemen mit Hilfe von heuristischen und wissenbasierten Methoden, Planung, Verstehen von Bildern und Sprache, Wahrnehmung und Lernen. Durch die Entwicklung leistungsfähiger Parallelrechner, die starke Verbreitung von

Netzwerken und die Übertragung der Beobachtung auf die KI, das viele Arbeitsaufgaben durch Menschen in Gruppen gelöst werden, ist dann die VKI entstanden.

VKI kann wieder in drei Forschungsbereiche untergliedert werden: *Verteilte Problemlösung* beschäftigt sich damit, wie ein Problem auf eine Menge von Modulen verteilt werden kann, die untereinander Wissen austauschen können über das zu lösende Problem und Lösungsmethoden. Im Bereich der *Parallelen Künstliche Intelligenz* wird an parallelen Computerarchitekturen, Programmiersprachen und Algorithmen für die KI gearbeitet. Der letzte, aber für diese Arbeit interessanteste, Bereich der VKI sind die sogenannten *Multiagentensysteme (MAS)*. Forschung im Bereich der Multiagentensysteme beschäftigt sich mit der Koordination von intelligentem Verhalten in einer Gruppe von intelligenten *Agenten*, wie sie ihr Wissen, Ziele, Fähigkeiten und Pläne zusammen koordinieren können um zusammen gewisse Handlungen zu vollziehen oder Probleme zu lösen. Agenten in einem MAS können zusammen auf ein einzelnes Ziel hinarbeiten, oder auf individuelle Ziele, die miteinander interagieren. Zwischen den einzelnen Agenten muß in einer gewissen Weise Koordination stattfinden, Informationen müssen zwischen ihnen ausgetauscht werden. [11]

Wie man jetzt klarer sieht, ist ein Animationssystem, das auf einem solchen Multiagentensystem aufsetzt, von seiner Grundstruktur sehr gut dazu geeignet, Gruppenprozesse aus der Sicht des Individuums zu simulieren. Wie ein solches System aufgebaut sein muß und welche Eigenschaften und Strukturen die autonomen Agenten aufweisen müssen, wird im folgenden näher erläutert.

2.4 3-Schichten Verhaltensmodell

Das Verhalten eines autonomen Agenten kann besser analysiert werden, wenn man es in drei Schichten aufteilt: *Motivation, Steuerung* und *Fortbewegung* [2,10,15].

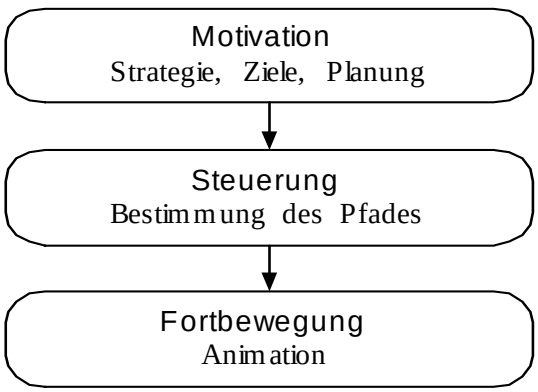


Bild 2-1: 3-Schichten Verhaltensmodell

An einem anschaulichen Beispiel kann dieses Schaubild erklärt werden: Ein Fahrradkurier bekommt von seinem Chef den Auftrag, ein Päcken in einer bestimmten Straße abzuliefern. Der Chef entspricht also der ersten Schicht, er gibt dem autonomen Agenten Kurier die Motivation, sich in Bewegung zu setzen. Dieser steuert (2. Schicht) sein Fahrrad (3. Schicht) in die gewünschte Strasse. In jeder Schicht ist es möglich, Komponenten auszutauschen. Der Kurier kann sich mit unterschiedlichen Motivationen in Bewegung setzen, die Art, wie er steuert kann sich ändern (z.B nach einem Fahrsicherheitstraining) und auch eine andere Art der Fortbewegung (sei es z.B. mit dem Motorrad oder dem Auto) ist relativ unabhängig von den anderen Schichten.

Die nachfolgende detaillierte Beschreibung der einzelnen Schichten beginnt mit der untersten Schicht, da an dieser Stelle das physikalische Grundmodell greift, auf dem die vorgestellte Lösung aufbaut.

2.5 Schicht 3: Fortbewegung

2.5.1 Physikalische Eigenschaften eines Agenten

Das hier vorgestellte Modell für die Fortbewegung (vgl. [15]) basiert auf *Massepunkt-Approximation*. Ein autonomer Agent wird durch einen Massepunkt definiert. Dieser Massepunkt hat eine Position (`Position`) und eine Masse (`Mass`). Zusätzlich besitzt er eine Geschwindigkeit (`Velocity`), die durch auf den Massepunkt angewandte Kräfte modifiziert wird. Da es sich um einen autonomen Agenten handelt, der ein Lebewesen repräsentiert, das diese Kräfte selbst aufbringt (z.B. durch Beschleunigung oder Abbremsung), sind die Kräfte beschränkt, hier vereinfacht durch den Parameter Maximale Kraft (`MaxForce`). Desweiteren wird die Geschwindigkeit durch die Parameter Maximale Geschwindigkeit (`MaxSpeed`) und Minimale Geschwindigkeit (`MinSpeed`) beschränkt. Durch die Orientierung (`Orientation`) kann zusammen mit der Position ein lokales Koordinatensystem für den Agenten definiert werden, an dem ein geometrisches Modell ausgerichtet werden kann.

```
Mass                    scalar
Position                vector
Velocity                vector
MaxForce                scalar
MinSpeed, MaxSpeed      scalar
Orientation             vector
```

2.5.2 Physik des Fortbewegungsmodells

Autonome Agenten bringen Kräfte auf, um die Position des Massepunkts im Raum zu verändern. Da die Agenten miteinander interagieren und folglich zu jedem Zeitpunkt die unterschiedlichsten Kräfte aufgebracht werden, ist es nicht möglich, die Bewegung direkt zu beschreiben (wenn dies der Fall gewe-

sen wäre, dann hätte man auch die Skript-Technik zur Animation verwenden können). Also muß aus den Kräften zu jedem Zeitpunkt berechnet werden, wie die Position des Agenten ausgehend vom aktuellen Status zu einem späteren Zeitpunkt sein wird [O-24]. Die dazu nötige Integration der sich aufgrund der *Newtonschen Gesetze* ergebenen Differentialgleichungen erfolgt durch das Verfahren der *Euler Integration*.

Bei der Euler-Integration handelt es sich um sehr einfaches Verfahren, um eine Differentialgleichung, die in den meisten Fällen nur sehr schwer oder überhaupt nicht auf analytischem Wege zu lösen ist, dennoch numerisch (aber meist nicht optimal) zu lösen. Es folgt eine kurze Darstellung des Verfahrens (vgl. [6,17]):

Gesucht wird: $x: \Re \rightarrow \Re$ mit $\frac{dx}{dt}(t) = f(x(t))$, $t \in \Re$ = Menge der reellen Zahlen

Die *Taylor-Näherung* für *x(t)* lautet:

$$x(t_n) \approx x(t_{n-1}) + x'(t_{n-1})(t_n - t_{n-1})$$

Setzt man nun $x(t_n) \approx x_n$ und $x(t_{n-1}) \approx x_{n-1}$, so erhält man eine explizite Form, um das System vom Zeitpunkt *n-1* zum Zeitpunkt *n* zu bewegen

$$x_n \approx x_{n-1} + x'(t_{n-1})(t_n - t_{n-1})$$

Setze $x'(t_{n-1}) = f(x_{n-1})$ und $\Delta t = (t_n - t_{n-1})$:

$$x_n \approx x_{n-1} + f(x_{n-1}) \cdot \Delta t$$

Wollte man das Verfahren geometrisch interpretieren, dann müßte man sagen: Der neue Funktionswert wird durch die Tangente im alten Funktionswert angenähert.

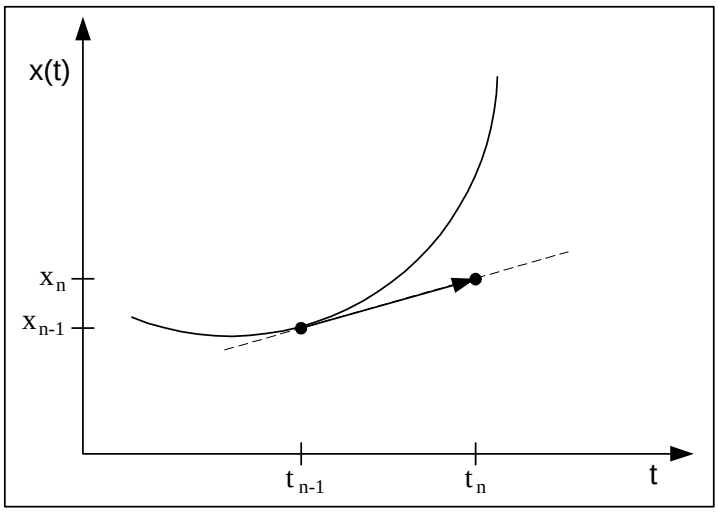


Bild 2-2: Verfahren der Euler-Integration

Das Euler-Integrationsverfahren ist insofern ungenau, als das während des ganzen Zeitschritts von *n-1* auf *n* der Funktionswert *f(x)* als konstant angesehen wird, was er aber in den meisten Fällen nicht ist. Trotzdem wird es im physikalischen Modell des Multiangentensystems verwendet und nicht eine der besseren numerischen Integrationsmethoden (zB. das *Verfahren von Heun*, die *Runge-Kutta Methode*), denn es hat den Vorteil, daß sich sehr einfach und schnell implementieren läßt. Es braucht keine zeitaufwendigen Berechnungen und ist von daher sehr schnell. Der Fehler (bestimmt durch die Schrittweite) hält sich auch bei dieser nicht optimalen Methode in Grenzen, wenn die Anzahl der Simulationsschritte (in jedem Schritt wird ja das Verfahren angewandt) pro Sekunde hoch ist, der Zeitabstand zwischen *n-1* und *n* also möglichst gering. Die gewonnen Ergebnisse werden auf das Modell des autonomen Agenten angewendet:

Bei jedem Simulationsschritt wird die in der Schicht 2 durch die Steuerverhalten bestimmte und durch den Parameter der Maximalen Kraft beschränkte Kraft (`DesiredSteeringForce`) auf den Massepunkt angewandt. Die dabei erzeugte Beschleunigung (`Acceleration`) kann durch die Division der Kraft mit der Masse berechnet werden. Die neue Geschwindigkeit wird gebildet, indem die Beschleunigung zur alten Geschwindigkeit addiert und durch die Parameter der Minimalen und Maximalen Geschwindigkeit beschränkt wird. Zum Schluß wird diese zur aktuellen Position addiert, um die neue Position zu erhalten.

```
SteeringForce := trunc (DesiredSteeringForce, MaxForce);[1]
Acceleration := SteeringForce / Mass;
Velocity := trunc(Velocity + Acceleration[2], MinSpeed, MaxSpeed);
Position := Position + Velocity[3]
Orientation := normalize(Velocity);
```

Wie zu sehen ist, wird die Orientierung der einzelnen Akteure in jedem Simulationsschritt durch inkrementelle Anpassung des vorhergehenden Simulationsschritt berechnet.

1. Die Funktion trunc liefert als Ergebnis einen Vektor, dessen Länge nach oben (bzw. unten) durch den 2. (und 3.) Parameter beschränkt wird.
2. Allgemein sieht die zugrundeliegende physikalische Gesetzmäßigkeit folgendermaßen aus: `Velocity := Velocity + Acceleration * Time`. Es fällt also auf, daß die Zeit `Time` vernachlässigt wurde. Dies ist in diesem Fall erlaubt, da die Simulation mit konstanten Zeitintervallen zwischen den einzelnen Berechnungsschritten arbeitet, die gleich 1 gesetzt werden.
3. Auch an dieser Stelle fällt wieder auf, daß die Zeit `Time` im Gegensatz zu der allgemeinen physikalischen Formel `Position := Position + Velocity * Time` nicht berücksichtigt wird. Dies ist aus dem Grund, der in FN 2.) dargelegt wurde, erlaubt.

2.6 Schicht 2: Steuerung

Auf die Darstellung der in dieser Arbeit analysierten Steuerverhalten folgen Ausführungen zur Kombination von Steuerverhalten [15].

2.6.1 Steuerverhalten

Die hier präsentierten Steuerverhalten liefern als Ergebnis einen Vektor für die auf den Massepunkt anzuwendene Kraft (`DesiredSteeringForce`), der als einziger Parameter von der Steuerschicht an die Fortbewegegungsschicht weitergeleitet und dort, wie oben beschrieben, verarbeitet wird. Bevor sie näher beschrieben werden, wird zuerst ein Modell vorgestellt, mit dem das Sehen für die autonomen Agenten simuliert werden kann.

Sichtfeld

Lebende Wesen nehmen ihre Umwelt auf den unterschiedlichsten Ebenen war, abhängig von den Sinnen, mit denen sie ausgestattet sind. Primär sind dies beim Menschen der Sehsinn, der Tastsinn und der Hörsinn. Andere Lebewesen nehmen ihre Umwelt auf anderen Ebenen war, reagieren vieleicht auf Licht in anderen Frequenzspektren, verlassen sich auf ihren Geruchssinn etc.. Die Evolution hat Lebewesen mit den jeweils nötigen und zweckdienlichen Möglichkeiten zur Wahrnehmung ihrer Umwelt ausgestattet.

Um das Verhalten von lebenden Wesen simulieren zu können, muß auch der Bereich der Wahrnehmung simuliert werden - genauso wie reale Akteure sich aufgrund ihrer Umwelt und ihrer Ziele bewegen, muß auch der autonome Agent seine Umwelt registrieren, analysieren und dann entsprechend seine Bewegung anpassen.

Das Sehen wird hier approximiert durch die Definition eines Sichtradius (`Radius`) und eines Sichtwinkels (`Angle`), die Winkelhalbierende des Sichtwinkels ist gleich orientiert mit dem aktuellen Ausrichtrung des autonomen Charakters. Dies bedeutet, daß der Agent nur in die Richtung schauen kann, in die er sich sich auch bewegt - ein Umherschauen ist also nicht möglich.

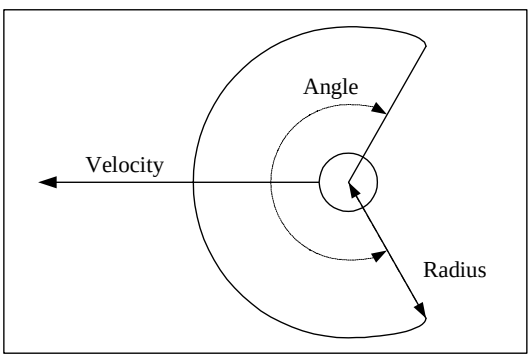


Bild 2-3: Sichtmodell eines autonomem Agenten

Diese Modell wird in der hier vorgestellten Lösung dazu benutzt, um es einem Agenten zu ermöglichen, andere Agenten oder ein Ziel in seinem durch das Blickfeld definierten Umfeld zu erkennen.

Im ersten Schritt werden alle autonomen Charaktere (`AgentsInViewCircle`) ermittelt, die sich im Sichtradius des Agenten (`Agent`) befinden. Dazu wird überprüft, ob die Länge des Differenzvektors (`DiffVector`) kleiner ist als der Radius (`Radius`) - ist dies der Fall, so befindet sich der im Moment überprüfte Agent innerhalb des durch den Radius definierten Kreises.

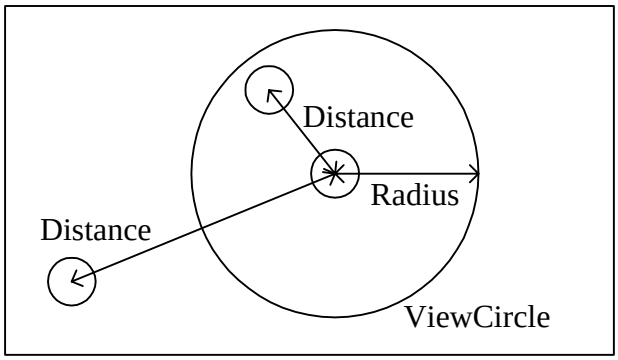


Bild 2-4: Inklusion im Sichtkreis des autonomen Agenten

```
for i = 0 to AgentList.Count
  if Agent != AgentList[i]
    then
      if length(Agent.Positon - AgentList[i].Position) <= Radius
        then
           AgentsInViewCircle.Add(AgentList[i])
```

Im zweiten Schritt werden die im ersten Schritt gefundenen Agenten daraufhin überprüft, ob sie sich auch im Sichtwinkel des Agenten befinden.

Grundlage für diesen Test ist die Definition des *Skalarproduktes*

$$ab = |a||b|\cos\varphi \quad wobei \quad \varphi \ (0 \le \varphi \le \pi)$$

$$\cos\varphi = \frac{ab}{|a||b|}$$

a und *b* bezeichnen die beiden Vektoren, die den Winkel φ einschliessen.

Ist jetzt der halbe Sichtwinkel (`0.5 * Angle`) größer als der Winkel zwischen den Vektoren der momentanen Geschwindigkeit (`Velocity`) und der Differenz der Positionen von Agent und ihn umgebenden Agent (`DistanceVector`), so befindet sich der überprüfte Agent im Blickwinkel des prüfenden Agenten.

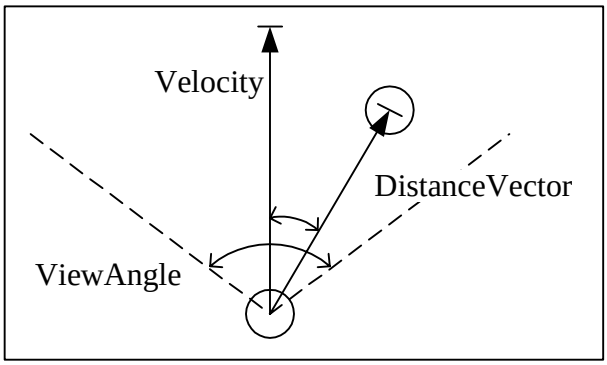


Bild 2-5: Inklusion im Blickwinkel des autonomen Agenten

```
VelocityLength  = length(Agent.Velocity);
CosViewAngleRad = cos(DegToRad(0.5 *Angle));

for i = 0 to AgentsInViewCircle.Count
  begin
    DistanceVector = length(Agent.Position
                            - AgentsInViewCircle[i].Position);
    Distance = length(DistanceVector);

    if (VelocityLength > 0) and (Distance > 0)
      then
        CosVectorAngle = Agent.Velocity * DistanceVector /
                      (VelocityLength * Distance);

    if CosVectorAngle <= CosViewAngleRad
      then
        NearbyAgentsList.Add(AgentsInViewCircle[i]);
  end;
```

In den folgenden Beschreibungen der Steuerverhalten (abgesehen vom *Hindernisse ausweichen*) wird davon ausgegangen, daß das Ziel (*Suchen und Fliehen, „Mottenflug"-Suchen und -Fliehen, Ankommen*) oder die Agenten im Blickfeld (*Ausrichten, Zusammenhalten, Abstand halten*) mit der oben vorgestellten Technik ermittel wurden.

Suchen und Fliehen

Suchen steuert den Agenten zu einer Position im Koordinatensystem. In den hier abgebildeten Schaubildern ist dies ein fester Punkt, in der Implementation ist es aber auch möglich, die Position eines sich bewegenden Akteurs als Ziel zu wählen.

Die gewünschte Geschwindigkeit (`DesiredVelocity`) ist ein Vektor von der Position des Agenten in Richtung des Ziels, seine Länge entspricht der maximalen Geschwindigkeit. Der endgültige Steuervektor ergibt sich aus der Differenz von gewünschter Geschwindigkeit und aktueller Geschwindigkeit.

```
DesiredVelocity := normalize(Position - Target) * MaxSpeed;
DesiredSteeringForce := DesiredVelocity - Velocity;
```

Soll der Charakter vor dem Ziel *fliehen*, so wird einfach die gewünschte Geschwindigkeit invertiert, so daß der Vektor in die entgegengesetzte Richtung zeigt (vgl. [10]).

