Flow Fields zur Kollisionsvermeidung in Spielen

Michał Jan Karpowicz

DIPLOMARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2011

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 26. September 2011

Michał Jan Karpowicz

Inhaltsverzeichnis

Vorwort

Obwohl einige wunderbare Menschen – insbesondere enge Freunde und Angehörige meiner Familie – eine Nennung an dieser Stelle verdient hätten, beschränke ich mich im Vorwort auf nur eine Person, deren Tod die Fertigstellung dieser Diplomarbeit überschattet hat.

So möchte ich diese Arbeit meinem Großvater Jerzy Karpowicz, der am 22. September 2011 verstorben ist, widmen. *Pokój jego duszy.*

Kurzfassung

Verfahren zur Kollisionsvermeidung spielen in verschiedenen Gebieten eine wichtige Rolle. So ist das Verhindern von Zusammenstößen zwischen sich bewegenden Objekten im Bereich der Computeranimation, in der Robotik sowie in Computerspielen unerlässlich. Bei letzteren hat sich das Verfahren von Craig W. Reynolds als Industriestandard etabliert. Da bei diesem nur das gefährlichste Hindernis berücksichtigt wird, kann es in Situationen mit vielen potenziellen Kollisionen zu widersprüchlichen Ergebnissen führen.

Flow Fields stellen eine vielversprechende Alternative zum Verfahren von Reynolds dar. Bei diesem Konzept, dessen Ursprünge im Bereich der Robotik liegen, werden alle Hindernisse als Repulsionen in ein Vektorfeld einberechnet, wodurch alle möglichen Kollisionen berücksichtigt werden.

In dieser Arbeit wurden beide Verfahren theoretisch aufgearbeitet und deren Implementierung im Spiel *Delta Strike* beschrieben. Mithilfe dieser Umsetzung wurde eine empirische Studie durchgeführt, um die Vor- und Nachteile der beiden Ansätze in einer realistischen Spielsituation zu untersuchen. Die Ergebnisse dieser Untersuchung werden am Ende der Arbeit vorgestellt und diskutiert.

Abstract

Methods for collision avoidance play a key role in various areas, such as CG animation, robotics, and computer games. The method proposed by Craig W. Reynolds has been established as the de facto standard for the latter. However, as only the most dangerous obstacle is taken into consideration while utilizing this method, situations with many potential collisions can lead to contradictory results.

The concept of *Flow Fields*, which has its origin in robotics, poses a budding alternative to Reynolds' approach. In this technique, all obstacles emit a repulsing force, which can be used to calculate a vector field. This facilitates the consideration of all possible collisions.

Both methods are discussed theoretically in this thesis and their implementation in the game *Delta Strike* is described in detail. Using these implementations, an empirical study was conducted to analyze the benefits and disadvantages of both approaches. The results of the analysis are being presented and discussed at the end of the thesis.

Kapitel 1

Einleitung

1.1 Motivation

> The game AI [artificial intelligence, M.K.] revolution is at hand.

Mit diesem Satz beendet Tozour in [34, S. 3] die Einleitung eines Artikels, in dem er die Entwicklung der künstlichen Intelligenz (kurz *KI*) in Computerspielen beschreibt – von einer Komponente die meist erst in letzter Minute vor der Veröffentlichung erledigt wurde, hin zu einem wichtigen Kriterium für den Erfolg eines Spiels.

Im folgenden Abschnitt wird ein kurzer Überblick über das Gebiet der akademischen *KI* (oder auch *Mainstream-KI*, vgl. [34, S. 5]) gegeben, auf dessen Basis auf die Ziele und Funktionsweisen der künstlichen Intelligenz in Spielen eingegangen wird.

1.1.1 Künstliche Intelligenz

Überblick

Eine Beschreibung des ganzen Gebiets der akademischen *KI* inklusive dessen Methoden, Tendenzen und Untergebiete wäre ein Unterfangen, das den Rahmen und das Ziel dieses Kapitels sprengen würde. Stattdessen soll hier ein kurzer Überblick über die Forschungsziele in dem Bereich gegeben werden, der eine Trennung zur künstlichen Intelligenz in Computerspielen ermöglicht.

Schon das Finden einer einheitlichen Definition des Begriffs *KI* stellt eine Herausforderung dar, nicht zuletzt durch den Mangel einer solchen für den allgemeinen Terminus *Intelligenz*. So werden in [29, S. 5] acht Zitate aufgelistet, die verschiedene Strömungen in der künstlichen Intelligenz darstellen.

Trotz der Unterschiede in diesen Tendenzen fallen zwei wichtige Merkmale auf, die sie gemeinsam haben: zum einen ist es das Bestreben dieser Wissenschaft menschliche Intelligenz, bzw. Rationalität, zu verstehen, zum anderen der Versuch eine Maschine (beispielsweise einen Computer) so zu programmieren, dass diese, einem Menschen gleich, eigenständig Probleme

bearbeiten und lösen kann. Den Unterschied zwischen dieser Vision einer *intelligenten Maschine* und dem Ziel der künstlichen Intelligenz in Computerspielen zeigt folgendes Zitat von Mat Buckland (aus [9, *Introduction*]):

> With regard to game AI I am firmly of the opinion that if the player *believes* the agent [[1], M.K.] he's playing against is intelligent, then it *is* intelligent. It's that simple. Our goal is to design agents that provide the *illusion of intelligence*, nothing more.

In vorangegangenem Zitat wird einer der wichtigsten Aspekte der künstlichen Intelligenz in Spielen aufgezeigt: Sie muss für den Benutzer glaubhaft wirken. Im Gegensatz zur akademischen *KI* ist es also nicht nötig rationales Verhalten nachzubilden – die computergesteuerten Agenten müssen lediglich die Illusion erwecken, intelligent zu sein. Dabei stehen zwei Faktoren im Vordergrund: Die *KI* in einem Spiel muss Spaß machen und gleichzeitig fordernd sein (s. [31]).

Beispiele für *KI* in Spielen

In jedem Genre gibt es Computerspiele, in welchen die Implementierung der künstlichen Intelligenz vorbildlich gelöst oder gar revolutioniert wurde. In der Fachliteratur (vgl. [20, Kap. 1] und [34]) werden einige Beispiele für solche genannt:

Im Bereich der *Ego-Shooter*[2] wird die *Half-Life* Reihe[3] von *Valve* häufig als Exempel für gut implementierte künstliche Intelligenz erwähnt. So werden besonders die Fähigkeit zum Fällen von komplexen Entscheidungen und das strategische Gruppenverhalten hervorgehoben. Zweites wird auch bei *Unreal Tournament* (1999) von *Epic Games* und *Tom Clancy's Ghost Recon* (2001) von *Red Storm Entertainment* unterstrichen.

In sogenannten *Stealth*-Computerspielen[4], in denen der Spieler durch Schleichen und das Vermeiden von direkten Konfrontationen eine möglichst heimliche Vorgehensweise anstrebt, wird viel Wert auf das Schaffen einer realistischen Atmosphäre gelegt. Dies kann durch die *KI* verstärkt werden, indem die computergesteuerten Gegner auf Sinneseindrücke in ihrer Umgebung reagieren. Zu Spielen, die wegweisend für diese Technik waren, gehören *GoldenEye 007* (1997) von *Rare*, *Thief: The Dark Project* (1998) von *Looking Glass Studios* und *Metal Gear Solid* (1998) von *Konami*.

[1] In der künstlichen Intelligenz gilt ein *Agent* als etwas, das seine Umgebung mit Sensoren wahrnehmen und diese mit Effektoren beeinflussen kann (vgl. [29, Kap. 2]). In Spielen bezeichnet man damit meist einen computergesteuerte Figur, wie z. B. einen Gegner.

[2] *Ego-Shooter*, auch *First-Person-Shooter* genannt, bezeichnen Spiele, in denen man aus der Sicht der Spielfigur dreidimensionale Welten erkundet, Rätsel löst und, meist mit Feuerwaffen, gegen Gegner kämpft, die vom Computer oder anderen Menschen gesteuert werden.

[3] *Half-Life* ist 1998 erschienen, dessen Nachfolger *Half-Life 2* im Jahr 2004.

[4] Der Begriff *Stealth* lässt sich mit *Heimlichkeit* übersetzen.

Eine weitere Gattung von Spielen, in denen eine funktionierende *KI* eine wichtige Rolle spielt, sind Strategiespiele, die sich grob in zwei Richtungen einteilen lassen: Echtzeitstrategiespiele[5]und rundenbasierte Strategiespiele[6]. Das *RTS Warcraft: Orcs & Humans* (1994) von *Blizzard Entertainment* wird für seine funktionierende Wegfindungstechnologie (vgl. Abschnitt 2.2) gelobt, dessen Nachfolger *Warcraft II: Tides of Darkness* (1995) (vom gleichen Entwickler) für die allgemein hohe Kompetenz der künstlichen Intelligenz. Diese wird auch bei *Age of Empires II: The Age of Kings* (1999) von *Ensemble Studios* hervorgehoben. Bei rundenbasierten Strategiespielen wird die *Civilization* Reihe[7] häufig als positives Beispiel genannt.

Modell der *KI* in Spielen

Trotz der Unterschiede in der Funktionsweise der künstlichen Intelligenz in den oben genannten Spielen, lassen sich unschwer Gemeinsamkeiten finden und als Modell generalisieren. Ein Beispiel dafür wird in [20, Abschnitt 1.2] angeführt – dieses wird in Abbildung 1.1 illustriert. Es handelt sich hierbei um ein hierarchisches Schema, wobei die oberen Schichten auf den unteren aufbauen. Ganz unten steht hierbei die *Fortbewegung*. Unter diesen Begriff fallen sämtliche Algorithmen, die einen Agenten von seinem momentanen Aufenthaltsort zu einem bestimmten Ziel bringen sollen – dieses wird von der nächsthöheren Schicht vorgegeben: der *Entscheidungsfindung*. Diese Ebene ist dafür zuständig, für eine bestimmte Situation die beste Verhaltensweise (z. B. Angreifen, Verstecken, Patroullieren, Fliehen) zu finden. Die oberste Schicht, die *Strategie*, beinhaltet Algorithmen, die auf mehrere Agenten Einfluss nehmen, und dadurch zu strategischem Gruppenverhalten führen.

Im letzten Abschnitt wurden Beispiele für Spiele mit herausstechender künstlicher Intelligenz genannt. Dabei wurden meist Eigenschaften und Implementierungen hervorgehoben, die den oberen zwei Schichten des Modells zuordenbar sind. Die Vorraussetzung für die Effektivität der Entscheidungsfindung und Strategie ist die Funktionsfähigkeit der Algorithmen zur Fortbewegung, zu denen auch Wegfindung und Kollisionsvermeidung gehören – eine Gruppe von Agenten, die eine ausgereifte Strategie hat, diese aber nicht

[5]Bei Echtzeitstrategiespielen, bzw. auf Englisch *real-time strategy games* oder kurz *RTS*, muss der Spieler meist Armeen kommandieren, um seine Feinde zu zerstören. Häufig müssen die Streitkräfte auch gebaut werden, was das Sammeln von Ressourcen und Bauen von Gebäuden nötig macht. Diese Aufgaben müssen alle gleichzeitig in Echtzeit ausgeführt werden, was den Namen des Genres erklärt.

[6]Rundenbasierte Strategiespiele, auf Englisch *turn-based strategy games* oder kurz *TBS*, laufen im Gegensatz zu *real-time strategy games* nicht in Echtzeit ab und erlauben dem Spieler unbeschränkt viel Zeit für das Ausführen seiner Aktionen. Spiele, die diesem Genre zuordenbar sind, haben nicht immer Krieg zum Motiv, wie ein Großteil der Echtzeitstrategiespiele. So gibt es z. B. eine Vielzahl an *TBS*, in denen man eine Firma verwalten oder eine ganze Zivilisation aufbauen muss.

[7]Die ersten zwei Titel wurden 1991 und 1996 von *MicroProse* entwickelt, deren drei Fortsetzungen 2001, 2005 und 2010 von *Firaxis*.

Abbildung 1.1: Modell der *KI* in Computerspielen gemäß [20, Kap. 1.2].

realisieren kann, weil sie ständig mit Hindernissen kollidiert, vermittelt keinesfalls den Eindruck von Intelligenz.

1.1.2 Kollisionsvermeidung

Das Ziel von Verfahren zur Kollisionsvermeidung ist das Verhindern von Zusammenstößen zwischen sich bewegenden Objekten. Dies ist nicht nur bei Computerspielen, in denen Kollisionen von computergesteuerten Charakteren den Spielfluss stören und als *dumm* wahrgenommen werden würden, unerlässlich. Auch im Bereich der Computeranimation und in der Robotik spielt die Kollisionsvermeidung eine wichtige Rolle (siehe Abschnitt 2.1). So gibt es in jedem Gebiet Verfahren, die für diese Aufgabe zuständig sind.

Hierbei existiert eine Technik, die sich im Spielebereich durchgesetzt hat: Auf einem Beitrag von Craig W. Reynolds basierend (s. [28]), in dem verschiedene Verhaltensweisen für selbstständige (bzw. *autonome*) Agenten beschrieben werden, analysiert in diesem Ansatz jede Einheit für sich selbst, ob sie Hindernisse vor sich hat, wählt gegebenenfalls das *gefährlichste* Objekt aus und weicht diesem aus, indem sie ihren Kurs verändert (siehe Abschnitt 2.3.3).

Im Jahr 2010 veröffentlichte das Unternehmen *Gas Powered Games* ein Werbevideo (siehe [33]) für das Echtzeitstrategiespiel *Supreme Commander 2*. In diesem wird ein alternativer Ansatz zur Kollisionsvermeidung, genannt *Flowfields* (bzw. *Flow Fields*), vorgestellt. Das präsentierte Konzept ist bereits aus dem Gebiet der Robotik bekannt, wo es genutzt wird um einem mobilen Roboter die Fähigkeit zu geben Hindernissen auszuweichen. Bei diesem Ansatz wird ein Vektorfeld erstellt, in welches alle statischen und dynami-

schen Objekte als Repulsionen einberechnet werden (siehe Abschnitt 2.4.3). Dies ermöglicht es auf alle möglichen Kollisionen gleichzeitig zu reagieren, wohingegen sich beim Ansatz von Reynolds widersprüchliche Situationen ergeben können (vgl. Abschnitt 2.5).

Das vielversprechende *Flow Field*-Konzept hat bis jetzt noch wenig Einfluss auf das Gebiet der künstlichen Intelligenz in Spielen. Das Analysieren des Ansatzes sowie ein Vergleich mit dem Verfahren von Reynolds stellen die Motivation hinter dieser Arbeit dar.

1.2 Zielsetzung der Arbeit

Das Ziel dieser Diplomarbeit ist die Gegenüberstellung von zwei Verfahren zur Kollisionsvermeidung in Spielen. So soll herausgefunden werden, in welchen Situationen die Verwendung von *Flow Fields* zur Vermeidung von Hindernissen intelligenter wirkt als der Ansatz von Reynolds, welcher als Industriestandard im Computerspielbereich gilt.

Zu diesem Zweck wird neben den theoretischen Hintergründen beider Ansätze eine mögliche Umsetzung sowie die Integration im Computerspiel *Delta Strike*, welches im Anhang A vorgestellt wird, beschrieben. Mithilfe der Implementierung im Spiel wurde eine empirische Studie unter Zuhilfenahme von Methoden der *Usability* durchgeführt, um die Stärken und Schwächen der zwei Verfahren zur Kollisionsvermeidung herauszufinden.

1.3 Aufbau der Arbeit

Nachdem in Kapitel 1 die Motivation für die wissenschaftliche Bearbeitung des Themas sowie die Zielsetzung dieser Arbeit beschrieben wurden, wird ein theoretischer Überblick über das Gebiet der Kollisionsvermeidung gegeben. Dies geschieht in Kapitel 2, wobei potenzielle Einsatzgebiete sowie eine Abgrenzung der Problemstellung die Einleitung bilden. Daraufhin werden die zwei Verfahren zur Kollisionsvermeidung, welche für diese Arbeit relevant sind, beschrieben. Hierbei wird sowohl auf deren Funktionsweise als auch deren Hintergründe und Probleme eingegangen. Das Kapitel wird mit einer Gegenüberstellung der Ansätze abgeschlossen.

Das dritte Kapitel der Arbeit widmet sich der Beschreibung der Implementierung der beiden Verfahren, welche in einer seperaten Testumgebung erfolgte. Nach der Schilderung der Überlegungen zur technischen Realisierung, die in den ersten zwei Abschnitten des Kapitels erfolgt, wird die Umsetzung mithilfe von Quelltexten detailliert erläutert. Abschließend wird auf die Integration der Verfahren zur Kollisionsvermeidung in das Spiel *Delta Strike* eingegangen.

Kapitel 4 behandelt die Evaluation der zwei Verfahren zur Kollisionsvermeidung, welche mithilfe einer Studie erfolgte. Nachdem die Überlegungen

zu dieser beschrieben wurden, wird auf die Durchführung und die Ergebnisse eingegangen.

Den Abschluss bildet das fünfte Kapitel, in dem über die Arbeit reflektiert wird. So werden die Erkenntnisse, welche durch die Implementierung und Evaluation gewonnen wurden, zusammengefasst und eine mögliche Fortsetzung der Forschung in diesem Bereich beschrieben.

Kapitel 2

Kollisionsvermeidung

2.1 Überblick

> Collision avoidance is the collision-free movement of two or more objects.

Dieses trivial anmutende Zitat aus [12, Kap. 2] beschreibt die Aufgabe von Verfahren zur Kollisionsvermeidung – die man selbst keinesfalls als trivial bezeichnen kann. Das Problem von Agenten mit der Fähigkeit anderen Objekten, die sowohl statische als auch dynamische Hindernisse sein können, auszuweichen, beschäftigt verschiedene Forschungsgebiete, wobei die entwickelten Ansätze häufig gebietsübergreifend eingesetzt werden können. Beispielhafte Anwendungsbereiche für Verfahren zur Kollisionsvermeidung, bzw. im Englischen *obstacle avoidance* genannt, sind Computeranimation, Robotik und Computerspiele.

2.1.1 Computeranimation

Im Bereich der Computeranimation, in dem mittels Computergrafik Animationen geschaffen werden, finden Verfahren aus der künstlichen Intelligenz zur Simulation von Gruppen Verwendung (vgl. [23, Abschnitt 4.5]), wie beispielsweise das in [27] vorgestellte Schwarmverhalten, welches als *flocking* bezeichnet wird. Bei diesem Ansatz besitzt jedes Mitglied des Schwarms drei Verhaltensmuster, die sich an den in der Nähe befindlichen Nachbarn orientieren und durch Kombination zum realistischen Gesamtverhalten der Herde führen sollen (s. [27, *Simulated Flocks*]):

1. **Collision Avoidance**: Das Mitglied des Schwarms steuert von zusammendrängenden Rudelmitgliedern in der Nähe weg.
2. **Velocity Matching**: Das Erreichen der Geschwindigkeit (und Ausrichtung) der Rudelmitglieder in der Nähe wird angestrebt.
3. **Flock Centering**: Das Mitglied des Schwarms steuert auf den Mittelpunkt der Rudelmitglieder in der Nähe zu.

Während die angeführten Verhaltensmuster Kollisionen zwischen den Mitgliedern des Schwarms verhindern, besteht noch immer das Problem von Hindernissen in der Umgebung. Als mögliche Lösungen hierfür werden in [27, *Avoiding Environmental Obstacles*] *Flow Fields* (s. Abschnitt 2.4) sowie ein *Steer-to-avoid* Verhalten erwähnt, bei dem ein Mitglied des Schwarms nur Objekte vor sich in Betracht zieht und diesen unter Berücksichtigung der Silhouette der Hindernisse ausweicht.

Mehr zum Thema *flocking* findet man u. a. in [8, Abschn. 4.1], eine mögliche Implementierung in [8, Abschnitt 4.2].

2.1.2 Robotik

Das Gebiet der Robotik ist sehr vielseitig – das Forschungsfeld reicht von stationären Industrierobotern, die in einer angepassten Arbeitsumgebung stehen und repetitive, handwerkliche Arbeiten erledigen, bis hin zu mobilen Robotern, die in dynamischen Umgebungen Aufgaben erfüllen und dabei auf unvorhersehbare Ereignisse reagieren können müssen (vgl. [21, Abschnitt 1.7]). Bei letzterem spielt die Vermeidung von Kollisionen eine wichtige Rolle.

Für die Kollisionsvermeidung gibt es, je nach Architektur des Robotersystems, verschiedene mögliche Lösungen. Bei rein reaktiven Systemen, die ohne Planung auskommen, stellt der *Flow Field* Ansatz eine solche dar. Dieses Verfahren basiert auf einem Vektorfeld, welches Ausrichtung und Geschwindigkeit eines mobilen Roboters, der sich in diesem bewegt, vorgibt. Dabei setzt sich das Feld aus Repulsionen von Hindernissen, die vermieden werden sollen, und anziehenden Kräften von Positionen, die erreicht werden sollen, zusammen. Auf die Hintergründe und die Funktionsweise des *Flow Field* Ansatzes wird in Abschnitt 2.4 detailliert eingegangen.

2.1.3 Computerspiele

Wie in Abschnitt 1.1 bereits angegeben, spielen Verfahren zur Kollisionsvermeidung eine wichtige Rolle in der künstlichen Intelligenz in Computerspielen – unabhängig vom Genre. Die für die Bewegung von Einheiten zuständigen Techniken, die auch für das Verhindern von Kollisionen verantwortlich sind, gehören in vielen Modellen der künstlichen Intelligenz in Spielen zur Basis, auf welcher die höheren Schichten, die für komplexere Aufgaben wie beispielsweise taktische Entscheidungen veranwortlich sind, aufbauen (so auch im Modell, welches in Abbildung 1.1 dargestellt wird).

Fehler in der Kollisionsvermeidung könnten schwere Folgen haben, denn Probleme bei einer derart grundlegenden Aufgabe der *KI* sind so offensichtlich, dass der Spieler schnell den Eindruck von nicht intelligentem Verhalten bekommen und frustriert werden kann, was sich maßgeblich auf den Erfolg des betroffenen Spiels auswirken könnte. In Computerspielen, in denen der Spieler seine Einheiten befehligt (wie beispielsweise in Echtzeitstrategiespie-

len), müssen Agenten zuverlässig und möglichst schnell ihre Zielpositionen erreichen – wenn diese kollidieren (bzw. an Hindernissen oder anderen Einheiten hängen bleiben) wird dies nicht gewährleistet. Bei Spielen wie *StarCraft II: Wings of Liberty* (2010) von *Blizzard Entertainment*, um das weltweit Turniere und Ligen ausgetragen werden, bei denen um hohe Geldbeträge gespielt wird, könnten solche Fehler der *KI* zwischen Sieg und Niederlage entscheiden.

Im Computerspielebereich gilt der Ansatz von Craig W. Reynolds, welcher in [28] veröffentlicht wurde, als Industriestandard (vgl. [30, S. 45]) – Auf diesen wird in Abschnitt 2.3 detailliert eingegangen.

2.2 Abgrenzung der Problemstellung

Es ist zwingend notwendig die Problemstellung, vor der die Kollisionsvermeidung steht, abzugrenzen – ein häufiger Fehler ist das Gleichsetzen dieser mit Techniken zur Wegfindung (bzw. im Englischen *Pathfinding*). In [28, *Introduction*] wird folgendes Beispiel angeführt, das die Unterscheidung erleichtern soll: Wegfindungsverfahren liefern eine Beschreibung welche Straßen zum Ziel genommen werden müssen, während das Entlangfahren dieser eine Aufgabe ist, für die Bewegungsalgorithmen (zu denen auch Kollisionsvermeidung gehört) verantwortlich sind.

Agenten in Computerspielen müssen häufig durch ihre Umgebung navigieren: Beispielweise können das Wachen in einem *Ego-Shooter* sein, die einen bestimmten Bereich patroullieren, oder Truppen in einem Echtzeitstrategiespiel, die zum Stützpunkt des Gegners geschickt werden. Wie in [20, S. 203] beschrieben, sollten die dabei gewählten Routen vernünftig und möglichst kurz sein, alles andere würde ineffizient sein und nicht intelligent wirken. Für das Finden solcher Wege sind *Pathfinding*-Algorithmen bestimmt.

Der in der Spielebranche dominierende (s. [20, S. 204]) Ansatz zur Wegfindung ist der sogenannte *A*-Algorithmus* (bzw. *A Stern* oder im Englischen *A star*), der auf dem *Dijkstra-Algorithmus* von Edsger Dijkstra basiert. Bei beiden handelt es sich um Algorithmen, die innerhalb eines Graphen[1] den kürzesten Weg finden, wobei der *A*-Algorithmus* dies durch den Einsatz einer Heuristik, die Wissen über das Ziel verwertet, schneller erledigt (vgl. [26, S. 3]).

Die zwei erwähnten Algorithmen können nicht direkt auf der Spielewelt angewendet werden – dafür muss diese vereinfacht und als Graph repräsentiert werden. Diese Konvertierung kann sowohl automatisch als auch manuell

[1] Ein Graph beschreibt in diesem Anwendungsfall eine Menge von Knoten (die Positionen in der Umgebung des Spiels darstellen), die durch Kanten verbunden sind. Diese sind mit Werten versehen, welche man *Kosten* nennt, die Auskunft über den Aufwand geben, der benötigt wird um von einem Knoten zu einem anderen zu gelangen (beispielsweise Distanz oder Beschaffenheit des Terrains dazwischen). Mehr dazu findet sich in [18].

durchgeführt werden. In [20, Abschn. 4.4] und [20, Abschn. 11.1] werden einige Möglichkeiten genannt, wie dies bewerkstelligt werden kann.

Liefert der angewandte Wegfindungsalgorithmus einen Weg, so kann ein Agent diesen nehmen, ohne mit statischem Terrain zu kollidieren (vorausgesetzt der Graph wurde korrekt generiert). Gibt es auf dieser Route allerdings dynamische Hindernisse (beispielsweise verschiebbare Kisten oder sich bewegende Einheiten) kommen Verfahren zur Kollisionsvermeidung zum Einsatz.