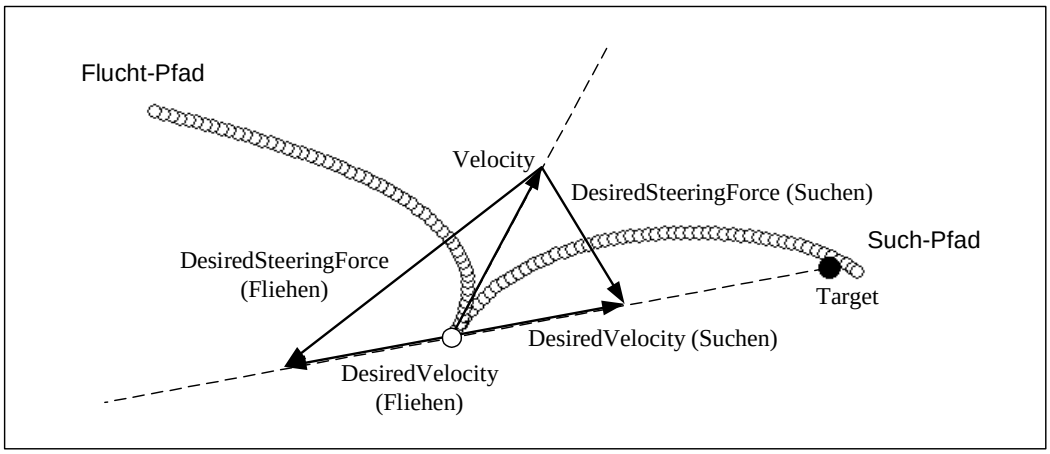


Bild 2-6: Schaubild: Suchen und Fliehen

Hat der Agent das Ziel erreicht, so durchquert er es und kehrt daraufhin wieder um, um es abermals zu durchqueren.

Simulation 2-1: schematic_SeekFlee.bgz
Simulation 2-2: anim_Seek.bgz
- eine Gruppe von Bugs, die einen Punkt im Raum „Suchen“

Film 2-6: Seek.mov

„Mottenflug“ - Suchen und Fliehen

Eine Vereinfachung von *Suchen* stellt dieses Steuerverhalten dar. Hierbei entspricht die vom Steuerverhalten gewünschte Kraft dem normalisierten Vektor zwischen Position und Ziel, bzw. dem Inversen dieses Vektors, wenn *Fliehen* simuliert werden soll.

```
DesiredSteeringForce := normalize(Position - Target);
```

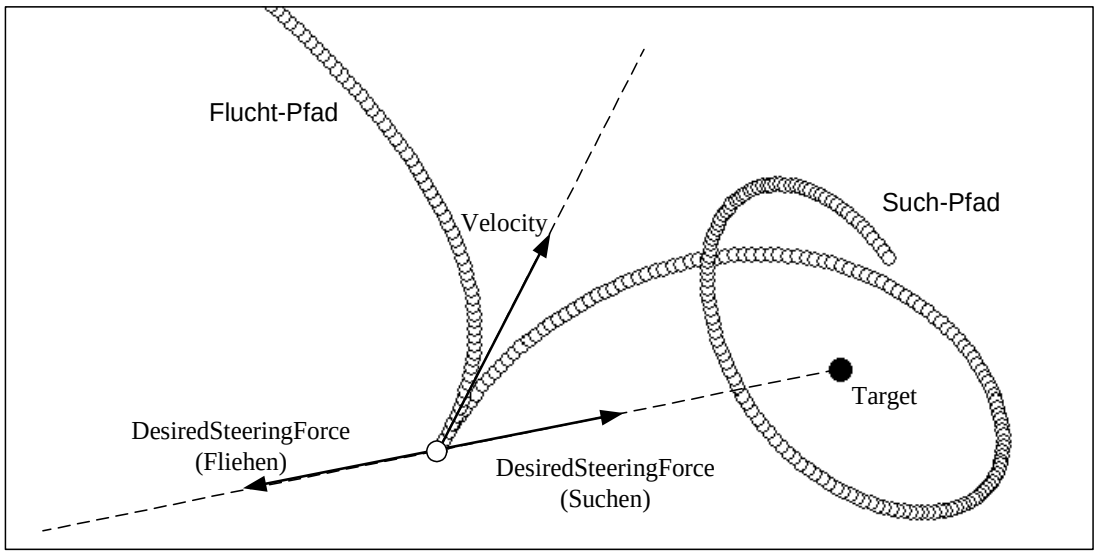


Bild 2-7: Schaubild: Suchen und Fliehen „Mottenflug“

Simulation 2-3: schematic_MothSeekFlee.bgz

Dieses Steuerverhalten veranlaßt den Charakter dazu, sich in elliptischen Bahnen um das Ziel bewegen, ähnlich einer Motte, die um das Licht schwirrt. Bewegt man das Ziel, so kann man mit diesem Steuerverhalten auf einfache Weise Bewegungsmuster erzeugen, die einem Fisch- oder Insektenschwarm sehr ähnlich sehen.

Simulation 2-4: anim_MothSeek.bgz
- Bugs, die auf einen sich bewegenden Bug „Motteflug-Suchen“ anwenden

Film 2-7: MothSeek.mov
Film 2-8: Fireflies.mov
Film 2-9: Flies.mov
Film 2-10: Fish.mov

Ankommen

Ein autonomer Agent der mit dem *Ankommen*-Steuerverhalten ausgestattet ist, bewegt sich zuerst mit dem Suchen-Steuerverhahlten auf sein Ziel zu. Ab einem definierbaren Abstand (`SlowingDistance`) zum Ziel, verringert er jedoch seine Geschwindigkeit kontinuierlich, um dann schließlich am Ziel die minimal mögliche Geschwindigkeit (`MinSpeed`) angenommen zu haben. Sollte die minimal mögliche Geschwindigkeit dem Wert 0 entsprechen, so bleibt der Agent beim Ziel stehen, er „kommt an“.

```
TargetOffset = length(Target - Position);
CurrentDistance = length(TargetOffset);
RampedSpeed = MaxSpeed * (CurrentDistance / SlowingDistance);
ClippedSpeed = minimum(RampedSpeed, MaxSpeed);
DesiredVelocity = (ClippedSpeed / Distance) * TargetOffset;
DesiredSteeringForce := DesiredVelocity - Velocity;
```

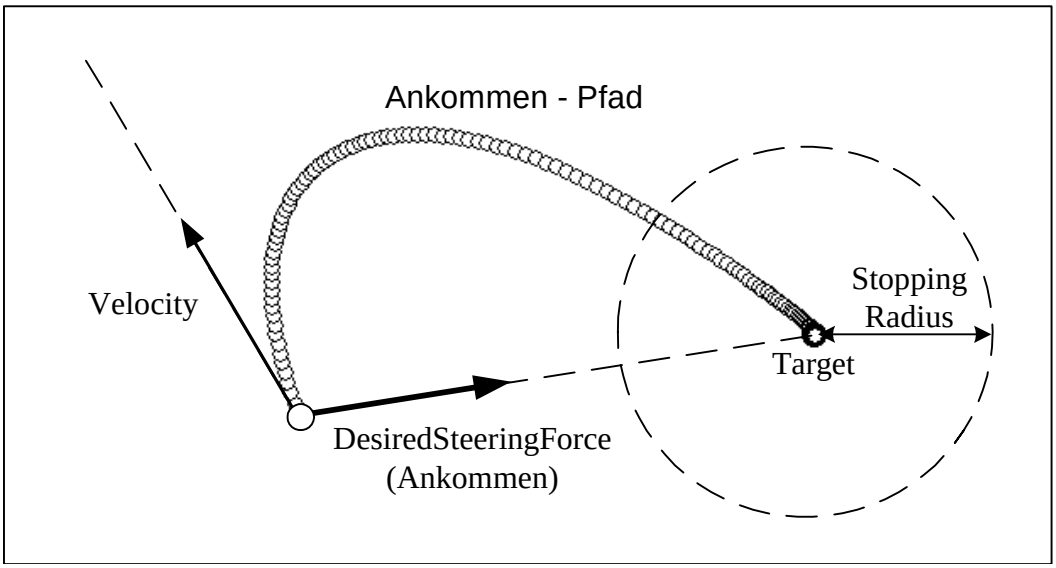


Bild 2-8: Schaubild: Ankommen

Simulation 2-5: schematic_arrival.bgz
Simulation 2-6: anim_arrival.bgz

Film 2-11: Arrival.mov

Ausrichten

Das Ausrichten Steuerverhalten ermöglicht es einem Agenten, sich selbst nach den andere Agenten in seinem Umfeld auszurichten. Der Agent findet alle anderen Agenten in seinem Sichfeld und aus den Geschwindigkeiten der einzelnen Geschwindigkeiten wird die mittlere Geschwindigkeit gebildet. Sie entspricht der gewünschten Geschwindigkeit des Agenten (`DesiredVelocity`), der Steuervektor entspricht der Differenz aus gewünschter Geschwindigkeit und aktueller Geschwindigkeit. Er richtet sich nach den anderen Agenten aus.

```
for i = 0 to NearbyAgentsList.Count
  SummVelocity += NearbyAgentsList[i].Velocity;

DesiredVelocity = SummVelocity / NearbyAgentsList.Count;
DesiredSteeringForce := DesiredVelocity - Velocity;
```

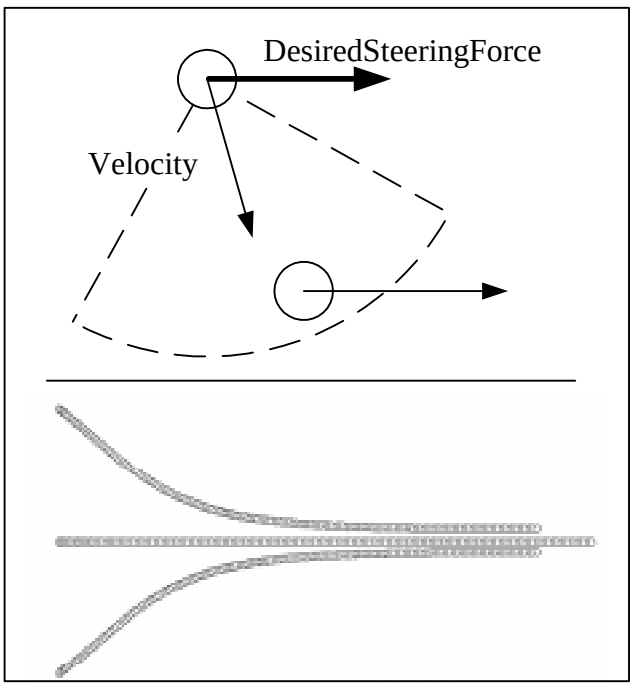


Bild 2-9: Schaubild: Ausrichten

Simulation 2-7: schematic_Alignment.bgz
Simulation 2-8: anim_Alignment.bgz

Film 2-12: Alignment.mov

Dieses Verhalten ist auch in der Natur zu beobachten, gerade in Gruppen mit vielen Mitgliedern ist es zwingend notwendig, ein ausgeglichenes Geschwindigkeitsniveau zu halten, um die Möglichkeit von Kollisionen und Konflikten möglichst gering zu halten.

Zusammenhalten

Autonome Agenten die sich nach dem *Zusammenhalten* Steuerverhalten bewegen, haben die Fähigkeit sich den Agenten in ihrem Blickfeld zu nähern und eine Gruppe mit ihnen zu bilden. Dazu wird aus den Positionen der Agenten im Blickfeld eine mittlere Position (`AveragePosition`) gebildet. In Richtung dieser mittleren Position soll der Agent steuern, also ergibt sich der Steuervektor aus der Differenz der mittleren Position mit der aktuellen Position des Agenten.

```
for i = 0 to NearbyAgentsList.Count
  SummPosition += NearbyAgentsList[i].Position;

AveragePosition = SummPosition / NearbyAgentsList.Count;
DesiredSteeringForce := AveragePosition - Position;
```

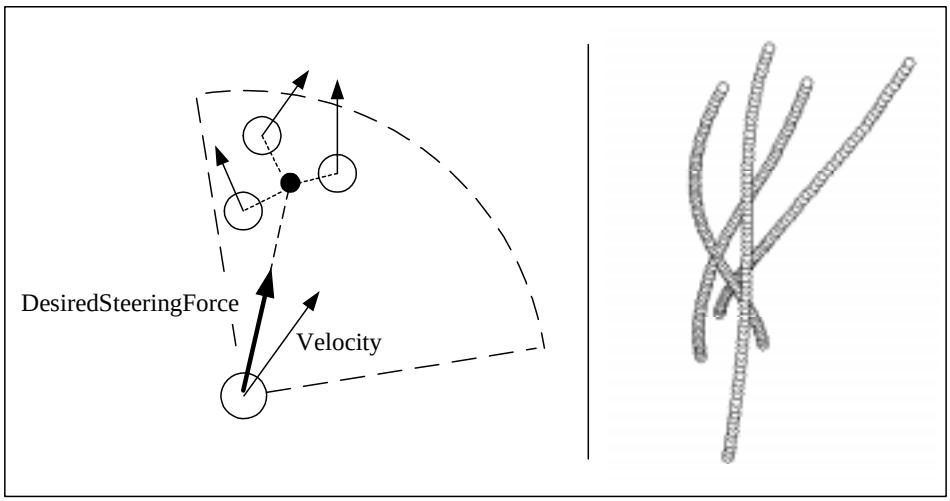


Bild 2-10: Schaubild: Zusammenhalten

Simulation 2-9: schematic_Cohesion.bgz

In der Natur ist das Innere einer Herde der sicherste und wärmste Platz - Tiere am Aussenrand sind weitaus gefährdeter gegenüber Angriffen und widrigen Umweltbedingungen. Also wird das Individuum versuchen, sich zwischen die anderen Gruppenmitglieder zu bewegen um so ihren Schutz geniessen zu können - genau dieses Verhalten wird hier simuliert, ohne auf die in der Natur an dem Prozess beteiligten Komponenten wie Rangordnung, Aufzucht und Schutz der Jungen etc. einzugehen.

Simulation 2-10: anim_Cohesion.bgz

Film 2-13: Cohesion.mov

Abstand halten

Das *Abstand halten* Steuerverhalten gibt einem Agenten die Fähigkeit, einen bestimmten Abstand von den anderen Agenten in seiner Umgebung zu halten. Dieses Steuerverhalten kann dazu genutzt werden, um Agenten davor zu bewahren, zusammenzustoßen oder sich auf einem Fleck anzusammeln.

Der Steuervektor dieses Verhaltens wird berechnet, indem die normalisierten Differenzvektoren von der Position des prüfenden Agenten zu den Positionen der zu prüfenden Agenten zuerst mit dem Faktor `1/(Anzahl der Agenten im Blickfeld)` gewichtet und dann aufsummiert werden. Der Gewichtungsfaktor ist kein festgesetzter unabänderlicher Wert, sondern einer, der zu guten Ergebnissen führt.

```
for i = 0 to NearbyAgentsList.Count
  begin
    DiffVector = Agent.Position - NearbyAgentsList[i].Position;
    SummVector += 1 /NearbyAgentsList.Count * normalize(DiffVector);
  end;

DesiredSteeringForce := SummVector;
```

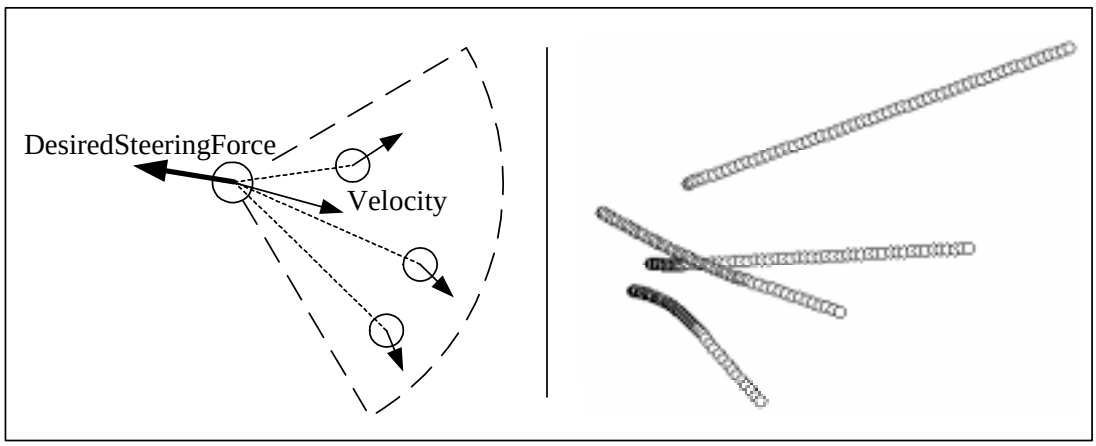


Bild 2-11: Schaubild: Abstand halten

Simulation 2-11: schematic_Separation.bgz
Simulation 2-12: anim_Separation.bgz

Film 2-14: Separation.mov

Dieses Steuerverhalten ist gerade bei gleichförmigen Bewegungen gut dazu geeignet, einen gewissen Abstand zwischen den einzelnen autonomen Agenten herzustellen. Bei nicht gleichförmiger Bewegung zeigen sich jedoch Schwächen, da die, für eine sichere Kollisionsvermeidung notwendige, Berechnung der zukünftigen Position der Agenten im Blickfeld fehlt.

Hindernissen ausweichen

Hindernisse ausweichen ist ein weiteres Steuerverhalten, das ein autonomer Agent aufweisen kann. Um Hindernisse wahrzunehmen, wird nicht das oben vorgestellte Konzept des Blickfeldes definiert durch Blickradius und Blickwinkel benutzt, sondern eine Methode, die man mit dem Tastsinn beim Menschen vergleichen könnte. An dem autonomen Agenten wird eine Strecke von vorgegebener Länge befestigt, die der Agent, wie ein Blinder einen Blindenstab, vor sich herführt. Diese Strecke hat die gleiche Orientierung wie der momentane Geschwindigkeitsvektor.

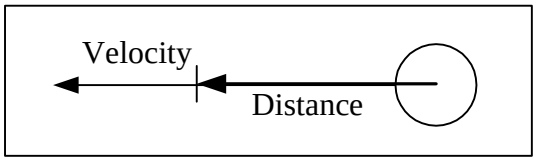


Bild 2-12: Tastmodell eines autonomen Agenten

Sobald dieser „Taststab“ auf ein Hindernis (`Obstacle`) trifft, wird der Schnittpunkt (`IntersectionPoint`) mit diesem Hindernis ermittelt und der *orthogonale Vektor* (`PerpendicularVector`) des Hindernisses im Schnittpunkt wird dem Agenten als gewünschter Steuervektor (`DesiredSteeringForce`) übergeben.

Es gibt zwei verschieden Arten von Hindernisse, die in diesem System behandelt werden: Strecken und Kreise. Wie exakt für den jeweiligen Hindernistyp Schnittpunkte berechnet und die orthogonalen Vektoren ermittelt werden, wird nun erläutert:

Der Schnitt von Sensorstab und einem linienförmigen Hindernis läßt sich als geometrische Problem „*Schnitt von zwei Strecken*“ lösen (vgl. [6,O-8])

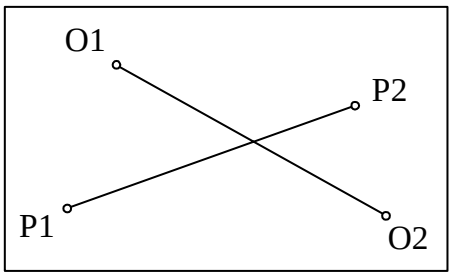


Bild 2-13: Schnitt von zwei Strecken

P sei die Menge aller Punkte U im $\Re^2$ mit den Komponenten (U.x, U.y)

$O1, O2 \in P$ bezeichnen die Endpunkte des Hindernisses
$P1, P2 \in P$ die Endpunkte des Sensorstabs mit

```
P1 = Agent.Position;
P2 = Agent.Position + normalize(Velocity)*SensorRange;
```

Die Gleichungen der Geraden durch die jeweiligen Endpunkte lauten:

$$P = \mathrm{P1} + m_P \cdot (\mathrm{P2} - \mathrm{P1})$$
$$O = \mathrm{O1} + m_O \cdot (\mathrm{P2} - \mathrm{O1})$$

Um den Schnittpunkt zu bestimmen, wird die Gleichung $P = O$ jeweils so umgeformt, daß sich eine Gleichung für m_P bzw. m_O ergibt.

$$P = O$$

$$\mathrm{P1.x} + m_P(\mathrm{P2.x} - \mathrm{P1.x}) = \mathrm{O1.x} + m_O(\mathrm{O2.x} - \mathrm{O1.x})$$

$$\mathrm{P1.y} + m_P(\mathrm{P2.y} - \mathrm{P1.y}) = \mathrm{O1.y} + m_O(\mathrm{O2.y} - \mathrm{O1.y})$$

$$m_P = \frac{(\mathrm{O2.x} - \mathrm{O1.x})(\mathrm{P1.y} - \mathrm{O1.y}) - (\mathrm{O2.y} - \mathrm{O1.y})(\mathrm{P1.x} - \mathrm{O1.x})}{(\mathrm{O2.y} - \mathrm{O1.y})(\mathrm{P2.x} - \mathrm{P1.x}) - (\mathrm{O2.x} - \mathrm{O1.x})(\mathrm{P2.y} - \mathrm{P1.y})}$$

$$m_O = \frac{(\mathrm{P2.x} - \mathrm{P1.x})(\mathrm{P1.y} - \mathrm{O1.y}) - (\mathrm{P2.y} - \mathrm{P1.y})(\mathrm{P1.x} - \mathrm{O1.x})}{(\mathrm{O2.y} - \mathrm{O1.y})(\mathrm{P2.x} - \mathrm{P1.x}) - (\mathrm{O2.x} - \mathrm{O1.x})(\mathrm{P2.y} - \mathrm{P1.y})}$$

Haben sowohl Zähler- als auch Nenner-Term den Wert 0, so liegen die beiden Geraden P und O aufeinander. Hat nur der Nenner den Wert 0, so sind die beiden Geraden parallel, hat er einen anderen Wert, so sind die obigen Gleichungen für die relativen Distanzen von P zu O entlang des Strecke (m_P bzw. m_O) definiert und der Schnittpunkt $I \in \mathrm{P}$ ergibt sich wie folgt:

$$\mathrm{I.x} = \mathrm{P1.x} + m_P(\mathrm{P2.x} - \mathrm{P1.x})$$
$$\mathrm{I.y} = \mathrm{P1.y} + m_P(\mathrm{P2.y} - \mathrm{P1.y})$$

Bei dem so ermittelten Schnittpunkt handelt es sich jedoch um den Schnittpunkt der Geraden, die Strecken schneiden sich, wenn $0 \le m_P \le 1$ und $0 \le m_O \le 1$.

```
denomin = (O2.y-O1.y)*(P2.x-P1.x) - (O2.x-O1.x)*(P2.y-P1.y);

if denomin != 0
  then
    begin
      mp = ((O2.x-O1.x)*(P1.y-O1.y) - (O2.y-O1.y)*(P1.x-O1.x)) / denomin;
      mo = ((P2.x-P1.x)*(P1.y-O1.y) - (P2.y-P1.y)*(P1.x-O1.x)) / denomin;

      if (mp >= 0) and (mp <= 1) and (mo >= 0) and ( mo <= 1)
        then
          begin
            I.x := P1.x + mp*(P2.x-P1.x);
            I.y := P1.y + mp*(P2.y-P1.y);
          end;
    end;
```

Wurde auf diese Weise ein Schnittpunkt ermittelt, so besteht die Gefahr, daß der autonome Agent mit dem Hindernis kollidieren kann, wenn er nicht die Richtung ändert. Der Steuervektor, den das Steuerverhalten erzeugen muß, ist der orthogonale Vektor zum Hindernis im Schnittpunkt.

Zwei Vektoren sind zueinander orthogonal, wenn das Skalarprodukt gleich 0 ist. Hieraus ergibt sich für einen orthogonalen Vektor (`Perpendicular`) zur Hindernisstrecke definiert durch *O1, O2*:

```
Difference = O1 - O2;
Perpendicular[0] = - Difference.y / Difference.x;
Perpendicular[1] = 1;
Perpendicular = normalize(Perpendicular);
```

Hat die x-Komponente des Differenzvektors den Wert 0 so ergibt sich für den orthogonalen Vektor.

```
Perpendicular.x = - 1;
Perpendicular.y = 0;
```

Der so gewonnenen normalisierte senkrechte Vektor muß nicht zwangsläufig in die Richtung zeigen, die den Agenten dazu veranlassen würde, vom Hindernis wegzusteueren. Um sicher zu gehen, daß dies der Fall ist, muß sicher gegangen werden, daß der Vektor in Richtung der Seite zeigt, auf der sich auch der autonome Charakter befindet.

Hierzu wird die eigentlich 2-dimensionale Simulation für die Berechnung indie 3. Dimension erweitert - das Kreuzprodukt wird aus dem Richtungsvektor des Hindernisse (`ov`) und dem orthogonalen Vektor (`pv`) gebildet, ebenso wie das Kreuzprodukt aus dem Richtungsvektor des Hindernisse und dem Richtungsvektor von O1 des Hindernisses zur Position des Agenten (`av`). Hierdurch wird ein neuer Vektor erzeugt, der senkrecht auf den beiden anderen steht und mit ihnen zusammen ein Rechtssystem erzeugt.

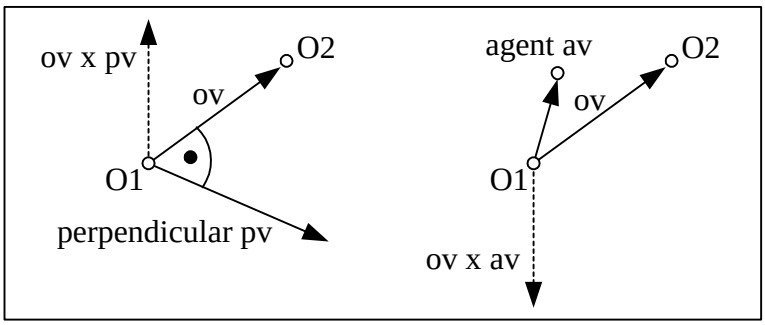


Bild 2-14: Orientierung des Ergebnisses eines Kreuzprodukts

Zeigen die beiden Vektoren auf verschiedene Seiten des Hindernisses, so wird sich das Vorzeichen der z-Komponente des erzeugten Vektors ändern. Sollte dies der Fall sein, so muß der orthogonale Vektor invertiert werden.

```
function DetermineSide(P1,P2,P3:vector):integer;
begin
  Diff1 = P1- P2;
  Diff2 = P1 - P3;
  if (Diff1.x * Diff2.y - Diff2.x * Diff1.y) < 0
    then Result := 1
    else Result := -1;
end;

AgentSideIndicator := DetermineSide(O1,O2,Agent.Position);
PerpendicularSideIndicator := DetermineSide(O1,O2,O1+Perpendicular);

if BugSideIndicator != PerpendicularSideIndicator
   then DesiredSteeringForce = -Agent.MaxSpeed * Perpendicular
   else DesiredSteeringForce = Agent.MaxSpeed * Perpendicular;
```

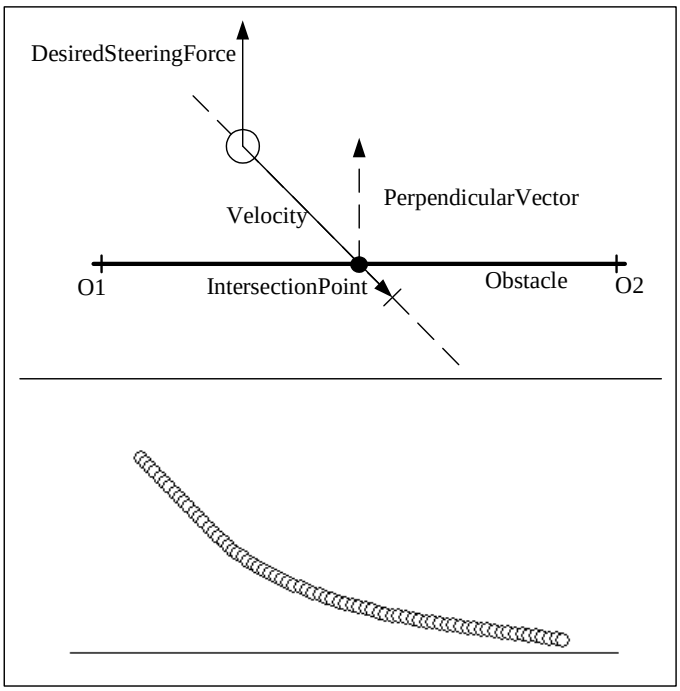


Bild 2-15: Hindernissen ausweichen, Linie

Simulation 2-13: schematic_LineObstacleAvoidance.bgz
Simulation 2-14: anim_LineObstacleAvoidance.bgz

Film 2-15: LineObstacleAvoidance.bgz

Soll der autonome Agent kreisförmigen Hindernissen ausweichen, so muß der Schnittpunkt zwischen Kreis und Strecke ermittelt werden (vgl. [O-8, O-12]). Hierzu wird davon ausgegangen, daß durch die beiden Punkte *P1* und *P2* eine Gerade beschrieben wird, die einen Kreis mit dem Radius *r* schneidet, der seinen Mittelpunkt im Ursprung hat. Um die Menge der Schnittpunkte zu bestimmen werden folgende Variablen für die Gerade definiert:

$$dx = P2.x - P1.x$$

$$dy = P2.y - P1.y$$

$$D = \begin{vmatrix} P1.x & P2.x \\ P1.y & P2.y \end{vmatrix} = P1.x \cdot P2.y - P2.x \cdot P1.y$$

Sowie für den Kreis

$$dr^2 = dx^2 + dy^2$$

Für die Schnittpunkte ergibt sich nach Auflösen und Umformen der quadratischen Gleichung folgende Lösung:

$$I.x = \frac{Ddy \pm sgn(dy)dx\sqrt{\Delta}}{dr^2}$$

$$I.y = \frac{-Ddx \pm |dy|\sqrt{\Delta}}{dr^2}$$

Δ bezeichnet die Diskriminante, sie ist folgendermassen aufgebaut:

$$\Delta = r^2dr^2 - D^2$$

Die Diskriminante bestimmt, wie sich die Gerade zum Kreis verhält:

$\Delta < 0$ Die Gerade schneidet den Kreis nicht.
$\Delta = 0$ Die Gerade liegt tangential zum Kreis, sie berührt ihn in einem Punkt.
$\Delta > 0$ Die Gerade schneidet den Kreis in zwei Punkten.

Das hier verwendete Verfahren bezieht sich auf einen Kreis, der seinen Mittelpunkt im Ursprung hat. Sollen auch Kreise mit andern Mittelpunkten verwendet werden, so brauchen nur alle in die Berechnung involvierten Punkte der durch die Verschiebung des Kreismittelpunkts in den Ursprung vorgegebenen Translationstransformation unterzogen werden und im Anschluß an die Berechnung wieder zurücktransformiert werden.

Durch die obige Berechnungsmethode erhält man eine Menge von Schnittpunkten, die entweder null, ein oder zwei Elemente enthält. Diese bezeichnen

die Schnittpunkte der Geraden mit dem Kreis. Da aber nicht die Schnittpunkte der Geraden berechnet werden sollen, sondern die der Strecke, müssen die gewonnenen Punkte (`I1, I2`) daraufhin untersucht werden, ob ihr Abstand (`Distance1, Distance2`) von der Position des Agenten (`Agent.Position = P1`) kleiner ist als die Länge des Sensorstabs (`SensorRange`). Ist dies der Fall, so muß muß überprüft werden, ob sich der Schnittpunkt in Blickrichtung (`SensorHeading`) des autonomen Charakters befindet, damit Punkte, die hinter dem Agenten liegen, nicht in die Berechnung mit einfliessen. Ist so die Menge der Schnittpunkte auf die Menge reduziert worden, die tatsächlich auf der Strecke liegen, muß noch bestimmt werden, welcher der Punkte den geringsten Abstand zum Agenten hat, denn nur auf diesen soll das Steuerverhalten reagieren, da er auf dem Hindernis liegt, das zum momentanen Simulationszeitpunkt die größte Gefahr darstellt.

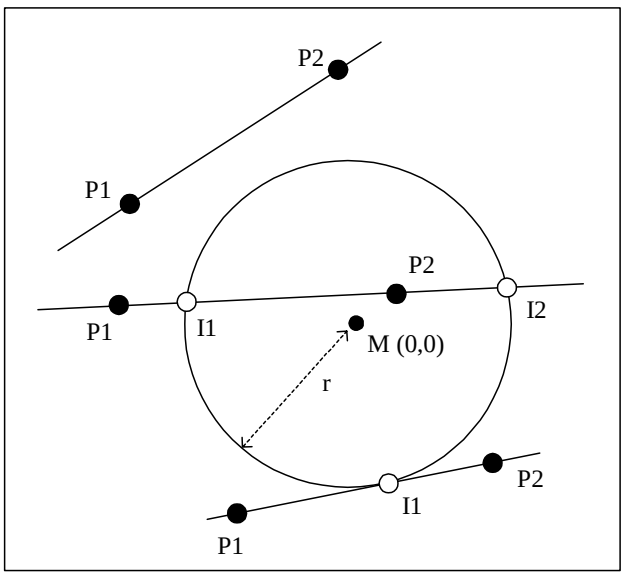


Bild 2-16: Schnitt von Gerade und Kreis

```
function DirectionEquals(a,b:Vector):boolean;
begin

  Result := (sgn(a[0]) = sgn(b[0])) and (sgn(a[1]) = sgn (b[1]));

end;

SensorHeading = P1 - P2;

dx = P2.x - P1.x;
dy = P2.y - P1.y;
drr = sqr(dx) + sqr(dy);
D = P1.x*P2.y - P2.x*P1.y;

Discriminant = sqr(r)*drr - sqr(D);
```

```
if not (Discriminant < 0)
  then
     begin

        temp1 = sqrt(sqr(r)*drr-sqr(d));
        temp2 = sgn(dy) * dx * temp1;
        temp3 = abs(dy) * temp1;

        I1.x = (d*dy + temp2) / drr;
        I1.y = (-d*dx + temp3) / drr;

        DiffVector = P1 - I1;
        Distance1 := length(DiffVector);

        Intersection1Valid = DirectionEquals(DiffVector, SensorHeading)
                             and ( Distance1 < SensorRange );

        if Intersection1Valid
           then
              IFinal = I1;

        if Discriminant > 0
          then
            begin

              I2.x := (d*dy - temp2) / drr;
               I2.y := (-d*dx - temp3) / drr;

              DiffVector = P1 - I2;
               Distance2 := length(DiffVector);

              if DirectionEquals(DiffVector, SensorHeading)
                   and (Distance2 < SensorRange)
                then
                  if IntersectionOneValid
                    then
                       begin

                         if (Distance2 < Distance1)
                           then
                             IFinal =I2;
                       end
                    else
                       IFinal =I2;

            end;
      end;
```

Der so gewonnenen Schnittpunkt (`IFinal`) ist der Punkt, an dem die Orthogonale zum Kreis angesetzt werden muß, deren Orientierung der vom Steuerverhalten gelieferte Vektor (`DesiredSteeringForce`) hat. Diese entspricht dem Differenzvektor vom (rücktransformierten) Mittelpunkt des Kreises (`Center`) und Schnittpunkt. Die Richtung ist entgegengesetzt zur Orientierung des Agenten, so werden Agenten im Kreis eingesperrt und Agenten ausserhalb abgelenkt.

```
if  length(Agent.Position - Center) < r
  then
     DesiredSteeringForce = Center - IFinal;
  else
     DesiredSteeringForce = IFinal - Center;

DesiredSteeringForce = normalize(DesiredSteeringForce);

DesiredSteeringForce = MaxSpeed * DesiredSteeringForce;
```

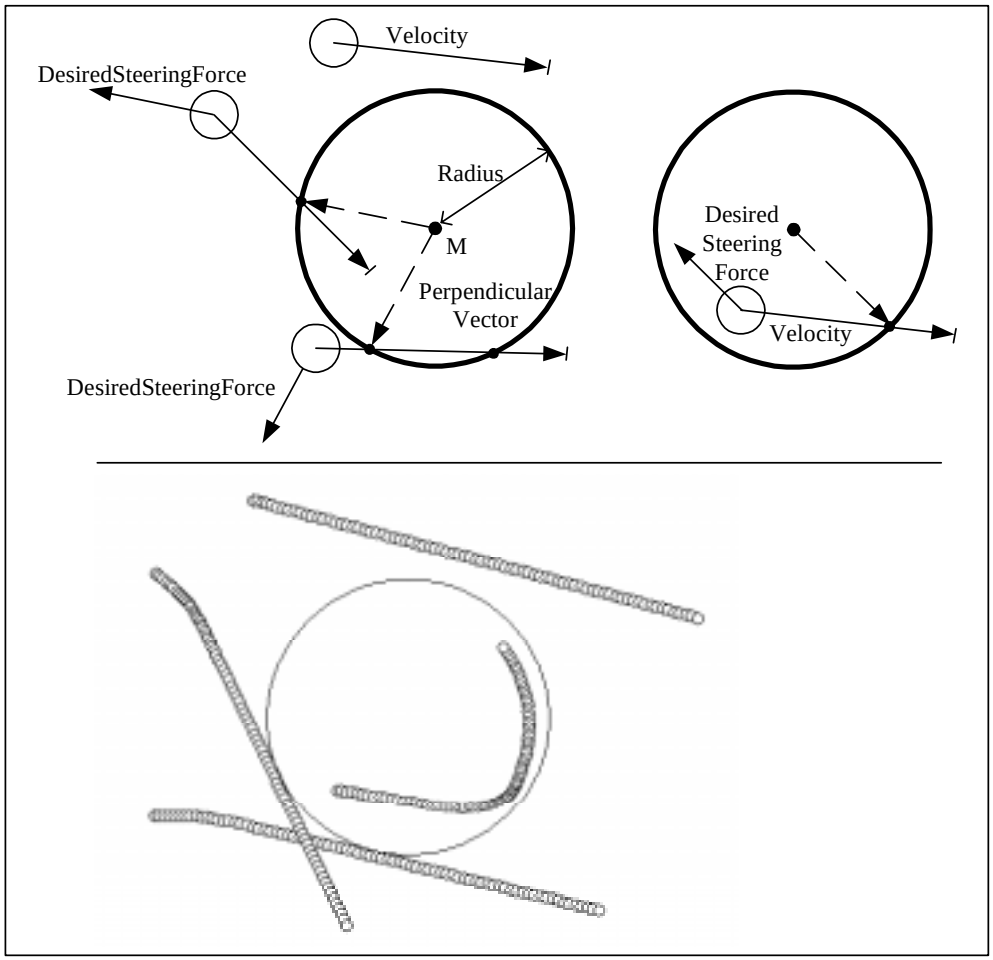


Bild 2-17: Schaubild: Hindernissen ausweichen, Kreis

Simulation 2-15: schematic_CircleObstacleAvoidance.bgz
Simulation 2-16: anim_CircleObstacleAvoidance.bgz

Film 2-16: CircleObstacleAvoidance.bgz

2.6.2 Kombination von Steuerverhalten

Besitzt ein autonomer Agent eins der oben beschriebenen Steuerverhalten, so lassen sich bereits dadurch, wie in den Beispielsimulationen zu sehen war, interessante und komplexe Bewegungsmuster sowohl für den einzelnen Agenten, als auch für eine Gruppe von Agenten erzeugen. Manche Arbeitsaufgaben erfordern es aber, daß sich der Agent nach mehr als einem Steuerverhalten richtet. So ist zB. durchaus denkbar, daß ein Agent sowohl Hindernissen ausweichen soll, als auch einen anderen Agenten verfolgen. Um solche Effekte zu erreichen, müssen die entsprechenden Steuerverhalten auf angemessene Weise kombiniert werden.

Im folgenden werden 2 verschiedene Möglichkeiten der Kombination vorgestellt.

Lineare Kombination

Jedem Steurverhalten wird ein Gewicht zugeordnet, mit dem es an dem endgültigen Steuervektor beteiligt ist. Haben die Gewichte den gleichen Betrag, so ergibt sich der Steuervektor als mittlerer Vektor der einzelnen Steuerungsvektoren der Steuerverhalten.

```
for i = 0 to SteeringBehaviourList.Count
with SteeringBehaviourList[i] do
  begin
    SummForce += Weight * DesiredSteeringForce;
    SummWeight += Weight;
  end;

DesiredSteeringForce = SummForce / SummWeight;
```

Das Problem dieser linearen Kombination ist, daß auch Steuerverhalten in die aktuelle Bewegung einfliessen, die zum aktuellen Simulationsschritt vieleicht überhaupt keinen Steuervektor erzeugen (zB. weil sich kein dafür notwendiges Ziel im Sichtbereich befindet). Abhilfe schafft die modifizierte lineare Kombination.

```
for i = 0 to SteeringBehaviourList.Count
  with SteeringBehaviourList[i] do
    if length(DesiredSteeringForce) <> 0
      then begin
        SummForce += Weight * DesiredSteeringForce;
        SummWeight += Weight;
      end;

DesiredSteeringForce = SummForce / SummWeight;
```

Hierbei bekommen nur noch die Steuerverhalten einen Einfluß auf die endgültigen Steuervektor, die auch wirklich selbst einen Steuervektor produzieren. Kritikpunkt an dieser Weise der linearen Kombination ist, daß sich der Einfluß

der für die Verhalten angegebenen Gewichte in Bezug auf die Menge aller Steuerverhalten bei einer kleineren Anzahl von Verhalten entsprechend verändert wird.