Zusammenfassend kann man sagen, dass Algorithmen zur Wegfindung Planungsprobleme lösen: Man durchsucht einen Graphen, der eine statische Umgebung repräsentiert, nach dem kürzesten Weg. Die in dieser Arbeit präsentierten Ansätze zur Kollisionsvermeidung sind hingegen rein *reaktiv* – der Agent plant nichts im Voraus, sondern reagiert nur auf die eingehenden Reize, die je nach Ansatz unterschiedlich sind.

2.3 Kollisionsvermeidung nach Reynolds

2.3.1 Geschichtliches

Im Jahr 1987 erschien ein Paper ([27]) von Craig W. Reynolds, in dem auf das Schwarmverhalten von Tieren und dessen Nachbau für Zwecke der Computeranimation eingegangen wurde (siehe auch Abschnitt 2.1.1). Wie in [30, S. 45] erwähnt, erweiterte Reynolds seine Forschung von Agenten, die ohne Planung auskommen, durch den 1999 erschienen Beitrag *Steering Behaviors for Autonomous Characters* ([28]). Die meisten Techniken, die in dieser Publikation vorgestellt wurden, können direkt im Computerspielbereich eingesetzt werden (vgl. [9, S. 85]), so auch das Verfahren zur Kollisionsvermeidung. Aufgrund der Relevanz für diese Diplomarbeit wird nachfolgend ein Überblick über die Arbeit von Reynolds gegeben.

2.3.2 Hintergrund

Die Arbeit von Craig W. Reynolds (s. [28]) baut auf der Grundidee von *selbstständigen Charakteren* auf. Mit diesem Ausdruck bezeichnet er Agenten, die in der Lage sind, sich ohne Planung auf möglichst realistische Art und Weise durch ihr Umfeld zu bewegen. Dabei basiert die Bewegung der Charaktere auf einer Reihe an Verhaltensweisen (den *steering behaviors*), die miteinander kombiniert werden.

Ein Gegner in einem *Ego-Shooter* ist ein gutes Beispiel um dieses Zusammenspiel zu illustrieren: Dieser soll die Figur des Spielers jagen, während er anderen Einheiten sowie Hindernissen in der Umgebung ausweicht. Sein Gesamtverhalten ist somit eine Mischung aus einem Verhalten zur *Verfolgung* und einem zur *Kollisionsvermeidung*.

Der Ansatz von Reynolds setzt sich zur Aufgabe den Einsatz der *steering behaviors* unabhängig von der gewählten Art der Fortbewegung zu ermögli-

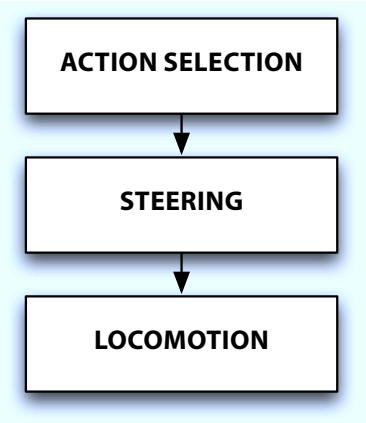


Abbildung 2.1: Modell gemäß [28].

chen. Dies geschieht durch eine Trennung von Lokomotion und Steuerung, die Reynolds in [28] mithilfe eines dreischichtigen Modells beschreibt, welches in Abbildung 2.1 dargestellt wird. Die oberste Schicht (*Action Selection*) beschreibt hierbei Strategie, Enscheidungsfindung und Planung – Herausforderungen, die außerhalb des Rahmens dieser Diplomarbeit (und der Arbeit von Reynolds) liegen. Die zweite Ebene, die für die Festlegung von Richtung und Geschwindigkeit verantwortlich ist, wird als *Steering* bezeichnet – Diese wird mit den oben erwähnten *steering behaviors* implementiert und stellt den Fokus der Forschung von Reynolds dar. Auf der untersten Schicht liegt *Locomotion*, welche die Ausführung der Kontrollsignale, die von der *Steering*-Ebene erhalten wurden, übernimmt.

In [28] werden die Begriffe *Action Selection*, *Steering* und *Locomotion* mit einem Beispiel erklärt, welches für das Verständnis hilfreich ist:

> Consider, for example, some cowboys tending a herd of cattle out on the range. A cow wanders away from the herd. The trail boss tells a cowboy to fetch the stray. The cowboy says "giddy-up" to his horse and guides it to the cow, possibly avoiding obstacles along the way. In this example, the trail boss represents *action selection*: noticing that the state of the world has changed (a cow left the herd) and setting a goal (retrieve the stray). The *steering* level is represented by the cowboy, who decomposes the goal into a series of simple subgoals (approach the cow, avoid obstacles, retrieve the cow). A subgoal corresponds to a steering behavior for the cowboy-and-horse team. Using various control signals (vocal commands, spurs, reins) the cowboy steers his horse towards the target. [...] The horse implements the *locomotion* level. Tak-

> ing the cowboy's control signals as input, the horse moves in the indicated direction.

Bei der Lokomotion gibt sich Reynolds mit einem simplen Fahrzeugmodell[2] zufrieden, welches auch in dieser Diplomarbeit verwendet und noch genauer beschrieben wird (s. Abschnitt 3.1.3). Bei diesem konzentriert sich die Masse des Objekts auf einem einzelnen Punkt. Weiters hat jedes Gefährt einen Vektor für die Geschwindigkeit.

Pro Zeitschritt wird aus der Division der *steering forces* (die Kräfte, die ausgehend von den eingesetzten Verhaltensweisen wirken) durch die Masse des Fahrzeugs die Beschleunigung errechnet, die zur letzten Geschwindigkeit addiert wird, welche gegebenenfalls durch festgelegte Maximalwerte begrenzt wird. Dabei gibt jedes angewendete *steering behavior* die gewünschte Kraft, die für die Realisierung des Verhaltens erforderlich ist, an das Fahrzeug. Bei mehreren werden diese *steering forces* meist zu einem Vektor summiert, wobei diese auch gewichtet werden können.

Im oben genannten Beispiel des verfolgenden Gegners könnte sich die Gesamtkraft, welche pro Zeitschritt auf diesen wirkt, durch die Summierung der benötigten Kräfte zum Verfolgen des Spielers und Ausweichen vor Hindernissen berechnen lassen.

Im Folgenden werden Beispiele für die in [28] vorgestellten *steering behaviors* genannt und kurz erklärt:

***Seek*:** Diese Verhaltensweise soll den Charakter zu einer angegeben Position steuern, indem mithilfe der gewünschten Geschwindigkeit (Vektor zwischen Agent und Ziel) und der aktuellen Geschwindigkeit die gewünschte Kraft errechnet wird.

***Arrival*:** Wird das *seek*-Verhalten angewendet um ein Ziel zu erreichen, so fährt der Charakter irgendwann daran vorbei und wendet, um es wieder anzusteuern. Reynolds vergleicht dies in [28] mit einer Motte, die um eine Glühbirne herumschwirrt. Dies soll beim *arrival*-Verhalten verhindert werden, indem das Fahrzeug ab einer bestimmten Distanz zum Ziel (die als Parameter übergeben wird) anfängt langsamer zu werden und an der gewünschten Position zum Halt kommt. Ansonsten ist dieses *steering behavior* identisch zu *seek* und verhält sich außerhalb der Bremsdistanz genau gleich.

***Flee*:** Dieses Verhalten ist zum Fliehen von einer angegebenen Position bestimmt und soll somit das Gegenteil des *seek*-Verhaltens liefern. Dies kann bewerkstelligt werden, indem es analog zu diesem implementiert wird, die gewünschte Geschwindigkeit allerdings negiert wird und somit weg vom *Ziel* zeigt.

[2] Craig W. Reynolds verwendet *vehicle*, auf Deutsch *Fahrzeug*, als Begriff unter den alle Arten der Fortbewegung fallen – ob es nun Panzer, Pferde oder die eigenen Beine sind.

***Pursuit*:** Während das *seek*-Verhalten für das Ansteuern von statischen Positionen bestimmt ist, sollen bei *pursuit* dynamische Ziele verfolgt werden. Dies wird anhand einer simplen Vorhersage umgesetzt: Basierend auf der Annahme, dass die verfolgte Einheit ihre Bewegungsrichtung und Geschwindigkeit nicht ändern wird, wird für einen Zeitpunkt in der (sehr nahen) Zukunft ein Punkt berechnet, an dem sich das dynamische Ziel voraussichtlich befinden wird. Diese Position wird dann analog zum *seek*-Verhalten angesteuert.

Ein weiteres *steering behavior* von Reynolds ist das zur Kollisionsvermeidung. Dieses wird meistens in Verbindung mit einem der oberen Verhalten angewendet. In Abschnitt 2.3.3 wird die genaue Funktionsweise veranschaulicht.

2.3.3 Funktionsweise

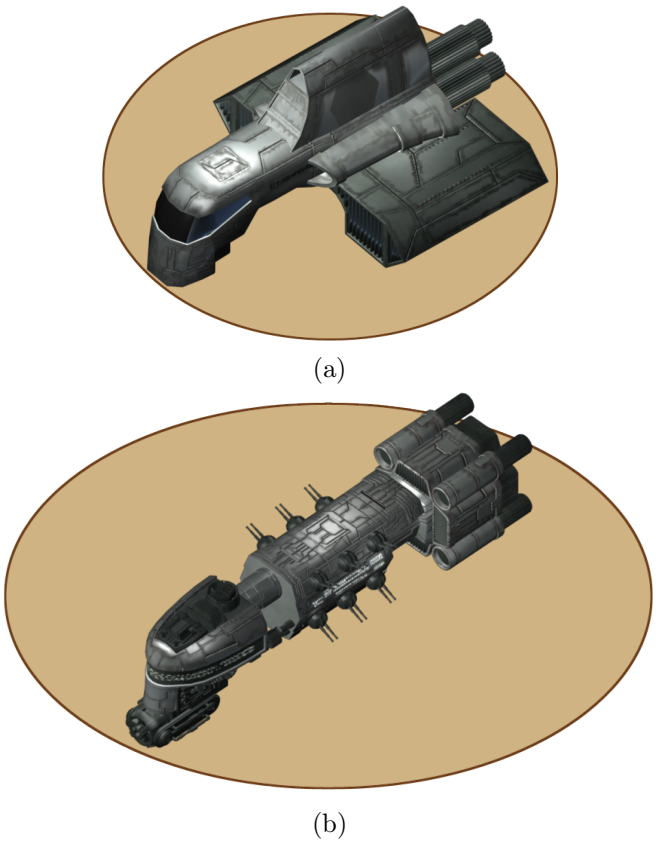
Wie in diesem Kapitel bereits beschrieben, sollen Verfahren zur Kollisionsvermeidung einem Agenten die Fähigkeit geben, durch Umgebungen mit statischen und dynamischen Hindernissen zu navigieren ohne dabei zu kollidieren. Das *obstacle avoidance behavior*, welches von Reynolds in [28] beschrieben wurde, setzt sich das Lösen dieser Problematik zum Ziel.

Es ist wichtig eine Unterscheidung zum *flee*-Verhalten, welches in Abschnitt 2.3.2 vorgestellt wurde, zu treffen. Während dieses einen Agenten von einer Position wegnavigiert, unabhängig davon, wo sich diese befindet, reagiert das *obstacle avoidance behavior* lediglich auf Hindernisse, die vor dem Charakter liegen. Dies wird in [28] am Beispiel eines Autos erklärt, welches an einer Wand parallel entlangfährt. In dieser Situation würde das *flee*-Verhalten das Fahrzeug sofort von dem Hindernis weg lenken, während das *obstacle avoidance behavior* den Kurs halten würde.

In [9, S. 99] wird die Grundidee des Verfahrens von Reynolds folgendermaßen erklärt:

> *Obstacle avoidance* is a behavior that steers a vehicle to avoid obstacles lying in its path. [...] This is achieved by steering the vehicle so as to keep a rectangular area – a detection box, extending forward from the vehicle – free of collisions. The detection box's width is equal to the bounding radius of the vehicle, and its length is proportional to the vehicle's current speed – the faster it goes, the longer the detection box.

Diese sich vom Agenten aus aufspannende *detection box*, oder im dreidimensionalem Raum ein Zylinder (s. [28]), überprüft bei jedem Zeitschritt, ob eine Kollision mit einem Objekt auftritt. Dabei setzt der Ansatz von Reynolds voraus, dass Hindernisse als Kreise, bzw. in 3D als Kugeln, angenähert werden können (vgl. Abbildung 2.2). Diese Voraussetzung hat einen Nachteil:



(a)

(b)

Abbildung 2.2: Annäherung als Kreis am Beispiel von zwei Raumschiffen aus *Delta Strike Origins* (siehe Anhang A). Zufriedenstellende Approximation (a), schlechte Annäherung aufgrund der länglichen Form (b).

Bei vielen geometrischen Körpern ist eine Approximation durch einen Kreis sehr ungenau, beispielsweise bei länglichen Formen, wie in Abbildung 2.2 (b) gezeigt wird. Diese Schwachstelle wird auch in [20, S. 94] beschrieben:

> The collision avoidance behavior assumes that targets are spherical. [...] More complex obstacles cannot be easily represented in this way. The bounding sphere of a large object, such as a staircase, can fill a room. We certainly don't want characters sticking to the outside of the room just to avoid a staircase in the corner.

Allerdings ermöglicht diese Annäherung ein schnelles Überprüfen, ob eine Kollision zwischen der *detection box* und einem Hindernis auftritt. So wird in jedem Zeitschritt jedes Objekt abgefragt. Der vom Agenten aus gesehene Mittelpunkt des runden Hindernisses wird auf die zur Ausrichtung normale

Ebene projiziert. Die Distanz zwischen diesem Punkt und der Position des Charakters wird mit der Summe der Radien der zwei Objekte verglichen, wodurch potenzielle Kollisionen schnell ausgeschlossen werden können. Gleiches geschieht bei Hindernissen, die als Ganzes hinter dem Agenten oder vor der *detection box*, bzw. vor dem Zylinder, liegen.

Können auf die beschriebene Art und Weise nicht alle Kollisionen mit Hindernissen in der Umgebung des Agenten ausgeschlossen werden, so muss ein anderer Kurs eingeschlagen werden, um den Zusammenstoß zu verhindern. Wenn es mehrere potenzielle Kollisionen gibt wird das Objekt, das die Vorwärts-Achse[3] am nächsten vor dem Agenten schneidet, als größte Gefahrenquelle bestimmt.

Wurde das *gefährlichste* Hindernis gefunden, muss nun eine Kraft berechnet werden, durch die der Charakter ausweicht. Während es sich gemäß Reynolds um eine reine Querkraft handelt, die anhand der Projektion des Mittelpunkts des Hindernisses auf die zur Ausrichtung des Agenten normale Ebene berechnet wird, kann zusätzlich auch eine Bremskraft verwendet werden, welche den Charakter verlangsamen soll (siehe [9, Kap. 3]). Für ein besseres Verständnis wird die Kollisionsvermeidung nach Reynolds in Abbildung 2.3 dargestellt und erklärt.

Abschließend ist es wichtig anzumerken, dass man bei der Verwendung des *obstacle avoidance behavior* in Verbindung mit zielsuchenden Verhalten (wie z. B. *arrival*) nur Hindernisse zwischen Agent und Ziel beachten muss. Dies wird in [28] mit folgendem Beispiel dargestellt:

> The mountain beyond the airport is ignored by the airplane, but the mountain between the plane and the airport is *very* important.

2.3.4 Probleme

Übersehen von Hindernissen

In [20, Abschnitt 3.3.14] wird ein Problem beschrieben, bei dem es zu einer Kollision zwischen zwei Agenten kommt, da diese sich gegenseitig nicht bemerken. Eine mögliche Situation, in der es zu diesem Fehlverhalten kommen kann, wird in Abbildung 2.4 dargestellt. In der gezeigten Situation übersehen sich die zwei Agenten, da keiner von ihnen den anderen in seiner *detection box* hat. Somit wird kein anderer Kurs eingeschlagen und es kommt zu einem Zusammenstoß.

Eine mögliche Lösung ist der Einsatz von *unaligned collision avoidance*. Dieses Verfahren wird in [28] beschrieben. Dabei zieht ein Agent alle beweglichen Hindernisse in Betracht und überprüft, ob es in Zukunft zu einem

[3] Die Vorwärts-Achse bezeichnet die Achse, an der sich der Charakter vor- und zurückbewegt – sie entspricht somit der Ausrichtung.

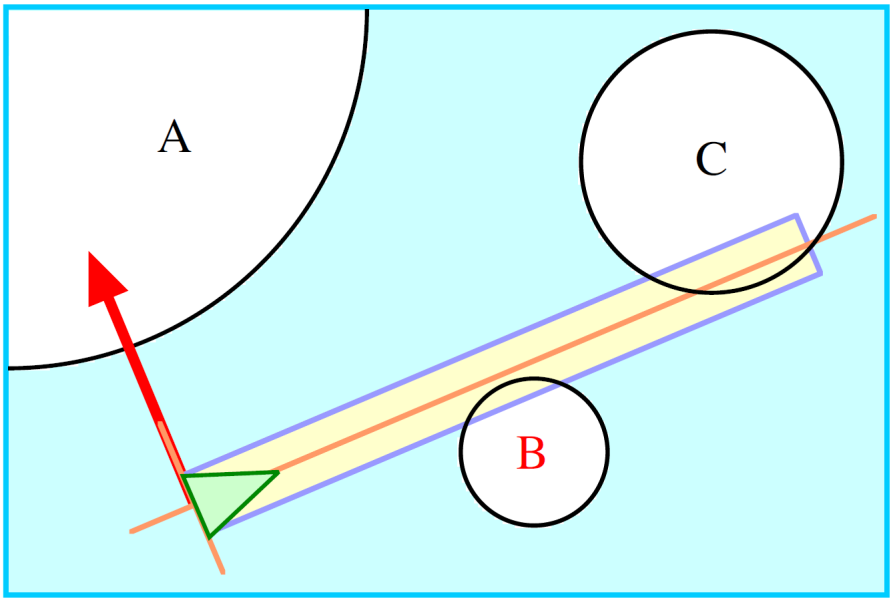


Abbildung 2.3: Darstellung der Kollisionsvermeidung nach Reynolds. Während die Hindernisse B und C die *detection box*, dargestellt als gelbes Rechteck, schneiden, liegt Hindernis A außerhalb dieser und kann somit ignoriert werden. Aufgrund der Nähe von Hindernis B wird diesem ausgewichen, wobei der Agent nach links steuern würde. Aus [28].

Zusammenstoß kommt, wobei angenommen wird, dass sich die Geschwindigkeiten der Objekte nicht ändern. Der Ort der nähesten potenziellen Kollision wird zur Berechnung eines Kraftvektors, welcher zum Vermeiden dieser Position benötigt wird, verwendet. Das Verfahren wird in Abbildung 2.5 skizziert.

Konflikte

Bei der Kollisionsvermeidung nach Reynolds wird immer nur einem Objekt ausgewichen. In Situationen, in welchen sich mehrere – dicht aneinander liegende – Hindernisse vor einem Agenten befinden, kann dies dazu führen, dass der Kraftvektor, welcher zur Vermeidung der *gefährlichsten* Kollision berechnet wird, einen Charakter auf ein anderes Objekt zusteuern lässt (vgl. [1, S. 159]).

Eine typische Situation, in der dieses Problem auftreten kann, wird in Abbildung 2.6 illustriert. Im dargestellten Fall befinden sich zwei Hindernisse, welche eine geringe Distanz zueinander aufweisen, vor einem Agenten. Weicht der Charakter vor einem dieser aus, steuert er auf das jeweils andere Objekt zu. Durch diese Richtungswechsel fängt der Agent an zu zittern, was ein ästhetisches Problem darstellt. Außerdem kann es zu einem Zusammenstoß mit einem Hindernis kommen.

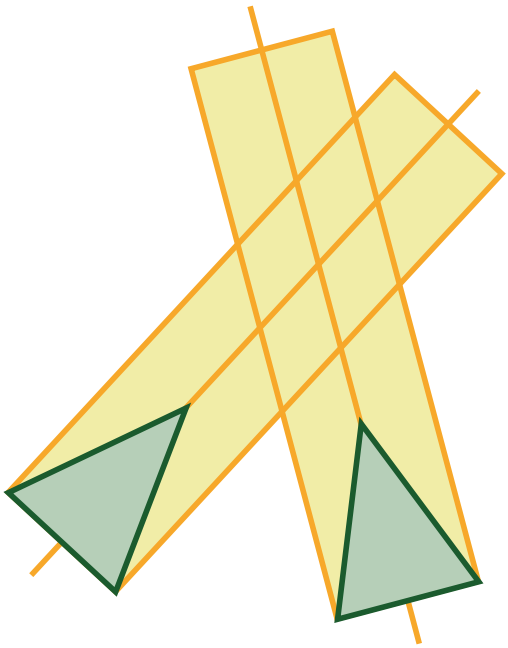


Abbildung 2.4: Zwei Agenten, die einander übersehen, wodurch es zu einer Kollision kommt.

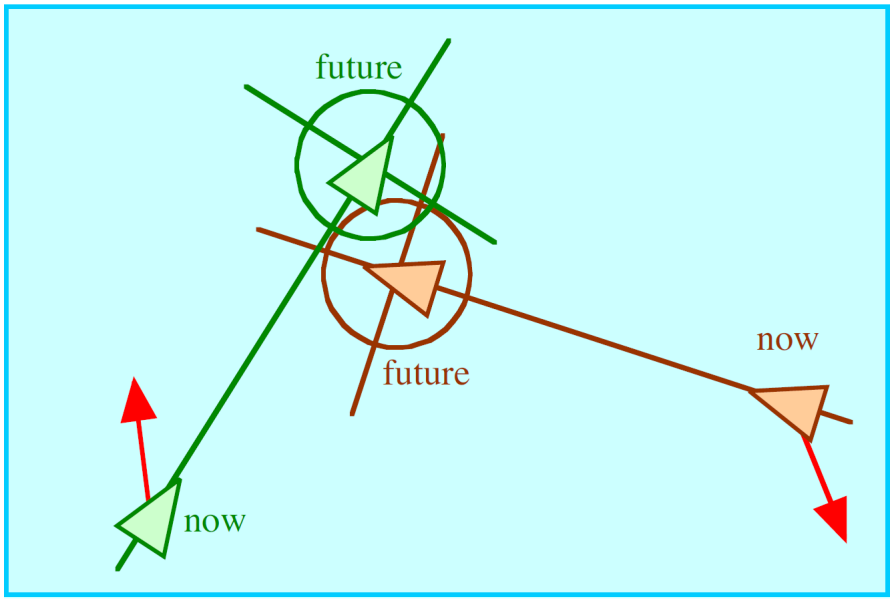


Abbildung 2.5: *Unaligned collision avoidance* gemäß Reynolds. Aus [28].

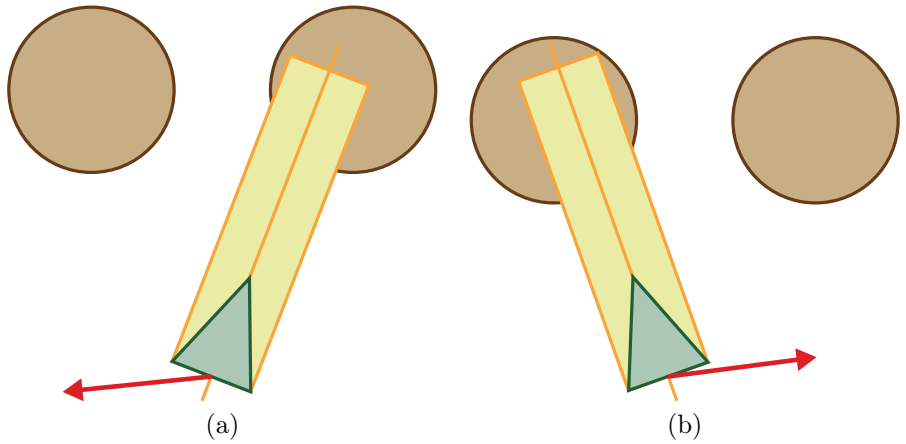


Abbildung 2.6: Konfliktsituation bei der Verwendung des Verfahrens von Reynolds. Ausweichen vor dem rechten Hindernis (a), Ausweichen vor dem linken Hindernis (b).

2.4 *Flow Fields*

2.4.1 Geschichtliches

Das *Flow Fields* Verfahren, auch als *Force Fields*, *Potential Fields* oder *Vector Fields* bezeichnet, hat seinen Ursprung im Forschungsgebiet der Robotik. Als Begründer gelten laut [4, S. 98] Oussama Khatib mit seiner 1985 erschienen Arbeit *Real-Time Obstacle Avoidance for Manipulators and Mobile Robots* und Bruce Krogh mit der Publikation *A Generalized Potential Field Approach to Obstacle Avoidance Control*, die 1984 veröffentlicht wurde. Beide setzen sich das Schaffen von ruckfreien Bahnen für intelligente, mobile Roboter zum Ziel. Dabei steht das Verfahren zur Vermeidung von Kollisionen im Mittelpunkt. So verspricht Khatib in [15] einen neuartigen Ansatz für diese Problemstellung, bei dem keine Planung für das Finden eines kollisionsfreien Pfades benötigt wird.

Durch dieses rein reaktive Verhalten zählt das von Khatib und Krogh vorgestellte *Potential Field* Verfahren als eines der bekanntesten Beispiele für das *Reactive Paradigm* (vgl. [21, Abschnitt 4.2.3]). Dieses ist neben dem *Hierarchical Paradigm* und dem *Hybrid Deliberative/Reactive Paradigm* eines der drei Paradigmen der Robotik und beschreibt eine Kopplung zwischen Wahrgenommenen und Aktionen eines Roboters, die ohne jegliche Planung auskommt. Zum besseren Verständnis von *Flow Fields* wird in Abschnitt 2.4.2 auf die drei Paradigmen eingegangen, wobei der Fokus auf dem *Reactive Paradigm* liegt.

2.4.2 Hintergrund

Paradigmen der Robotik

Wie in [21, S. 5] beschrieben, bezeichnen die Paradigmen die Beziehung zwischen den drei Primitiven der Robotik: *SENSE*, *PLAN* und *ACT*. Diese werden an gleicher Stelle folgendermaßen definiert:

> The functions of a robot can be divided into three very general categories. If a function is taking in information from the robot's sensors and producing an output useful by other functions, then that function falls in the SENSE category. If the function is taking in information (either from sensors or its own knowledge about how the world works) and producing one or more tasks for the robot to perform (go down the hall, turn left, proceed 3 meters and stop), that function is in the PLAN category. Functions which produce output commands to motor actuators fall into ACT (turn 98°, clockwise, with a turning velocity of 0.2mps).

Es haben sich drei Paradigmen in der Robotik etabliert, die in folgender Aufzählung nach ihrem Alter aufgelistet werden:

1. *Hierarchical Paradigm*
2. *Reactive Paradigm*
3. *Hybrid Deliberative/Reactive Paradigm*

Im Folgenden wird ein kurzer Überblick über jedes Paradigma gegeben. Da die Gedanken hinter dem *Reactive Paradigm* den Grundstein für das *Flow Field* Verfahren gelegt haben, wird auf dieses genauer eingegangen.

Hierarchical Paradigm

Das *Hierarchical Paradigm* ist das älteste Paradigma in der Robotik und war von 1967 bis 1990 vorherrschend (s. [21, S. 6]). Es beschreibt einen rein sequentiellen Ablauf: Zuerst wird die Umgebung des Roboters wahrgenommen (*SENSE*). Mithilfe der erhaltenen Informationen wird ein Weg zum Erreichen eines bestimmten Ziels geplant (*PLAN*), der dann ausgeführt wird (*ACT*). Nach jeder solchen Sequenz fängt der Kreislauf wieder von vorne an. Abbildung 2.7 soll diesen Sachverhalt verdeutlichen.

Der Grund für den sequentiellen Aufbau des Paradigmas sind Annahmen über die menschliche Denkweise, welche auf Selbstbeobachtung gestützt sind – dies zeigt folgendes Zitat aus [21, S. 6]:

> Under it [the *Hierarchical Paradigm*, M.K.], the robot operates in a top-down fashion, heavy on planning [...]. This was based on an introspective view of how people think. "I see a door, I decide to head toward it, and I plot a course around the chairs."

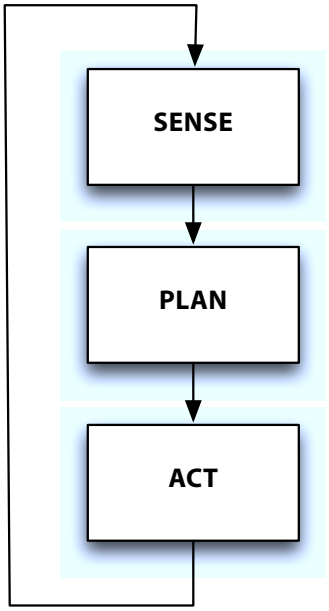


Abbildung 2.7: Sequentieller Ablauf gemäß dem *Hierarchical Paradigm*.

In [21, Kap. 2] wird ein großes Problem des *Hierarchical Paradigm* beschrieben: Die Informationen aus der Wahrnehmungsphase werden bei diesem Paradigma meist als eine einzige Datenstruktur an die Planungsfunktion übergeben. Das Erstellen eines solchen Modells der Umgebung des Roboters ist eine schwierige Aufgabe, da dieses *world model* alle Informationen beinhalten muss, die zur Bewerkstelligung einer Aufgabe benötigt werden. Dies wird als *closed world assumption* bezeichnet. Es müssen sämtliche Situationen, die in der Umgebung auftreten können, berücksichtigt werden. Somit ist die Fähigkeit des Programmierers/Robotikers, alle Details in einem solchen Modell einzubauen, ausschlaggebend für den Erfolg des Roboters.

Die *closed world assumption* führt zu einem weiteren Problem: Durch die Berücksichtigung aller erforderlichen Faktoren in der Umgebung kann das Modell sehr groß werden, wodurch die Planung der Lösung einer Aufgabe hohe Rechenleistung erfordert. Obwohl dies, wie in [21, S. 44] beschrieben, vor allem in den 60er-Jahren wegen der damals fehlenden Rechenkraft problematisch war, wurde durch Forschung im Bereich der Biologie klar, dass ein hierarchischer Ansatz für das Lösen von Navigationsaufgaben, die meist eine schnelle Reaktionszeit erfordern, allgemein nicht sinnvoll ist. Dies führte zur Entstehung des *Reactive Paradigm*, welches im nächsten Unterabschnitt erklärt wird.

Reactive Paradigm

Wie in [21, S. 8] beschrieben, war das *Reactive Paradigm* von 1988 bis 1992 besonders populär in der Robotik, danach wurde es mehr und mehr von hybriden Ansätzen (wie dem *Hybrid Deliberative/Reactive Paradigm*) abgelöst. Während die Gedanken hinter dem *Hierarchical Paradigm*, wie im letzten Abschnitt beschrieben, auf Introspektion basieren, entstand das *Reactive Paradigm* durch die Hoffnung einiger Robotiker, dass Forschung in Bereichen wie Biologie und Psychologie Nutzen für die Entwicklung von Robotern haben könnte. Im Folgenden werden die hierbei einflussreichsten Gebiete aufgelistet (vgl. [4, Abschnitt 2.1]):

- **Neurowissenschaften** beschäftigen sich mit der Analyse von Nervensystemen.
- **Psychologie** beschreibt die Erforschung von Verstand und Verhalten.
- **Ethologie** bezeichnet die Untersuchung des Verhaltens von Tieren in deren natürlicher Umgebung.

Eine detaillierte Beschreibung der Einsichten, die durch die Forschung in den oben genannten Bereichen gewonnen wurden, würde den Rahmen dieser Arbeit sprengen – außerdem wird diese bereits vielerorts in entsprechender Fachliteratur gegeben, beispielsweise in [4, Kap. 2], [21, Kap. 3] und [2].

Zusammenfassend kann festgestellt werden, dass Forschung in der Tierwelt eine große Inspiration für das Gebiet der Robotik darstellt – vor allem Verhaltensweisen von Tieren hatten einen direkten Einfluss auf die Entwicklung von intelligenten Robotersystemen (vgl. [21, Abschnitt 4.1]). Das folgende Zitat aus [21, S. 99] definiert den Begriff *Verhalten* im Kontext der Robotik:

> A behavior is the fundamental element of biological intelligence, and will serve as the fundamental component of intelligence in most robot systems. A *behavior* is defined as a mapping of sensory inputs to a pattern of motor actions which then are used to achieve a task.

Eine Verhaltensmuster stellt somit eine direkte Verbindung zwischen Wahrnehmung (*sensory inputs*) und Handlung (*motor actions*) dar. Anhand dieser Definition lässt sich der radikale Grundgedanke des *Reactive Paradigm* erklären: Das Paradigma verzichtet zur Gänze auf den Planungsschritt, der im *Hierarchical Paradigm* eine wichtige Rolle spielt. Dabei wird, wie in [4, Abschnitt 3.1.1] beschrieben, auf ein Modell der Umgebung, wie das *world model* im *Hierarchical Paradigma*, verzichtet, wodurch auch die damit verbundenen Probleme wegfallen. Weiters wird durch den Verzicht auf ein solches Modell Berechnungszeit eingespart, was vor allem in dynamischen und gefährlichen Umgebungen, in denen schnelle Reaktionen erforderlich sind, von Vorteil ist.

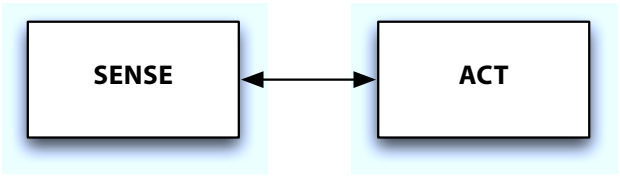


Abbildung 2.8: Darstellung eines *behavior* gemäß dem *Reactive Paradigm*.

Den Grundstein für das *Reactive Paradigm* stellen *behaviors* dar, die als direkte Kopplungen zwischen *SENSE* und *ACT* realisiert sind, wie in Abbildung 2.8 skizziert wird. Die Handlung, die ausgeführt wird, ist eine direkte Reaktion auf die von den Sensoren wahrgenommene Umgebung. Auf den ersten Blick begrenzt diese Vorgehensweise einen Roboter auf nur eine mögliche Verhaltensweise. In [21, S. 8–9] wird gezeigt, dass dem nicht so ist:

> The robot has multiple instances of SENSE-ACT couplings [...]. These couplings are concurrent processes, called behaviors, which take local sensing data and compute the best action to take independently of what the other processes are doing. One behavior can direct the robot to "move forward 5 meters" (ACT on drive motors) to reach a goal (SENSE the goal), while another behavior can say "turn 90°" (ACT on steer motors) to avoid a collision with an object dead ahead (SENSE obstacles). The robot will do a combination of both behaviors, swerving off course temporarily at a 45° angle to avoid the collision. Note that neither behavior directed the robot to ACT with a 45° turn; the final ACT emerged from the combination of the two behaviors.

Die Kombination von mehreren *behaviors* ist der Funktionsweise der *steering behaviors* von Reynolds (vgl. Abschnitt 2.3.2) ähnlich – auf diese Similarität wird in Abschnitt 2.5 näher eingegangen.

Auf Basis der vorgestellten Gedanken haben sich verschiedene Architekturen etabliert, die den Grundsätzen des *Reactive Paradigm* entsprechen. Die zwei bekanntesten sind die *Subsumption Architecture* von Rodney Brooks (s. [21, Abschnitt 4.3]) und die *Flow Fields* Methodologie, deren Funktionsweise in Abschnitt 2.4.3 detailliert erklärt wird.

Hybrid Deliberative/Reactive Paradigm:

Gemäß [21, S. 9] wurde deutlich klar, dass das vollständige Weglassen der Planung eine zu extreme Maßnahme darstellt. Aufgrund der Vorteile des *Reactive Paradigm*, wie der Einsparung von wertvoller Rechenzeit, wurde

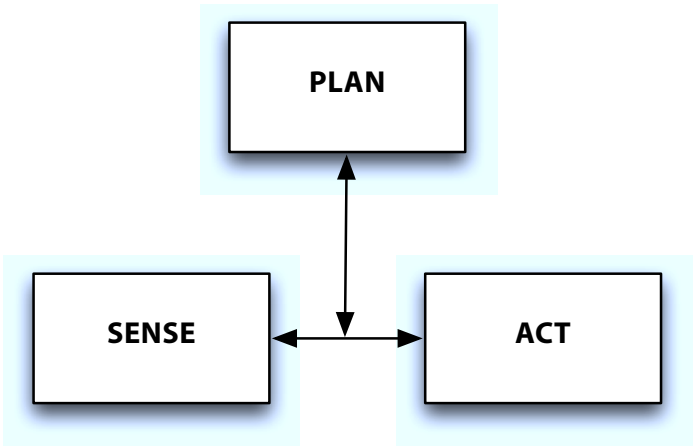


Abbildung 2.9: Organisation der Primitiven der Robotik gemäß dem *Hybrid Deliberative/Reactive Paradigm*.

es als Basis für das *Hybrid Deliberative/Reactive Paradigm*, das in den 90er Jahren aufgekommen ist, verwendet.

Die Organisation der Funktionen *SENSE*, *ACT* und *PLAN* wird in Abbildung 2.9 skizziert und wird im folgenden Zitat aus [21, S. 259] beschrieben:

> The organization of a Hybrid Deliberative/Reactive system can be described as: PLAN, then SENSE-ACT. [...] The PLAN box includes all deliberation and global world modeling, not just task or path planning. The robot would first plan how to accomplish a mission (using a global world model) or a task, then instantiate or turn on a set of behaviors (SENSE-ACT) to execute the plan (or a portion of the plan). The behaviors would execute until the plan was completed, then the planner would generate a new set of behaviors, and so on.

Durch diese Vorgehensweise sollen die Vorteile des *Hierarchical Paradigm* und des *Reactive Paradigm* in einem Paradigma vereint werden. Während bei einer hierarchischen Organisation bei jedem Zyklus ein Planungsschritt vorkommt, was hohe Rechenzeiten verursacht, werden beim *Hybrid Deliberative/Reactive Paradigm* die *behaviors* (als *SENSE-ACT* Paare) und die *PLAN* Funktionen asynchron ausgeführt.

Beispielhafte Architekturen, die dem *Hybrid Deliberative/Reactive Paradigm* zuzuordnen sind, werden u. a. in [4, Kap. 6] und [21, Kap. 7] vorgestellt.

2.4.3 Funktionsweise

Wie in Abschnitt 2.4.2 beschrieben, stellt die *Flow Fields* Methodologie eine der zwei bekanntesten Architekturen innerhalb des *Reactive Paradigm* dar. Die Idee dahinter ist die Realisierung verschiedener Verhaltensweisen durch *Vektorfelder* (vgl. [21, Abschnitt 4.4]). Bevor auf die Kollisionsvermeidung mit *Flow Fields* eingegangen wird, wird das Konzept von Vektorfeldern und deren allgemeine Benutzung innerhalb des *Reactive Paradigm* erklärt.

Definition

Im *Reactive Paradigm* bezeichnet ein Vektorfeld eine Matrix aus Steuerungsvektoren. Die Richtung dieser zeigt an, wohin sich ein Roboter (bzw. ein Agent in einem Computerspiel) bewegen soll, während deren Länge die Stärke der Kraft beschreibt. Wie in [21, Abschnitt 4.4.1] beschrieben, ist das Vektorfeld in den meisten Anwendungsfällen in der Robotik zweidimensional und repräsentiert somit die Umgebung des Roboters aus der Vogelperspektive. Dies macht auch in vielen Computerspiel-Genres Sinn, wie beispielsweise bei Echtzeitstrategiespielen, bei denen das Spielgeschehen meist auf einer Ebene passiert. Durch den Einsatz von dreidimensionalen Vektoren in einem Raum könnte das Konzept problemlos für die Navigation von fliegenden oder tauchenden Robotern (vgl. [4, S. 144]) oder in anderen Computerspiel-Genres eingesetzt werden.

Vektorfelder im *Reactive Paradigm*

Wie bereits erwähnt, werden bei dieser Architektur des *Reactive Paradigm* Vektorfelder dazu benutzt, um verschiedene *behaviors* zu implementieren. Dabei stellen die Kraftvektoren im Feld die – für die Verhaltensweise gewünschte – Richtung und Geschwindigkeit dar. Ein Beispiel dafür wäre ein Verhalten, durch welches ein mobiler Roboter ein Ziel erreicht: Jeder Vektor im Feld zeigt auf die Position, die erreicht werden soll. Die Länge (bzw. Stärke) hängt von der Distanz ab. In der Nähe des Ziels sollte der Roboter abbremsen, was durch ein Verkürzen der dortigen Vektoren bewerkstelligt wird. Ansonsten wird die Maximallänge verwendet. Das Vektorfeld für dieses Verhalten wird in Abbildung 2.10 skizziert.

In vielen Fällen wirken mehr als nur eine Verhaltensweise auf einen mobilen Roboter ein – es muss zum Beispiel ein Ziel erreicht werden, während gleichzeitig Kollisionen vermieden werden. In [4, Abschnitt 4.4.2] wird beschrieben, wie mehrere Verhalten, welche als Vektorfelder realisiert wurden, koordiniert werden. Für eine Position wird durch Summierung der sich dort befindlichen Vektoren aller eingesetzten Verhalten eine Gesamtkraft berechnet. Die Größe dieser wird dann gegebenenfalls anhand von Maximalwerten gekürzt. Ein Beispiel für die Kombination von Feldern wird in Abbildung 2.13 dargestellt.

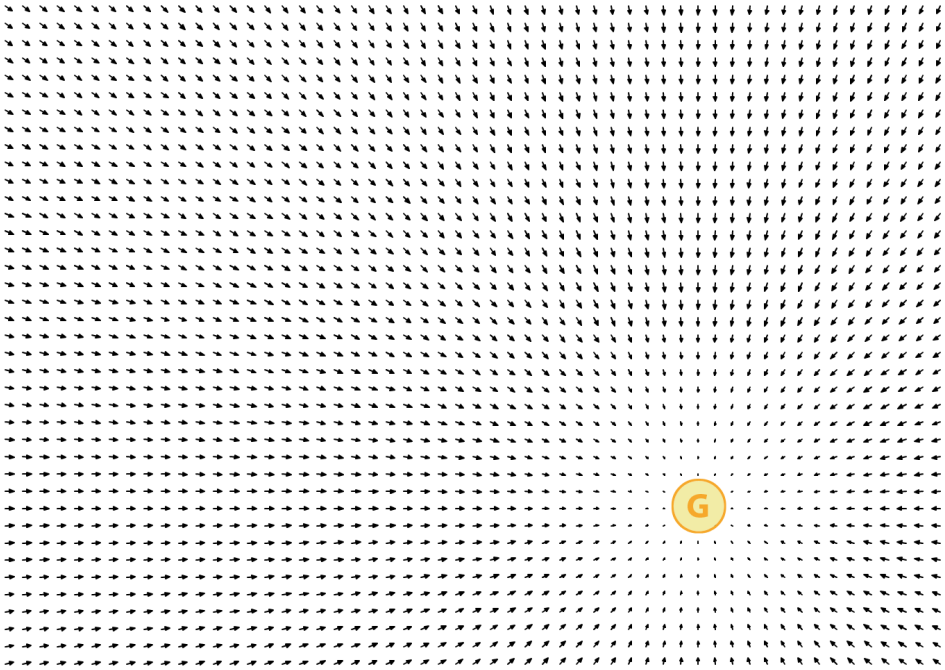


Abbildung 2.10: Vektorfeld für das Erreichen der Zielposition G.

Beispiele für weitere mögliche Verhaltensweisen werden u. a. in [3] und [4, S. 146–151] vorgestellt.

Kollisionsvermeidung mit Vektorfeldern

> During the past few years, potential field methods (PFM) for *obstacle avoidance* have gained increased popularity among researchers in the field of robots and mobile robots. [...] In these approaches obstacles exert repulsive forces onto the robot, while the target applies an attractive force on the robot. [...] One of the reasons for the popularity of this method is its *simplicity* and *elegance*.

Dieses Zitat aus [17, Kap. 1] beschreibt die Grundidee der Kollisionsvermeidung. Es wird jedes Hindernis durch ein repulsierendes Vektorfeld repräsentiert, welches, im Falle von dynamischen Hindernissen, mitbewegt und gegebenenfalls rotiert oder (meist abhängig von der Geschwindigkeit) skaliert wird. Im Weiteren wird für diese Vektorfelder der Ausdruck *Vermeidungsfelder* verwendet – abgeleitet vom englischen Begriff *avoidance fields*, der in [1] vorgeschlagen wird.

Radiale Repulsion: Das einfachste algorithmisch erzeugte Vermeidungsfeld stellt eine radiale Repulsion dar (vgl. [1]), die in Abbildung 2.11 in verschiedenen Formen dargestellt wird. Es handelt sich dabei um ein Feld mit Vektoren, die vom Ursprung, der sich meist in der Mitte des Hindernisses befindet, wegzeigen. Ein Vektorfeld dieser Art ist meist durch eine bestimmte Reichweite beschränkt, so dass die Repulsion nur innerhalb einer gewissen Distanz wirkt.

Die Erstellung eines solchen Vermeidungsfeldes ist meist simpel, wobei der Berechnung der Länge der Kraftvektoren Aufmerksamkeit geschenkt werden muss. So ist die Größe der Vektoren des Feldes in Abbildung 2.11 (a) konstant, was gemäß [21, Abschnitt 4.4.2] den Nachteil hat, dass ein Roboter, der in die Reichweite des Feldes kommt, ruckartige Bewegungen machen würde – im Falle von Agenten in Computerspielen würde dies neben dem funktionalen Aspekt auch zu ästhetischen Problemen führen.

Dieses Problem kann behoben werden, indem die Größe der Vektoren in Abhängigkeit vom Abstand zum Hindernis abnimmt. Eine Möglichkeit dafür ist eine lineare Abnahme, die in Abbildung 2.11 (b) dargestellt wird, und mit

$$v_{length} = \begin{cases} \frac{(D-d)}{D} & \text{für } d \leq D, \\ 0 & \text{für } d > D \end{cases} \tag{2.1}$$

beschrieben werden kann (vgl. [21, S. 129]), wobei v_{length} die Länge des Vektors, d die Distanz zum Mittelpunkt und D die maximale Reichweite der Repulsion darstellen. Die resultierende Größe v_{length} liegt zwischen 0.0 (außerhalb des Einflussbereiches) und 1.0 (direkt an der Quelle der Repulsion). Durch den Einsatz einer solchen Lösung würde der mobile Roboter bei der Annäherung an das Hindernis immer stärkere Repulsionskräfte erfahren.

Obwohl der Einsatz einer linearen Abnahme eine Verbesserung gegenüber einer radialen Repulsion mit konstanter Größe darstellt, ist der schnelle Übergang am Rand des Einflussbereiches noch immer ein Problem (vgl. [21, S. 128]). Dies kann durch den Einsatz einer exponentiellen Abnahme verbessert werden. Diese wird in Abbildung 2.11 (c) dargestellt und kann mit

$$v_{length} = \begin{cases} \frac{(D-d)^2}{D^2} & \text{für } d \leq D, \\ 0 & \text{für } d > D \end{cases} \tag{2.2}$$

beschrieben werden. Auch hier liegt v_{length} zwischen 0.0 und 1.0, wobei die Werte im Randbereich langsamer steigen, wodurch ruckartiges Verhalten des mobilen Roboters (bzw. des Agenten im Computerspiel) verhindert wird.

Statische Vektorfelder: In [1] werden zusätzlich statische Vektorfelder für die Verwendung in Computerspielen vorgeschlagen, um die gleichbleibende Umgebung zu beschreiben. Damit können damit beispielsweise Terrain oder Gebäude in der Spielwelt repräsentiert werden. Solche Vektorfelder

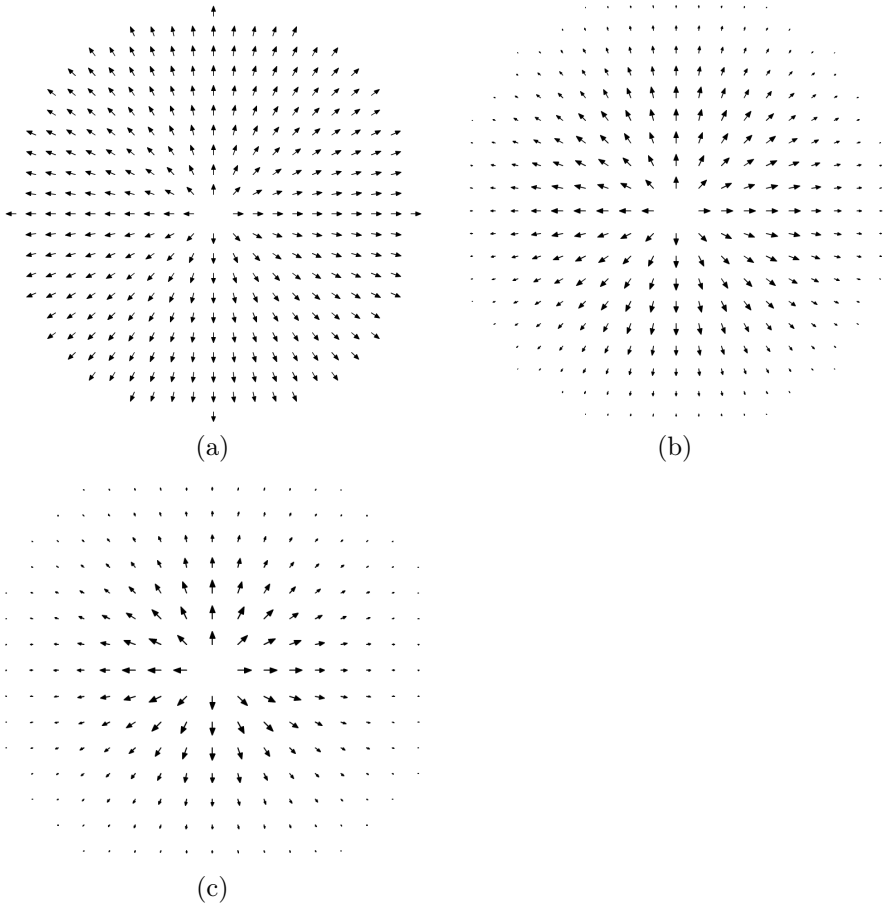


Abbildung 2.11: Arten von radialen Repulsionen. Repulsion mit konstanter Kraft (a), Repulsion mit linearer Abnahme der Kraft (b), Repulsion mit exponentieller Abnahme der Kraft (c).

werden meist nicht algorithmisch, sondern mit einem dafür vorgesehenem Werkzeug entworfen.

Die Verwendung von statischen Vermeidungsfeldern hat zwei wichtige Vorraussetzungen, die bei der Verwendung beachtet werden müssen:

- Es muss an jedem Punkt in der Umgebung ein Vektor berechnet werden können, um eine Kraft für den Agenten zu erhalten – eine kontinuierliche Repräsentation des Vermeidungsfeldes wäre wünschenswert.
- Um das statische Vermeidungsfeld speichern zu können ist eine diskrete Repräsentation notwendig.

Auf die Problematik, die sich durch diese zwei Bedingungen ergibt, sowie eine mögliche Implementierung wird in Abschnitt 3.2.2 eingegangen.

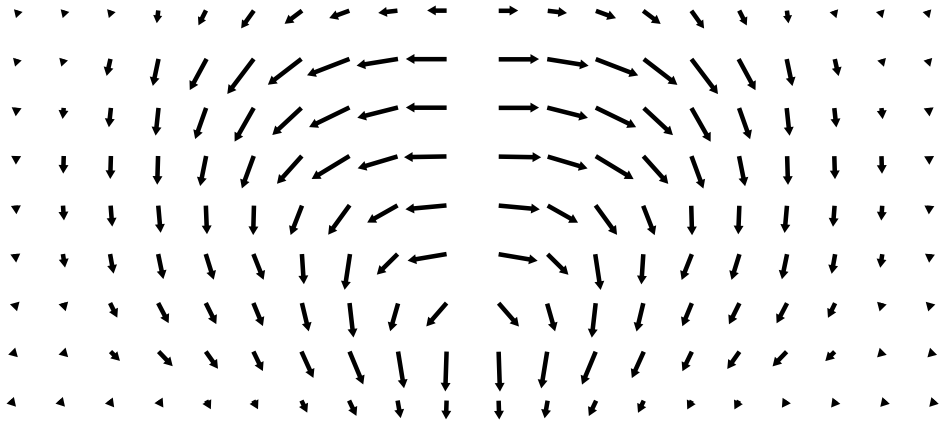


Abbildung 2.12: Benutzerdefiniertes Vermeidungsfeld gemäß [1, Abb. 3.1.9].