Die Kombination von Steuerverhalten auf linearem Weg ergibt zwar in vielen Fällen gute Ergebnisse, hat aber wenigstens zwei Nachteile: Zum einen ist diese Methode nicht besonders effizient in Bezug auf Rechenzeit, denn in jedem Simulationsschritt müssen für jeden autonomen Agenten alle Steuerverhalten ausgeführt und dann kombiniert werden. Zum anderen können die Steuervektoren sich zu ungünstigen Zeitpunkten entweder aufheben oder dafür sorgen, daß der endgültige Steuervektor in eine Richtung zeigt, die nicht gewünscht ist. Man stelle sich nur ein Charakter vor, der an einer Kreuzung steht, ausgestattet mit zwei Steuerverhalten. Das erste liefert einen Vektor, der geradeaus zeigt, das zweite einen, der nach rechts zeigt - obwohl jeder der beiden Vektoren den Charakter dazu veranlassen würde, einen gültigen Weg einzuschlagen, führt die einfache lineare Kombination zu keinem gewünschten Ergebnis.

Kombination von priorisierten Steuerverhalten

Eine Lösung des oben beschriebenen Problems ist es, den einzelnen Steuerverhalten Prioritäten zuzuweisen. Zuerst wird das Steuerverhalten ausgewertet, dessen Priorität am höchsten ist, nur wenn dieses einen Steuervektor liefert, der den Betrag 0 hat, wird das darauffolgende Verhalten ausgewertet und so weiter.

```
FinalDesiredSteeringForce = (0,0);
i = 0;

while (length(FinalDesiredSteeringForce) = 0)
       or (i < OrderdSteeringBehaviourList.Count) do
   begin
     FinalDesiredSteeringForce =
                    OrderdSteeringBehaviourList[i].DesiredSteeringForce;
     i++;
   end;
```

Diese Art der Kombination von Steuervektoren erweist sich als besonders nützlich, wenn man in bestimmten Situationen sicher gehen will, daß ein Steuerverhalten die Bewegung des Agenten bestimmt. Gibt man zB. der Ausweichen von Hindernissen die höchste Priorität, so kann man sich sicher sein, daß der autonome Agent dies auch ausführt, wenn es benötigt wird.

Simulation 2-17: anim_GroubObstacleAvoidance.bgz
- Bugs versuchen, nicht mit dem Hindernis zusammenzustoßen (Hindernissen ausweichen), nicht untereinander zusammenzustoßen (Abstand halten) und ein gemeinsames Ziel zu erreichen (Suchen) (Reihenfolge entspricht der Priorisierung)

Film 2-17: GroubObstacleAvoidance.mov

Simulation 2-18: anim_PathFlocking.bgz
- Bugs versuchen, nicht mit den Hindernissen zusammenzustoßen (Hindernissen ausweichen), nicht untereinander zu kollidieren (Abstand halten), ihre Geschwindigkeit den anderen in der Umgebung anzupassen (Ausrichten) und eine Gruppe mit den Bugs im Sichfeld zu bilden (Zusammenhalten)

Film 2-18: PathFlocking.mov

2.6.3 Anmerkungen zu Steuerverhalten

In einem Multiagentensystem, in dem sich die autonomen Charaktere nach den Regeln bewegen, die durch die vorgestellten Steuerverhalten vorgegeben werden, kann man nicht davon ausgehen, daß alle Regeln zu allen Zeitpunkten dafür sorgen, daß keine Konfliktituationen auftauchen. Dies ist darin begründet, daß sich Steuerverhalten gegenseitig widersprechen können - obwohl zB. zwei Charaktere versuchen, sich auszuweichen, kann es trotzdem passieren, daß sie sich berühren.

Dies ist ein Problem, das man in einem solchen offenen System schwer oder garnicht ausmerzen, wohl aber durch geschickte Wahl der physikalischen Eigenschaften und Steuerverhalten verkleinern kann. Selbst in der realen Welt funktioniert eine konfliktlose Bewegung zusammen mit vielen anderen Personen auch nicht immer reibungslos. Immer wieder kommt es vor, daß zwei Personen zusammenstossen oder sich einfach nur gegenseitig im Weg stehen, vieleicht sogar beim Ausweichen die gleiche Seite wählen, um sich dann wieder gegenüber zu stehen.

2.7 Schicht 1: Motivation

Die Gruppe bietet dem Individuum viele Vorteile: Sie bietet Schutz und Sicherheit vor Angreifern. In Fischschwärmen macht die schiere Anzahl der einzelnen Fischen es einem Räuber sehr schwer, ein einzelnes Opfer zu fixieren. Der verwundbare Umfang einer Herde steigt nur linear, während sich der sichere innere Bereich der Herde quadratisch vergrößert, wenn neue Tiere hinzukommen. Die Gruppe eröffnet neue Möglichkeiten der Problemlösung, egal ob es sich dabei um Jagdstrategien oder die Aufzucht der Jungen handelt. Nicht zuletzt bietet die Gruppe Geborgenheit, Wärme und die Möglichkeit einen

Partner für die Fortpflanzung zu finden. Obwohl die Motivation für Gruppenbildung und -verhalten einen interessanten Forschungsbereich darstellt, so ist sie denoch nicht dazu notwendig, um Gruppenverhalten zu simulieren.

Komplexe Bewegungsmuster können auch allein auf Basis der präsentierten Steuerverhalten erzeugt werden. Diese legt in der Computeranimation der Animator fest, der die Ziele und Pläne der autonomen Agenten kennt. Er hat, meist durch das Drehbuch und das Storyboard fest vorgegebene Vorstellungen davon, wie sich die Gruppe von Akteuren bewegen soll. Rechnergestützte Methoden zur Wahl der Steuerverhalten würden hier z.B. im Anwendungsfall eines Computerspiels mit autonom handelnen Agenten (etwa einem Strategiespiel) greifen.

2.8 Verbesserungsmöglichkeiten

Das hier vorgestellte System dient als erste Annäherung an die Problematik der Simulation von Gruppenbewegungen. Es zeigen sich viele Verbesserungsmöglichkeiten an fast allen präsentierten Bereichen.

2.8.1 Physikalisches Modell

Das benutzte physikalische, auf Massepunkt-Aproximation basierende, Modell ist zwar einfach zu verstehen und effizient in der Berechnung, hat aber den Nachteil, daß es nicht besonders realistisch ist. Es gibt keine Massepunkte in der Natur, sie sind nur eine Vereinfachung der wahren physikalischen Beschaffenheiten. Jedes Objekt mit Masse muß eine räumliche Ausdehnung haben und folglich auch ein, nicht simuliertes, Trägheitsmoment. Würde man dieses simulieren, so könnte man dadurch die manchmal auftretenen, nicht erwünschten großen Änderungen in der Orientierung von einem Simulationsschritt zum nächsten mindern.

Die Bewegung des autonomen Charakters wird nur duch eine einzige Kraft bestimmt, die immer die gleiche Orientierung hat wie der Charakter. Dies hat zur Folge, daß Bewegungen wie Rutschen und Schleudern nicht simuliert werden können. Bei realistischeren Modellen eines Agenten würde es mehrere Vektorgrößen zur Kontrolle geben, so könnte zB. ein Auto vier Steuervektoren haben (steuern rechts, steuern links, beschleunigen, abbremsen), die auch in ihrer maximalen Länge unterschiedlich begrenzt wären - das Auto könnte beispielsweise schneller bremsen als beschleunigen.

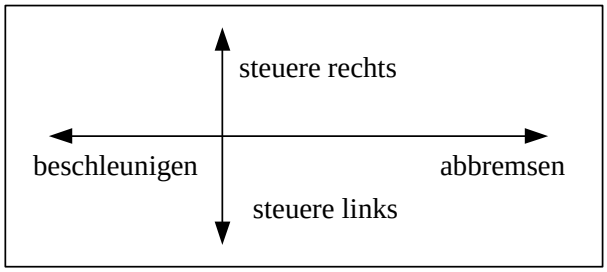


Bild 2-18: Alternatives 4-Kräfte-Modell für autonome Agenten

2.8.2 Steuerverhalten

Abgesehen davon, daß sich die Liste der sinnvollen Steuerverhalten sich sicher noch erweitern liesse (z.B. mit solchen, die Algorithmen zur Wegplanung beinhalten), könnte auch noch auf anderem Wege intelligenteres Verhalten erzeugt werden. Es wäre durchaus denkbar, jedem Agenten oder wahlweise auch der Gruppe aller Agenten, quasi als kollektives Gedächtnis, eine lernfähige Struktur z.B. in Form eines neuronalen Netzes zu verleihen. Dieses könnte auf Basis der lokalen Umwelt und der bereits gemachten Erfahrungen ein der momentanen Situation entsprechendes Steuerverhalten auswählen - die reine Reaktion würde um die Lernkomponente erweitert werden. Die Schicht 1 des 3-Schichten-Modells wäre nicht mehr nur dem Animator überlassen.

Innerer Zustände mit zugeordneten Mengen von Steuerverhalten wären ein weiteres adäquates Mittel, um die Bewegung der Agenten der Situation anzupassen bzw. ein größeres Verhaltensspektrum zu simulieren. So wie eine Gazelle die in der Savanne grast (innerer Zustand: Fressen) sicher vollkommen andere Steuerverhalten zeigt, als die gleiche Gazelle, die von einem Geparden angegriffen wird (innerer Zustand: Flucht). Neben den innerern Zuständen mit den zugehörigen Steuerverhalten müßte auch noch ein System entwickelt werden, das die Wahl des jeweiligen inneren Zustandes aufgrund der Umwelt des Agenten steuert.

2.8.3 Simulationsraum

Bei dem hier vorgestellten System handelte es sich um ein System, in dem sich die Agenten rein nach ihren Steuerverhalten bewegen. Im Simulationsraum, der Arena, befindet sich ausser den autonomen Charakteren selbst nichts. Selbst Hindernisse sind keine richtigen Hindernisse, sondern nur Parameter des *Hindernisse ausweichen* Steuerverhaltens, sie existieren in der Simulation nicht als physische Objekte. Dies hat folgende Konsequenzen: Da Hindernisse nicht physisch sind, können Agenten sich problemlos durch sie hindurch bewegen, ein Kollisionscheck erfolgt nicht. Auch wird durch Hindernisse die Sicht nicht eingeschränkt, ein Ziel hinter einem Hindernis ist für einen sich da-

vor befindenen Agenten ohne Probleme auszumachen.

Das gleiche Problem gilt natürlich auch für die Agenten selbst, da sie ja nur Massepunkte sind und keine wirkliche physische Ausdehnung haben, können sie sich auch zur gleichen Zeit am exakt selben Ort befinden und sich überlappen.

Der gesamte, bis jetzt noch recht leere, Simulationsraum kann realistischer gestaltet sein, die Simulation von Gelände mit Höhenunterschieden und unterschiedlichen Bodenbeschaffenheiten wäre nur der erste Schritt.

2.8.4 Die Wahrnehmung von Hindernissen

Das Erkennen von Hindernissen erfolgt in der vorgstellten Lösung mit Hilfe eines Sensorstabs. In vielen Fällen ist dies ausreichend, aber wenn sich der Agent direkt neben einem Hindernis befindet, so kann es passieren, daß er in einem Simulationsschritt durch ein Steuerverhalten auf die andere Seite eines Hindernisses gesteuert wird. Um dies zu verhindern, könnte man mehrere Sensorstäbe am Charakter befestigen, besonders an den Seiten - diese müssen nicht besonders lang sein, da sie nur dafür sorgen müssen, daß der Agent nicht zufällig „durch das Hindernis gedrückt“ wird.

Ein weiteres Manko des bisherigen Systems zeigt sich in der Tatsache, daß dem Agenten beim Steuerverhalten *Hindernissen ausweichen* die Orientierung der Orthogonalen im Schnittpunkt des Sensorstabs mit dem Hindernis übergeben wird. Dies führt in einigen Fällen dazu, daß der Agent seinen Weg unnötig stark von der eigentlichen Route abändert.

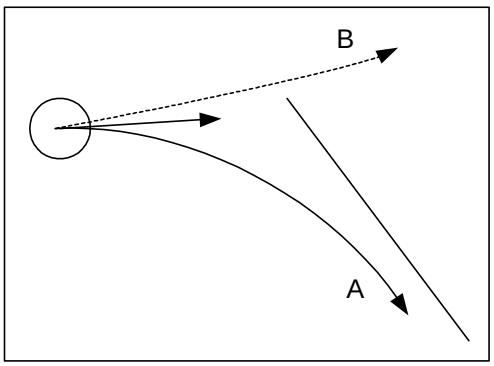


Bild 2-19: Alternativpfad beim Ausweichen von Hindernissen

Wie im Schaubild zu erkennen, würde ein autonomer Agent den Pfad A nehmen, um dem Hindernis auszuweichen. Der Pfad B wäre aber besser geeignet um dem Hindernis auszuweichen, denn er benötigt weniger Kraftaufwand und er verändert die Richtung des Agenten nicht sonderlich.

Ein weiteres Problem dieser Art des Ausweichens ist, daß zu jedem Simulationszeitpunkt der autonome Agent nur auf ein Hindernis reagieren kann. Es ist aber ohne Probleme möglich, durch eine konkave Anordung von Hindernissen dafür zu sorgen, daß ein Agent ein Hindernis nicht mehr frühzeitig genug erkennt und dann nicht mehr rechtzeitig ausweichen kann.

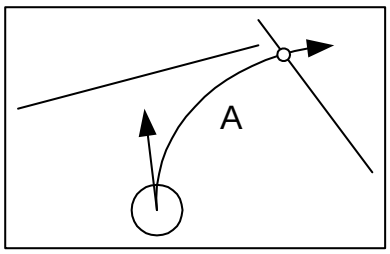


Bild 2-20: Kollisionsmöglichkeit bei konkaver Anordung von Hindernissen

Diesen Problemen kann mit eine größeren Anzahl von Sensorstäben begegnet werden (wie oben schon angesprochen), deren Wahrnehmung entsprechend intelligent verarbeitet wird - oder einer andere Methode, die Umwelt zu analysieren. Als eine solche Methode würde sich ein auf Bildverarbeitung basierendes Verfahren eigenen, z.B. das der Analyse eine *Z-Puffer* Bildes [16].

Ein solches Z-Puffer Bild kann durch die Methode der Strahlenverfolgung erstellt werden, bekannt aus dem Bereich des Raytracing. Typischerweise liefert ein solcher Algorithmus ein Bild mit Graustufeninformationen zurück, die die verschiedenen Tiefen des Raumes in Bezug auf die Messstelle wiedergeben. Für diesen Anwendungsfall reicht aber auch ein aus wenigen Testuntersuchungen gewonennes binäres Bild, das anzeigt, in welchem Teil des Bildes sich die Stellen befinden, deren mittlerer Grauwert am geringsten ist. Durch diese wird der längste hindernisfreie Pfad gekennzeichnet. In mehreren Iterationen kann das jeweilige Teilbild soweit auf diese Weise verfeinert werden, bis man zu einem zufriedenstellenden Ergebnis kommt.

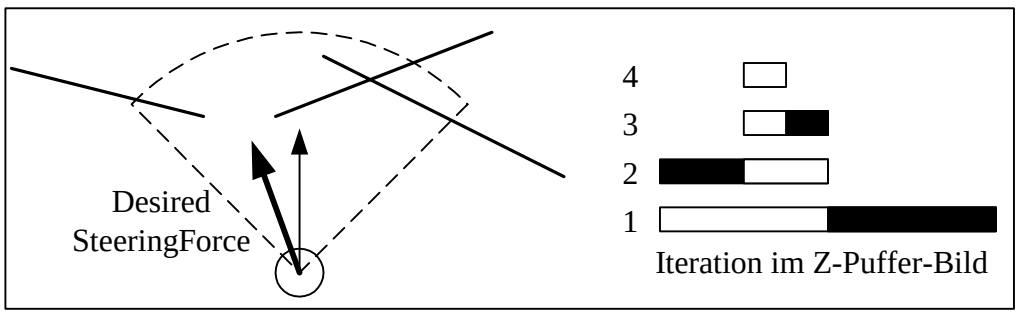


Bild 2-21: Hindernissen ausweichen durch Z-Puffer-Analyse

Dieses Verfahren verspricht intelligenteres Ausweichen von Hindernissen, hat aber den Nachteil, daß es relativ berechnungsintensiv ist.

3 Implementation

„bugz“ (amerikanischer Slang: Käfer/Insekten) ist der Name der Software, in der, aufbauend auf den in Kapitel 2 vorgestellten Ergebnissen, Gruppenbewegung simuliert werden können. Der Nutzer hat die Möglichkeit, Gruppen zu definieren und diesen Bugs zuzuordnen, die er auf einer Fläche positionieren kann. Den physikalischen Eigenschaften eines jeden Bugs können Werte zugewiesen werden, genauso wie es möglich ist, den Bugs Steuerverhalten zuzuordnen. Aufbauend auf diesen Daten, simuliert die Software dann die Bewegung der einzelnen Akteure.

Die nun folgenden Abschnitte befassen sich nicht mit der Bedienung des Programms (für diesen Zweck wurde ein eigenes Dokument geschrieben, das sich im Anhang befindet), sondern vielmehr mit der softwaretechnischen Realisierung. Eine detaillierte Beschreibung der Implementierungsdetails incl. sämtlicher Klassen, deren Beziehungen untereinander und Interaktionsschemata sowie des GUI kann an dieser Stelle aufgrund des großen Umfangs der Software leider nicht gegeben werden, es würde den Rahmen dieser Ausarbeitung sprengen. Wohl aber wird der Grundaufbau der Applikation erklärt und gezeigt, wie sie funktioniert. Die Programmstruktur wir erläutert und wichtige Konzepte, die bei der Implementierung in die Software eingeflossen sind, vorgegstellt.

3.1 Objektorientierte Programmierung

Die Forderung nach autonomen Agenten benötigt auch bei der Implementierung ein geeignetes Konzept. Viele Gemeinsamkeiten finden sich im Klassenkonzept der *Objektorientierten Programmierung*: Instanzen von Klassen kapseln ihre gesamte Funktionalität, alle Änderungen, die Eigenschaften dieser Objekte betreffen, nimmt das Objekt selber vor. Objekte handeln sozusagen in Bezug auf die sie betreffenden Änderungen autonom. Objekte kommunizieren über Methoden, so wie die realen Gegenstücke der autonomen Agenten miteinander und mit ihrer Umwelt über Sinneseindrücke die produziert und aufgenommen werden kommunizieren.

3.2 Die Wahl der Entwicklungsumgebung: Delphi 4.0

Für die Implementierung von dezentralen Problemen, wie es die verhaltensgesteuerte Animation darstellt, bietet sich, wie oben festgestellt, die Verwendung einer objektorientierten Sprache geradezu an. Die Entwicklungsumgebung Delphi[1] bietet eine solche in Form von Objekt-Pascal

an. Object-Pascal berücksichtigt fast alle wichtigen Konzepte der Objektorientierung, die auch in C++ vorhanden sind. Object-Pascal orientiert sich vom Syntax aber stark an dem prozeduralen Pascal.

Da es sich bei Delphi um eine visuelle Entwicklungsplattform handelt, lassen sich in kurzer Zeit einfache, und in überschaubarem Zeitrahmen auch aufwendige Benutzeroberflächen erzeugen, wie sie in der Anwendung benötigt werden. Die Entwicklungsumgebung erlaubt einen nahtlosen Übergang zwischen Quelltext-Generierung und Erzeugung der Nutzeroberfläche, komplizierte Makroaufrufe wie bei einigen C++ Systemen sind nicht notwenig.

Schon im sehr frühen Entwicklungsstadium zeichnete sich ab, daß aufgrund der rechenintensiven Simulation eine schnelle, aber dennoch komfortable Entwicklungsumgebung gewählt werden mußte. So eine stellt Delphi dar, die Geschwindigkeit der generierten Programme liegt nur knapp hinter der von mit C++ Compilern wie Visual C++ erstellten Programmen. Der Geschwindigkeitsaspekt war auch der einzige Grund, warum nicht ein JAVA-System zur Entwicklung gewählt wurde. Ein solches hätte sich in Anbetracht von Struktur und Sprachumfang auch hervorragend geeignet.

Die Komponentenbasierung erlaubt es, nicht in der Basis-Entwicklungsumgebung vorhandene, aber für die Realisierung benötigte Funktionalitäten durch externe Komponenten bereitzustellen, bzw. selbst eigene Komponenten zu entwickeln. Diese werden nahtlos in die Klassenhirarchie der VCL (Visual Component Library) integriert, die die Basis von Delphi darstellt und fast alle der durch die MFC (Microsoft Foundation Classes) bereitgestellten Funktionen kapselt, um sie wesentlich anwenderfreundlicher wieder zur Verfügung zu stellen.