Benutzerdefinierte Felder für dynamische Hindernisse: Zusätzlich zum Einsatz bei statischer Levelgeometrie werden benutzerdefinierte Vermeidungsfelder gemäß [1] auch bei dynamischen Hindernissen verwendet. Wie bereits erwähnt, werden solche Vektorfelder meist mit einem speziellen Werkzeug erstellt, wodurch sie im Gegensatz zu algorithmisch erzeugten Feldern, wie der in diesem Abschnitt beschriebenen radialen Repulsion, stehen. Neben dem Problem des Findens einer sinnvollen Repräsentation (s. oben) kommt bei statischen Vermeidungsfeldern für dynamische Objekte noch ein weiterer Aspekt hinzu: Es muss möglich sein das Feld zu bewegen, rotieren und skalieren, damit das Hindernis immer richtig umgeben wird.

Der Sinn von solchen Vermeidungsfeldern ist das Lösen von Problemen, die durch den Einsatz simpler, algorithmisch erzeugter Felder auftreten können, wie beispielsweise der *T-Bone* Situation, die in Abschnitt 2.4.4 beschrieben wird. Weiters kann die Verwendung von benutzerdefinierten Vektorfeldern ästhetische Vorteile beim Ausweichverhalten von Agenten mit sich bringen. Ein Beispiel für ein statisches Vermeidungsfeld, welches zum Einsatz bei dynamischen Hindernissen gedacht ist, wird in Abbildung 2.12 dargestellt.

Kombination mit anderen Verhalten: Wie bereits beschrieben, spielt die Verbindung verschiedener Verhalten im *Reactive Paradigm* eine wichtige Rolle und erfolgt durch Vektorsummierung. Dieses Vorgehen lässt sich am besten durch ein Beispiel erklären: Ein mobiler Roboter steht vor der Aufgabe ein Ziel zu erreichen, wobei sich in der Umgebung auch ein Hindernis befindet, dem ausgewichen werden muss. Für diese Problemstellung eignet sich das bereits vorgestellte Verhalten zur Navigation zu einer Position (s. Abbildung 2.10) perfekt. Zur Kollisionsvermeidung wird ein Vermeidungsfeld mit radialer Repulsion (s. Abbildung 2.11) an die Stelle des Hindernis-

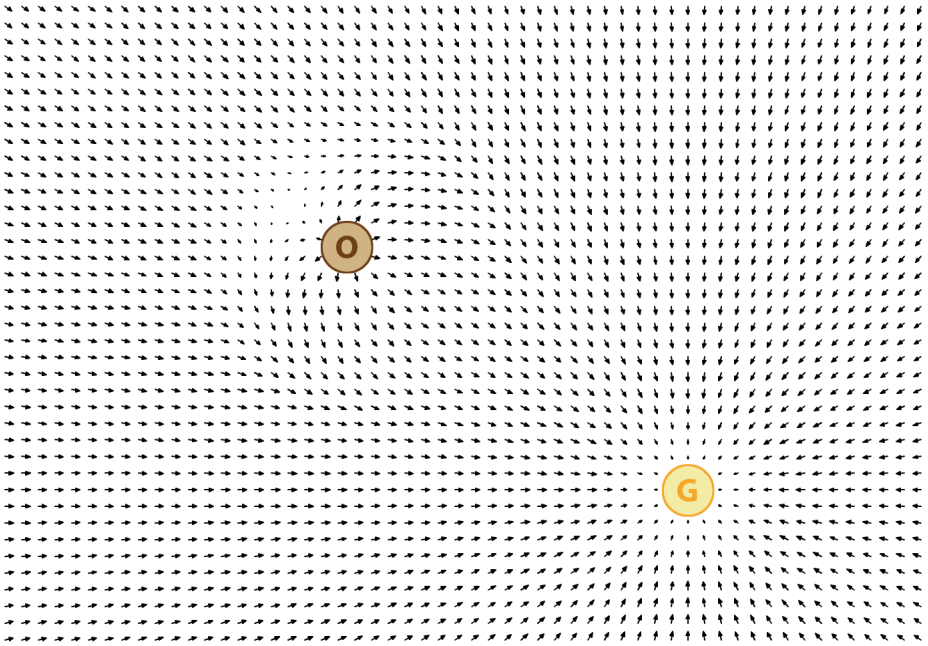


Abbildung 2.13: Vektorfeld für das Ausweichen vor dem Hindernis O und Erreichen der Position G.

ses gesetzt. Das resultierende Vektorfeld wird in Abbildung 2.13 illustriert. Anhand dieser Skizze lässt sich das Verhalten eines mobilen Roboters, der durch ein Vektorfeld navigiert, mithilfe der folgenden Analogie aus [21, S. 124] erklären:

> One way of thinking about potential fields is to imagine a force field acting on the robot. Another way is to think of them as a potential energy surface in three dimensions (gravity is often represented this way) and the robot as a marble. In that case, the vector indicates the direction the robot would "roll" on the surface. Hills in the surface cause the robot to roll away or around (vectors would be pointing away from the "peak" of the hill), and valleys would cause the robot to roll downward (vectors pointing toward the bottom).

An dieser Stelle ist die Vorgehensweise bei dynamischen Hindernissen erwähnenswert. Das Vermeidungsfeld (beispielsweise eine radiale Repulsion) wird bei einer Änderung der Position des Objekts auch mitbewegt und gegebenfalls rotiert oder skaliert. Aufgrund dessen muss das Feld, welches durch Summierung der Vektoren der eingesetzten Verhalten berechnet wurde, aktualisiert (und somit neu kalkuliert) werden.

2.4.4 Probleme

Der Einsatz von *Flow Fields* zur Kollisionsvermeidung bringt auch bestimmte Nachteile mit sich. Diese werden in diesem Abschnitt – gemeinsam mit möglichen Lösungen – vorgestellt.

Lokale Minima

Ein Problem, welches in entsprechender Fachliteratur (beispielsweise in [15, Kap. 10], [4, Abschnitt 3.3.2] und [21, Abschnitt 4.4.5]) häufig erwähnt wird, sind lokale Minima. Diese beschreiben den Fall, in dem durch die Kombination von Feldern Stellen entstehen, an denen die Summierung einen Vektor mit der Länge 0.0 ergibt, was zum Stillstand des mobilen Roboters (bzw. Agenten in einem Computerspiel) führen kann. Den Grund, warum dies als *lokales Minimum* bezeichnet wird, beschreibt Murphy in [21, S. 133] folgendermaßen:

> This is called the local minima problem, because the potential field has a minima, or valley, that traps the robot.

In Abbildung 2.14 wird ein Vektorfeld mit lokalem Minimum illustriert. Bei der dargestellten Situation handelt es sich um eine Abänderung eines bereits präsentierten Beispiels (vgl. Abbildung 2.13), bei dem ein Verhalten zur Navigation zu einer Position mit dem Vermeidungsfeld eines Hindernisses kombiniert wird. In Abbildung 2.14 befinden sich der mobile Roboter, das Hindernis und das Ziel auf einer Geraden – Zu Beginn *spürt* der Roboter nur die Attraktion, welche von der zu erreichenden Position ausgeht. Diese wird durch die Repulsion vom Hindernis zunehmend vermindert, bis sich die Kräfte an einem bestimmten Punkt, welcher in der Skizze durch einen roten Kreis markiert ist, ausgleichen, und den Roboter zum Stehen bringen.

Eine der frühesten (vgl. [21, Abschnitt 4.4.8]) und gleichzeitig einfachsten (vgl. [4, Abschnitt 4.4.2]) Lösungen für diese Problematik ist die Generierung eines Feldes mit *Störgeräuschen* (auf Englisch *noise*) – dies sind Vektoren mit geringer Stärke, die in zufällige Richtungen zeigen. Ein Beispiel für ein solches Vektorfeld wird in Abbildung 2.15 dargestellt. Gemäß [3, S. 268] sollen die geringen Impulse dazu ausreichen, dass ein mobiler Roboter ein lokales Minimum verlässt. In manchen Fällen verhindert ein Feld aus solchen Störgeräuschen auch das Betreten eines lokalen Minimums, wie in [4, S. 155] beschrieben wird. So kann das Problem, welches in Abbildung 2.14 gezeigt wurde, mithilfe von *noise* gelöst werden – dies wird in Abbildung 2.16 dargestellt.

Eine weitere Möglichkeit zur Lösung des Problems wurde in [10] vorgeschlagen und erfolgt durch die Verwendung von harmonischen Funktionen, deren größter Vorteil die Absenz von lokalen Minima ist (vgl. [16, S. 3]) – Diese Möglichkeit wird in [10], [16] und [11] detailliert beschrieben.

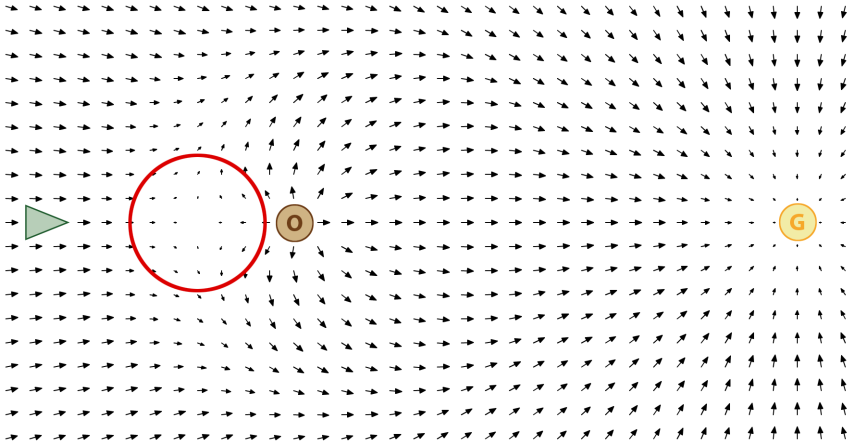


Abbildung 2.14: Vektorfeld mit lokalem Minimum. Bewegt sich der Roboter, dargestellt als grünes Dreieck, auf das Ziel G zu, kommt er an der markierten Position zum Stillstand.

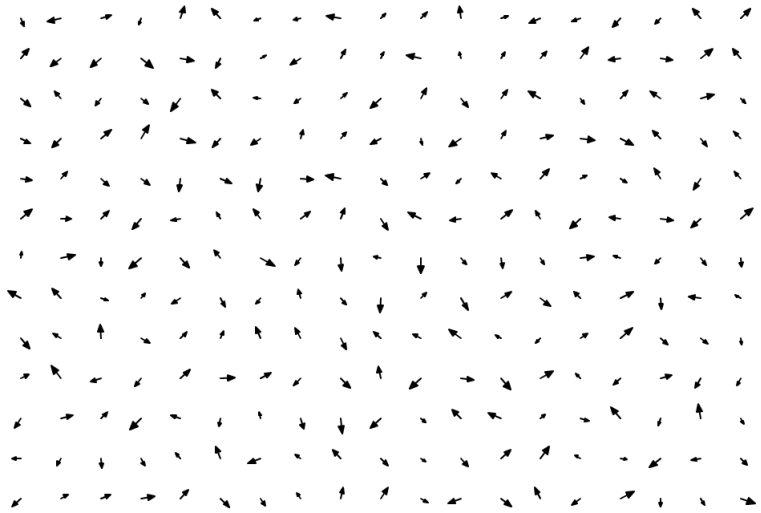


Abbildung 2.15: Vektorfeld mit Störgeräuschen.

T-Bone Situation

Die *T-Bone* Situation, die in [1] aufgezeigt wird, beschreibt ein Problem, welches durch den Einsatz von radialen Repulsionen (s. oben) auftreten kann. So passiert es oft, dass zwei Fahrzeuge beim Vermeiden einer Kollision einen ungewünschten Kurs einschlagen und sich schließlich parallel zueinander bewegen. Dies wird in Abbildung 2.17 (a) dargestellt.

Als mögliche Lösung für dieses Problem nennt Alexander in [1] alternative Vermeidungsfelder – beispielsweise das Feld in Abbildung 2.12. Der Einsatz

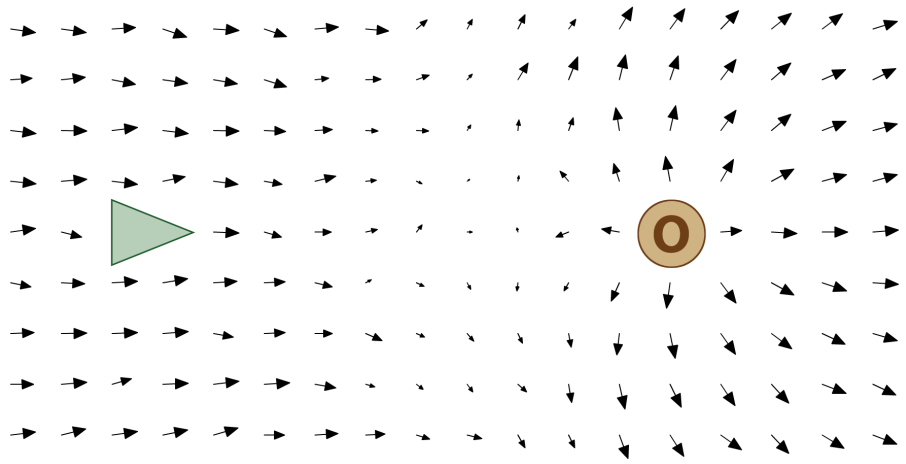


Abbildung 2.16: Verhinderung eines lokalen Minimums durch den Einsatz von *noise*.

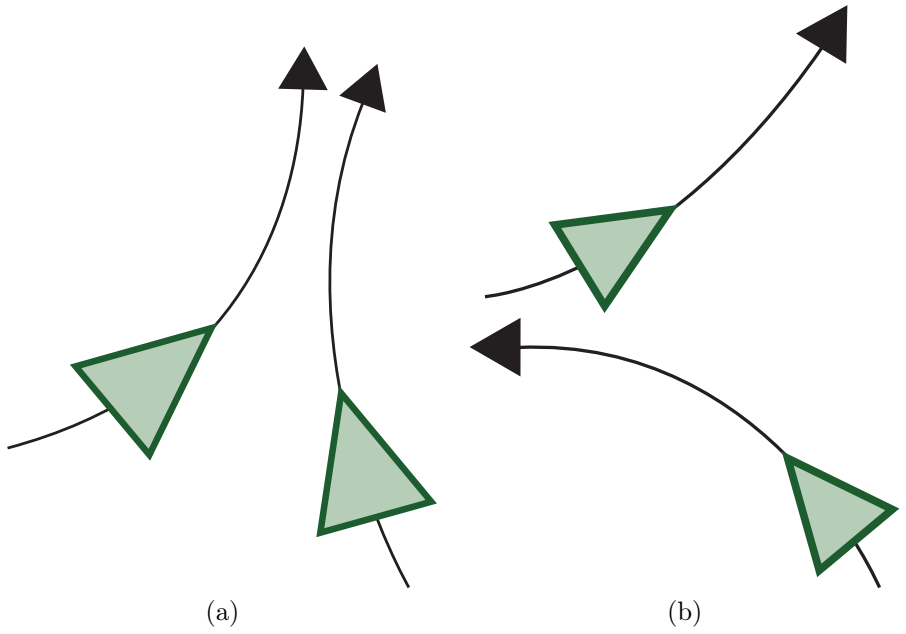


Abbildung 2.17: *T-Bone* Situation gemäß [1, Abbildung 3.1.8]. Ungewünschtes Verhalten beim Ausweichen (a), besseres Ergebnis durch den Einsatz des Vermeidungsfeldes aus Abbildung 2.12 (b).

eines solchen Feldes würde zu einem besseren Ergebnis beim Ausweichen führen, wie Abbildung 2.17 (b) zeigt.

2.5 Gegenüberstellung der Ansätze

In Abschnitt 2.4.2 wurde auf die Ähnlichkeit der Grundgedanken hinter den *steering behaviors* von Reynolds und den *behaviors* im *Reactive Paradigm* eingegangen. Reynolds selbst erwähnt in [28] die reaktive Strömung in der Robotik als Inspiration für seine Forschung. Dabei erkennt er die Analogie zwischen seinen *steering behaviors* und den *behaviors* des *Reactive Paradigm*, welche mithilfe von Vektorfeldern dargstellt werden. Aufgrund dessen kommt man mit beiden Verfahren in vielen Situationen zum gleichen Ergebnis. Man könnte beispielsweise ein *arrival*-Verhalten (wie in Abschnitt 2.3.2 beschrieben) mittels *Flow Fields* so modellieren, dass keine Unterschiede im Verhalten eines Agenten erkennbar wären. Bei beiden Implementierungen würde dieser auf das Ziel zusteuern und kurz vor der Ankunft langsamer werden. In Abschnitt 2.4.3 wurde beschrieben, wie ein solches Verhalten mit einem Vektorfeld realisiert werden kann. Dieses wird in Abbildung 2.10 skizziert.

Kollisionsvermeidung ist ein Fall, in dem die Implementierungen stark unterschiedliche Resultate (und somit auch anderes Verhalten der Agenten) liefern. Während das *obstacle avoidance behavior* von Reynolds das *gefährlichste* Hindernis auswählt und den Charakter ausweichen lässt, üben bei der Kollisionsvermeidung mittels *Flow Fields* alle Objekte in der Nähe des Agenten eine Repulsion auf diesen aus. In [1, S. 159] wird auf die Problematik beim Verfahren von Reynolds, die in Abschnitt 2.3.4 bereits genauer beschrieben wurde, eingegangen, wobei der Vorteil, der durch die Benutzung von *Flow Fields* entsteht, hervorgehoben wird:

> Any algorithms in AI can produce conflicting results. For example, in collision avoidance, avoiding one object can result in hitting another. The AI must resolve these conflicts and find a solution that avoids all objects simultaneously. [...] However, by using flow fields, this problem can be solved for all objects simultaneously.

Neben dem Vermeiden von widersprüchlichen Ergebnissen kann durch den Einsatz von *Flow Fields* auch das Problem von sich übersehenden Agenten gelöst werden. Würde die Kollisionsvermeidung in der Situation, welche in Abbildung 2.4 dargestellt wurde, mit Vektorfeldern erfolgen, könnte ein Zusammenstoß verhindert werden, da beide Fahrzeuge aufgrund der abstoßenden Kräfte, welche durch die Vermeidungsfelder entstehen, einen anderen Kurs einschlagen würden.

Aufgrund der beschriebenen Vorteile, durch welche typische Probleme beim Verfahren von Reynolds vermieden werden können, sowie der simplen Implementierung bieten sich *Flow Fields* als vielversprechende Lösung zur Kollisionsvermeidung an.

Kapitel 3

Umsetzung

Im Rahmen der Diplomarbeit wurden die zwei Ansätze zur Kollisionsvermeidung, welche in Kapitel 2 vorgestellt wurden, implementiert. Die erste Umsetzung erfolgte hierbei in einer seperaten Testumgebung. Anhand der daraus gewonnenen Erkenntnisse wurden die zwei Verfahren in das Computerspiel *Delta Strike*, welches im Anhang A beschrieben wird, integriert.

In den Abschnitten 3.1 und 3.2 werden Überlegungen zur Implementierung dokumentiert, auf die konkrete Umsetzung wird in Abschnitt 3.3 eingegangen. Im letzten Abschnitt des Kapitels wird auf die Integration der Verfahren in *Delta Strike* eingegangen.

3.1 Allgemeine Überlegungen

3.1.1 Testumgebung

Beide Verfahren zur Kollisionsvermeidung wurden in einer seperaten Testumgebung implementiert. Dabei lag der Fokus auf der Umsetzung der künstlichen Intelligenz zur Steuerung von Einheiten, sodass Erkenntnisse hinsichtlich der Implementierung gesammelt werden konnten, welche für die Integration in *Delta Strike* relevant waren.

Die Zielsetzungen an diese Testumgebung werden im Folgenden näher beschrieben:

- Einheiten sollten die Fähigkeit besitzen sich von ihrer derzeitigen Position zu einer anderen zu bewegen.
- Kollisionen mit Hindernissen (sowie anderen Einheiten) sollten verhindert werden, wobei ausgewählt werden kann, ob die Kollisionsvermeidung mithilfe von Vektorfeldern oder gemäß dem Ansatz von Reynolds geschieht.
- Die Bewegung der Einheiten sollte innerhalb einer zweidimensionalen Ebene ablaufen, da dies zum einen eine übersichtliche Visualisierung ermöglicht und zum anderen für das Spiel *Delta Strike* ausreichend ist,

da dessen Spielgeschehen ebenfalls im 2D-Raum stattfindet.
- Statische Vektorfelder zur Repräsentation von Levelgeometrie sollten implementiert werden.

Als Programmiersprache für die Implementierung der Testumgebung wurde *ActionScript 3.0* gewählt, da visuelle Prototypen in kurzer Zeit umsetzbar sind. Weiters wird das Spiel *Delta Strike* in ebendieser Sprache programmiert, wodurch die Wiederverwendung von Programmcode aus der Testumgebung bei der Integration ermöglicht wird.

3.1.2 Verhaltensweisen

Wie bereits beschrieben, stellen *behaviors* im Rahmen des *Reactive Paradigm* und *steering behaviors* in der Arbeit von Reynolds den Grundstein dar und dienen zur Steuerung eines mobilen Roboters (bzw. Agenten im Falle eines Computerspiels). Während *behaviors* beim *Flow Fields*-Verfahren durch Vektorfelder, welche die gewünschten Kraftvektoren für einen Roboter beinhalten, repräsentiert werden, werden *steering behaviors* mithilfe der erforderlichen geometrischen Kalkulationen beschrieben, durch welche die Kraft, die auf einen Agenten wirken soll, zur Laufzeit berechnet werden kann.

Trotz der Unterschiede hinsichtlich Repräsentation (bzw. Beschreibung) haben die Verhaltensweisen aus den zwei Gebieten wichtige gemeinsame Merkmale. In der folgenden Auflistung wird auf diese eingegangen, wobei der Begriff *behavior* sowohl für *behaviors* aus dem *Flow Fields* Verfahren als auch für *steering behaviors* gemäß der Arbeit von Reynolds verwendet wird.

- Zu jedem Zeitschritt berechnet ein *behavior* exakt eine Kraft, die an den Roboter, bzw. Agenten, weitergegeben wird.
- Ein *behavior* ist meist für eine spezifische Aufgabe vorgesehen, wie beispielsweise das Verfolgen von beweglichen Zielen oder das Vermeiden von Kollisionen mit Hindernissen.
- Durch die Kombination von mehreren *behaviors* (z. B. durch Summierung der Vektoren) können komplexe Gesamtverhalten geschaffen werden.

Aufgrund der Tatsache, dass jedes *behavior* exakt einen Kraftvektor berechnet, können Verhaltensweisen aus beiden Ansätzen analog verwendet und zu komplexeren Gesamtverhalten kombiniert werden – die einzige Voraussetzung ist ein Fahrzeug, welches Kraftvektoren zur Bestimmung der Bewegungsrichtung und -geschwindigkeit verwendet.

Benötigte Verhaltensweisen

Anhand der Zielsetzungen an die Testumgebung, die in Abschnitt 3.1.1 beschrieben wurden, lassen sich die Verhaltensweisen, die benötigt und implementiert werden müssen, ableiten. So wird ein *behavior* benötigt, welches

einen Agenten zu einer bestimmten Zielposition steuert. Für diese grundlegende Aufgabe würde sich sowohl das Vektorfeld, das in Abbildung 2.10 dargestellt und erklärt wurde, als auch das *arrival steering behavior* von Reynolds (s. Abschnitt 2.3.2) eignen. Da beide Verhaltensweisen, wie in Abschnitt 2.5 erklärt, zum selben Ergebnis führen, kann frei gewählt werden, welches in der Testumgebung umgesetzt wird. Aufgrund der Popularität der Arbeit von Reynolds existiert eine Vielzahl an Fachliteratur, in der das *steering behavior arrival* thematisiert wird und beispielhafte Implementierungen beschrieben werden (z. B. [9, S. 93] und [20, Abschnitt 3.3.4]) – Deshalb wurde es für den Einsatz in der Testumgebung gewählt.

Neben der Fähigkeit ein Ziel zu erreichen, müssen die Agenten in der Testumgebung dynamischen sowie statischen Hindernissen ausweichen können, wobei dies zu Testzwecken sowohl mit Vektorfeldern als auch gemäß Reynolds implementiert werden muss. Beide Ansätze müssen als *behaviors* implementiert werden, die einen Vektor mit der Kraft, welche zum Ausweichen benötigt wird, berechnen und an das Fahrzeug weitergeben.

Auf die konkrete Implementierung der benötigten Verhaltensweisen wird in Abschnitt 3.3.4 eingegangen.

Kombination von Verhaltensweisen

Es gibt verschiedene Methoden, wie aus den einzelnen Vektoren, welche von den eingesetzten *behaviors* kalkuliert werden, eine einzelne Kraft berechnet werden kann. Die Tatsache, dass alle Verhaltensweisen exakt einen Kraftvektor berechnen, erleichtert die Kombination – dadurch wurde eine Vielzahl an Lösungen vorgeschlagen (siehe [28] und [9]), die im Folgenden beschrieben werden:

- **Gewichtete Summierung:** Dieser Ansatz stellt das einfachste Verfahren zur Kombination von Kräften dar. Das Ergebnis von jedem eingesetzten *behavior* wird zuerst mit einem festgelegten Faktor multipliziert. Danach werden die resultierenden gewichteten Vektoren zu einer einzelnen Kraft summiert, die vom Fahrzeug verwendet wird.

  Die Kombination mittels gewichteter Summierung bringt allerdings Probleme mit sich. So stellt das Finden von geeigneten Faktoren für die Gewichtung der Kräfte eine zeitaufwändige Aufgabe dar (vgl. [9, S. 120]). Schlecht gewählte Werte können zu seltsamen Gesamtverhalten des Agenten führen. Ein weiterer Nachteil ist die Rechenzeit, da jedes *behavior* zu jedem Zeitschritt einen Vektor berechnet.
- **Gewichtete Summierung mit Priorisierung:** Dieser Ansatz soll die Probleme der simplen gewichteten Summierung lösen, indem die eingesetzten Verhaltensweisen priorisiert werden. In der Testumgebung, in der Verhaltensweisen zum Erreichen eines Ziels und zum Ausweichen vor Hindernissen eingesetzt werden, wäre es beispielsweise sinnvoll der

Vermeidung von Kollisionen eine höhere Priorität zu geben.

Bei der Kombination wird ein leerer Vektor zur Speicherung des finalen Kraftvektors angelegt. Danach werden die eingesetzten *behaviors* der Reihe nach (gemäß deren Priorität) aufgerufen. Der Ergebnisvektor jeder Verhaltensweise wird – analog zum vorherigen Verfahren – mit einem Faktor multipliziert und zum finalen Kraftvektor addiert. Übersteigt dessen Länge eine festgelegte maximale Schwelle, so werden keine weiteren *behaviors* mehr aufgerufen und somit nicht mehr hinzugefügt.

- ***Prioritized dithering:*** Bei dieser Lösung, die von Reynolds vorgeschlagen wurde, wird pro Zeitschritt per Zufall bestimmt, welches *behavior* aufgerufen wird. Dabei hat jede Verhaltensweise einen festgelegten Wert, der bestimmt, wie wahrscheinlich ein Aufruf dieser ist. Wie der Name bereits andeutet kann auch bei diesem Ansatz priorisiert werden.

  Analog zum letzten vorgestellten Verfahren werden alle *behaviors* gemäß deren Priorität durchgegangen. Bei jeder Verhaltensweise wird eine Zufallszahl generiert, die mit der Wahrscheinlichkeit des Aufrufs verglichen wird. Je nach Ergebnis wird entweder der Vektor des aktuellen *behaviors* als Steuerungsvektor verwendet oder der gleiche Schritt für die nächste Verhaltensweise wiederholt.

  Dieser Ansatz benötigt minimale Rechenzeit, bringt allerdings den Nachteil mit sich, dass die Festlegung der Wahrscheinlichkeiten für die Verhaltensweisen eine ähnlich aufwändige Angelegenheit ist, wie die Findung von geeigneten Werten bei der gewichteten Summierung.

Für den Einsatz in der Testumgebung wurde sowohl die gewichtete Summierung mit Priorisierung als auch *prioritized dithering* implementiert, um so einen Vergleich dieser Lösungen zu ermöglichen.

3.1.3 Fahrzeugphysik

Wie bereits beschrieben, muss das Fahrzeug mehrere Verhaltensweisen verwalten können und anhand des Kraftvektors, welcher von diesen *behaviors* vorgegeben wurde, seine Bewegungsrichtung und -geschwindigkeit ändern. Da das simple Modell von Reynolds, welches bereits kurz erklärt wurde und im Folgenden detailliert beschrieben wird, die Voraussetzung erfüllt, wurde es für den Einsatz in der Testumgebung gewählt.

Sollte komplexere Fahrzeugphysik benötigt werden, so wäre ein Austausch problemlos (und ohne Veränderung der verwendeten *behaviors*) möglich. Dies wird auf Basis des Cowboy Beispiels, welches in Abschnitt 2.3.2 beschrieben wurde, von Reynolds in [28] verdeutlicht:

> As described above, a cowboy's horse can be considered as an example of the locomotion layer. The rider's steering decisions

> are conveyed via simple control signals to the horse who converts them into motion. The point of making the abstract distinction between steering and locomotion is to anticipate "plugging in" a new locomotion module. Imagine lifting the rider off of the horse and placing him on a cross-country motorcycle. The goal selection and steering behavior remain the same. All that has changed is the mechanism for mapping the control signals (go faster, turn right, ...) into motion.

Das Modell von Reynolds basiert auf der Annahme, dass ein Vehikel als Punkt, in dem dessen Masse vereinigt ist, repräsentiert werden kann. Diese Annahme hat den großen Nachteil, dass sie unrealistisch ist – real besitzt jedes physikalische Objekt sowohl ein Volumen als auch einen Trägheitsmoment. Dafür rechtfertigen die Vorteile dieses simplen Modells, wie z. B. die einfache und Rechenzeit sparende Implementierung, den Einsatz in der Testumgebung.

Die grundlegenden Bestandteile des Fahrzeugmodells von Reynolds werden im Folgenden aufgezählt und beschrieben (vgl. [28] und [9, S. 87–91]):

- Das Fahrzeug besitzt einen Skalar, der die **Masse** repräsentiert.
- Die **Position** des Vehikels wird als Vektor gespeichert. Im Falle der Testumgebung ist dieser zweidimensional mit jeweils einer Komponente für den x und einer für den y Wert.
- Die **Geschwindigkeit** wird ebenfalls durch einen zweidimensionalen Vektor repräsentiert und wird in jedem Zeitschritt durch einwirkende Kräfte von den *behaviors* verändert.
- Bei den meisten Fahrzeugen wird die Kraft, die zur Bewegung benötigt wird, vom Vehikel selbst generiert. Diese Aufgabe übernimmt beispielsweise der Motor bei einem Automobil oder Düsen bei einem Kampfjet. Der maximale Schub, den solche Antriebsarten erzeugen können, wird im simplen Modell von Reynolds durch einen Skalar für die **Maximale Kraft** repräsentiert. Dieser Wert wird verwendet um den Kraftvektor, welcher von den *behaviors* vorgegeben wurde, zu begrenzen.
- Meist besitzen Fahrzeuge eine **Maximale Geschwindigkeit**, die durch das Gegenspiel von Beschleunigung und verlangsamenden Faktoren, wie z. B. Reibung, entsteht. Im simplen Fahrzeugmodell wird diese als Skalar gespeichert, mithilfe dessen der Vektor für die Geschwindigkeit begrenzt wird.
- Ein normalisierter Vektor beinhaltet die **Ausrichtung** des Fahrzeugs.

In jedem Zeitschritt wirkt eine Kraft, die, wie im letzten Abschnitt beschrieben, durch die Kombination von mehreren Verhaltensweisen zustande kommt, auf das Fahrzeug. Ist die Länge dieser größer als die festgelegte maximale Kraft wird der Vektor verkürzt. Mithilfe des Kraftvektors wird nun die Geschwindigkeitszunahme gemäß dem zweiten Newtonschen Gesetz mittels

$$a = \frac{F}{m} \tag{3.1}$$

berechnet (vgl. [7, Kap. 1]), wobei a die Beschleunigung, F die Kraft und m die Masse darstellen. Der daraus resultierende Vektor wird schließlich mit der Zeit, die seit dem letzten Zeitschritt vergangen ist, multipliziert und zur aktuellen Geschwindigkeit addiert, wobei das Ergebnis gegebenenfalls auf den Maximalwert reduziert werden muss. Zuletzt wird eine Aktualisierung der Ausrichtung (durch Normalisierung der Geschwindigkeit) sowie der Position (durch Addition der Geschwindigkeit) durchgeführt.

Die Simulation der Physik des Fahrzeugs liegt dem *Eulerschen Polygonzugverfahren* zugrunde. Hier dient dieses der Annäherung der Veränderung der Geschwindigkeit Δv pro Zeitschritt Δt. Diese kann mit

$$\Delta v = a\Delta t \tag{3.2}$$

dargestellt werden (vgl. [7, S. 173]).

Durch die Simplizität des Modells von Reynolds ist dieses auch mit Nachteilen verbunden (vgl. [28]). Da die Ausrichtung des Fahrzeuges immer der Bewegungsrichtung entspricht, können manche Effekte, wie beispielsweise *Driften* aus dem Motorsport, nicht simuliert werden.

Weiters ermöglicht das Fahrzeugmodell große Änderungen der Ausrichtung innerhalb kleiner Zeitschritte – so kann sich ein Vehikel beispielsweise im Stand drehen, was nur wenige reale Fahrzeuge können. Gemäß Reynolds könnte dieser negative Effekt – falls gewünscht – durch das Einführen eines maximalen Wertes für die Drehung pro Zeitschritt, das Limitieren der seitlichen Kräfte innerhalb der *behaviors* oder das Simulieren des Trägheitsmoments verhindert werden.

3.2 Überlegungen zu *Flow Fields*

3.2.1 Berechnung der Kraft

In Abschnitt 2.4.3 wurden verschiedene Beispiele für die Funktionsweise von *Flow Fields* beschrieben und mit Skizzen dargestellt. In diesen wurde immer das gesamte Feld angezeigt, wodurch man zum falschen Schluss gelangen kann, dass dieses in jedem Zeitschritt zur Gänze berechnet wird. Dies wird in folgendem Zitat aus [4, S. 99] geklärt:

> Another seemingly significant problem with the use of the potential fields method is the amount of time required to compute the entire field. Reactive robotic systems eliminate this problem by computing each field's contribution at the instantaneous position merely where the robot is currently located [...]. One of the major misconceptions in understanding reactive methods based

> on potential fields is a failure to recognize the fact that the only computation needed is that required to assess the forces from the robot's current position within the world. This method is thus inherently very fast to compute as well as highly parallelizable. When the entire field is represented in a figure, it is only for the reader's edification.

In dynamischen Umgebungen, in denen sich Hindernisse bewegen können, ist eine häufige Neuberechnung der Kraft, die auf einen Roboter (bzw. Agenten) wirkt, unerlässlich. Eine Kalkulation des ganzen Vektorfeldes würde erhebliche Rechenzeit verursachen und ist nur für Visualisierungen, die beispielsweise der Fehlersuche oder Überprüfung der Richtigkeit dienen können (vgl. [21, Abschnitt 4.4.2]), sinnvoll.

Für Hindernisse, deren Vermeidungsfelder als radiale Repulsion implementiert sind, kann der Einfluss auf eine bestimmte Position sehr schnell berechnet werden, wie in den Gleichungen 2.1 und 2.2 gezeigt wurde. Bei benutzerdefinierten Vermeidungsfeldern, die in Abschnitt 2.4.3 vorgestellt wurden, kann die wirkende Kraft auf verschiedene Arten kalkuliert werden, beispielsweise durch bilineare Interpolation, auf die in Abschnitt 3.2.2 eingegangen wird.

3.2.2 Benutzerdefinierte Vektorfelder

Wie in Abschnitt 2.4.3 beschrieben wurde, können benutzerdefinierte Vektorfelder, die zur Darstellung von statischer Geometrie der Umgebung oder als ästhetischere Vermeidungsfelder für dynamische Hindernisse (z. B. bei der *T-Bone* Situation, die in Abschnitt 2.4.4 präsentiert wurde) verwendet werden, meist nicht algorithmisch erzeugt werden, was die Berechnung der Kräfte erschwert. So wurde bereits auf folgende Problematik hingewiesen: Während zum Speichern eines benutzerdefinierten Vektorfeldes eine diskrete Repräsentation benötigt wird, muss trotzdem an jeder Position ein Vektor kalkuliert werden können.

Ein möglicher Ansatz, der von Alexander in [1] vorgeschlagen wird, ist die Speicherung des Feldes als Raster mit gleich großen Vierecken, an deren Ecken Vektoren liegen. Es gibt verschiedene Möglichkeiten, um aus dieser diskreten Darstellung kontinuierliche Werte für jede Position im Feld zu erhalten. Dies kann beispielsweise durch bilineare Interpolation erreicht werden, die anschließend erklärt wird. Der Ansatz von Alexander ist für den Einsatz im zweidimensionalen Raum vorgesehen, was für *Delta Strike*, bei dem das gesamte Spielgeschehen auf einer Ebene abläuft, ausreichend ist. In 3D könnte man die Vektoren in einer dreidimensionalen Matrix speichern und beispielsweise durch den Einsatz trilinearer Interpolation kontinuierliche Werte zwischen den Ecken der Quader berechnen.

Abbildung 3.1 skizziert die bilineare Interpolation bei einem Quadrat mit

vier Vektoren an den Ecken. Wie in [1] beschrieben, werden diese vier anhand ihrer relativen Distanz zur Position P, an der die benötigte Kraft berechnet werden soll, skaliert: Die für diese Skalierung verwendeten Faktoren s_x (für die x-Achse) und s_y (für die y-Achse) können hierbei mit

$$s_x = \frac{P_x - A_x}{B_x - A_x} \tag{3.3}$$

und

$$s_y = \frac{P_y - A_y}{C_y - A_y} \tag{3.4}$$

berechnet werden. Mithilfe dieser Faktoren kann gemäß [1] nun der Vektor V_P an Position P mit

$$V_{P_x} = (1 - s_y) \cdot ((1 - s_x) \cdot V_{A_x} + s_x \cdot V_{B_x}) + s_y \cdot ((1 - s_x) \cdot V_{C_x} + s_x \cdot V_{D_x}) \tag{3.5}$$

für die x-Komponente und

$$V_{P_y} = (1 - s_y) \cdot \big((1 - s_x) \cdot V_{A_y} + s_x \cdot V_{B_y}\big) + s_y \cdot \big((1 - s_x) \cdot V_{C_y} + s_x \cdot V_{D_y}\big) \tag{3.6}$$

für die y-Komponente kalkuliert werden.

3.2.3 Eingesetzte Vermeidungsfelder

Gemäß den Voraussetzungen, welche die Testumgebung erfüllen soll (vgl. Abschnitt 3.1.1), müssen sowohl Hindernisse als auch statische Levelgeometrie anhand von Vektorfeldern repräsentiert werden.

Für die Vermeidung von dynamischen und statischen Hindernissen wurden simple radiale Repulsionen verwendet. Dabei wurden die drei Möglichkeiten, welche in Abschnitt 2.4.3 vorgestellt wurden, umgesetzt.

Zur Umsetzung von gleichbleibenden, globalen Vektorfeldern, welche beispielsweise zur Darstellung der Spielwelt benutzt werden können, wurden benutzerdefinierte Vektorfelder gewählt.

3.3 Implementierung

In diesem Abschnitt wird detailliert auf die Implementierung der beiden Verfahren in der Testumgebung eingegangen. Zusätzlich zur Beschreibung der Umsetzung werden Programmtexte (in *ActionScript 3.0*), bzw. Auszüge aus diesen, zum besseren Verständnis aufgeführt.

Eine wichtige Konvention, welche beim Lesen der Quelltexte beachtet werden muss, ist die Namensgebung. Ist die Sichtbarkeit einer Variable auf

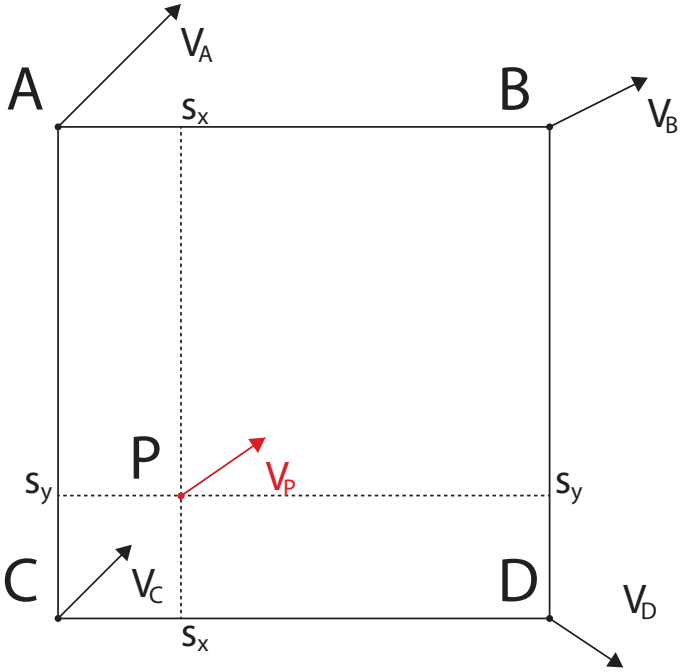


Abbildung 3.1: Bilineare Interpolation gemäß [1, Abb. 3.1.2].

`private` gesetzt, und somit nur von der Klasse, in der sie sich befindet, verwendbar, so wird dies durch ein `f` am Anfang des Namens angezeigt. Bei öffentlichen Zugriffsfunktionen wird auf diesen Buchstaben verzichtet.

3.3.1 Benötigte Mathematik

Vektoren

Für die Implementierung der *steering behaviors* und der Vektorfelder in der zweidimensionalen Testumgebung wird eine Klasse zur Verwaltung eines Vektors mit zwei Komponenten benötigt.

Zu diesem Zweck wurde die Klasse `Vec2` geschrieben. Neben den Komponenten x und y enthält diese Funktionen für grundlegende Rechenoperationen. Diese werden immer von einem Objekt der Klasse `Vec2` aufgerufen und im Folgenden aufgelistet sowie erklärt, da sie für das Verständnis der Programmtexte in diesem Kapitel wichtig sind.

- Die Funktionen `add` und `subtract` erhalten einen Vektor als Parameter und führen eine Addition (bzw. Subtraktion) der Komponenten durch.
- Die Funktion `mult` erhält einen Skalar als Parameter und multipliziert die Komponenten des Vektors mit diesem.
- Die Funktion `length` berechnet die Länge des Vektors und gibt diese zurück.

- Die Funktion `normalize` normalisiert den Vektor.
- Die Funktion `truncate` erhält einen Skalar als Parameter. Ist der Vektor länger als der übergebene Wert, wird die Länge diesem angepasst. Die Ausrichtung ändert sich dabei nicht.
- Die Funktion `perpendicular` dreht den Vektor so, dass die neue Ausrichtung lotrecht zur alten Orientierung ist.
- Die Funktion `dotProduct` berechnet das Skalarprodukt zwischen dem übergebenen und dem aufrufenden Vektor.

Die beschriebenen Funktionen verändern die Werte im `Vec2` Objekt, auf dem sie aufgerufen wurden. Da dies nicht immer erwünscht ist, wurden Alternativen zur Verfügung gestellt, die das Ergebnis in einem neuen Vektor speichern und diesen zurückgeben – diese werden durch das Wort `get` am Anfang des Funktionsnamens (z. B. `getNormalized` bei `normalize`) einheitlich gekennzeichnet.

Umrechnung zwischen Koordinatensystemen

Die Umrechnung zwischen dem lokalen und dem globalen Koordinatensystem (im Englischen *local space* und *world space*, bzw. *global space*) erleichtert die Implementierung der Kollisionsvermeidung gemäß Reynolds. Der Unterschied zwischen diesen wird in [9, S. 26–27] beschrieben:

> The world space representation is normally what you see rendered to your screen. Every object is defined by a position and orientation *relative to the origin of the world coordinate system* [...]. A soldier is using world space when he describes the position of a tank with a grid reference, for instance.
>
> Local space, however, describes the position and orientation of objects relative to a specific entity's local coordinate system. In two dimensions, an entity's local coordinate system can be defined by a facing vector and a side vector (representing the local x- and y-axis, respectively), with the origin positioned at the center of the entity (for three dimensions an additional up vector is required).

Bei der Kollisionsvermeidung gemäß Reynolds können unwichtige Hindernisse durch die Ermittlung ihrer Positionen im lokalen Koordinatensystem des Agenten schnell disqualifiziert werden.

In der Klasse `MathUtils` wurden zwei Funktionen, welche für das *obstacle avoidance behavior* benötigt werden, implementiert:

- Die Funktion `getPointInLocalSpace` rechnet eine Position in das lokale Koordinatensystem um.
- Die Funktion `getVectorInGlobalSpace` transformiert einen Vektor von der *local space* in die *world space*.

3.3.2 Fahrzeug

Das Fahrzeug wurde gemäß den Überlegungen, welche in Abschnitt 3.1.3 vorgestellt wurden, implementiert. Hierfür wurde die Klasse `Vehicle` umgesetzt, welche die Variablen

- `fMass` für die Masse des Fahrzeuges,
- `fPosition` für die Position als `Vec2`,
- `fVelocity` für die aktuelle Geschwindigkeit als `Vec2`,
- `fMaxForce` für die maximale Kraft,
- `fMaxSpeed` für die maximale Geschwindigkeit
- `fOrientation` für die Ausrichtung als `Vec2`,
- `fDestination` für die Zielposition als `Vec2`, sowie
- `fRadius` für die Breite des Fahrzeugs, welche für die Kollisionsvermeidung gemäß Reynolds benötigt wird,

beinhaltet. Weiters wird das Feld `fSteeringBehaviors` verwendet um die eingesetzten Verhaltensweisen zu verwalten – dabei werden entsprechende Funktionen bereitgestellt, durch welche zusätzliche *behaviors* hinzugefügt werden können.

Die Simulation der Vehikelphysik findet in der `update` Methode statt. Diese wird jeden Zeitschritt aufgerufen, wobei die vergangene Zeit als Parameter übergeben wird:

```
1  public function update(elapsedTime : Number) : void {
2    var steeringForce : Vec2 = ForceCalculator.
       calculateWeightedSumWithPrioritization(fSteeringBehaviors, fMaxForce);
3    //var steeringForce : Vec2 = ForceCalculator.calculateByPrioritizedDithering(
       fSteeringBehaviors, fMaxForce);
4    var acceleration : Vec2 = steeringForce.getMult(1 / fMass);
5    acceleration.mult(elapsedTime);
6    fVelocity.add(acceleration);
7    fVelocity.truncate(fMaxSpeed);
8    fPosition.add(fVelocity);
9
10   if (fVelocity.length > MIN_THRESHOLD) {
11     fOrientation = fVelocity.getNormalized();
12   }
13 }
```

Mithilfe der Klasse `ForceCalculator` wird der Kraftvektor der eingesetzten Verhaltensweisen berechnet – im Falle der Testumgebung entweder durch gewichtete Summierung mit Priorisierung oder durch *prioritized dithering*. Anhand dieser Kraft wird nun die Beschleunigung des Fahrzeuges berechnet und zur Geschwindigkeit addiert. Diese wird so gekürzt, dass `fMaxSpeed` nicht überschritten wird, und die Aktualisierung der Position des Fahrzeuges durchgeführt. Die Ausrichtung wird nur dann verändert, wenn die Geschwindigkeit einen sehr kleinen Wert – repräsentiert durch die Konstante `MIN_THRESHOLD` – übersteigt. Der Grund hierfür wird in [9, S. 90] beschrieben:

> A vehicle's heading should always be aligned with its velocity so this is updated, making it equal to the normalized velocity vector. But – and this is important – *the heading is only calculated if the vehicle's velocity is above a very small threshold value.* This is because if the magnitude of the velocity is zero, the program will crash with a divide by zero error, and if the magnitude is non-zero but *very* small, the vehicle may (depending on the platform and operating system) start to move erratically a few seconds after it has stopped.

3.3.3 Vektorfelder

Schnittstelle

Da jedes Vektorfeld, unabhängig davon, ob es ein Hindernis oder beispielsweise eine ganze Spielwelt repräsentiert, an jeder Position eine bestimmte Kraft berechnet, ist es sinnvoll eine Schnittstelle zu erstellen, welche die Verwaltung von mehreren Feldern erleichtert. Für diesen Zweck wurde das Interface `Influencing` geschaffen, welches alle Vektorfelder implementieren müssen:

```
1 public interface Influencing {
2   function getForceAt(x:Number, y:Number):Vec2;
3 }
```

Vermeidungsfelder für Hindernisse

Da sich die beschriebenen Arten von abstoßenden Vektorfeldern nur in der Berechnung der Kraft unterscheiden, wurde eine gemeinsame Basisklasse erstellt, die grundlegende Attribute vereinen soll. So beinhaltet die Klasse `DynamicForce`, welche die Schnittstelle `Influencing` implementiert, die abstrakte[1] Funktion `getForceAt`, sowie Variablen zur Speicherung der Position als `fX` und `fY`, des Radius `fSpread`, in welchem die Kraft wirken soll, sowie der maximalen Stärke der Repulsion `fMaxForce`.

Mithilfe dieser Basisklasse wurden die drei Arten der Repulsion, welche in Abschnitt 2.4.3 beschrieben wurden, für den Einsatz in der Testumgebung implementiert. Dazu wurden die Klassen

- `ConstantRadialRepulsion` (für Repulsionen mit konstanter Kraft),
- `LinearDropOffRadialRepulsion` (für Repulsionen mit linearer Abnahme der Kraft), sowie
- `ExponentialDropOffRadialRepulsion` (für Repulsionen mit exponentieller Abnahme der Kraft)

[1] Da *ActionScript 3.0* die Erstellung von abstrakten Funktionen nicht ermöglicht, wurden diese im Rahmen der Testumgebung so implementiert, dass zur Laufzeit ein Fehler geworfen wird, sobald es zu einem Aufruf kommt.

umgesetzt. Da sich diese lediglich in der Implementierung der Funktion `getForceAt` unterscheiden, welche bei allen ähnlich ausfällt, wird exemplarisch nur auf eine dieser Klassen eingegangen.

In der Klasse `LinearDropOffRadialRepulsion` wurde die Repulsion mit linearer Abnahme folgendermaßen implementiert:

```
1 override public function getForceAt(x : Number, y : Number) : Vec2 {
2   var d : Number = Math.sqrt(Math.pow(fX - x, 2) + Math.pow(fY - y, 2));
3   if (d > fSpread) {
4     return new Vec2(0, 0);
5   }
6   var result : Vec2 = new Vec2(x - fX, y - fY);
7   result.truncate(((fSpread - d) / fSpread) * fMaxForce);
8   return result;
9 }
```

Anhand der übergebenen Koordinaten wird der Abstand `d` berechnet. Ist dieser größer als der Radius der Repulsion, so kann sofort eine leere Kraft zurückgegeben werden. Ansonsten wird der Vektor zwischen Ursprung der Repulsion und übergebener Position berechnet. Dieser wird abhängig von der Distanz (vgl. Gleichung 2.1) sowie der maximalen Stärke gekürzt und letztendlich zurückgegeben.