3.3 OOA - Objektorientierte Analyse

Ziel der objektorientierten Analyse ist es, die Anforderungen an ein neues Software-Produkt mit Hilfe von Konzepten aus der Objektorientierung zu modellieren. Das Problem wird in einzelne, für sich abgeschlossene, sinnvolle Bereiche aufgeteilt, so daß aus diesen später Klassen modeliert werden können. Beziehungen und Abhängigkeiten von Objekten in Form von Aggregationen, Assoziationen und Kardinalitäten werden erkannt und in das theoretische Softwaremodell übertragen.
Als Beschreibungssprache wird UML (Unified Modeling Language) genutzt - die im Moment übliche standardisierte Methode, um Klassen und Objektstrukturen zu modellieren.

1. Delphi wurde von der Firma Inprise (ehemals Borland) entwickelt und liegt mittlerweile in der Version 5.0 vor.

Mit diesem Ansatz wurde das vorliegende Problem der Erstellung eines Multiagentensystems zur Generierung von Massenszenen untersucht und ein Klassendiagramm entwickelt, das eine mögliche Sichtweise auf das Problem darstellt.
Aufgrund seiner Größe ist es im Anhang zu finden ist. Es wird angeraten, bei der Vorstellung der einzelnen Klassen dieses parrallel zu verfolgen.

Im folgenden werden die wichtigsten Komponenten dieses Systems beschrieben.

3.3.1 TSimulation

Die Klasse `TSimulation` ist die Basisklasse einer jeden Simulation. Die Instanz dieser Klasse (zur Laufzeit gibt es immer nur eine) enthält Listen mit allen Objekten, die in einer Simulation relevant sind. Dieses wären z.B. eine Instanz der Klasse `TSpace`, im folgenden *Arena* genannt, die Eigenschaften des Raumes kapseln, auf dem sich die Bugs bewegen. Desweiteren gibt es Listen der an der Simulation beteiligten Gruppen (`TBugGroup`) von Bugs (`TBug`), sowie benötigte Listen von Hindernissen (`TObstacle`) und Trigger (`TTrigger`) für die Kontrolle des Bewegungsmusters der Bugs. Instanzen der so assoziierten Klassen werden nur durch die Klasse `TSimulation`[1] erzeugt, verändert und gelöscht. Desweiteren ist `TSimulation` zuständig für den Ablauf der Simulation: Durch diese Klasse wird sie gestartet und gestoppt, sie regt die Instanzen von `TBug` dazu an, sich entsprechend ihrer Eigenschaften auf der Arena zu bewegen. Somit ist `TSimulation` die Verwaltungsinstanz, der *Controller* des gesamten Simulationsablaufs.

3.3.2 TBugGroup

Bugs sind in Gruppen (`TBugGroup`) organisiert. Da auch in der Natur sich meist Gruppen von Individuen bilden, die zu einem gewissen Grad die gleichen Eigenschaften haben, ist es sinnvoll, einer Instanz von `TBugGroup` zu erlauben, die Eigenschaften aller Gruppenmitglieder gleichzeitig zu bearbeiten. Hierzu besitzt `TBugGroup` ein Objekt der Klasse `TDefaultBug`, das als Vorlage für die Bugs dient. Wird diese Vorlage geändert, so können auf einfachem Wege auch die assozierten Bugs geändert werden. Bugs sind aber nicht abhängig von den Vorgaben durch das jeweilige `TDefaultBug` - Objekt der übergeordneten Gruppe, vielmehr können alle Eigenschaften individuell angepaßt werden. In der Implementierung ist die vorherige Erschaffung einer Gruppe zwingend notwendig, damit man einen Bug erzeugen kann. Ein Bug muß also immer einer Gruppe zugeordnet sein. In der Visualisation der Geschehnisse im virtuellen

1. Natürlich verwaltet nicht die Klasse `TSimulation` die assoziierten Objekte, sondern eine Instanz der Klasse `TSimulation`. Aus Gründen der Lesbarkeit wird aber auf die eigentlich richtige Schreibweise verzichtet.

Raum werden Mitglieder einer Gruppe zur besseren Zuordnung durch eine einheitliche Farbe kenntlich gemacht.

3.3.3 TBasisBug

`TBasisBug` dient als abstrakte Oberklasse für `TBug` und `TDefaultBug`. Attribute und Methoden, die diesen beiden Klassen gemein sind, werden hier definiert. Zum einen sind dies ein Großteil der physikalischen Attribute der Bugs, die Liste der Steuerverhalten (`TSteeringBehaviour`) und die Liste der Trigger (`TTrigger`).

3.3.4 TBug

Die Klasse `TBug` kapselt zusätzlich zu den Attributen aus `TBasisBug` die Eigenschaften des autonomen Agenten, die er für die Simulation benötigt. Auch die in der Simulation nötigen Methoden für die Bestimmung der Handlung eines Bugs haben hier ihren Platz. Einem Bug ist zusätzlich noch ein Objekt der Klasse `TShape` zugeordnet. Dieses bestimmt, wie er aussieht, bzw. was für eine Form er hat. Ein Bug ändert einige seiner Attribute (zB. die Geschwindigkeit) häufig im Laufe der Simulation - damit die Startwerte bei einem Neustart der Simulation wieder vorliegen, werden diese zusätzlich gespeichert.
Da ein Objekt der Klasse `TBug` auch grafisch repräsentiert werden soll, ist ihm auch ein Objekt von `TGraphicElement` zugeordnet, der Klasse, die für die visuelle Ausgabe des Bugs auf der Arena verantwortlich ist.

3.3.5 TDefaultBug

`TDefaultBug` dient dazu, Vorgabewerte für Eigenschaften und Steuerverhalten zum Zwecke der späteren Übertragung auf Bugs zu speichern. Hierbei werden die einzelnen physikalischen Attribute um Varianzen ergänzt, so daß man statt mit diskreten Werten auch mit Wertbereichen arbeiten kann. Hiermit wird es ermöglicht, die in der Natur absolut üblichen Schwankungen bei physikalischen Attributen innerhalb einer Gruppe nachzubilden.

3.3.6 TSteeringBehaviour

Steuerverhalten sind in der Klasse `TSteeringBehaviour` modelliert. Zwar gibt es viele verschiedene Steuerverhalten und man hätte für jedes eine eigene Klasse modellieren können, aber aus Gründen der Erleichterung bei der Programmierung (hier besonders dem Laden und Speichern) werden alle Steuerverhalten in einer Klasse zusammengefasst und nur durch einen Typ unterschieden. Zwar kann es dann vorkommen, das Daten aus einer Instanz von `TSteeringBehaviour` nicht genutzt werden, aber die Übersichtlichkeit des Programmcodes und die Vereinfachung bei der Implementierung machen den Speichernachteil vernachlässigbar.

3.3.7 TObstacle

Die Klasse TObstacle kapselt die Attribute und Methoden eines Hindernisses. Wie ein Bug, so hat auch das Hindernis eine grafische Repräsentation und von daher auch eine Assoziation zu einer Instanz von `TGraphicElement`.

3.3.8 TGraphicDoc, TGraphicElement

Bei der Bildschirmausgabe wird das *Model-View Konzept* angewendet. Die interne Verarbeitung ist streng von der Bildschirmausgabe getrennt, alle sichtbaren Objekte (*Model*) sind mit weiteren Objekten verknüpft, die sie auf dem Bildschirm visualisieren (*View*). `TGraphicElement` stellt hier die Klasse dar, deren Instanzen mit Bugs oder Hindernissen verknüpft sind und durch die diesse visualisiert werden. `TGraphicElement` enthält alle Informationen, die nötig sind für die Visualisierung - Attribute wie `Position` beziehen sich rein auf das optische Ausgabefenster, sie müssen nicht im Wert zwangsläufig den Positionen der zugeordneten Objekte im Simulationsraum (der durch die `Arena` als Instanz von `TSpace` definiert wird) entsprechen. `TGraphicDoc` übernimmt die Verwaltung der Instanzen von `TGraphicElement` und sorgt für die fehlerfreie Grafikausgabe.
Nachteilig wirkt sich diese Struktur im erhöhten Verwaltungsaufwand aus, doch diese Lösung hat den großen Vorteil, daß die Visualisierung vollkommen von der Verarbeitungslogik getrennt ist und nur eine (vereinfachte) Sicht darauf liefert. Das Austauschen der Visualisationsart ist ohne größere Probleme möglich, denkbar wäre z.B. eine Implementierung der Grafikausgabe unter Nutzung einer geeigneten Grafik-API wie etwa Open GL von Silicon Graphics.

3.3.9 TTrigger

Instanzen der Klasse `TTrigger` dienen als sogenannte *Trigger*. Ein Trigger ist ein Hilfsmittel, mit dem es möglich ist, Steuerverhalten von Bugs zu einem bestimmten Zeitpunkt zu ändern. Die Werte dafür werden durch eine Assoziation zu `TDefaultBug` vorgegeben.

3.4 Programmstrukturen

Nach dieser kurzen Darstellung der wichtigsten Klassen, wird anhand einiger ausgewählter Arbeitsschritte beschrieben, wie diese in Standardsituationen zusammenarbeiten.

3.4.1 Objektverwaltung

Das Anlegen, Ändern und Löschen von Objekten zur Laufzeit ist ein Bereich, der einen beträchtliche Teil der Implementierung ausmacht, denn zu jedem Zeitpunkt muß die Integrität der Daten gewährleistet werden.

Sämtliche Verwaltung läuft nach dem gleichen Schema ab - sobald der Nutzer eine der oben genannten Aktionen anfordert, wendet sich der zuständige Programmteil an die aktuelle Instanz der Simulationsklasse `TSimulation`. Diese führt die angeforderte Aktion aus und sorgt dafür, daß alle nötigen Objektverknüpfungen entsprechend den neuen Bedingungen angepaßt werden.

Beispielhaft werden die Verwaltungsabläufe für einen Bug gezeigt, die durch `TSimulation` erledigt werden:

Bug erstellen

- Bug als Instanz der Klasse `TBug` erzeugen
- Bug in die Mitgliederliste der zugehörigen Gruppe eintragen
- Attribute des Bugs aufgrund der Defaultwerte der zugehörigen Gruppe anpassen
- Repräsentation im GUI (`Object Browser`, Visualisierung in der Arena) erzeugen

Bug ändern

- Attribute entsprechend der durch den Nutzer gemachten Vorgaben ändern
- Repräsentation im GUI (`Object Browser`, Visualisierung in der Arena) anpassen

Bug löschen

- sämtliche Referenzen aus Steuerverhalten, egal ob es sich um Referenzen von anderen Bugs, Gruppen- oder Triggervoreinstellungen handelt, entfernen
- Bug aus der Liste der Mitglieder seiner ehemaligen Gruppe löschen
- sämtliche Repräsentationen im GUI (`Object Browser`, Visualisierung in der Arena) entfernen
- Speicherplatz freigegeben

Die Verwaltung der anderen Objekte (Gruppen, Hindernisse, Trigger, Simulation) geschieht nach dem gleichen Prinzip - natürlich unter Berücksichtigung der Eigenheiten eines jeden Objekts.

3.4.2 Interaktion mit der GUI, 2-Schichtenmodell

Wie jedes Windows-Programm ist auch *bugz* ereignisorientiert, d.h. Methoden werden aufgrund von externen (durch den Nutzer verursachten), oder internen Ereignissen ausgeführt. Der Großteil der dazu nötigen Funktionalität wird durch Delphi bereits zur Verfügung gestellt, der Bereich der hier betrachtet wird, bezieht sich auf die Arbeit mit Attributen von Objekten der Simulation, mit denen der Nutzer direkt interagieren kann.
Für eine solche Situation, steht für jeden direkt editierbaren Objekttyp ein sogenanntes `Property Window` zur Verfügung. Jedes dieser Fenster besitzt ein Interface-Objekt, eine Instanz der Klasse, die dem Fenster zugehörig ist. Bei jeglicher Nutzung des Fensters wird zuerst das zu editierende Objekt auf dieses Interface-Objekt kopiert. Die Attribute des Interface-Objekts werden dann

in den Eingabefeldern des GUI-Fensters angezeigt. Änderungen an diesen Eingaben betreffen vorerst nur das Interface-Objekt, erst wenn der Nutzer das `Property Window` mit dem Wunsch schließt, die gemachten Eingaben zu übernehmen (er drückt den `OK-Button`), wird das Interface-Objekt auf das eigentlich zu editierende Objekt kopiert. Auf diese Weise ist es möglich, Graphical User Interface und Anwendungskern zu trennen, und somit diese beiden Schichten voneinander unabhängiger zu machen.

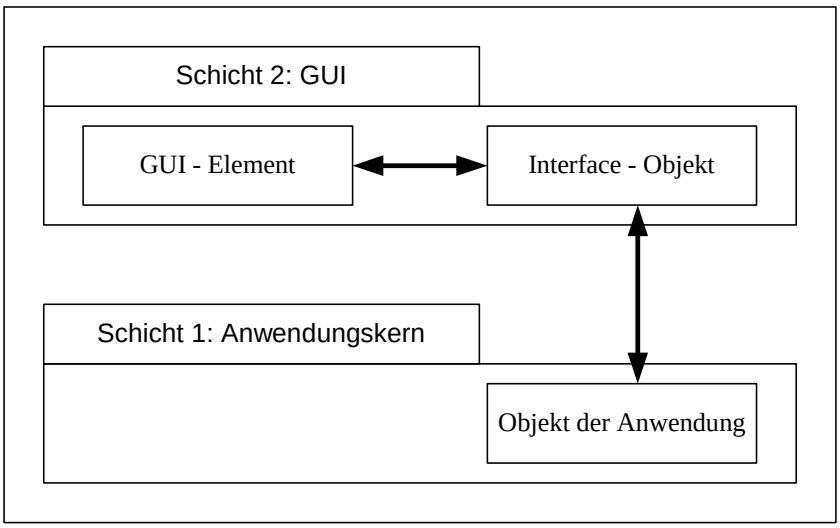


Bild 3-1: 2-Schichtenmodell

Eine solche Schichtenarchitektur (wie sie übrigens auch durch das weiter oben vorgestellte Model-View Konzept erreicht wird) hat softwaretechnische Vorteile: Sie erlaubt eine übersichtliche Strukturierung in Abstraktionsebenen und fördert so die Wiederverwendbarkeit, Änderbarkeit, Wartbarkeit, die Portabilität und nicht zuletzt auch die Testbarkeit [1].

3.4.3 Simulationsablauf

Der eigentliche Simulationsablauf wird durch die aktuelle Instanz der Klasse `TSimulation` gesteuert. Sie veranlaßt alle Bugs dazu, die notwendigen Funktionen zu starten, um deren Bewegung zu ermitteln. Jeder Bug für sich ermittelt dann aufgrund seiner Steuerverhalten einen Steuervektor mit dem dann, unter Berücksichtigung der eigenen physikalischen Eigenschaften, die Veränderung der Position im Simulationsraum bestimmt wird. Im Anschluß an den gesamten Vorgang stößt `TSimulation` auch die Aktualisierung der Grafikausgabe an. Die Simulationsklasse bringt den Controller der Grafikausgabe (`TGraphicDoc`) dazu, seinerseits die Grafikausgabe der ihm untergeordneten Objekte der Klasse `TGraphicElement` zu starten.

Schematisch läßt sich dieser Vorgang folgendermassen darstellen:

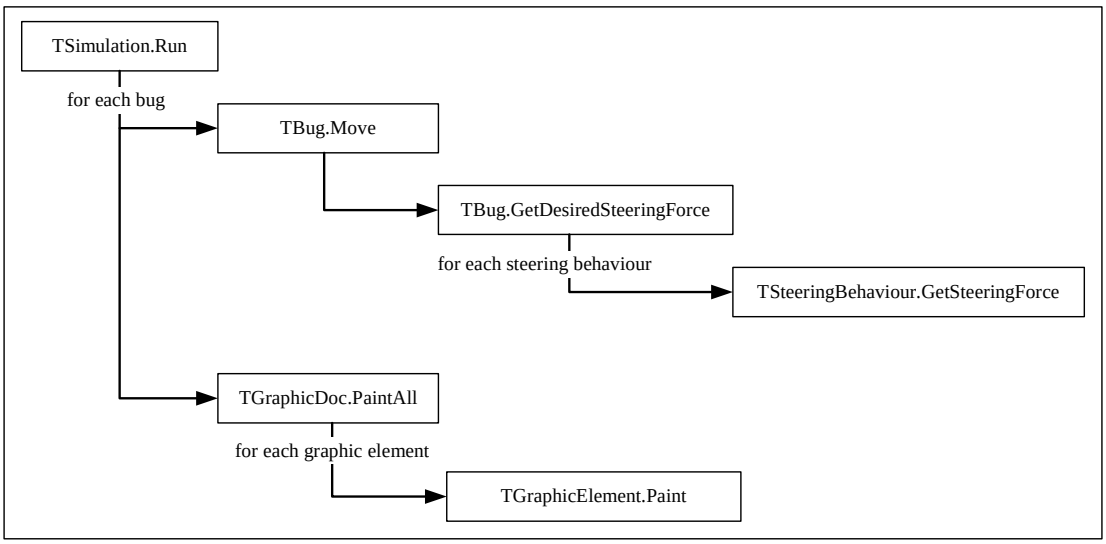


Bild 3-2: Hierachie der Methodenaufrufe im Simulationsablauf

3.5 Schnittstelle zum Animationssystem

3.5.1 Export von Bewegungsdaten

Während des Simulationsprozesses fallen eine große Menge Daten an, die sich für die Visualisierung von Gruppenbewegungen in einer 3D-Applikation nutzen lassen. Vor allem ist dies die Positionen eines jeden Bugs zu den verschiedenen Zeitpunkten der Simulation. In *bugz* lassen sich diese Daten während der Simulationsprozesse aufzeichnen und in einer Textdatei abspeichern. Diese Textdatei, die die kompletten Bewegungsdaten der Mitglieder einer Gruppe enthält, stellt die Schnittstelle zur Animationssoftware dar.

Für die bei ID-TV hauptsächlich benützten Softwarepakete *Lightwave* und *Maya* ist der Import der Daten über ein Plugin für die jeweilige Animationssoftware realisiert. Die durch die Dateischnittstelle bereitgestellten Daten können hiermit in der Arbeitsumgebung zur Verfügung gestellt werden. Dort können sie dann weiter bearbeitet werden. Dies geschieht durch die Nutzung der für die Animationswerkzeuge spezifischen und durch die zugehörigen Skriptsprachen zur Verfügung gestellten Befehle.

Die Orientierung der Bugs wird aus Gründen der Datenreduktion nicht gespeichert; fast jedes 3D-Animationspaket (so auch Lightwave und Maya) erlaubt es, einen Körper an seinem Bewegungspfad auszurichten. Da das bisherige Simulationsmodell es auch nicht erlaubt, daß sich die Orientierung eines Bugs von dessen Blickrichtung unterscheidet, ist es deshalb auch nicht nötig, diese Daten zu exportieren.

3.5.2 Lightwave 6.0

Lightwave aus dem Hause NewTek war eines der ersten 3D-Animationsprogramme, zuerst war es nur für den Amiga, später auch für PC, SGI und Alpha erhältlich. Aufgrund der intuitiven Benutzerführung, der qualitativ hochwertigen Renderergebnisse, der Vielseitigkeit und nicht zuletzt auch dem guten Preis/Leistungsverhältnis ist es eins der am weitesten verbreiteten professionellen 3D-Programme und wird hauptsächlich in der Film- und Videoproduktion eingesetzt.

Lightwave bietet mit *LScript* eine objektbasierte Skriptsprache an, mit der sich Arbeitsprozesse automatisieren lassen, mit der Objekte prozedural animiert oder auch Oberflächenstrukturen definiert werden können (Shader). Auf Basis von LScript wurde auch ein Plugin realisiert, mit dem sich die Bewegungsdaten in Lightwave importieren lassen, um sie dort weiter zu bearbeiten.

Die Positionsdaten zusammen mit dem zugehörigen Zeitpunkt werden eingelesen und mit Hilfe der Keyframe-Technik aus diesen Daten eine Animation für die jeweiligen Objekte generiert. Die einzelnen Keyframes werden durch Splines interpoliert, so daß, auch bei im Verhältnis zum importierten Zeitraum geringer Anzahl von übertragenen Positionen, der Eindruck eines flüssigen Bewegungsablaufes entsteht. Dieses Verfahren entspricht dem Importieren von Motion-Capturing-Daten, wie es häufig in der Charakteranimation verwendet wird.

3.5.3 Maya 2.5

Bei Maya aus dem Hause Alias|Wavefront handelt es sich um eine High-End-Animationssoftware, die hauptsächlich im Bereich der Charakteranimation eingesetzt wird. Es finden sich jedoch auch äußerst leistungsfähige Funktionen, die, z.B. mittels aufwendiger Simulation, die Visualisierung komplexer physikalischer Vorgänge (wie z.B. Deformation von Kleidern am Körper, berstendes Glas, das Verhalten von Nebel, wenn sich Objekte durch diesen bewegen, sich im Wind wiegendes Gras etc.) ermöglichen. Als Referenz für die Leistungsfähigkeit dieser Software stehen Computeranimationen, die in großen Kinoproduktionen wie z.B. Star Wars: Episode 1, Matrix oder Toy Story 2 zu sehen sind.

Die streng objektorientierte Systemarchitektur von Maya ist mit C++ realisiert worden und bietet neben einem API auch die Skriptsprache *MEL* (Maya Embeded Language) zur Erweiterung des bereits vorhandenen und extrem umfangreichen Funktionssatzes an.

Das Plugin zum Import von Bewegungsdaten ist unter Nutzung von MEL realisiert. Im Gegensatz zu Lightwave werden die Bewegungen nicht durch die Animationstechnik der Schlüsselbilder realisiert sondern durch Pfadanimation. Aus der Menge der einzelnen Positionen wird eine Spline-Kurve erstellt, an die im nachhinein ein Objekt gebunden wird. Der Grund dafür ist, daß man in Maya ein Objekt nicht an einem durch Keyframes vorgegebene Bewegungspfad ausrichten kann (wie dies in Lightwave möglich ist), sondern nur an einer geometrischen Kurve im Raum.

3.6 Performance

bugz wurde nicht geschwindigkeitsoptimiert. Vielmehr geht es in der Applikation darum, die theoretisch gewonnenen Erkenntnisse nachvollziehbar zu implementieren. Dennoch wäre diese Arbeit nicht vollständig, wenn sie nicht einen groben Überblick über die Leistung des Systems liefern würde.

Interessant sind hier vor allen Dingen die Steuerverhalten, die sich auf die anderen Bugs in der Umgebung beziehen. Bei der hier verwendeten naiven Implementation der Algorithmen steigt die Komplexität quadratisch mit der Anzahl der verwendeten Bugs ($O(N^2)$). Dies rührt daher, weil für die entsprechenden Steuerverhalten (*Ausrichten, Zusammenhalten, Abstand halten*) zu jedem Zeitpunkt jeder Bug alle anderen Bugs in seiner Gruppe daraufhin überprüfen muß, ob sie sich in seinem Sichtfeld befinden. Wird die Anzahl der Bugs verdoppelt, so vervierfacht sich die zur Berechnung benötigte Zeit.

Das Problem rührt zu einem Teil daher, daß zur Berechnung nur ein einzelner Prozessor genutzt wird - ganz im Widerspruch zur Natur, wo jeder beteiligte Akteur mit einem eigenen „Prozessor“ ausgestattet ist. Eine mögliche Weise, dem Problem zu begegnen, ist es also, für jeden Bug einen eigenen Prozessor zu nutzen. Auf diese Weise wären, selbst bei naiver Implementation, die Algorithmen mit einem Aufwand von $O(n)$ zu bewerkstelligen.

Ein weiterer Weg den Aufwand zu reduzieren besteht darin, aufgrund der Annahme, daß sich die Umwelt eines Bugs nicht sonderlich von der im vorherigen Simulationsschritt unterscheidet, für eine gewisse Anzahl von Simulationsschritten die einmal gefundenen benachbarten Bugs wiederzuverwenden. Gerade bei relativ gleichförmigen Gruppenbewegungen verspricht dieses Verfahren gute Erfolge.

Auch an anderen Stellen des Programms zeigt sich noch Raum für Optimierung der Geschwindigkeit durch Approximation. Eine dieser Stellen wäre die sehr häufig benötigte Bestimmung der Länge eines Vektors oder einer Strecke. Statt der hier verwendeten *Euklidischen Distanz* kann auch ein anderes, weniger re-

chenintensives Distanzmaß genutzt werden, wie z.B. die *City-Block-Distanz*. Dieses muß nicht unbedingt ein schlechteres Ergebnis zur Folge haben, da der visuelle Gesamteindruck einer Gruppenbewegung dadurch nicht zunichte gemacht werden muß. - Problematisch wird eine solche Art der Approximation nur beim Ausweichen von Hindernissen, in den meisten Fällen werden die Bugs die räumlichen Begrenzungen nicht mehr einhalten.

3.7 Probleme bei der Implementierung

Die Implementierung des eigentlichen Anwendungskerns stellte, sicher auch aufgrund der vorherigen intensiven Planungsphase, kein großes Problem dar. Schwierigkeiten ergaben sich vor allem im Bereich der Erstellung der grafischen Benutzerschnittstelle. Zwar bietet Delphi die Möglichkeit, in kurzer Zeit Dialoge und Anwendungsfenster mit Komponenten zu füllen und diese mit der Anwendung zu verknüpfen, doch bei aufwendigeren Benutzeroberflächen, wie z.B. einem in der Anwendung realisierten dynamischen GUI (speziell ist das frei erweiterbare Tab-Sheet für die Eingabe der Steuerverhalten gemeint), zeigen sich schnell Beschränkungen von Delphi. Es ist nämlich nicht möglich, von einer Klasse die alle Informationen für einen Dialog enthält (wie z.B. im Dialog enthaltene Komponenten und deren Interaktion mit dem Rest des Softwaresystems), ohne Probleme mehr als eine Instanz zur Laufzeit zu erzeugen. Stattdessen müssen jegliche dynamisch erzeugte visuelle Komponenten explizit in der Objektverwaltung für das GUI angemeldet werden, obwohl dies bei nur einer Instanz ohne Probleme automatisch erfolgt. Auch ergeben sich speziell bei der Entwicklung von neuen Komponenten in denen schon vorgegebene Komponenten genutzt werden häufig nicht nachvollziehbare Fehler und Abstürze der Entwicklungsumgebung, die darauf hindeuten, daß eins der wichtigsten Prinzipien der Objektorientierung, die Kapselung von Daten und Funktionen, nicht immer durchgängig in das Design der VCL eingeflossen ist.

Aufgrund solcher Probleme ist ein beträchtlicher Teil der Entwicklungszeit (mehr als 50%) für die Generierung der grafischen Benutzerschnittstelle entfallen, mehr als zuerst dafür geplant war. Deshalb mußten auch Abstriche in der Funktionalität und vor allem in der Optimierung der Performance gemacht werden.

3.8 Anforderung an Hardware und Software

Hardwareplattform

Für den Betrieb der Software wird ein Rechner der Pentiumklasse mit mindestens 16, besser 32 MB RAM benötigt. Grundsätzlich läßt sich aber sagen, daß das System, auf dem mit der Software gearbeitet wird, so schnell sein sollte, wie möglich, abhängig von der Komplexität der erstellten Simulationen.

Für den Betrieb werden ca. 700 KB freier Festplattenspeicher benötigt, zum Speichern von Simulationen und Bewegungsdaten ist entsprechend mehr Platz zur Verfügung zu stellen.

Betriebssystem

Die Applikation wurde unter Windows NT 4.0 SP 6 entwickelt und ist auf dieser Plattform lauffähig. Unter Windows 95/98/2000 sollte sie gleichfalls laufen, da keine NT-spezifischen Eigenheiten genutzt wurden. Kompatibilität mit den oben genannten Betriebssystem ist nicht getestet.

Entwicklungsumgebung

Wie zu Beginn dieses Kapitels bereits beschrieben, wurde die Implementierung unter Verwendung der Entwicklungsumgebung Delphi 4.0, Professional Version entwickelt. Diese ist notwenig, um den auf dem Datenträger beigefügten Quellcode zu kompilieren. Kompatibilität mit älteren oder mittlerweile erhältlichen neueren Versionen von Delphi wurde nicht getestet und kann von daher nicht garantiert werden.

Verwendete Komponenten

Zusätzlich zu den schon im Umfang von Delphi enthaltenen, wurden folgende als Freeware erhältliche Komponenten verwendet:

- TDFSColorButton v2.56

Eine Komponente zu Auswahl von Farben, erstellt von Brad Stowers. Erhältlich unter: http://www.pobox.com/~bstowers/delphi/ .

- TFloatEdit v1.2

Eine Komponente, die die einfache Eingabe von Fließkommazahlen erlaubt. Sie wurde erstellt von Markus Stephany und ist unter http://home.t-online.de/home/MirBir.St/ zu beziehen.

- TreeNT

Bei TreeNT von Dipl. Ing. Mike Lischke handelt es sich um eine Erweiterung der schon im Delphi Umfang enthaltenen Komponente TTreeView. Zu der Zusatzfunktionalität gehört vor allen Dingen das benötigte Referenzieren eines Baumeintrags über seine Verknüpfung mit einem Objekt. Zu finden ist diese Komponennte in bekannten Delphi-Komponenten Sammlungen im Internet, wie z.B. http://www.torry.ru .

Die Komponenten werden zur fehlerfreien Kompilierung des Quellcodes benötigt - sie sind neben den genannten Internetadressen auch auf dem beigefügten Datenträger zu finden.

4 Zusammenfassung und Ausblick

4.1 Zusammenfassung

In dieser Arbeit wurde der Begriff *Autonomer Charakter* definiert und mit dem *3-Schichten Verhaltensmodell* ein möglicher Ansatz präsentiert, der zeigt, wie das Bewegungsverhalten eines solchen Charakters in die Ebenen *Motivation*, *Steuerung* und *Fortbewegung* aufgeteilt werden kann. Desweiteren wurde mit dem Modell der *Massepunkt-Approximation* ein vereinfachtes physikalisches System für die Fortbewegungsschicht vorgestellt und 9 Steuerverhalten in ihrer Wirkung analysiert. Mit dem Multiagentensystem *bugz*, dessen softwaretechnische Aspekte beleuchtet wurden, wurden die theoretischen Ergebnisse realisiert und gezeigt, wie sich auf Basis der präsentierten Steuerverhalten und deren Kombination, komplexe Bewegungsmuster erzeugen lassen. Durch eine Dateischnittstelle und einen Importfilter wurden die Bewegungsmuster in das Animationswerkzeug Lightwave 6.0 sowie Maya 2.5 überführt und dort zur Weiterbearbeitung zur Verfügung gestellt.

4.2 Ausblick

Die hier vorgestellten Ergebnisse können nur als erste Annäherung an die Thematik der Generierung von Gruppenbewegungen zur Nutzung in der Computeranimation gelten. Möglichkeiten der Verbesserung zeigen sich an mehreren Stellen: Zwar erlaubt die Definition von zeitabhängigen Triggern (-> User Manual) einen gewissen Einfluß auf das Bewegungsmuster der Bugs, doch ist dieses Mittel der Steuerung leider noch nicht ausreichend um wirkliche Kontrolle über die Menge der autonomen Agenten zu haben. Trigger, die im Raum verteilt sind und dort einen einen gewissen Einflußbereich haben, wären ein Schritt die Bewegung stärker zu kontrollieren. Auf diese Weise könnte man Effekte simulieren, wie Tiere, die suchend durch die Savanne streifen, in einem gewissen Bereich das Wasserloch registrieren, sich in dessen Richtung bewegen um dann, dort angekommen, zur Tränke verweilen.

Das Vorgeben von gerichteten Pfaden oder Kraftfeldern würde den Prozess vereinfachen, die Gruppe einen bestimmten Weg entlangzuführen. Ebenso wäre dazu ein der Simulationsfläche unterlegtes Graustufenbild geeignet, bei dem Bereiche, die die Bugs meiden sollen, andersfarbig sind, als Bereiche, die Bugs suchen sollen. Ideal wäre die vollkommende Integration eines auf verhaltensgesteuerter Animation basierendem System mit einem System, das auf klassischen Animationstechniken basiert. Auf diese Weise könnte man einige der Bugs durch Pfad- oder Keyframe-Animation steueren, während der Rest

sich nach Verhaltensregeln bewegt. Der Animator hätte die volle Kontrolle dort, wo er sie verlangt - spezielle, z.B. durch das Storyboard vorgegebene Positionen von Bugs zu bestimmten Zeitpunkten könnten sehr einfach realisiert werden.

Ein großer Bereich, nämlich der der Eigenbewegung der autonomen Agenten, wurde noch überhaupt nicht beachtet. Wenn ein Mensch sich zu Fuß fortbewegt, so tut er dies auf unterschiedlichste Weise: Schleichen, Gehen, Laufen, Rennen - dies sind nur einige der verschiedenen Gangarten, die ein Mensch, abhängig von seiner Geschwindigkeit, nutzen kann. Soll ein realistischer Eindruck entstehen, so muß dies alles bei der Generierung einer Massenszene beachtet werden. Das System müßte dahingehend erweitert werden, als das es möglich sein sollte, bereits vorgefertigte Animation und Bewegungsabläufe auf die visuelle Repräsentation des autonomen Agenten zu legen. Damit der Eindruck einer fließenden Bewegung nicht gestört wird, sollten diese Bewegungsabläufe ineinander zu den passenden Zeitpunkten geblendet werden können.

Doch nicht nur die reine Fortbewegung des einzelnen autonomen Agenten ist interessant für die Animation, auch weitere Arbeitsaufgaben, wie eine tanzende, jubelnde oder auch zum Gebet niederknieende Menge, sollten idealerweise mit einem Gruppenanimationssystem gelöst werden. Hierzu muß der Animator die Möglichkeit haben, zu einem bestimmten Zeitpunkt einen Reiz an alle Mitglieder der Gruppe auszusenden, worauf diese hin mit einem entsprechenden vorgefertigtem Bewegungsablauf reagieren. Automatische, zufällige leichte Abwandlung der dem Reiz zugehörigen Bewegungsabläufe wäre ein weiteres Mittel, um die gesamte Bewegung der Gruppe organisch anmuten zu lassen.

Bei der Realsierung von Massenszenen zeigen sich aber auch noch weitere Probleme, die nichts mit der Kontrolle der Menge zu tun haben. Gerade bei sehr großen Massen ist es nicht möglich, für jeden Akteur eine eigene Objektgeometrie zu definieren. Der Speicher- und Arbeitsaufwand der zur Verwaltung der Menge der Einzelgeometrien nötig wäre, ist mit heutigen Rechnersystem nicht wirtschaftlich zu bewältigen, auch die Berechnung der Bilder wird sehr viel Zeit verbrauchen. Es gibt aber einige mögliche Ansätze, dem Problem zu begegnen: Mehrfache Verwendung der gleichen Geometrie bei verschiedenen Akteuren (*Instanz-Objekte*), Arbeiten mit mehreren separaten *Tiefeneben*, *Level-of-Detail* Verfahren, oder auch *Sprite-Rendering*, bei dem das endgültige Bild nicht mit der Objektgeomtrie selbst gerechnet wird, sondern mit einem schon vorher produziertem Bild der Objektgeometrie aus dem spezifischen Kamerawinkel.

Unter dem Gesichtspunkt, daß es zum Zeitpunkt der Erstellung der Arbeit noch kein kommerziell erhältliches System gibt, daß die oben genannten Eigenschaften aufweißt und Möglichkeiten bietet, erscheint weitere Forschung und Entwicklung in diesem Gebiet attraktiv.

4.3 Alternative Anwendungsmöglichkeiten

Der Fokus dieser Arbeit liegt vor allem auf der Simulation von Gruppenbewegungen zur Nutzung in der Computeranimation, zur Generierung von Massenbewegungen für den Einsatz in Spielfilmen und Visualisierungen. Weitere Möglichkeiten des Einsatzes für ein solches Multiagentensystems liegen in der Planung von Architekturen. Ein auf dem gleichen Ansatz basierendes System könnte (wenn ihm ein realistischeres physikalisches und verhaltenstechnisches Modell unterliegt) dazu genutzt werden, um bauliche Schwachpunkte aufzudecken. Zum Beispiel könnte man auf diese Weise, bevor überhaupt ein Spatenstich getan wurde, schon im Rechner überprüfen, wie sich eine Massenpanik in einem Fussballstadion auswirken würde, und ob die Fluchtwege den Ansturm der Massen aushalten würden. Sicherheitstechnische Mängel können noch in der Planungsphase mit Hilfe einer solchen Umgebung aufgedeckt werden. Aber auch die Gestaltung von Gehwegen, Galerien und Museen, Einkaufspassagen, Gänge in einem Geschäft - all dies könnte daraufhin untersucht werden, wie Menschenmassen sich in ihnen verhalten und ob sich Engpässe ergeben, ob bauliche Massnahmen es ermöglichen, den Menschenfluß zu steuern.

Ein weiteres Einsatzgebiet ergibt sich für ein solches Modell in der Simulation einer virtuellen Realität, sei es im VR oder im Entertainmentbereich.

Auch in der Verhaltensforschung (in der eng mit dem Forschungsfeld der Simulation von Leben in all seinen Erscheinungsformen, *Artificial Life*, zusammengearbeitet wird), kann ein solches Multiagentensystem gute Dienste leisten. Oft ist es nämlich sehr schwer, aufwendig und teuer, gerade bei Gruppenprozessen, Feldforschung zu betreiben. Wenn dann die Gruppe auch noch vorher genau definierten Ereignissen gegenüber gestellt werden soll, um ihr Verhalten zu beobachten, dann ist dies häufig unmöglich.

Anhang

A Klassendiagramm - *bugz*

Aufgrund seiner Größe ist es nicht möglich, das Klassendiagramm in dieses Dokument einzubinden. Es liegt daher als separates Faltblatt bei.

B User Manual - *bugz*

Das Handbuch zur Software *bugz*. Es ist in englischer Sprache gefaßt, da es bei ID-TV, der Firma bei der die Diplomarbeit erstellt wurde, üblich ist, die Dokumentation so zu schreiben, daß auch die nicht deutschsprachigen Mitarbeiter sie verstehen können.

C Inhalt der beigefügten CD

Die beigefügte CD wird gestartet, indem man die die Datei index.htm in einem Webbrowser aufruft. Sie enthält:

- die komplette Arbeit im PDF-Format
- Klassenschema im MDL-Format von Rational Rose 98
- Beispielanimation auf die sich im Text bezogen wird
- die implementierte Software *bugz* als direkt von der CD ausführbare Datei
- Beispielsimulation auf die sich im Text bezogen wird, sowie einige zusätzliche Simulationen
- das Handbuch zu *bugz* im PDF-Format
- die Skripte zum Import von mit *bugz* generierten Bewegungsdaten in Maya 2.5 und Lightwave 6.0
- den Quellcode zu *bugz* im für die Entwicklungsumgebung Delphi 4.0 lesbaren Format
- zusätzlich benötigte Komponenten

D Quellcode

Aufgrund des großen Umfangs kann an dieser Stelle nicht der gesamte Quellcode wiedergegeben werden. Deshalb beschränkt sich der vorgestellte Teil auf die wichtigsten Funktionen der Implementierung sowie den Quelltext der Plugins für Lightwave und Maya.