Globale Vektorfelder

Für globale Vektorfelder wurde die Klasse `FlowField` erstellt, die gemäß der Schnittstelle `Influencing` umgesetzt wurde. Um das Gitter aus Vierecken, welches in Abschnitt 3.2.2 beschrieben wurde, zu verwalten, wird ein zweidimensionaler `Vector fForces`, welcher mit `Vec2` Objekten gefüllt ist, als Variable gespeichert. Weiters wird der Wert `fSamplingDist` zum Bestimmen der Distanz zwischen zwei Vektoren verwendet.

Wird nun die Funktion `getForceAt` aufgerufen, kann eine Kraft für jede Position innerhalb des Vektorfeldes kalkuliert werden: Mithilfe der übergebenen Koordinaten sowie der Distanz zwischen zwei Vektoren wird das *Viereck* bestimmt, in welchem die Position liegt. Weiters werden die dazugehörigen `Vec2` Objekte aus `fForces` gelesen. Anhand des Abstandes der übergebenen Position zu den Eckpunkten werden die Skalierungsfaktoren berechnet, mit deren Hilfe die Komponenten des Ergebnisvektors durch bilineare Interpolation (vgl. Abschnitt 3.2.2) der vier Vektoren bestimmt werden können.

Zu Testzwecken wurde weiters die Klasse `FieldGenerator` umgesetzt. In dieser werden statische Funktionen bereitgestellt, die ein schnelles Befüllen der Matrix – beispielsweise mit Vektoren mit Zufallswerten zur Erschaffung eines *noise*-Feldes (vgl. Abschnitt 2.4.4) – ermöglichen.

Verwaltung

Bei der Kollisionsvermeidung mit Vektorfeldern wirken alle Repulsionen (und sonstigen Einflüsse, wie beispielsweise benutzerdefinierte Felder) gleichzeitig

auf den Agenten. Deshalb ist es sinnvoll diese so zu verwalten, dass zu jedem Zeitschritt exakt ein Vektor berechnet werden kann, welcher die Summe der Kräfte aller Vermeidungsfelder an einer bestimmten Position beinhaltet.

Aus diesem Grund wurde die Klasse `InfluenceManager` erstellt, welche der Verwaltung von Objekten des Typen `Influencing` dient – Somit können sowohl Vermeidungsfelder von Hindernissen als auch ganze Vektorfelder darin gespeichert werden. Diese werden im Feld `fInfluences` verwaltet und können durch Zugriffsfunktionen hinzugefügt werden.

Muss die Kraft an einer bestimmten Position berechnet werden, so wird bei jedem Objekt im Feld `fInfluences` die Funktion `getForceAt` aufgerufen. Die Gesamtkraft wird durch simple Summierung der einzelnen Vektoren kalkuliert.

3.3.4 Verhaltensweisen

Basisklasse

Um die Verwaltung mehrerer Verhaltensweisen zu erleichtern, wurde eine Basisklasse für diese erstellt, welche die Gemeinsamkeiten aller *behaviors* vereint. So besteht die Klasse `SteeringBehavior` aus einem Wert, welcher die Gewichtung, die für die Kombination notwendig ist, speichert, und der abstrakten Funktion `getSteeringForce`, welche den berechneten Kraftvektor als `Vec2` Objekt zurückgibt. Weiters beinhaltet die Klasse die Variable `probability`. Diese bestimmt die Wahrscheinlichkeit des Aufrufs und kommt bei *prioritized dithering* zum Einsatz.

Arrival

Das *arrival behavior*, welches der Bewegung eines Agenten zu einer bestimmten Position dient, wurde als Unterklasse von `SteeringBehavior` implementiert. Zum Ausführen benötigt diese Verhaltensweise das Vehikel, auf das es angewendet wird, da dieses die aktuelle Position und Geschwindigkeit, die maximale Geschwindigkeit sowie die Zielposition beinhaltet. Weiters muss die Distanz angegeben werden, ab welcher der Agent anfängt zu bremsen.

Der Programmtext der Funktion `getSteeringForce` der Verhaltensweise, welche gemäß der Arbeit von Reynolds (s. [28]) umgesetzt wurde, wird im Folgenden dargestellt und erklärt:

```
1  override public function getSteeringForce() : Vec2 {
2    if (!fVehicle.destination) {
3      return new Vec2();
4    }
5
6    if (fVehicle.position.x == fVehicle.destination.x && fVehicle.position.y ==
       fVehicle.destination.y) {
7      return new Vec2();
8    }
9
10   var targetOffset : Vec2 = fVehicle.destination.getSubtract(fVehicle.position);
```

```
11   var dist : Number = targetOffset.length;
12   var rampedSpeed : Number = fVehicle.maxSpeed * (dist / fSlowingDist);
13   var clippedSpeed : Number = Math.min(rampedSpeed, fVehicle.maxSpeed);
14   var desiredVelocity : Vec2 = targetOffset.getMult(clippedSpeed / dist);
15   return desiredVelocity.getSubtract(fVehicle.velocity);
16 }
```

Falls keine Zielposition angegeben wurde (bzw. diese bereits erreicht wurde) wird ein leerer Kraftvektor erstellt und zurückgegeben. Ansonsten wird der Vektor `targetOffset` zwischen Ziel und aktueller Position berechnet, dessen Länge `dist` die Distanz angibt. Mit dieser Information wird nun die gewünschte Geschwindigkeit berechnet – innerhalb der Bremsdistanz kommt es zu einem linearen Abfall, außerhalb treten größere Werte auf, welche mithilfe der maximalen Geschwindigkeit des Fahrzeugs limitiert werden. Der Unterschied zwischen der gewünschten und der aktuellen Geschwindigkeit bestimmt den Kraftvektor, der zurückgegeben wird.

Obstacle avoidance

Analog zum *arrival behavior* wurde die Verhaltensweise zur Kollisionsvermeidung als Unterklasse von `SteeringBehavior` implementiert. Neben dem Fahrzeug, auf dem es angewendet wird, benötigt das *steering behavior* eine Liste der Hindernisse, welche sich in der Welt befinden. Zu diesem Zweck wurde die Schnittstelle `Obstacle` erstellt – Da die Kollisionsvermeidung gemäß Reynolds davon ausgeht, dass jedes Hindernis durch einen Kreis angenähert werden kann, besteht diese lediglich aus Position und Radius:

```
1 public interface Obstacle {
2     function get radius() : Number;
3     function get position() : Vec2;
4 }
```

Um eine Anpassung der Verhaltensweise zu ermöglichen, wurde das *obstacle avoidance behavior* um die Skalare `fDBoxBaseLength` (zur Bestimmung der Länge der *detection box*) und `fBrakingWeight` (zur Einstellung der Bremsstärke) erweitert.

Der relevante Programmtext der Verhaltensweise befindet sich innerhalb der Funktion `getSteeringForce` und wurde gemäß [28] sowie [9, S. 99–104] implementiert. Zum besseren Verständnis wird dieser aufgeteilt und jeweils im Nachhinein erklärt:

```
1 var dBoxLength : Number = fDBoxMinLength + ((fVehicle.velocity.length / fVehicle
      .maxSpeed) * fDBoxBaseLength);
2 var obstaclesInRange : Vector.<Obstacle> = tagCloseObstacles(dBoxLength);
```

Im ersten Schritt wird die Länge der *detection box* berechnet – umso höher die aktuelle Geschwindigkeit, desto länger soll diese sein. Weiters werden anhand der Funktion `tagCloseObstacles` alle Hindernisse ausgewählt, die für die Vermeidung in Frage kommen, also eventuell innerhalb der *detection box* liegen. Da die Funktion lediglich durch alle Hindernisse iteriert und jene,

deren Distanz zum Fahrzeug kleiner als `dBoxLength` ist, zurückgibt, wird auf eine Darstellung des Programmtextes verzichtet.

Nun wird das näheste (und somit *gefährlichste*) Hindernis bestimmt:

```
1 var closestObstacle : Obstacle;
2 var distToClosestObstacle : Number;
3 var localPositionOfClosestObstacle : Vec2;
4
5 for (var i : int = 0; i < obstaclesInRange.length; ++i) {
6   var localPosition : Vec2 = MathUtils.getPointInLocalSpace(obstaclesInRange[i].
      position, fVehicle.position, fVehicle.orientation);
7
8   if (0 <= localPosition.x) {
9     var expandedRadius : Number = fVehicle.radius + obstaclesInRange[i].radius;
10
11    if (Math.abs(localPosition.y) < expandedRadius) {
12      var temp : Number = Math.sqrt(expandedRadius * expandedRadius -
      localPosition.y * localPosition.y);
13      var intersectionPoint : Number = localPosition.x - temp;
14      intersectionPoint = (intersectionPoint < 0) ? localPosition.x + temp :
      intersectionPoint;
15
16      if (!closestObstacle || intersectionPoint < distToClosestObstacle) {
17        closestObstacle = obstaclesInRange[i];
18        distToClosestObstacle = intersectionPoint;
19        localPositionOfClosestObstacle = localPosition;
20      }
21    }
22  }
23 }
```

Es wird durch alle Hindernisse in der Nähe des Fahrzeuges iteriert. Dabei wird die Position des aktuellen `Obstacle` in das lokale Koordinatensystem des Vehikels transformiert. Liegt diese im negativen Bereich der x-Achse, so befindet sich das Hindernis hinter dem Fahrzeug und kann ignoriert werden. Ansonsten wird anhand der Projektion auf die lokale y-Achse überprüft, ob das `Obstacle` innerhalb der *detection box* liegt – Hierfür genügt der Vergleich mit der Summe der Radien des Fahrzeugs und des Hindernisses. Wenn ja, wird der Schnittpunkt der kreisförmigen Annäherung des `Obstacle` mit der x-Achse berechnet. Ist dies bis jetzt der näheste Schnittpunkt wird dieser, gemeinsam mit dem zugehörigen Hindernis sowie dessen Position im lokalen Koordinatensystem, gespeichert.

Im letzten Schritt muss die Kraft, die zum Ausweichen benötigt wird, berechnet werden:

```
1 var steeringForce : Vec2 = new Vec2();
2 if (closestObstacle) {
3     steeringForce.y = -localPositionOfClosestObstacle.y;
4     steeringForce.x = (closestObstacle.radius - localPositionOfClosestObstacle.x
      ) * fBrakingWeight;
5 }
6 return MathUtils.getVectorInGlobalSpace(steeringForce, fVehicle.orientation);
```

Zum seitlichen Ausweichen (y-Komponente des Kraftvektors) wird die negative y-Komponente der lokalen Position des *gefährlichsten* Hindernisses verwendet. Außerdem wird, wie von Buckland in [9, S. 103] vorgeschlagen, eine

Bremskraft verwendet, deren Stärke abhängig von der Distanz zum Hindernis ist. Diese bestimmt die x-Komponente des Kraftvektors. Da das Ergebnis aus der Sicht des Fahrzeuges berechnet wurde, muss dieses zum Schluss noch in das globale Koordinatensystem transformiert werden.

Flow following

Die Klasse `FlowFollowing` wurde als Unterklasse von `SteeringBehavior` implementiert und dient der Steuerung eines Agenten anhand von Kräften, die ein festgelegtes Objekt des Typen `Influencing` vorgibt. Dieses wird in der Variable `fInfluence` gespeichert und kann beispielsweise ein einzelnes Vektorfeld des Typen `FlowField` sein, oder ein `InfluenceManager`, welcher alle Hindernisse der Spielwelt verwaltet.

Die Implementierung der Funktion `getSteeringForce` gestaltet sich einfach: Es wird lediglich die `getForceAt` Funktion von `fInfluence` mit der Position des Fahrzeugs aufgerufen – der resultierende Vektor wird als Steuerungskraft an das Fahrzeug zurückgegeben.

Kombination der Verhaltensweisen

Wie bereits beschrieben (vgl. Abschnitt 3.1.2), wurden zwei Lösungen zur Kombination von Verhaltensweisen in der Testumgebung implementiert: die gewichtete Summierung mit Priorisierung und *prioritized dithering*. Hierfür wurde die Klasse `ForceCalculator` erstellt, welche gemäß [9, S. 121–124] umgesetzt wurde.

Gewichtete Summierung mit Priorisierung: Zur Berechnung werden die *behaviors*, gespeichert in einem Feld mit dem Basistyp `SteeringBehavior`, sowie ein Wert, welcher die maximale Kraft bestimmt, an die Funktion `calculateWeightedSumWithPrioritization` übergeben, die im Folgenden dargestellt wird:

```
1 public static function calculateWeightedSumWithPrioritization(steeringBehaviors
      : Vector.<SteeringBehavior>, maxForce : Number) : Vec2 {
2   var result : Vec2 = new Vec2();
3
4   for (var i : int = 0; i < steeringBehaviors.length; ++i) {
5     var force : Vec2 = steeringBehaviors[i].getSteeringForce();
6     force.mult(steeringBehaviors[i].weight);
7     if (!accumulateForce(result, force, maxForce)) {
8       return result;
9     }
10  }
11
12  return result;
13 }
```

Die Funktion `calculateWeightedSumWithPrioritization` geht alle übergebenen *behaviors* der Reihe nach durch, wobei angenommen wird, dass diese

gemäß deren Priorität sortiert sind. Der Kraftvektor jeder Verhaltensweise wird mit der zugehörigen Gewichtung multipliziert und an die statische Hilfsfunktion `accumulateForce` übergeben, die der laufenden Summierung der Vektoren dient. Sobald diese (mithilfe eines *booleschen* Wertes) signalisiert, dass kein Platz mehr für weitere Kräfte existiert, wird das Ergebnis der Kombination zurückgegeben.

Im Folgenden wird der Programmtext der Funktion `accumulateForce` dargestellt und erklärt:

```
1 private static function accumulateForce(runningSum : Vec2, forceToAdd : Vec2,
      maxForce : Number) : Boolean {
2   var magnitudeSoFar : Number = runningSum.length;
3   var magnitudeRemaining : Number = maxForce - magnitudeSoFar;
4
5   if (magnitudeRemaining <= 0) {
6     return false;
7   }
8
9   var magnitudeToAdd : Number = forceToAdd.length;
10
11  if (magnitudeToAdd <= magnitudeRemaining) {
12    runningSum.add(forceToAdd);
13  } else {
14    forceToAdd.truncate(magnitudeRemaining);
15    runningSum.add(forceToAdd);
16  }
17
18  return true;
19 }
```

Anhand des Parameters `maxForce` und der aktuellen Summe wird bestimmt, wieviel Kraft noch addiert werden kann. Ist diese kleiner als, bzw. gleich, null, so wird der *boolesche* Wert *falsch* zurückgegeben. Ansonsten wird die Länge des Vektors, der hinzugefügt werden soll, überprüft und so verkürzt, dass die maximale Kraft nicht überschritten wird. Letzendlich wird dieser zur laufenden Summe addiert.

Prioritized dithering: Dieses Verfahren zur Kombination von Verhaltensweisen wurde in der Funktion `calculateByPrioritizedDithering` implementiert, wobei analog zur gewichteten Summierung mit Priorisierung ein Feld mit den eingesetzten *behaviors* sowie ein Wert, welcher die maximale Kraft bestimmt, benötigt werden:

```
1 public static function calculateByPrioritizedDithering(steeringBehaviors :
      Vector.<SteeringBehavior>, maxForce:Number) : Vec2 {
2   for (var i : int = 0; i < steeringBehaviors.length; ++i) {
3     if (i == steeringBehaviors.length - 1) {
4       return steeringBehaviors[i].getSteeringForce().getTruncated(maxForce);
5     }
6     var rand : Number = Math.random();
7     if (rand <= steeringBehaviors[i].probability) {
8       if (steeringBehaviors[i].getSteeringForce().length != 0) {
9         return steeringBehaviors[i].getSteeringForce().getTruncated(maxForce);
10      }
11    }
```

```
12   }
13
14   return new Vec2();
15 }
```

Auch bei dieser Funktion wird angenommen, dass die Verhaltensweisen gemäß deren Priorität sortiert sind. Es wird durch alle Verhaltensweisen iteriert, wobei bei jedem Durchlauf eine Zufallszahl generiert wird, durch die entschieden wird, ob das aktuelle *behavior* zum Einsatz kommt. Wenn ja, wird der Steuerungsvektor der Verhaltensweise berechnet – ist dieser kein leerer Kraftvektor, wird er gegebenenfalls auf den Maximalwert `maxForce` reduziert und zurückgegeben. Ansonsten fährt die Schleife fort – bei der letzten Iteration wird die aktuelle Verhaltensweise unabhängig von der Zufallszahl verwendet.

3.4 Integration in *Delta Strike*

In diesem Abschnitt wird auf die Integration der Kollisionsvermeidung im Computerspiel *Delta Strike* eingegangen, wobei die Erkenntnisse, welche anhand der Testumgebung gewonnen wurden, beschrieben werden. Weiters werden Möglichkeiten zur Verbesserung der Ansätze (beispielsweise hinsichtlich Effizienz) präsentiert.

3.4.1 Erkenntnisse aus der Testumgebung

Repulsionen

Bei der Verwendung der Testumgebung ist klar geworden, dass große Hindernisse nur schwer mit den vorgestellten radialen Repulsionen dargestellt werden können. Obwohl die repulsierende Kraft mit der Nähe zum Zentrum des Objekts zunimmt, kann es aufgrund dessen Größe zu Kollisionen kommen.

Das Problem kann durch das Anlegen einer Zone, welche nicht durchdrungen werden soll, gelöst werden. Dabei besitzt jedes Hindernis zusätzlich zum Wirkradius der Repulsion einen inneren Radius, innerhalb dessen eine maximale Abstoßung wirkt. Die Kraftvektoren zwischen den Radien werden durch lineare oder exponenentielle Abnahme berechnet. Ein Beispiel für ein solches Vermeidungsfeld wird in Abbildung 3.2 dargestellt.

Da das Verändern des inneren Radius die Repräsentation von sowohl großen als auch kleinen Hindernissen ermöglicht und die Ergebnisse bei linearer Abnahme zufriedenstellend sind, wurde diese Art der Repulsion für den Einsatz in *Delta Strike* gewählt.

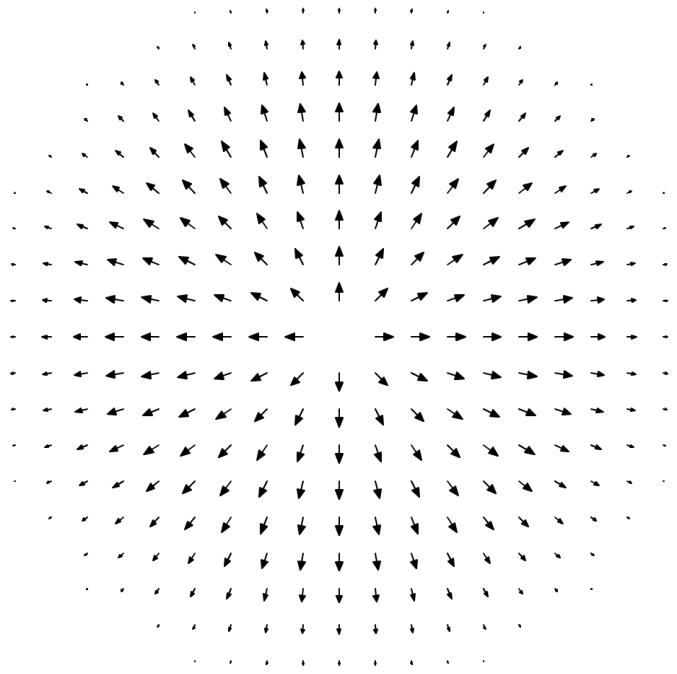


Abbildung 3.2: Radiale Repulsion mit maximaler Abstoßung innerhalb eines bestimmten Radius.

Globale Vektorfelder

Während globale Vektorfelder, welche gleichbleibende Levelgeometrie repräsentieren sollen, eine simple Möglichkeit darstellen einfache Umgebungen zu beschreiben, würden sie bei komplexen Spielwelten keine Algorithmen zur Wegfindung ersetzen und aufgrund der großen Anzahl an Vektoren einen hohen Speicheraufwand verursachen.

Aufgrund der Tatsache, dass das Spielgeschehen von *Delta Strike* im Weltraum stattfindet (vgl. Anhang A) und deshalb keine komplexe Spielwelt mit statischen Hindernissen existiert, wurde auf den Einbau von globalen Vektorfeldern verzichtet.

Kombination von Verhaltensweisen

Zur Kombination von *behaviors* wurden zwei Ansätze in der Testumgebung implementiert und getestet: gewichtete Summierung mit Priorisierung und *prioritized dithering*. Dabei kamen die Verhaltensweisen zum Erreichen einer Position (*arrival*), zum Ausweichen vor Hindernissen gemäß Reynolds (*obstacle avoidance*) und zur Kollsionsvermeidung mit Vektorfeldern (*flow following*) zum Einsatz. Aufgrund der Wichtigkeit der Vermeidung von Kollisionen bekamen die entsprechenden Verhaltensweisen eine höhere Priorität

zugewiesen als das *arrival behavior*.

Während die Ergebnisse beim Einsatz der gewichteten Summierung mit Priorisierung mit geeigneten Werten für die Gewichtungen zufriedenstellend waren, führte die Verwendung von *prioritized dithering* zu zwei Problemen:

- Auch wenn die Wahrscheinlichkeit des Aufrufs der Verhaltensweise zur Kollisionsvermeidung hoch eingestellt wurde, kam es zu Situationen, in denen kurz vor einem Hindernis das *arrival behavior* aufgerufen wurde, was zu einer Kollision führte.
- Durch die Zufälligkeit des Verfahrens kam es häufig zu Situationen, in denen die eingesetzten Verhaltensweisen abwechselnd aufgerufen wurden – Bei stark unterschiedlichen Kräften zuckte der Agent in jedem Zeitschritt in eine andere Richtung, was ein ästhethisches Problem darstellt.

Aufgrund der negativen Erfahrungen mit *prioritized dithering* wurde die gewichtete Summierung mit Priorisierung in *Delta Strike* implementiert.

3.4.2 Raumpartitionierung

Ein Problem, welches sowohl bei den Verfahren zur Kollisionsvermeidung als auch in Gebieten wie Computergrafik und Kollisionserkennung besteht, ist die Rechenzeit, welche für das Finden von Objekten in einem bestimmten Bereich benötigt wird. Bei beiden vorgestellten Ansätzen müssen alle Hindernisse in der Nähe eines Agenten in Betracht gezogen werden – entweder zum Finden des *gefährlichsten* Hindernisses oder zur Berechnung der repulsierenden Kraft. Würde man hierfür alle Entitäten in der Spielwelt beachten, würde dies zu einem hohen Rechenaufwand pro Zeitschritt führen. Eine Möglichkeit dies zu verhindern stellt die Partitionierung der Welt dar, bei welcher der Raum in mehrere Bereiche aufgeteilt wird. Dadurch können Objekte, welche sich in einer bestimmten Region befinden, schnell gefunden werden (vgl. [9, S. 126]). Im Anhang B wird auf verschiedene Möglichkeiten zur Partitionierung eines Raumes eingegangen.

Raumpartitionierung in *Delta Strike*

Für die Raumpartitionierung in *Delta Strike* wird ein *quadtree* verwendet. Dieser funktioniert analog zum *octree*, wird aber im zweidimensionalen Raum benutzt – Die Spielwelt wird nicht mit Würfeln aufgeteilt, sondern mit Quadraten. Somit hat jeder Knoten eines *quadtree* vier Unterknoten, wie in Abbildung 3.3 skizziert wird.

In der Testumgebung ermittelt jede Einheit für sich selbst, welche Kräfte auf sie wirken (beispielsweise unter Zuhilfenahme eines `InfluenceManager`). Da jedes Vermeidungsfeld eine andere Reichweite hat, müssen alle Felder in der Spielwelt durchgegangen werden, wodurch Verfahren zur Raumpartitionierung keinerlei Vorteil hinsichtlich Rechenzeit liefern würden. Aus diesem

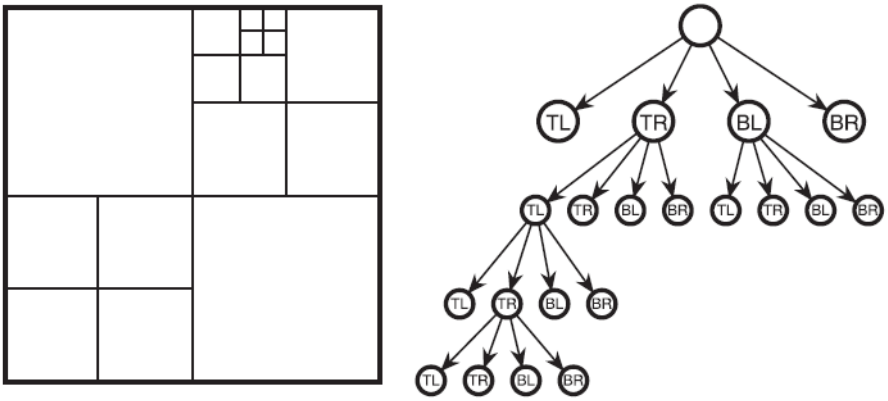


Abbildung 3.3: Raumpartitionierung anhand eines *quadtree*. Links wird ein partitionierter Bereich dargestellt, rechts der dazugehörige Baum. Aus [24, Kap. 23].

Grund wurde die Ermittlung der Kräfte, welche auf einen Agenten wirken, anders implementiert. In jedem Zeitschritt analysiert jedes Vermeidungsfeld, welche Fahrzeuge in dessen Wirkradius liegen, berechnet an deren Positionen die Kraftvektoren und übergibt diese an die entsprechenden Vehikel. Da jedes Vermeidungsfeld nur die Einheiten in dessen Reichweite benötigt, kann die Suche dieser anhand des *quadtree* optimiert werden.

3.4.3 Komponentenorientierte Entwicklung

Das Computerspiel *Delta Strike* wurde anhand des komponentenbasierten Ansatzes (*CBGD*, bzw. *Component based Game Development*) entwickelt, bei welchem sämtliche Teile des Spiels als Bausteine implementiert werden. Das folgende Zitat aus [25, S. 20] beschreibt das Prinzip eines komponentenorientierten Systems:

> Würde man in der klassischen Spieleprogrammierung beim Entwurf eines Weltraumspiels die Klasse „Raumschiff“ modellieren, so würde diese Klasse in einem komponentenorientierten System nicht existieren. Das Spielobjekt „Raumschiff“ würde sich rein aus der Aggregation von kleinen Komponenten wie: Antrieb, Position, Waffe und Visualisierung zusammensetzen.

So wurden in *Delta Strike* sowohl Verhaltensweisen als auch Vermeidungsfelder als Komponenten implementiert, wodurch diese sämtlichen Objekten in der Spielwelt zugewiesen werden können. Da die Verwaltung der Komponenten in *Delta Strike* mithilfe von *XML*-Konfigurationsdateien erfolgt, können die Parameter der Verhaltensweisen und Felder, wie beispielsweise

die Bremsdistanz des *arrival behavior* oder die Reichweite einer radialen Repulsion, mit wenig Aufwand verändert und optimiert werden, was das Testen und Konfigurieren erleichtert.

Kapitel 4

Evaluation

Wie in Kapitel 1 beschrieben wurde, setzt sich die *KI* in Computerspielen zum Ziel eine *Illusion von Intelligenz* zu schaffen. Da die Verfahren zur Kollisionsvermeidung auch zu dieser beitragen, wurde eine empirische Studie durchgeführt, in welcher die Ansätze hinsichtlich ihrer Wirkung auf den Spieler untersucht werden, wobei wissenschaftliche Methoden der *Usability* verwendet werden.

4.1 Überlegungen zur Studie

Wie in [6, Abschnitt 1.1] beschrieben wird, beschäftigt sich die Computerspiel-Industrie seit ihrer Frühzeit mit der Erfassung der Nutzererfahrung (oder im Englischen *User Experience*, bzw. kurz *UX*) beim Spielen, wobei herausgefunden werden soll, ob ein bestimmter Titel *Spaß macht*. Insbesondere in den letzten zehn Jahren wurde die Evaluierung der *User Experience* in Computerspielen durch Methoden aus dem Forschungsgebiet der *Human-Computer Interaction* inspiriert – so können gemäß [19, Abschnitt 7.2.1] herkömmliche Verfahren der *Usability* bei der Erfassung von Nutzererfahrung bei Spielen angewendet werden.

Zur Evaluation der Verfahren zur Kollisionsvermeidung im Spiel *Delta Strike* wurde eine Studie durchgeführt, wobei Methoden der *Usability* verwendet wurden. Die Überlegungen zu dieser Untersuchung, sowie deren Vorbereitung, werden in den folgenden Abschnitten beschrieben.

4.1.1 Zielsetzung und Überblick

Das Ziel der Studie ist das Vergleichen der vorgestellten Ansätze zur Kollisionsvermeidung, wobei herausgefunden werden soll, welcher Ansatz unter welchen Bedingungen einen besseren Eindruck vermittelt.

Um den Vergleich der Ansätze zu ermöglichen wurde ein *Test-Level* in *Delta Strike* implementiert. In diesem soll eine typische Situation, in welcher

man sich als Spieler eines Echtzeitstrategie- oder Rollenspiels befindet, simuliert werden. Es wurden zwei Versionen des *Levels* vorbereitet, welche sich lediglich im eingesetzten Verfahren zur Kollisionsvermeidung unterscheiden. Das *Test-Level* wird in Abschnitt 4.1.2 detailliert beschrieben.

Das implementierte *Level* wurde von Teilnehmern, welche für die Studie ausgewählt wurden (siehe Abschnitt 4.1.4), gespielt. Dabei wurden die Erfahrungen der Spieler anhand von wissenschaftlichen Methoden der *Usability*, welche in Abschnitt 4.1.3 vorgestellt werden, erfasst.

4.1.2 *Test-Level* und Aufgabenstellung

> The basic rule for test tasks is that they should be chosen to be as representative as possible of the uses to which the system will eventually be put in the field.

Aus diesem Zitat aus [22, S. 185] geht hervor, dass die Aufgaben, welche während des Tests von den Probanden gelöst werden, möglichst realitätsnah gestaltet werden sollen. Deshalb wurde zur Evaluation der beiden Verfahren zur Kollisionsvermeidung ein *Test-Level* in *Delta Strike* implementiert, welches so aufgebaut wurde, dass eine typische Situation in einem Echtzeitstrategie- bzw. Rollenspiel simuliert wird. Dabei kontrolliert der Spieler sowohl einzelne Einheiten als auch eine Gruppe, wobei die Steuerung, analog zu *Diablo* (1996) von *Blizzard Entertainment*, über die Maus erfolgt.

Im Folgenden wird auf die Aufgabenstellung, deren Bearbeitung in etwa fünf bis zehn Minuten dauert, eingegangen. Dabei werden die einzelnen Aufgaben in der Reihenfolge, in welcher diese im *Test-Level* gelöst werden, aufgelistet.

1. Zu Beginn des *Levels* steuert der Spieler ein einzelnes Raumschiff. Mit diesem sollen mehrere *Items*[1], welche sich in der Nähe der Anfangsposition befinden, aufgesammelt werden. Diese simple Aufgabe soll dem Spieler helfen, sich an die Bedienung von *Delta Strike* zu gewöhnen.
2. Nach dem Aufsammeln der *Items* soll der Spieler das Raumschiff durch ein Asteroidenfeld steuern. Die Asteroiden wurden beim Starten des *Levels* zufällig innerhalb des Feldes angeordnet und sind unbeweglich.
3. Während des Flugs durch das Asteroidenfeld trifft man auf computergesteuerte Gegnergruppen, welche bekämpft werden müssen. Damit sich der Spieler auf das Verhalten der Einheiten beim Vermeiden von Kollisionen konzentrieren kann, wurden die Attribute der Gegner so konfiguriert, dass sie dem gesteuerten Raumschiff keinen ernsthaften Schaden zufügen können.

[1] In Computerspielen werden Gegenstände, welche vom Spieler aufgenommen werden können, als *Items* bezeichnet.

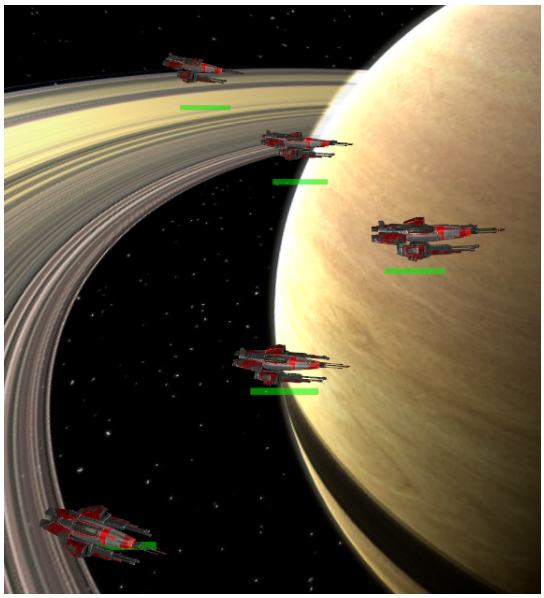


Abbildung 4.1: *V-Formation* im *Test-Level*.

4. Nach dem Verlassen des Asteroidenfeldes begegnet man vier computergesteuerten Agenten, welche sich der Spielfigur anschließen und gemeinsam mit dieser eine *V-Formation* (siehe Abbildung 4.1) bilden. Ab diesem Zeitpunkt steuert der Spieler fünf Einheiten und soll mit diesen, analog zu Punkt 1, mehrere *Items* aufsammeln. Dabei sollen auch Formationswechsel durchgeführt werden, um die Bewegung der Raumschiffe bei diesen Manövern zu beobachten.
5. Analog zu Punkt 2 soll wieder ein Asteroidenfeld durchquert werden, wobei diesmal eine Formation befehligt wird.
6. Im Asteroidenfeld wird der Spieler wieder von Gegnergruppen angegriffen, welche mithilfe der gesteuerten Formation bekämpft werden müssen.
7. Das Verlassen des Asteroidenfeldes stellt das Ende des *Test-Levels* dar.

Das *Test-Level* lässt sich in zwei Abschnitte unterteilen: In der ersten Hälfte wird nur ein Raumschiff gesteuert, im zweiten Teil befehligt der Spieler eine ganze Formation. Somit kann sowohl das Einzelverhalten als auch das Gruppenverhalten von Agenten bei der Vermeidung von Kollisionen beobachtet werden.

4.1.3 Methoden

Während manche Vorgehensweisen, wie beispielsweise das Messen der Zeit, welche der Benutzer für die Erledigung einer Aufgabe benötigt (vgl. [22, Ab-

schnitt 6.7]), den Vorteil haben, dass die erfassten Werte leicht quantifizierbar und somit auch vergleichbar sind, wäre die Sinnhaftigkeit des Einsatzes einer solchen Methode bei der Studie, welche für diese Diplomarbeit durchgeführt wurde, begrenzt, da der Fokus auf dem subjektiven Empfinden des Probanden beim Spielen des *Test-Levels* liegt. Aus diesem Grund wurden zwei Methoden ausgewählt, welche dieses erfassen sollen. Im Folgenden werden die eingesetzten Vorgehensweisen beschrieben.

Methode des lauten Denkens

Bei der Methode des lauten Denkens, bzw. *Thinking Aloud-Methode*, werden Probanden dazu aufgefordert die Gedanken, welche sie beim Bearbeiten der Aufgabenstellung haben, zu äußern (vgl. [14, S. 566]). Während diese Methode zum Erfassen von quantifizierbaren Daten nicht geeignet ist, hat sie den Vorteil, dass bereits bei einer geringen Zahl an Testteilnehmern eine große Menge qualitativer Informationen aufgezeichnet werden kann (siehe [22, Abschnitt 6.8]). So können gemäß [5, S. 256] Schlüsse hinsichtlich den Eindrücken und Gefühlen eines Probanden gezogen werden – wodurch sich diese Methode für den Einsatz bei der Evaluation der Verfahren zur Kollisionsvermeidung anbietet.

Allerdings ist die Methode mit einem Nachteil verbunden, welcher in [14, S. 566] beschrieben wird:

> Nachteil bei dieser Methode ist, dass es für die Testpersonen meist sehr ungewöhnlich ist, sämtliche Gedankengänge laut auszusprechen und jede Handlung zu kommentieren. Deshalb kann es während der Durchführung der Tests auch oft vorkommen, dass die Testpersonen vergessen, ihre Vorgehensweise laut zu kommentieren und dann vom Expertenteam daran erinnert werden müssen.

Durch die Unterbrechungen des Testleiters und das Formulieren der Gedanken kann es zu Störeffekten kommen, welche das Verhalten beim Lösen der Aufgaben beeinflussen können (siehe [5, S. 257]).

Es existiert eine Variation der Methode des lauten Denkens, bei welcher das beschriebene Problem vermieden wird, indem zwei Probanden die Testaufgaben gemeinsam lösen und dabei kommunizieren (vgl. [14, S. 566]). Der Vorteil dieser Vorgehensweise, welche als *Constructive Interaction* oder *Co Discovery*-Methode bezeichnet wird, wird in [22, S. 198] zusammengefasst:

> The main advantage of constructive interaction is that the test situation is much more natural than standard thinking-aloud tests with single users, since people are used to verbalizing when they are trying to solve a problem together. Therefore, users

> may make more comments when engaged in constructive interaction than when simply thinking aloud for the benefit of an experimenter [...].

Um mögliche Störungen und Unterbrechungen, welche bei der Methode des lauten Denkens auftreten können, zu verhindern, wurde bei der Evaluation der Verfahren zur Kollisionsvermeidung *Constructive Interaction* verwendet. Obwohl das *Test-Level* für nur einen Spieler konzipiert ist, kann die Anwesenheit eines beobachtenden Probanden zur Gewinnung wertvoller Erkenntnisse beitragen, da dieser weniger vom Spielen des *Levels* abgelenkt ist und somit dem eigentlichen Spielgeschehen, insbesondere der Bewegung der Einheiten, mehr Aufmerksamkeit schenken kann.

Interview

Gemäß [22, S. 209] können viele Aspekte, wie beispielsweise die subjektive Zufriedenheit mit bestimmten Eigenschaften eines Systems, durch das Befragen eines Probanden erforscht werden. Dabei kommen meist Fragebögen oder Interviews zum Einsatz. Während sich diese ähnlich sind, haben sie doch Unterschiede, wie in [22, S. 210] folgendermaßen erläutert wird:

> Questionnaires and interviews are very similar methods since both involve asking users a set of questions and recording their answers. Questionnaires are printed on paper or presented interactively on a computer and can be administered without the need to have any other people present beside the user answering the questions. In contrast, interviews involve having an interviewer read the questions to the respondent, and the answers are recorded by the interviewer instead of being filled in by the respondent.

Fragebögen beinhalten vorwiegend Fragen mit Antwortvorgaben, während offene Fragestellungen aufgrund von verschiedenen Gründen (siehe [14, s. 534–535]) vermieden werden. Dies macht die Daten, welche mithilfe dieser Methode erhoben wurden, meist leicht quantifizierbar (vgl. [22, Abschnitt 7.2]). Im Gegensatz zu Fragebögen werden bei Interviews vorwiegend offene Fragestellungen verwendet, wobei der Interviewer auch Anschlussfragen stellen kann. Weiters kann dieser Probleme, welche beim Verständnis von Fragen entstehen können, durch das Umformulieren von Fragestellungen beheben.

Bei der Studie, welche im Rahmen dieser Diplomarbeit durchgeführt wurde, wurden Interviews als Methode gewählt, da diese durch offene Fragestellungen und Anschlussfragen tiefere Einblicke in die Erfahrungen, welche die Probanden beim Testen der Verfahren zur Kollisionsvermeidung gemacht haben, ermöglichen. Für diesen Zweck wurde ein *semi-strukturiertes Interview* (siehe [14, S. 530]) verwendet. Bei dieser Art des Interviews werden zentrale

Fragestellungen der Studie im Vorhinein vorbereitet, wobei spontane Anschlussfragen, welche von den Antworten des Probanden abhängig sind, im Laufe des Gesprächs formuliert werden.

4.1.4 Testteilnehmer

Für jede Methode der *Usability* wird eine Mindestanzahl an Testteilnehmern benötigt. Gemäß [22, Tabelle 10] sind fünf Personen für ein Interview erforderlich, während für die Methode des lauten Denkens drei bis fünf Probanden notwendig sind – beim Einsatz von *Constructive Interaction* wird mindestens die doppelte Anzahl benötigt (vgl. [22, S. 198]). So wurden zwölf Personen für die Evaluation der Kollisionsvermeidung herangezogen – Die Kriterien für die Auswahl der Probanden sowie die Aufteilung der Testaufgaben unter diesen wird im Folgenden beschrieben.

Auswahl der Probanden

> The main rule regarding test users is that they should be as representative as possible of the intended users of the system.

Wie anhand des Zitates aus [22, S. 175] ersichtlich wird, sollen Probanden, welche für eine Studie herangezogen werden, möglichst der Nutzergruppe des Produktes entsprechen. So wurden die Testteilnehmer für die Evaluation der Kollisionsvermeidung gemäß der Zielgruppe des Spiels *Delta Strike* ausgewählt. Während potenzielle Spieler von *Delta Strike* weder gemäß deren Alter noch deren Geschlecht eingeschränkt werden können, ist deren Erfahrung mit Computerspielen relevant. So stellen Personen, welche hauptsächlich Echtzeitstrategie- sowie Rollenspiele spielen, die Zielgruppe dar.

Die Auswahl von Personen, welche zu dieser Zielgruppe gehören, hat den Vorteil, dass diese bereits Erfahrung mit den genannten Computerspielen haben. Somit haben diese weniger Probleme mit der Bedienung und Funktionsweise von *Delta Strike* als Benutzer, die zum ersten Mal mit einem Echtzeitstrategie- bzw. Rollenspiel konfrontiert werden. Dadurch kann der Kollisionsvermeidung, auf welcher der Fokus der Evaluation liegt, mehr Aufmerksamkeit geschenkt werden.

Aufgabenaufteilung

Gemäß [14, Abschnitt 12.2] gibt es bei Experimenten, in welchen mehrere Testbedingungen existieren – bei der Evaluation der Verfahren zur Kollisionsvermeidung wären dies die zwei Versionen des *Test-Levels* – zwei Möglichkeiten, wie Aufgaben unter den Testteilnehmern verteilt werden können:

- **Between-Subjects Testing:** Beim *Between-Subject Testing* nimmt jeder Benutzer an nur einem Test teil – jeder Testbedingung wird eine Gruppe an Personen zugewiesen. Ein Problem, welches bei dieser

Art der Aufgabenaufteilung besteht, ist der große Unterschied zwischen den Testteilnehmern, welcher die Ergebnisse der Studie maßgeblich beeinflussen kann. Dies kann durch den Einsatz von mehr Testpersonen verhindert werden (vgl. [22, Abschnitt 6.2]).

- **Inter-Subjects Testing:** Beim *Inter-Subject Testing* nimmt jeder Benutzer an jedem Test teil, wodurch Unterschiede zwischen Testpersonen keine Rolle spielen. Allerdings hat diese Art der Aufgabenaufteilung den Nachteil, dass ein Lerneffekt auftreten kann: So kann es passieren, dass ein Benutzer durch die Erfahrungen, welche dieser bei vorherigen Tests gemacht hat, beeinflusst wird (vgl. [22, Abschnitt 6.2]). Dies kann durch das Balancieren der Reihenfolge der Testbedingungen vermieden werden (siehe [14, S. 564]).

Bei der Evaluation der beiden Verfahren zur Kollisionsvermeidung wurde das *Inter-Subjects Testing* verwendet, damit die Ergebnisse der Studie nicht von den Unterschieden zwischen den Probanden beeinflusst werden.

4.2 Durchführung der Studie

4.2.1 Testteilnehmer

Gemäß den Voraussetzungen, welche in Abschnitt 4.1.4 formuliert wurden, wurden Probanden für die Studie ausgewählt und zur Evaluation der Verfahren zur Kollisionsvermeidung herangezogen.

Es nahmen zwölf Personen an der Studie teil, wobei deren Alter zwischen 21 und 28 lag. Drei der Probanden waren weiblich. Alle Testteilnehmer wiesen Erfahrung mit Computerspielen auf, wobei deren bevorzugte Genres unterschiedlich waren. So waren sowohl Anhänger von Echtzeitstrategie- und Rollenspielen anwesend als auch Spieler die Sportsimulationen und *Ego-Shooter* bevorzugten.

Um den Lerneffekt, welcher beim *Inter-Subject Testing* auftreten kann, zu vermeiden, wurden die Testteilnehmer in zwei Gruppen aufgeteilt, welche sich hinsichtlich der Reihenfolge, in der die Versionen des *Levels* gespielt wurden, unterschieden. Die Zuordnung erfolgte per Zufall.

4.2.2 Testumgebung

Die Evaluation der Verfahren zur Kollisionsvermeidung wurde im Büro der *Pro 3 Games GmbH*, in welches die Probanden eingeladen wurden, durchgeführt. Hierfür wurde ein Computer vorbereitet, auf welchem beide Versionen des *Test-Levels* spielbar waren.

Während der Durchführung der Studie wurde auf eine leise Umgebung geachtet, um mögliche Störquellen zu vermeiden. Außer den aktuellen Probanden war nur der Autor dieser Diplomarbeit, der als Testleiter fungierte, anwesend.

4.2.3 Aufbau der Testsitzungen

Wie bereits beschrieben, wurden die Probanden, welche an der Studie teilnahmen, in zwei Gruppen aufgeteilt. Innerhalb dieser wurden per Zufall Paare gebildet, um den Einsatz von *Constructive Interaction* zu ermöglichen. So wurden die Testteilnehmer paarweise zu den Testsitzungen eingeladen. Der Aufbau dieser wird im Folgenden beschrieben.

1. Am Anfang jeder Testsitzung wurden die zwei Probanden durch den Testleiter begrüßt. Dabei wurden sie über die Ziele der Studie sowie die eingesetzten Methoden aufgeklärt. Weiters wurden die Testteilnehmer darauf hingewiesen, dass die Ergebnisse vertraulich behandelt werden und die Sitzung auf Wunsch der Probanden jederzeit abgebrochen werden kann.
2. Nach der Begrüßung folgte eine Einführung in die Funktionsweise und Bedienung des Spiels *Delta Strike*. Danach übernahm ein Proband die Rolle des Spielers, während der andere Testteilnehmer als Beobachter fungierte (vgl. Abschnitt 4.1.3).
3. Die Aufgabenstellung wurde erklärt und eine Version des *Test-Levels* gestartet. Die Probanden wurden aufgefordert die Aufgaben zu lösen und ihre Erfahrungen beim Spielen zu verbalisieren. Der Testleiter machte dabei Notizen und verhielt sich ruhig, um den Test nicht zu beeinflussen.
4. Um negative Effekte durch Ermüdung oder mangelnde Konzentration der Probanden zu vermeiden, wurde eine kurze Pause eingelegt.
5. Analog zum dritten Punkt wurde das *Test-Level* von den Probanden gespielt, wobei diesmal ein anderes Verfahren zur Kollisionsvermeidung zum Einsatz kam.
6. Abschließend wurde ein Interview zum Erfassen der Eindrücke, welche die Probanden beim Spielen der zwei Versionen des *Levels* bekommen haben, durchgeführt.

4.2.4 Pilottest

Nielsen beschreibt in [22, S. 174–175] die Wichtigkeit eines Pilottests. Bei diesem soll der Aufbau einer Studie vor der eigentlichen Durchführung überprüft werden, so dass Probleme, wie beispielsweise zu schwierige Aufgaben, im Vorhinein erkannt und behoben werden können.

Gemäß Nielsen werden nur wenige Probanden für einen Pilottest benötigt. So wurden im Falle dieser Studie zwei Angestellte der *Pro 3 Games GmbH* sowie zwei Personen, welche den in Abschnitt 4.1.4 formulierten Voraussetzungen entsprechen, herangezogen.

Durch die Abwicklung des Pilottests konnten Probleme bei der Gestaltung des *Test-Levels* sowie der Formulierung der Fragen des Interviews ge-

funden werden. Vor der eigentlichen Durchführung der Studie wurden diese behoben.

4.3 Ergebnisse der Studie

Im Folgenden werden die Ergebnisse der Studie vorgestellt. Die Informationen, welche durch den Einsatz von *Constructive Interaction* und die Durchführung der Interviews gewonnen wurden, werden in Abschnitt 4.3.1 beschrieben. Anschließend werden diese in Abschnitt 4.3.2 analysiert und diskutiert.

4.3.1 Auswertung

In diesem Abschnitt werden die Notizen, welche während der Testsitzungen gemacht wurden, ausgewertet. Hierfür wurden die Aussagen der Probanden sortiert und zusammengefasst, so dass ein kompakter Überblick ohne redundante Informationen gegeben werden kann. Dabei wird jedem Verfahren zur Kollisionsvermeidung ein Unterabschnitt, innerhalb dessen separat auf das Einzel- und Gruppenverhalten der Agenten eingegangen wird, gewidmet. Abschließend werden die Meinungen und Präferenzen hinsichtlich der getesteten Verfahren, welche im Laufe der Interviews von den Probanden formuliert wurden, vorgestellt.

Kollisionsvermeidung nach Reynolds

Einzelverhalten: Das Einzelverhalten von Raumschiffen bei der Vermeidung von Kollisionen mit einzelnen Objekten gemäß dem Ansatz von Reynolds war überwiegend zufriedenstellend. So empfand der Großteil der Probanden das Ausweichen als *flüssig* und *ruckfrei*. Dies war insbesondere bei der Vermeidung von Kollisionen mit einzelnen unbeweglichen Hindernissen der Fall, welche von den Testteilnehmern aufgrund des geringen Geschwindigkeitsverlustes beim Ausweichen als *elegant* und *natürlich* bezeichnet wurde.

Allerdings führte die Vermeidung von Kollisionen mit mehreren Objekten zu Problemen. So kam es aufgrund der zufälligen Positionierung der Asteroiden zu Situationen, in welchen ein Agent vor einer Ansammlung von statischen Hindernissen ausweichen musste. In solchen Fällen empfanden manche Testteilnehmer das Verhalten des Agenten als *desorientiert*, da dieser anfing zu zittern und manchmal sogar mit den Asteroiden kollidierte. Da es nur selten zu diesen Situationen kam und diese aufgrund des temporeichen Spielgeschehens kaum wahrgenommen wurden, stellte dieses ungewünschte Verhalten kein großes Problem für die Testteilnehmer dar.

Gruppenverhalten: Das Gruppenverhalten von Einheiten beim Einsatz des Verfahrens von Reynolds wurde von den meisten Probanden bemängelt,

Abbildung 4.2: Verhalten von Gegnergruppen beim Einsatz des Verfahrens von Reynolds.

wobei Kampfsituationen als besonders frustrierend empfunden wurden. So entstanden während der Gefechte Situationen, in welchen die computergesteuerten Gegnergruppen dicht zusammengedrängt waren, was die Probanden als unübersichtlich und *chaotisch* empfanden.Weiters merkten manche Testteilnehmer an, dass das Verhalten der Gegner das Anvisieren von bestimmten Einheiten unmöglich machte. Dies wurde kritisiert, da das Auswählen eines anzugreifenden Ziels eine wichtige Komponente in Echtzeit- sowie Rollenspielen darstellt. Für ein besseres Verständnis wird das beschriebene Problem in Abbildung 4.2 dargestellt.

Neben den Problemen in Kampfsituationen beobachteten die Testteilnehmer auch unerwünschtes Verhalten der Agenten bei Formationswechseln. Anstatt aneinander vorbeizufliegen, kollidierten die meisten Raumschiffe bei diesem Manöver, wie in Abbildung 4.3 gezeigt wird. So machte das Verhalten der Gruppe einen negativen Eindruck auf alle Probanden und wurde auch hier als *chaotisch* beschrieben.

Kollisionsvermeidung mit *Flow Fields*

Einzelverhalten: Das Verhalten von einzelnen Einheiten bei der Kollisionsvermeidung mithilfe von Vektorfeldern machte insgesamt einen positiven Eindruck auf alle Probanden. So empfanden die Testteilnehmer das Vermeiden von Kollisionen als zufriedenstellend, da alle Hindernisse berücksichtigt wurden und es zu keinen Zusammenstößen kam. Dabei hoben manche Probanden hervor, dass es auch bei Ansammlungen von Hindernissen keine Kollisionen auftraten.