Vektoroperationen

```
TYPE      Vector = array of extended;
VAR       DimMatrix    : integer; // number of vektor-elements

// AddVector
// Add (U)*(Sign) to (V), store result in (W)
PROCEDURE AddVector (Sign:integer;U,V:Vector;var W:Vector);
var i        : integer;
begin
  for i:=0 to pred(DimMatrix) do
    W[i]:=U[i]+sign*V[i];
end;

/*-------------------------------------------------*/
// SummVector
// Addition of two vectors, (U) and (V), result in (W)
PROCEDURE SummVector(U,V:Vector;var W:Vector);
begin
  AddVector(1,U,V,W);
end;

/*-------------------------------------------------*/
// DifVector
// Returns (W) = (U)-(V) ;
PROCEDURE DifVector(U,V:Vector;var W:Vector);
begin
  AddVector(-1,U,V,W);
end;

/*-------------------------------------------------*/
// ScaleVector
// Multiply the vector (U) with skalar (x), store the result in vector (V)
PROCEDURE ScaleVector(x:extended;U:Vector;var V:Vector);
var i : integer;
begin
  for i:=0 to pred(DimMatrix) do
    V[i]:=x*U[i];
end;

/*-------------------------------------------------*/
// ScalarProdukt
// Returns the scalar product of the vector (U) and (V)
FUNCTION  ScalarProduct(U,V:Vector)  : extended;
var i : integer;
    s : extended;
begin
  s:=0;
  for i:=0 to pred(DimMatrix) do
    s:=s+U[i]*V[i];
  Result:=s;
end;
```

```
/*-------------------------------------------------*/
// NormVector
// Returns the euclidian norm of vector (U)
FUNCTION  NormVector(U:Vector):extended;
var i : integer;
    n : extended;
begin
  for i:=0 to pred(DimMatrix) do
    n:=n+sqr(U[i]);
  Result:=sqrt(n);
end;

/*-------------------------------------------------*/
// MinVector
// Determins the minimum of (U),(V) and stores it in (W)
Procedure MinVector(U,V:Vector; var W:Vector);
begin
  if NormVector(u) < NormVector(v)
    then CopyVector(u,w)
    else CopyVector(v,w);
end;

/*-------------------------------------------------*/
// Sets the length of (U) to (Skalar) and stores it in (V)
Procedure SetVectorLength(Skalar:Extended; U:Vector; var V:Vector);
var Norm:Extended;
BEGIN
  Norm := NormVector(U);
  if Norm <> 0
    then ScaleVector(Skalar / Norm,u,v)
    else CopyVector(u,v);
END;

/*-------------------------------------------------*/
// CopyVector
// Copys the elements of (U) to (V)
procedure CopyVector(U:Vector; const V:Vector);
var i: byte;
begin
  for i := 0 to pred(DimMatrix) do
    V[i] := U[i];
end;

/*-------------------------------------------------*/
// NullVector
// Returns TRUE, if vector (U) is a null-vector
FUNCTION NullVector(u:Vector):boolean;
var i:integer;
    help:boolean;
begin
  help := (u[0] = 0);
  for i:=1 to pred(DimMatrix) do
    begin
      result := help and (u[i]=0);
      help := result;
    end;
end;
```

```
/*-------------------------------------------------*/
// NormalizeVector
// Normalizes vector (U), stores result in (V)
PROCEDURE NormalizeVector(U:Vector;var V:Vector);
var magnitude:extended;
begin
  magnitude := NormVector(U);
  if magnitude <> 0
     then
       ScaleVector(1/magnitude, U,V);
end;

/*-------------------------------------------------*/
// TruncateVector
// Truncates vector-elements of (U) by (Skalar)
Procedure TruncateVector(Skalar:Extended;U:Vector;var V:Vector);
var Norm:Extended;

BEGIN
  Norm :=  NormVector(U);
  if Skalar < 0
    then begin
      if (Norm < -Skalar) and (Norm > 0)
        then ScaleVector(- Skalar / Norm,u,v)
    end
    else begin
      if Norm > Skalar
       then ScaleVector(Skalar / Norm,u,v)
       else v := u;
    end;
END;
```

Objekte im Sichtfeld

```
/*-------------------------------------------------*/
// InFieldofView
// Returns (true) and (TargetOffset) if single (Target) in field of view
function TSteeringBehaviour.InFieldOfView(Bug: TBug;
                                          var TargetOffset: Vector):boolean;

var DistLength, VeloLength, CosAngle, CosVecAngle:extended;
    SensorCheck:boolean;
begin
  Result := false;
  SensorCheck := false;

  If TargetBugActive
    then CopyVector(TargetBug.Position,Target);
  DifVector(Target, Bug.Position, TargetOffset);

// Target always in field of view, when field of view is unlimited
  if not SensorRangeActive and not AngleActive
    then result := true
    else
      begin
        with Bug do
          begin

// Target in view circle, if distance to target smaller than radius of view circle

            DistLength := NormVector(TargetOffset);
            if SensorRangeActive and (DistLength <= SensorRange)
              then
                begin
```

```
                result := true;
                SensorCheck := true;
              end;

// check if target is also in viewing angel

            if AngleActive
              then
                begin
                  CosAngle := cos(DegToRad(Angle/2));
                  VeloLength := NormVector(Velocity);
                  CosVecAngle := ScalarProduct(Velocity,TargetOffset) /
                                 (VeloLength * DistLength);
                  if (CosVecAngle > CosAngle) and
                     (not SensorRangeActive or SensorCheck)
                  then
                    Result := true
                  else
                   Result := false;
            end;

      end;
  end;

  if not Result
    then
      begin
        TargetOffset[0] := 0;
        TargetOffset[1] := 0;
      end;
end;

/*-------------------------------------------------*/
// GetNearbyBugs
// Returns list of bugs, found in the viewing angel of (Bug)
function TSteeringBehaviour.GetNearbyBugs(Bug: TBug): TList;
var BugList:TList;
    NearbyBugList:TList;
    BugCounter:Cardinal;
    DistanceVector:Vector;
    VectorAngle:extended;
    ViewAngleRad, DistanceLength, VelocityLength:extended;

begin
  NearbyBugList:= TList.Create;
  setlength(DistanceVector,DimMatrix);
  BugList := TBugGroup(Bug.ParentGroup).MemberList;

  // if no field of view defined, all bugs are nearby
  if (not SensorRangeActive) and (not AngleActive)
    then
       CopyList(BugList,NearbyBugList)
    else
      begin
        if SensorRangeActive
          then
            for BugCounter := 0 to BugList.Count - 1 do
              begin
                // don't take bug performing the search into account
                if TBug(BugList[BugCounter]) = Bug then continue;

               DifVector(Bug.Position,TBug(BugList[BugCounter]).Position,
                           DistanceVector);
                // Add bug to NearbyBugList, if it is in viewing radius
```

```
                if NormVector(DistanceVector) < SensorRange
                  then
                    NearbyBugList.Add(BugList[BugCounter]);
              end
            else
              CopyList(Buglist, NearbyBugList);

       if AngleActive and (NearbyBugList.Count >0)
          then
            begin
              ViewAngleRad:= cos(DegToRad(0.5 *Angle));
              for BugCounter := NearbyBugList.Count - 1 downto 0 do
                begin
                  if TBug(NearbyBugList[BugCounter]) = Bug then continue;

                 DifVector(Bug.Position,TBug(NearbyBugList[BugCounter]).Position,
                           DistanceVector);
                  VelocityLength :=  NormVector(Bug.Velocity);
                  DistanceLength :=  NormVector(DistanceVector);

                  if (VelocityLength > 0) and (DistanceLength > 0)
                    then
                     VectorAngle := ScalarProduct(Bug.Velocity, DistanceVector)/
                                    (VelocityLength * DistanceLength);

                  // delete bug from NearbyBugList, if it is not viewing angle
                  if VectorAngle >  ViewAngleRad
                     then NearbyBugList.Delete(BugCounter);

                end;
           end;
      end;
   Result := NearbyBugList;
end;
```

Zielbasierte Steuerverhalten

```
/*-------------------------------------------------*/
// SeekSteering
// Return steering vector for seek-steering
function TSteeringBehaviour.SeekSteering(Bug: TBug): Vector;
begin
  setlength(Result,DimMatrix);
  with Bug do
    begin
      if InFieldOfView(Bug,Result)
        then
          begin
            NormalizeVector(Result,Result);
            ScaleVector(maxspeed,Result,Result);
            DifVector(Result,Velocity,Result);
         end;
   end;
end;
```

```
/*-------------------------------------------------*/
// FleeSteering
// Return steering vector for flee-steering
function TSteeringBehaviour.FleeSteering(Bug: TBug): Vector;
begin
  setlength(Result,DimMatrix);
  with Bug do
    begin
      if InFieldOfView(Bug,Result)
        then
          begin
            ScaleVector(-1,Result,Result);
            NormalizeVector(Result,Result);
            ScaleVector(maxspeed,Result,Result);
            DifVector(Result,Velocity,Result);
          end;
   end;
end;

/*-------------------------------------------------*/
// MothSeekSteering
// Return steering vector for mothseek-steering
function TSteeringBehaviour.MothSeekSteering(Bug: TBug): Vector;
begin
  setlength(Result,DimMatrix);
  with Bug do
    begin
      if InFieldOfView(Bug,Result)
        then
          begin
            NormalizeVector(Result,Result) ;
            ScaleVector(Simulation.Fps,Result,Result);
          end;
    end;
end;

/*-------------------------------------------------*/
// MothFleeSteering
// Return steering vector for mothflee-steering
function TSteeringBehaviour.MothFleeSteering(Bug: TBug): Vector;
begin
  setlength(Result,DimMatrix);
  with Bug do
    begin
      if InFieldOfView(Bug,Result)
        then
          begin
            NormalizeVector(Result,Result);
            ScaleVector(-1 * Simulation.Fps,Result,Result);
          end;
    end;
end;

/*-------------------------------------------------*/
// ArrivalSteering
// Return steering vector for arrival-steering
function TSteeringBehaviour.ArrivalSteering(Bug: TBug): Vector;
var TargetOffset:Vector;
    Distance, RampedSpeed:extended;
begin
  setlength(Result,DimMatrix);
  with Bug do
   begin
     if InFieldOfView(Bug,Result)
```

```
       then
         begin
           distance := NormVector(Result);
           RampedSpeed := MaxSpeed * (distance/StoppingRadius);
           ScaleVector(( Min(RampedSpeed, MaxSpeed))/Distance, Result, Result);
           DifVector(Result,Velocity,Result);
         end;
   end;
end;
```

Gruppenbasierte Steuerverhalten

```
/*------------------------------------------------*/
// SeparationSteering
// Return steering vector for separation-steering
function TSteeringBehaviour.SeparationSteering(Bug: TBug): Vector;

var NearbyBugList:TList;
    PositionOffset:Vector;
    i:cardinal;
begin
  setlength(Result, DimMatrix);
  setlength(PositionOffset, DimMatrix);
  NearbyBugList := GetNearbyBugs(Bug);
  if NearbyBugList.Count > 0
    then
      begin
        for i := 0 to NearbyBugList.Count - 1 do
          begin
            DifVector(Bug.Position, TBug(NearbyBugList[i]).Position,
                      PositionOffset);
            NormalizeVector(PositionOffset,PositionOffset);
            ScaleVector(1/NearbyBugList.Count,PositionOffset, PositionOffset);
            SummVector(Result, PositionOffset, Result);
          end
      end;
  NearbyBugList.Destroy;
end;

/*------------------------------------------------*/
// AligmentSteering
// Return steering vector for aligment-steering
function TSteeringBehaviour.AlignmentSteering(Bug: TBug): Vector;
var NearbyBugList:TList;
    i:cardinal;
begin
  setlength(Result,DimMatrix);
  NearbyBugList := GetNearbyBugs(Bug) ;
  if NearbyBugList.Count > 0
    then
      begin
        for i := 0 to NearbyBugList.Count - 1 do
          SummVector (Result,TBug(NearbyBugList[i]).Velocity,Result);
        ScaleVector(1/NearbyBugList.Count, Result, Result);
        DifVector(Result,Bug.Velocity, Result);
      end;
  NearbyBugList.Destroy;
end;
```

```
/*-------------------------------------------------*/
// CohesionSteering
// Return steering vector for cohesion-steering
function TSteeringBehaviour.CohesionSteering(Bug: TBug): Vector;
var NearbyBugList:TList;
    i:cardinal;
begin
  setlength(Result,DimMatrix);
  NearbyBugList := GetNearbyBugs(Bug);
  if NearbyBugList.Count > 0
    then
      begin
        for i := 0 to NearbyBugList.Count - 1 do
          SummVector (Result,TBug(NearbyBugList[i]).Position,Result);
        ScaleVector(1/NearbyBugList.Count, Result, Result);
        DifVector(Result,Bug.Position, Result);
      end;
  NearbyBugList.Destroy;
end;
```

Hindernisbasierte Steuerverhalten

```
/*-------------------------------------------------*/
// ObstacleAvoidanceSteering
// Return steering vector for ObstacleAvoidanceSteering
function TSteeringBehaviour.ObstacleAvoidanceSteering(Bug: TBug): Vector;
var i:cardinal;
    NearestObstacle:TObstacle;
    IntersectionPoint, RangePoint, Perpendicular:Vector;
    distance:extended;
    BugSideIndicator, PerpendicularSideIndicator:extended;
    intersectionOneValid:boolean;

/*-------------------------------------------------*/
// check whether obstacles in range and return the nearest obstacle and its
// distance to the bug
function ObstacleNear(var NearestObstacle:TObstacle;
                      var distance:extended):boolean;
var i:integer;
    Obstacle:TObstacle;
    oldDistance:extended;
    var temp:vector;

begin
  setlength(temp, dimmatrix);
  Result := false;
  olddistance := sensorrange + 1;

  // check all obstacles for intersection
  for i := 0 to ObstacleList.Count -1 do
    begin
      Obstacle := TObstacle(ObstacleList.Items[i]);
      case Obstacle.ObstacleType of
        otLine: begin
                  if LinesIntersect(Bug.Position,RangePoint,Obstacle.Pos1,
                                    Obstacle.Pos2, temp)
                    then
                      begin
                        CopyVector( Temp, IntersectionPoint);
                        DifVector(Bug.Position,Temp,Temp);
                        distance := NormVector(Temp);
                        // check if found obstacle is nearer than obstacles
                        // already found
                        if olddistance > distance
```

```
                            then
                              begin
                                Olddistance := distance;
                                result := true;
                                NearestObstacle := Obstacle;
                              end;
                          end;
                end;
        otEllipse: begin
                     if LineCircleIntersect(Bug.Position, RangePoint,
                      Obstacle.Center,Obstacle.radius,sensorrange,distance,temp)
                       then
                           begin
                             CopyVector( Temp, IntersectionPoint);
                             if olddistance > distance
                               then
                                 begin
                                   Olddistance := distance;
                                   result := true;
                                   NearestObstacle := Obstacle;
                                 end;
                           end;
                   end;
            end;
    end;
   Temp := Nil;
end;

// B on right side of line(p1,p2) then 1 else -1
function DetermineSide(P1o,P2o,B:vector):integer;
var p1,p2:vector;
begin
  setlength(p1,Dimmatrix);
  setlength(p2,Dimmatrix);
  CopyVector(p1o,p1);
  CopyVector(p2o,p2);
  DifVector(p1,p2,p2);
  DifVector(p1,b,p1);
  if (p1[0]*p2[1]-p2[0]*p1[1]) < 0
    then Result := 1
    else Result := -1;
  p1 := nil;
  p2 := nil;
end;

begin
  setlength(IntersectionPoint, DimMatrix);
  setlength(RangePoint, DimMatrix);
  setlength(Perpendicular, DimMatrix);
  setlength(Result,DimMatrix);

  // determine end point (range point)of the probe-line
  NormalizeVector(Bug.Velocity, RangePoint);
  ScaleVector(SensorRange, RangePoint,RangePoint);
  SummVector(Bug.Position, RangePoint, RangePoint);

  if ObstacleNear(NearestObstacle, Distance)
    then begin
      case NearestObstacle.ObstacleType
        of otLine: begin
                     // create vector perpendicular to obstacle line
                   DifVector(NearestObstacle.Pos1, NearestObstacle.Pos2, Result);
                     if Result[0] <> 0
                       then begin
```

```
                        Perpendicular[0] := - Result[1] / Result[0];
                        Perpendicular[1] := 1;
                        NormalizeVector(Perpendicular,Perpendicular);
                      end
                      else begin
                        Perpendicular[0] := - 1;
                        Perpendicular[1] := 0;
                      end;
                // check if perpendicular vector points to the side, the bug is in
                     BugSideIndicator := DetermineSide(NearestObstacle.Pos1,
                                          NearestObstacle.Pos2,Bug.Position);
                     SummVector(NearestObstacle.Pos1,Perpendicular,Result);
               PerpendicularSideIndicator := DetermineSide(NearestObstacle.Pos1,
                                                 NearestObstacle.Pos2, Result);
         // if perpendicular vector does not head in the right direction, revers it
                     if BugSideIndicator <> PerpendicularSideIndicator
                        then
                           ScaleVector(- Bug.MaxSpeed,Perpendicular,Result)
                        else
                          ScaleVector(  Bug.MaxSpeed,Perpendicular,Result);
                      end;

           otEllipse: begin
                        // check if bug inside circle to determine the direction
                        // perpendicular vector must head to
                       if PointInsideCircle(Bug.Position,NearestObstacle.Center,
                                            NearestObstacle.Radius)
                          then
                        DifVector(NearestObstacle.Center,IntersectionPoint,Result)
                          else
                       DifVector(IntersectionPoint,NearestObstacle.Center,Result);
                          NormalizeVector(Result,Result);
                          ScaleVector(Bug.Maxspeed,Result,Result);
                     end;
                end;
      end;
 IntersectionPoint := Nil;
 RangePoint := Nil;
 Perpendicular := Nil;
end;

/*-------------------------------------------------*/
// LinesIntersect
// line intersection in 2d
function TSteeringBehaviour.LinesIntersect(a, b, c, d: Vector;
  var ResV: Vector): boolean;
var ua,ub, denomin :extended;
begin
  Result := false;
  denomin := (d[1]-c[1])*(b[0]-a[0]) - (d[0]-c[0])*(b[1]-a[1]);
  //lines parrallel?
  if denomin <> 0
    then begin
      ua := ((d[0]-c[0])*(a[1]-c[1]) - (d[1]-c[1])*(a[0]-c[0])) / denomin;
      ub := ((b[0]-a[0])*(a[1]-c[1]) - (b[1]-a[1])*(a[0]-c[0])) / denomin;
      // intersection point between given points?
      if (ua >= 0) and (ua<=1) and (ub>= 0) and (ub<=1)
        then begin
           ResV[0] := a[0] + ua*(b[0]-a[0]);
           ResV[1] := a[1] + ua*(b[1]-a[1]);
           Result := true;
        end;
    end;
end;
```

```
/*-------------------------------------------------*/
// LineCircleIntersect
// Line Circle intersection in 2D
function TSteeringBehaviour.LineCircleIntersect(point1,point2,Center:Vector;
r,SensorRange:extended; var distance:extended; var ResV:Vector):boolean;
var d,discriminant,distance1, distance2:extended;
    dx,dy,drr:extended;
    temp1,temp2,temp3, temp4:extended; //temp values used for speeding up calc.
    TempIntersectionPoint, DiffVector,p1,p2:vector;
    sensorHeading:vector;
    IntersectionOneValid:boolean;

// determine the sign of a number
function sgn(a:extended):integer;
begin
  if a < 0
    then Result := -1
    else Result := 1;
end;


// check if vector a,b head in the same direction
function DirectionEquals(a,b:Vector):boolean;
begin
  Result := (sgn(a[0]) = sgn(b[0])) and (sgn(a[1]) = sgn (b[1]));
end;

begin
  setlength(sensorHeading, Dimmatrix);
  setlength(TempIntersectionPoint, Dimmatrix);
  setlength(DiffVector, Dimmatrix);
  setlength (p1, DimMatrix);
  setlength (p2, DimMatrix);
  result := false;
  CopyVector(point1,p1);
  CopyVector(point2,p2);
  DifVector(point1,point2,SensorHeading);

  DifVector(p1,center,p1);
  DifVector(p2,center,p2);

  dx := p2[0] - p1[0];
  dy := p2[1] - p1[1];
  drr := sqr(dx) + sqr(dy);
  if drr = 0 then drr := 1;
  D := p1[0]*p2[1] - p2[0]*p1[1];

  discriminant := sqr(r)*drr- sqr(d);

  // intersects line circle?
  if not (discriminant < 0)
    then begin
       // compute first intersection point
       temp1 :=  sqrt(sqr(r)*drr-sqr(d));
       temp2 :=  sgn(dy) * dx * temp1;
       temp3 :=  abs(dy) * temp1;

       ResV[0] := (d*dy + temp2) / drr;
       ResV[1] := (-d*dx + temp3) / drr;
       SummVector(Resv,Center,Resv);
       DifVector(point1,ResV,DiffVector);

       // check if intersection in range of bug
       if DirectionEquals(DiffVector, SensorHeading)
```

```
         then
           begin
             distance1 := NormVector(DiffVector);
             if Distance1 < SensorRange
               then
                 begin
                   distance := distance1;
                   result := true;
                   IntersectionOneValid := true;
                end
           end
        else IntersectionOneValid := false;

       // line not tangent to circle
       if discriminant > 0
       then begin
             // compute second intersection point
             TempIntersectionPoint[0] := (d*dy - temp2) / drr;
             TempIntersectionPoint[1] := (-d*dx - temp3) / drr;
             SummVector(TempIntersectionPoint,Center,TempIntersectionPoint);
             DifVector(point1,TempIntersectionPoint,DiffVector);
             // check if intersection 2 is valid and closer than intersectgion 1
             if DirectionEquals(DiffVector, SensorHeading)
                then
                  begin
                    distance2 := NormVector(DiffVector);
                    if not (distance2 > SensorRange)
                      then
                        begin
                          if IntersectionOneValid
                            then
                              begin
                                if (Distance2 < Distance1)
                                  then
                                    begin
                                      CopyVector(TempIntersectionPoint, ResV);
                                      distance := distance2;
                                      Result := true;
                                    end;
                              end
                          else
                            begin
                              CopyVector(TempIntersectionPoint, ResV);
                              distance := distance2;
                              Result := true;
                            end;
                        end;
              end;
        end;
    end;
  p1 := nil; p2 := nil;
  sensorHeading := nil;
  TempIntersectionPoint := Nil;
  DiffVector := Nil;
end;
```

```
/*-------------------------------------------------*/
//PointInsideCircle
//check if distance from point to center of circle is smaller than radius
function TSteeringBehaviour.PointInsideCircle(Point, Center: Vector;
  Radius: Extended): boolean;
var Distance:extended;
  DiffVector:vector;
begin
  setlength(DiffVector, Dimmatrix);
  DifVector(Point, Center, DiffVector);
  Distance := NormVector(DiffVector);
  if Distance < Radius
    then
      Result := true
    else
      Result := false;
  DiffVector := nil;
end;
```

Kombination von Steuerverhalten

```
/*-------------------------------------------------*/
// GetDesiredSteeringForce
// combine different steering behaviours to get one steering vector
function TBug.GetDesiredSteeringForce: Vector;
var i:integer;
    weightsum:integer;
    SB:TSteeringBehaviour;
    tempVector:Vector;
begin
  setlength(Result,DimMatrix);
  setlength(tempVector, DimMatrix);
  weightsum := 0;

  // choose from possible combine modes
  case SBCombineMode of
    // linear combination
    sbctLinearCombination:
      begin
        with SteeringBehaviourList do
          if Count > 0
            then
              begin
                // get different steering vectors
                for i := 0 to Count-1 do
                 begin
                tempVector := TSteeringBehaviour(Items[i]).GetSteeringForce(self);
                    if not NullVector(tempVector)
                      then
                        begin
                          SummVector(Result,tempVector, Result);
                          inc(weightsum);
                        end;
                 end;
                if weightsum > 0
                  then
                    ScaleVector(1/(weightsum),Result,Result);
            end;
       end;
    // linear combination with weight factors
    sbctWeightFactors:
        begin
          with SteeringBehaviourList do
            if Count > 0
```

```
              then
                begin
                  for i := 0 to Count-1 do
                    begin
                      SB := TSteeringBehaviour(Items[i]);
                      tempVector := SB.GetSteeringForce(self);
                      if not NullVector(tempVector)
                        then
                          begin
                            ScaleVector(SB.Weight,tempVector,tempVector);
                            SummVector(Result,tempVector,Result);
                            weightsum := weightsum + SB.weight;
                          end;
                    end;
                  if weightsum > 0
                    then
                      ScaleVector(1/weightsum,Result,Result);
               end;
        end;

    // combination priority
    sbctPriority:
       begin
          with SteeringBehaviourList do
            if Count > 0
              then
                begin
                  i := 0;
                  repeat
                    tempVector[0] := 0;
                    tempVector[1] := 0;
                    SB := TSteeringBehaviour(Items[i]);
                    tempVector := SB.GetSteeringForce(self);
                    i := i+1;
               until (tempVector[0] <> 0) or (tempVector[1] <> 0) or (i > Count-1);
                  CopyVector(tempVector,Result);
                end;
       end;
   end;

// if no steering force is computed, take the one from last simulation step
  if (Result[0] = 0) and (Result[1] = 0)
    then CopyVector(Velocity,Result);

end;
```

Bewegung eines autonomen Agenten

```
/*------------------------------------------------*/
// Move
// Move the bug according to his steering behaviours and its physical properies
procedure TBug.Move;
var SimulStepVelocity:vector;
begin
  CheckTriggerEvents;

  // get SteeringForce from the Steering Behaviours
  setlength(SimulStepVelocity,DimMatrix);
  DesiredSteeringForce := GetDesiredSteeringForce;
  // apply force to the vehicles point mass
  TruncateVector(maxForce,DesiredSteeringForce,SteeringForce);

  if Mass <> 0 then
    ScaleVector(1/Mass,SteeringForce,Acceleration);
```

```
  SummVector(FVelocity,Acceleration,FVelocity);
 // Clip to maxSpeed
  TruncateVector(maxSpeed ,FVelocity,FVelocity);
 // Clip to MinSpeed
  if MinSpeed <> 0 then begin
   TruncateVector(-minSpeed, FVelocity,FVelocity);
  end;

  ScaleVector(1/Simulation.Fps,FVelocity,SimulStepVelocity);
  SummVector(FPosition,SimulStepVelocity,FPosition);

  // check if bug is outside arena and if it is, mak him come again on the opposite
  // side
  AdjustPosition;

  // adjust orientation
  UpdateOrientationSpeed;
  //SetVisualisationValues;
  SetVisualisationPosition;
end;
```

Simulationsablauf

```
// Run
// Run the Simulation process
procedure TSimulation.Run;
var i,j:integer;
begin
  DissableBugCreation;
  SimulationRunning := true;

  // run the simulation till it is paused
  while SimulationRunning do
    begin
      for i := 0 to pred(GroupCount)do
        with TBugGroup(GroupList.Items[i]) do
          for j := 0 to pred(MemberList.Count) do
            TBug(MemberList.Items[j]).Move;

  // update visual representation
  GraphicObjects.PaintAll;

  // update current time
  if CurrentFrame = (FPS - 1)
     then begin
       CurrentSecond := CurrentSecond + 1;
       CurrentFrame  := 0;
      end
      else
        CurrentFrame  := CurrentFrame +1;

   // check for loop
   if Loop and (CurrentSecond = LoopSecond) and (CurrentFrame = LoopFrame)
       then  ResetToStartValues;

    Application.ProcessMessages;
    sleep(delaytime);
  end;
end;
```

Plugin für Maya 2.5

```
// Import bugz-Motion Data
// Carsten.Kolve@gmx.de

proc bmdbmdImport (string $fileName, float $scaleFactor, string $objekt, int $par-
entBool, int $alignBool)
{
string $line;
int $numTokens;

   print("Import bugzMotionData (c) 04.00 Carsten.Kolve@gmx.de\n");

   // fileDialog canceled?

   if ( $fileName == "" )
   {
      print "No file selected, quitting!";
      return;
      }

// fileName valid?

int $fileID = fopen($fileName, "r");

if ( $fileID == 0)
{
      print ("Error opening file: "+$fileID+"\n");
      return;
      }

   // read group properties

   string $groupname;
   int $membercount;
   string $buffer[];
   string $curvename[];

   $line= fgetline($fileID);
   $numTokens = ‘tokenize $line ";" $buffer‘;

   $groupname = $buffer[0];
   $membercount = $buffer[1];

   // create group locator

   if ($parentBool)
   {
     print("Creating Group: "+$groupname+"\n");

     spaceLocator;
     rename $groupname;
   }

   // load members

   string $membername;
   float $help;
   int $keycount;
   int $keyindex;
   int $memberindex;
   int $helpindex;

   for ($memberindex = 0; $memberindex < $membercount; $memberindex++)
```

```
{
  $line= fgetline($fileID);
  $numTokens = ‘tokenize $line ";" $buffer‘;

  $membername = $buffer[0];
  $keycount = $buffer[1];

  // create member locator

  print("Creating Member: "+$membername+"\n");

  spaceLocator;
  rename $membername;


  print("Assigning Motion Data.\n");
  for ($keyindex = 0; $keyindex < ($keycount+1); $keyindex++)
  {
    $line = fgetline($fileID);

   // substitute ’,’ with ’.’, as maya excepts no ’,’ in floating point numbers
    // as mel supports no regular expression search on multiple occurences of
   // an charakter,  doing the substitution 3 times will make sure no ’,’ is
    // in $line

    for ($helpindex = 0;$helpindex < 3;$helpindex++)
    {
      $line = ‘substitute "," $line "."‘;
    }

     $numTokens = ‘tokenize $line ";" $buffer‘;

     // create keyframes

     $help = $buffer[1];
     $buffer[1] = $help * $scaleFactor;

     $help = $buffer[2];
     $buffer[2] = $help * $scaleFactor;

     $help = $buffer[3];
     $buffer[3] = $help * $scaleFactor;

     currentTime $buffer[0];
     move -a -os -wd -xyz $buffer[1] $buffer[2] $buffer[3];
     $curvename = ‘setKeyPath‘;
     // align to path

  }


  // align to path
  if ($alignBool)
   {
    pathAnimation -edit -follow on -c $curvename[0];
   }

  // replace locator with objekt

  if ($objekt != "")
  {
   parent($objekt, $membername);
  }
```

```
     // parent to group
     if ($parentBool)
     {
       parent($membername,$groupname);
     }
   }

// close file

   fclose($fileID);

   print "\nImport of Motion Data finished! \n";
}


proc bmdGetFileName()
{
 // get filename from fileDialog and put it in the filename edit

 string $filename =  'fileDialog -directoryMask "*.bmd"';
 global string $bmdFileGroup;
 if (!($filename == ""))
 {
   textFieldButtonGrp -e -fi $filename $bmdFileGroup;
 }
}

proc bmdGetSelectedObject()
{
  // get current selected object and put name in edit

  global string $objGroup;
  string $selection[];
  $selection = 'selectedNodes';
  textFieldGrp -e -text $selection[0] $objGroup;
}

proc bmdStartImport()
{
// get all values from gui and start import procedure

 global string $objGroup;
 global string $bmdFileGroup;
 global string $ParentGroupCheckbox;
    global string $AlignPathCheckbox;
 global string $scaleValue;

 $fn = 'textFieldButtonGrp -query -fi $bmdFileGroup';
 $sv = 'floatFieldGrp -query -value1 $scaleValue';
 $ob = 'textFieldGrp -query -text $objGroup';
 $ap = 'checkBoxGrp -query -value1 $AlignPathCheckbox';
 $pg = 'checkBoxGrp -query -value1 $ParentGroupCheckbox';

    bmdbmdImport($fn,$sv,$ob,$pg,$ap);
}

proc bmdMainWindow ()
{
  // skript job for obtaining currently selected object

  global int $jobId;

  $jobId = 'scriptJob -ct "SomethingSelected" "bmdGetSelectedObject"';
```

```
  // gui window setup

  window -title "bugzMotionData Import (c) 04.00 carsten.kolve@gmx.de"
         -minimizeButton false
         -maximizeButton false
         -width 440
         -sizeable false;


  string $layout1 = ‘columnLayout‘;

  global string $bmdFileGroup;
  global string $objGroup;
  global string $ParentGroupCheckbox;
  global string $AlignPathCheckbox;
  global string $scaleValue;

  $bmdFileGroup = ‘textFieldButtonGrp -label "Choose bmd-File" -fi "*.bmd" -but-
tonLabel "Browse..." -bc "bmdGetFileName" ‘;
  $scaleValue = ‘floatFieldGrp -numberOfFields 1 -label "Scale Values by" -value1
1‘;

  $objGroup = ‘textFieldGrp -label "Choose Object"‘;
  text -label "                                                    (select object in
scene or leave blank to create Locator)";

  $ParentGroupCheckbox = ‘checkBoxGrp -numberOfCheckBoxes 1 -label "" -value1 true
-label1 "Parent Members to Group" -cw 2 200‘ ;
  $AlignPathCheckbox = ‘checkBoxGrp -numberOfCheckBoxes 1 -label "" -value1 true
-label1 "Align Member Heading to Path" -cw 2 200‘ ;


  separator -height 20 -width 440;

  rowColumnLayout -numberOfColumns 3
    -columnWidth 1 100
    -columnWidth 2 90
    -columnWidth 3 90;

  text -label "";
  button -label "OK" -c "bmdStartImport()";
  button -label "Cancel" ;

  showWindow;
}


global proc bmdImport()
{
  bmdMainWindow();
}

bmdImport();
```

Plugin für Lightwave 6.0

```
// bugz-Motion Data Import
// carsten kolve, 04.2000

@version 2.0
@warnings
@script generic


generic
{

// create gui

    return if !reqbegin("bugzMotionData Import (c) 04.00 Carsten.Kolve@gmx.de");

    c0 = ctlfilename("Choose bmd-File","*.bmd");
    c1 = ctlnumber("Scale Values by",1.0);
    c2 = ctlfilename("Choose lwo-File","*.lwo");
    c3 = ctltext("","(leave blank to create NULL-Objects)");
    c4 = ctlcheckbox("Parent Members to Group",true);
    c5 = ctlcheckbox("Align Member Heading to Path",true);

// get values from gui

    if(reqpost())
    {
        bmdFileName = getvalue(c0);
        scaleFactor = getvalue(c1);
        lwoFileName = getvalue(c2);
        parentBool =  getvalue(c4);
        pathBool = getvalue(c5);
    }
    else
      return;

    reqend();


   directory = recall("directory","C:\\");

// error handling

   if (!fileexists(bmdFileName))
   {
      error("Specified file does not exist, quitting!");
      return;
   }

   lwoExchange = fileexists(lwoFileName);

   base = split(bmdFileName);

   directory = string(base[1],base[2]);

   store("directory",directory);

// open file

   bmdFile = File(bmdFileName,"r");

   if(bmdFile == nil)
   {
```

```
      error("Cannot open bmdMotion File \"",bmdFileName,"\""+", quitting!");
      return;
   }

// get scene information

   scene = getfirstitem(SCENE);
   fps = scene.fps;

// get group information

   groupdata = bmdFile.parse(";");
   groupname = groupdata[1];
   membercount = integer(groupdata[2]);

// create group null

   if (parentBool)
   {
     AddNull(groupname);
     group = getfirstitem(groupname);
   }

// load group members

   for index1 = 1 to membercount do
   {

// get member information

      memberdata = bmdFile.parse(";");
      membername = memberdata[1];
      keycount = integer(memberdata[2]);


// create member null


      AddNull(membername);
      member = getfirstitem(membername);

      if (parentBool)
      {
        ParentItem(groupname);
      }

// store first key (lw bug: first key must be set last)

      firstkeydata = bmdFile.parse(";");

      for index2 = 1 to keycount do
      {

// read key information and set key

         keydata = bmdFile.parse(";");
         Position(number(keydata[2])*scaleFactor,number(keydata[3])*scaleFac-
tor,number(keydata[4])*scaleFactor);
         CreateKey(integer(keydata[1])/fps);
      }

// set first key

     Position(number(firstkeydata[2])*scaleFactor,number(firstkeydata[3])*scale-
```

```
Factor,number(firstkeydata[4])*scaleFactor);
      CreateKey(integer(firstkeydata[1])/fps);

// allign member null heading to motion path

      if (pathBool)
      {
         HController(2);
      }

// exchange null with object file

if (lwoExchange)
{
   ReplaceWithObject(lwoFileName);
}

   }

   bmdFile.close();
```

Abbildungsverzeichnis

Literaturverzeichnis

[1] Helmut Balzert, „Lehrbuch der Softwaretechnik“, Spektrum Akademischer Verlag, Heidelberg, 1996.

[2] Colin Beardon, Victor Ye, „Using behavioural rules in animation“, Computer Graphics International, Leeds, Juni 1995.

[3] Bruce Blumberg, Tinsley Galyean, „Multi-Level Direction of Autonomous Creatures for Real-Time Virtual Environments“, MIT Media Lab, Cambridge, MA, 1995.

[4] Valentino Braitenberg, „Vehicles: Experiments in Synthetic Psycholoy“, MIT Press, Cambridge, MA, 1984.

[5] Der Brockhaus in fünf Bänden, 8. Auflage, F.A. Brockhaus, Leipzig, 1993.

[6] Ilja Bronstein, Konstantin Semendjajew, „Taschenbuch der Mathematik“, Verlag Harri Deutsch, Frankfurt/Main, 1991

[7] Rodney Brooks, „Intelligence Without Reason“, MIT - A.I. Lab, A.I. Memo No.1293, Cambridge, MA, 1991.

[8] Otto Bruhns, Theodor Lehmann, „Elemente der Mechanik“, Vieweg, Braunschweig, 1993.

[9] James Foley, John Hughes, „Grundlagen der Computergrafik“, Addison-Wesley, Bonn, 1994.

[10] Carlos Ho, „A Framework for Behavioural Control in Computer Animation“, Thesis for the degree of Doctor of Philosophy, University of Sussex, September 1996.

[11] Stuart Mealing, „State-of-the-Art: Behavioural Animation“ in „The Art and Science of Computer Animation“, Intellect, Oxford, 1992, 211-227

[12] Glenn Proctor, Chris Winter, „Information Flocking: Data Visualisation in Virtual Worlds Using Emergent Behaviours“, BT Laboratories, 1997.

[13] William Reeves, „Particle Systems - A Technique for Modeling a Class of Fuzzy Objects“, Computer Graphics, vol. 17 no. 3, 1983, 359-376.

[14] Craig Reynolds, „Flocks, Herds, and Schools: A Distributed Behavioral Model“, Computer Graphics, 21(4), Juli 1987, 25-34.

[15] Craig Reynolds, „Steering Behaviors for Autonomous Characters“, Game Developers Conference 1999

[16] Craig Reynolds, „Not Bumping Into Things“, Notes on „obstacle avoidance“ for the course on Physically Based Modelling, Siggraph, 1988.

[17] Helmut Schwarz, „Numerische Mathematik“, B.G. Teubner, Leipzig, 1997.

[18] Peter Small, „Magical A-Life Avatars“, Manning Publications Co., Greenwich, 1998

[19] Demetri Terzopoulus, Tamer Rabie, „Animat Vision: Active Vision in Artificial Animals“, MIT Press, Videre: Journal of Computer Vision Research Vol. 1 Nr.1, Cambridge, MA, 1997.

[20] Demetri Terzopoulus, Xiaoyuoan Tu, Radek Grzeszeczuk, „Artificial Fishes: Autonomous Locomotion, Perception, Behavior, and Learning in a Simulated Physical World“, Artifical Life, 1(4), 1994, 327-351.

[21] Elmar Warken, „Professionelle Programmierung: Delphi 4“, Addison-Wesley, Bonn, 1999.

[22] Klaus Weltner, „Mathematik für Physiker“, Vieweg, Braunschweig, 1990.

[23] Wolfgang Wickler, „Soziales Verhalten als ökologische Anpassung“, Verh. Dtsch. Zool. Ges. 64, Hoffmann und Campe, Hamburg, 1970, 291-304.

Verzeichnis der Online-Quellen

[O-1] 3k Delphi Tips
http://members.iworld.net/khouse/3kdt/

[O-2] Agent Technologies
http://www.insead.fr/calt/Encyclopedia/ComputerSciences/Agents/

[O-3] Artificial Life Links
http://www.alcyone.com/max/links/alife.html

[O-4] Bob’s LightWave Lair
http://www.lightwave-outpost.com/employees/bobh/

[O-5] The Computational Beauty of Nature
http://mitpress.mit.edu/books/FLAOH/cbnhtml/index.html

[O-6] C++ Boids
http://www.media.mit.edu/~ckline/cornellwww/boid/boids.html

[O-7] Computational Geometry Pages
http://compgeom.cs.uiuc.edu/~jeffe/compgeom/

[O-8] Computational Geometry, Paul Bourke
http://www.swin.edu.au/astronomy/pbourke/geometry/

[O-9] Craig Reynolds, Homepage
http://www.red.com/cwr/

[O-10] Delphi, To make Controls in Runtime
http://www.q3.nu/trucomania/truco.cgi?21&ing

[O-11] DeskTop Mathematik
http://cdrom2.ub.uni-leipzig.de/DTMath/daten/auto/sonder/inhalt.htm

[O-12] Eric Weisstein's World of Mathematics
http://mathworld.wolfram.com/

[O-13] Flocking & Schooling
http://www.susqu.edu/facstaff/b/brakke/complexity/hagey/flock.htm

[O-14] Gamasutra, "Implementing A Group Behavioral Control System Using Maya"
http://www.gamasutra.com/features/19991011/mayaflock_01.htm

[O-15] Gamasutra, "A Modular Framework for Artificial Intelligence Based on Stimulus Response Directives"
http://gamasutra.com/features/19991110/guy_01.htm

[O-16] Gamasutra, "AI for Games and Animation"
http://gamasutra.com/features/19991206/funge_01.htm

[O-17] Geometry in Action
http://www.ics.uci.edu/~eppstein/geom.html

[O-18] Graphics Algorithms FAQ
http://www.cc.iastate.edu/olc_answers/packages/graphics/algos.faq.html

[O-19] Lightsource - Lightwave Resource Site
http://www.lightsource-3d.com/lscript.htm

[O-20] The Math Forum Internet Mathematics Library
http://forum.swarthmore.edu/library/

[O-21] The Ridge 6444 Game Programming Site
http://www.geocities.com/SiliconValley/Ridge/6444/

[O-22] Torry's Delphi Page
http://www.torry.ru

[O-23] Using forces to animate Objects, Theory Center, Cornell University
http://www.tc.cornell.edu/Visualization/Education/cs417/SECTIONS/dynamics.html

[O-24] VFX Pro - The Art, Technology and Business of Visual Effects
http://vfxpro.com

Erklärung

Hiermit versichere ich, daß die vorliegende Arbeit von mir selbständig und ohne fremde Hilfe verfaßt wurde. Sämtliche verwendete Quellen und Hilfsmittel sind im Text oder in den Quellenverzeichnissen nachgewiesen.

Carsten Kolve
Dortmund, 28. April 2000