Ein Sachverhalt, welcher allen Probanden aufgefallen ist und als störend

Abbildung 4.3: Formationswechsel beim Einsatz des Verfahrens von Reynolds.

empfunden wurde, ist ein vorwiegend ästhetisches Problem, welches hauptsächlich beim Vermeiden von Kollisionen mit unbeweglichen Hindernissen auftrat. Steuerte ein Agent in einem steilen Winkel auf ein Hindernis zu, kam es zu einem Brems- und Beschleunigungsverhalten, welches von den Testteilnehmern als *unnatürlich* beschrieben wurde. So bremste das Raumschiff beim Vermeiden einer Kollision erst stark ab, wich dann aus und beschleunigte außergewöhnlich schnell.

Gruppenverhalten: Alle Probanden waren mit dem Gruppenverhalten von Agenten beim Einsatz von Vektorfeldern zur Kollisionsvermeidung zufrieden. Neben der funktionierenden Vermeidung von Zusammenstößen hoben viele Testteilnehmer die Übersichtlichkeit in Kampfsituationen hervor. So hielten Einheiten immer Abstand zueinander, wodurch auch Gefechte mit einer großen Anzahl von Raumschiffen als übersichtlich empfunden wurden. Abbildung 4.4 zeigt eine solche Kampfsituation.

Auch das Verhalten von Einheiten beim Formationswechsel wurde von den Testteilnehmern als positiv empfunden. Neben der Tatsache, dass es zu keinen Zusammenstößen kam, hoben die Probanden hervor, dass die Manöver *störungsfrei* und *ästhetisch* durchgeführt wurden. Das Verhalten beim Formationswechsel wird in Abbildung 4.5 dargestellt.

Präferenzen der Probanden

Im Laufe der Interviews wurden die Probanden gefragt, welches Verfahren zur Kollisionsvermeidung sie in welcher Situation bevorzugen würden. Dabei konnte eine klare Tendenz festgestellt werden: Während die Probanden das Einzelverhalten der Agenten bei der Verwendung des Verfahrens von Reynolds bevorzugten, präferierten sie beim Gruppenverhalten den Einsatz von *Flow Fields*.

Abbildung 4.4: Verhalten von Gegnergruppen beim Einsatz von Vektorfeldern.

Abbildung 4.5: Formationswechsel beim Einsatz von Vektorfeldern.

Obwohl die Verwendung von *Flow Fields* zu einem Einzelverhalten von Agenten führte, bei welchem alle Hindernisse ohne Zusammenstöße vermieden wurden, empfanden die Testteilnehmer das starke Abbremsen und schnelle Beschleunigen beim Ausweichen vor unbeweglichen Objekten als sehr *störend*. Im Gegensatz dazu fand bei der Kollisionsvermeidung nach Reynolds nur ein geringer Geschwindigkeitsverlust statt, wodurch die Probanden das Verhalten bei diesem Verfahren als *eleganter* wahrnahmen und dieses somit auch präferierten.

Während die Probanden das Gruppenverhalten beim Einsatz des Verfahrens von Reynolds als *chaotisches Durcheinander* empfanden, führte die Kollisionsvermeidung mit *Flow Fields* zu zufriedenstellenden Ergebnissen. So beschrieben die Testteilnehmer sowohl Gefechte als auch Formationswechsel als übersichtlich und frei von Kollisionen. Dadurch wurde der Einsatz von Vektorfeldern in Situationen mit mehreren Einheiten für geeigneter befunden als das Verfahren von Reynolds.

4.3.2 Analyse

Im Folgenden werden die von den Probanden gewonnenen Eindrücke, welche in Abschnitt 4.3.2 vorgestellt wurden, analysiert. Dabei wird die Wirkung von beiden Verfahren zur Kollisionsvermeidung in separaten Unterabschnitten behandelt. Abschließend werden beide Ansätze gegenübergestellt und eine mögliche Hybridlösung, welche die Vorteile beider Verfahren vereint, vorgestellt.

Analyse der Kollisionsvermeidung nach Reynolds

Bei der Evaluation der Kollisionsvermeidung nach Reynolds wurde deutlich, dass die Probleme, welche bei der theoretischen Ausarbeitung bereits beschrieben wurden (vgl. Abschnitt 2.3.4), im praktischen Einsatz auftreten und einen negativen Eindruck auf den Spieler machen.

So kam es in Situationen, in denen sich mehrere Hindernisse – meist Ansammlungen von Asteroiden oder andere Einheiten beim Wechsel der Formation – vor einem Agenten befanden, zu Problemen. Der Grund hierfür ist die Vorgehensweise beim Ansatz von Reynolds, bei welchem nur das *gefährlichste* Hindernis berücksichtigt wird. Dadurch kommt es zu widersprüchlichen Situationen, in denen das Vermeiden eines potenziellen Zusammenstoßes zum Ansteuern eines anderen Hindernisses führt, wodurch der Agent anfängt zu zittern oder schlimmstenfalls mit einem Objekt kollidiert. Dieses ungewünschte Verhalten wurde bereits in Abbildung 2.6 skizziert.

Die negativen Eindrücke, welche die Probanden beim Verhalten von Gegnergruppen erhielten, lassen sich auch mit einem Problem, das bereits beschrieben wurde, begründen. In Kampfsituationen bewegten sich meist mehrere feindliche Einheiten von einer Richtung auf das Raumschiff des Spielers zu. Da beim Verfahren von Reynolds lediglich Hindernisse vor einem Agenten beachtet werden, übersahen die Gegner einander – wie in Abbildung 2.4 dargestellt – und es kam zu Kollisionen.

Analyse der Kollisionsvermeidung mit *Flow Fields*

Im theoretischen Teil dieser Arbeit wurde bereits der Vorteil, welcher bei der Verwendung von *Flow Fields* in Situationen mit vielen Objekten entsteht, erläutert (vgl. Abschnitt 2.5). So können – im Gegensatz zur Vorgehensweise

von Reynolds – alle potenziellen Kollisionen gleichzeitig berücksichtigt werden, da alle Hindernisse eine Abstoßung auf einen Agenten ausüben. Diese Tatsache machte sich auch bei der Evaluation der Kollisionsvermeidung mit *Flow Fields* bemerkbar. Selbst in Situationen mit vielen Hindernissen kam es zu keinen Zusammenstößen, was einen guten Eindruck bei den Probanden hinterließ.

Beim Gruppenverhalten, welches von den Testteilnehmern hauptsächlich aufgrund der Übersichtlichkeit gelobt wurde, zeigte sich ein weiterer Vorteil, der durch den Einsatz von Vektorfeldern entsteht. Während beim Ansatz von Reynolds nur Hindernisse vor einem Agenten berücksichtigt werden, wirkt bei der Verwendung von radialen Repulsionen die Abstoßung einer Einheiten in alle Richtungen. Im *Test-Level* führte dies dazu, dass Raumschiffe in Gruppen immer Abstand zueinander hielten, wie in Abbildung 4.4 gezeigt wurde. Außerdem tritt das Problem von sich übersehenden Einheiten (siehe Abbildung 2.4) bei der Benutzung von Vektorfeldern nicht auf, da die seitlichen Repulsionen zu einem frühen Ausweichen führen.

Allerdings zeigte die Evaluation ein ästhetisches Problem auf, welches beim Einsatz von *Flow Fields* entstehen kann. So bemängelten die Testteilnehmer das Brems- und Beschleunigungsverhalten bei der Kollisionsvermeidung, zu welchem es kam, wenn sich ein Agent in einem steilen Winkel auf ein Hindernis zubewegte. Dabei wirkte vor dem eigentlichen Ausweichen eine starke Bremskraft auf die Einheit. Nach der Vermeidung des Zusammenstoßes kam es durch die abstoßenden Kräfte des Hindernisses zu einer verstärkten Beschleunigung. Für ein besseres Verständnis werden die wirkenden Kraftvektoren in Abbildung 4.6 dargestellt.

Gegenüberstellung und mögliche Synthese

Die durchgeführte Evaluation zeigt auf, dass sich die Annahmen, welche im theoretischen Teil dieser Arbeit beschrieben wurden (siehe Abschnitt 2.5), beim praktischen Einsatz der beiden Verfahren bestätigen. Während die Verwendung der Methode von Reynolds zufriedenstellendes Ausweichverhalten bei der Vermeidung von Kollisionen mit einzelnen Objekten zur Folge hat, führen die Probleme des Ansatzes – hauptsächlich das Übersehen von anderen Einheiten sowie unerwünschtes Verhalten bei vielen Hindernissen (siehe Abschnitt 2.3.4) – zu Störungen beim Spielgeschehen. Diese können durch den Einsatz von *Flow Fields* verhindert werden, wobei dessen Stärken insbesondere in Gruppensituationen zur Geltung kommen. Diese gewinnen durch die Verwendung von Vektorfeldern auch an Übersichtlichkeit, die u. a. in Echtzeitstrategiespielen von großer Wichtigkeit ist.

Um die Vorteile der beiden Verfahren zu verbinden, bietet sich eine Hybridlösung an, bei welcher die Vermeidung von Kollisionen mit Agenten mithilfe von *Flow Fields* umgesetzt wird, während der Ansatz von Reynolds für das Verhindern von Zusammenstößen mit anderen Hindernissen in der

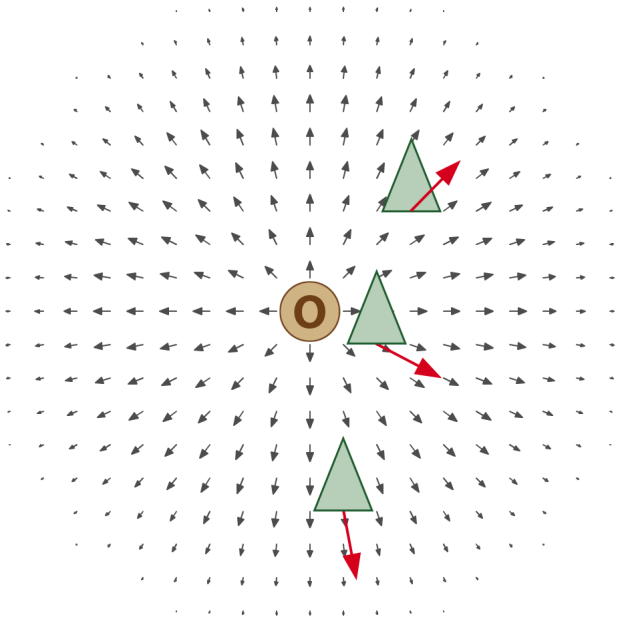


Abbildung 4.6: Veranschaulichung der Kräfte, welche bei der Vermeidung eines Hindernisses an verschiedenen Positionen auf einen Agenten wirken.

Spielwelt eingesetzt wird. Dies könnte unter Zuhilfenahme der bereits implementierten Verhaltensweisen realisiert werden:

- Mithilfe des *obstacle avoidance behavior* könnten Kollisionen mit unbeweglichen Hindernissen, welche hierfür als Kreise angenähert werden müssten (vgl. Abschnitt 2.3.3), verhindert werden.
- Die Verhaltensweise *flow following* könnte benutzt werden, um einen Agenten anhand von Kräften, die durch die Kombination der Vermeidungsfelder der Einheiten in der Spielwelt berechnet werden, zu bewegen.

Anhand der vorgestellten Methoden zur Kombination von Verhaltensweisen (siehe Abschnitt 3.1.2) könnten die Kraftvektoren, welche durch die *behaviors* kalkuliert wurden, zu einem Steuerungsvektor vereinigt werden. Mithilfe dieser Kraft könnte ein Agent unbeweglichen Hindernissen – beispielsweise Asteroiden in *Delta Strike* – gemäß der Vorgehensweise von Reynolds ausweichen, während die Vermeidung von Kollisionen mit anderen Einheiten unter Zuhilfenahme von *Flow Fields* erfolgen würde.

Kapitel 5

Fazit

5.1 Reflexion

Im Bereich der Computerspiele sind Verfahren zur Vermeidung von Zusammenstößen unerlässlich, da diese einen großen Einfluss auf das Spielerlebnis des Spielers haben. Aus diesem Grund lag der Fokus dieser Arbeit auf diesem Teilgebiet der künstlichen Intelligenz in Spielen, wobei zwei Verfahren – der Ansatz von Reynolds, welcher als Industriestandard im Bereich der Videospiele gilt, sowie das vielversprechende *Flow Fields*-Konzept, welches ihren Ursprung im Gebiet der Robotik hat – vorgestellt wurden.

Das Ziel dieser Arbeit war eine Gegenüberstellung dieser beiden Verfahren, welche sowohl theoretisch als auch praktisch durchgeführt wurde. So wurden beide Ansätze im ersten Teil der Arbeit detailliert beschrieben, wobei auf die Funktionsweise sowie die Probleme der Verfahren eingegangen wurde. In der zweiten Hälfte wurde die Implementierung der Ansätze im Computerspiel *Delta Strike* vorgestellt, anhand welcher eine Evaluation mithilfe einer empirischen Studie durchgeführt wurde. Abschließend wurden die Ergebnisse dieser Untersuchung analysiert, wobei der Versuch unternommen wurde, diese mit den Erkenntnissen, welche im theoretischen Teil der Arbeit gewonnen wurden, in Verbindung zu bringen.

Anhand dieser Gegenüberstellung konnten die Stärken und Schwächen der vorgestellten Ansätze in Erfahrung gebracht werden. Während der Einsatz des Verfahrens von Reynolds zu besseren Resultaten beim Einzelverhalten von Agenten führte, erwies sich die Kollisionsvermeidung mit *Flow Fields* in Situationen mit vielen Einheiten als geeigneter.

5.2 Ausblick

Die theoretische und praktische Gegenüberstellung der Verfahren zeigte auf, dass beide Ansätze sowohl positive als auch negative Punkte aufweisen. Um die Stärken beider Verfahren zu verbinden, wurde in Abschnitt 4.3.2 eine

mögliche Hybridlösung vorgeschlagen, welche eine Kombination der beiden Ansätze darstellt.

Die fertige Implementierung der beiden vorgestellten Verfahren im Computerspiel *Delta Strike* bildet eine gute Basis für weitere Forschung in diesem Bereich. So soll in Zukunft an der vorgeschlagenen Hybridlösung gearbeitet werden, wodurch eine Fortsetzung dieser Arbeit nicht ausgeschlossen wird.

Anhang A

Delta Strike

Die erste Version des Computerspiels *Delta Strike* wurde 2005 von Studenten der Fachhochschule Oberösterreich Campus Hagenberg entwickelt, wobei die Schaffung eines Weltraum-Echtzeitstrategiespiels auf Basis von *Adobe Flash* als Motivation diente.

Nach einjähriger Pause wurde die Idee von den ursprünglichen Entwicklern Andrzej Kozłowski, Michael Plank und Alexander Seifert erneut aufgegriffen und das Spiel innerhalb eines Jahres als Studienprojekt von Grund auf neu umgesetzt (vgl. [32]). Das Resultat, welches heute als *Delta Strike Origins* bezeichnet wird, war ein Echtzeitstrategiespiel, bei welchem zwei oder mehr Personen gegeneinander antreten konnten. Das Spiel beinhaltete neben dem Kampf von gruppierten Kampfeinheiten auch genreübliche Elemente wie Ressourcen-Verwaltung und Forschung. Elemente wie Basenbau oder Kollisionsvermeidung wurden jedoch nicht verwirklicht.

Im Jahr 2010 gründeten die drei Personen hinter *Delta Strike Origins* die Firma *Pro 3 Games GmbH*[1]. Seit der Gründung des Unternehmes wird an einer neuen Version von *Delta Strike* gearbeitet, welche ursprünglich eine Neuentwicklung von *Delta Strike Origins* mit zusätzlichen Elementen darstellte. So etwa wurden auch erste, rudimentäre Verfahren zur Kollisionsvermeidung entwickelt.

Mit März 2011 wurde das Spielkonzept von *Delta Strike* gekippt und vom Genre der Echtzeitstrategiespiele in das der Rollenspiele verlagert. An der Steuerung der Einheiten – einzeln als auch in Gruppen – änderte sich jedoch nur die Kamerasteuerung. Diese verbleibt auf die Haupteinheit fixiert, welche die Repräsentation des Spielers in der Spielwelt darstellt und welcher andere Raumschiffe der selben Gruppe in Formation folgen.

Nicht nur das Konzept wurde geändert, sondern auch die verwendete Grafiktechnologie, welche das Spiel durch den Einsatz von *Adobe Flash 11* (sowie dessen Bestandteil *Stage 3D*) von einer zweidimensionalen Anzeige zu einer dreidimensionalen Darstellung beförderte. Doch auch hier verblieb die

[1] http://www.pro3games.com

Steuerung im Grunde gleich, genauso wie auch das Spielgeschehen auf einer zweidimensionalen Ebene gehalten wurde.

Das neue *Delta Strike* soll im Herbst 2011 als *Betaversion* veröffentlicht und von da ausgehend im laufenden Betrieb weiterentwickelt werden. Die Methoden zur Kollisionsvermeidung, welche im Rahmen dieser Arbeit entwickelt wurden, sind dabei fester Bestandteil und werden voraussichtlich aufgrund der Erkenntnisse dieser Arbeit weiterentwickelt und verbessert.

Anhang B

Verfahren zur Raumpartitionierung

Es existiert eine Vielzahl von Ansätzen zur Raumpartitionierung, welche in [13, Kap. 7] grob in drei Typen unterteilt werden – *grids*, *trees* sowie *spatial sorting*. Diese werden im Folgenden beschrieben.

B.1 *Grids*

Bei der Raumpartitionierung mit einem Raster (im Englischen *grid*) wird die Spielwelt in gleich große, quadratische (bzw. im dreidimensionalen Raum würfelförmige) Bereiche unterteilt. Jedes Objekt in der Welt wird den Regionen, in welchen es sich befindet, zugeordnet – bei einer Änderung der Position müssen diese aktualisiert werden. Aufgrund der Einheitlichkeit des Rasters können Zellen, sowie Objekte innerhalb dieser, schnell ausgelesen werden, wodurch die Rechenzeit reduziert werden kann.

Ein Problem, welches bei der Benützung von *grids* zur Raumpartitionierung entsteht, ist das Finden einer geeigneten Größe für die Zellen. Bei zu kleinen Regionen müssen viele Aktualisierungen durchgeführt werden, wenn sich Objekte bewegen. Sind die Zellen allerdings zu groß wird aufgrund der hohen Anzahl an Entitäten pro Region nur wenig Rechenzeit gespart. *Hierarchical Grids* (vgl. [13, Abschnitt 7.2]) stellen eine mögliche Lösung für dieses Problem dar. Bei diesem Ansatz werden mehrere Raster für den selben Raum erzeugt, wobei diese unterschiedlich große Regionen beinhalten.

B.2 *Trees*

Die Partionierung der Spielwelt kann anhand eines Baumes (im Englischen *tree*) durchgeführt werden. Repräsentativ für diese Art der Raumpartitionierung sind *octrees*, welche zum Aufteilen eines dreidmensionalen Raumes verwendet werden.

Dazu wird ein Kubus, welcher an den Achsen ausgerichtet ist, über die Spielwelt gelegt. Dieser stellt den Wurzelknoten des Baumes dar und wird in acht kleinere Würfel unterteilt, welche dessen Unterknoten sind, und selber wiederum in kleinere Kuben aufgespaltet werden. Dieser Vorgang wird rekursiv ausgeführt, bis eine Abbruchbedingung erfüllt wurde. Meist ist diese eine bestimmte Mindestgröße, welche die Würfel nicht unterschreiten dürfen, oder eine maximal erreichbare Tiefe der Baumstruktur.

Jeder Knoten beinhaltet die Objekte, welche im Volumen des dazugehörigen Würfels liegen. Durch die Baumstruktur und dessen simple geometrische Darstellung können Objekte innerhalb einer bestimmten Region schnell gefunden werden.

B.3 *Spatial sorting*

Ein Problem, welches bei der Verwendung der bereits vorgestellten Ansätze zur Raumpartitionierung ensteht, ist die Einordnung von Objekten, welche in mehreren Regionen gleichzeitig liegen. Dies soll durch räumliche Sortierung vermieden werden. Ein möglicher Ansatz ist die *sort and sweep* Methode. Hierbei werden die *Bounding Boxes*[1] aller Objekte in der Spielwelt auf die Achsen projiziert. Für jede Achse wird eine Liste verwaltet, welche die Projektionspunkte in sortierter Reihenfolge beinhaltet. Dies ermöglicht eine schnelle Findung von Objekten in einem bestimmten Bereich.

[1] Eine *Bounding Box* ist ein Rechteck, welches ein Objekt umschließt.

Anhang C

Inhalt der CD-ROM

Format: CD-ROM, Joliet-Format

C.1 PDF-Dateien

Pfad: /

Karpowicz_Michal_2011.pdf Diplomarbeit

C.2 Quellkode

Pfad: /src

testumgebung/ . . . . . Quellkode der Testumgebung

C.3 Sonstiges

Pfad: /vid

SupremeCommander2-Flowfield.mp4 Quelle zu [33]

Literaturverzeichnis

[1] Alexander, B.: *Flow Fields for Movement and Obstacle Avoidance.* In: Rabin, S. (Hrsg.): *AI Game Programming Wisdom 3*, Kap. 3.1, S. 159–172. Charles River Media, Boston, MA, USA, 2006.

[2] Arbib, M. A. (Hrsg.): *The Handbook of Brain Theory and Neural Networks.* MIT Press, Cambridge, MA, USA, 2. Aufl., 2003.

[3] Arkin, R. C.: *Motor schema based navigation for a mobile robot: An approach to programming by behavior.* In: *1987 IEEE International Conference on Robotics and Automation*, S. 264–271, Raleigh, NC, USA, 1987.

[4] Arkin, R. C.: *Behavior-based Robotics.* MIT Press, Cambridge, MA, USA, 1998.

[5] Arndt, H.: *Integrierte Informationsarchitektur: Die erfolgreiche Konzeption professioneller Websites.* Springer-Verlag Berlin Heidelberg, 2006.

[6] Bernhaupt, R.: *User Experience Evaluation in Entertainment.* In: Bernhaupt, R. (Hrsg.): *Evaluating User Experience in Games*, Kap. 1, S. 3–7. Springer London, 2010.

[7] Bourg, D. M.: *Physics for Game Developers.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[8] Bourg, D. M. und G. Seemann: *AI for Game Developers.* O'Reilly Media, 2004.

[9] Buckland, M.: *Programming Game AI by Example.* Wordware Publishing, Inc., Plano, TX, USA, 2005.

[10] Connolly, C. I., J. B. Burns und R. Weiss: *Path Planning Using Laplace's Equation.* In: *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*, S. 2102–2106, Cincinnati, OH, USA, 1990.

[11] Connolly, C. I. und R. A. Grupen: *Applications of Harmonic Functions to Robotics.* Journal of Robotic Systems, 10:931–946, 1993.

[12] Egbert, P. K. und S. H. Winkler: *Collision Free Object Movement Using Vector Fields*. IEEE Computer Graphics and Applications, 16(4):18–24, 1996.

[13] Ericson, C.: *Real-Time Collision Detection*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[14] Grechenig, T., M. Bernhart, R. Breiteneder und K. Kappel: *Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten*. Pearson Studium, München, Deutschland, 2010.

[15] Khatib, O.: *Real-Time Obstacle Avoidance for Manipulators and Mobile Robots*. The International Journal of Robotics Research, 5(1):90–98, 1986.

[16] Kim, J. O. und P. Khosla: *Real-Time Obstacle Avoidance Using Harmonic Potential Functions*. IEEE Transactions on Robotics and Automation, 8(3):338–349, 1992.

[17] Koren, Y. und J. Borenstein: *Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation*. In: *Proceedings of the IEEE Conference on Robotics and Automation*, S. 1398–1404, Sacramento, CA, USA, 1991.

[18] Matthews, J.: *Basic A* Pathfinding Made Simple*. In: Rabin, S. (Hrsg.): *AI Game Programming Wisdom*, Kap. 3.1, S. 105–113. Charles River Media, Inc., Hingham, MA, USA, 2002.

[19] McAllister, G. und G. R. White: *Video Game Development and User Experience*. In: Bernhaupt, R. (Hrsg.): *Evaluating User Experience in Games*, Kap. 7, S. 107–128. Springer London, 2010.

[20] Millington, I.: *Artificial Intelligence for Games*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[21] Murphy, R. R.: *Introduction to AI Robotics*. MIT Press, Cambridge, MA, USA, 2000.

[22] Nielsen, J.: *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[23] Parent, R.: *Computer Animation: Algorithms and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[24] Penton, R.: *Data Structures for Game Programmers*. Premier Press, Cincinnati, OH, USA, 2003.

[25] Plank, M.: *Visuelle komponentenbasierte Spieleentwicklung.* Diplomarbeit, Digitale Medien; FH Oberösterreich – Fakultät für Informatik, Kommunikation und Medien, Hagenberg, Austria, 2008.

[26] Rabin, S.: *Common Game AI Techniques.* In: Rabin, S. (Hrsg.): *AI Game Programming Wisdom 2*, Kap. 1.1, S. 3–14. Charles River Media, Inc., Hingham, MA, USA, 2004.

[27] Reynolds, C. W.: *Flocks, herds and schools: A distributed behavioral model.* In: *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, Bd. 21, S. 25–34, New York, NY, USA, 1987.

[28] Reynolds, C. W.: *Steering Behaviors for Autonomous Characters.* In: *Game Developers Conference*, S. 763–782, San Jose, CA, USA, 1999.

[29] Russell, S. J. und P. Norvig: *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Inc., Englewood Cliffs, NJ, USA, 1995.

[30] Schwab, B.: *AI Game Engine Programming.* Charles River Media, Inc., Hingham, MA, USA, 2004.

[31] Scott, B.: *The Illusion of Intelligence.* In: Rabin, S. (Hrsg.): *AI Game Programming Wisdom*, Kap. 1.2, S. 16–20. Charles River Media, Inc., Hingham, MA, USA, 2002.

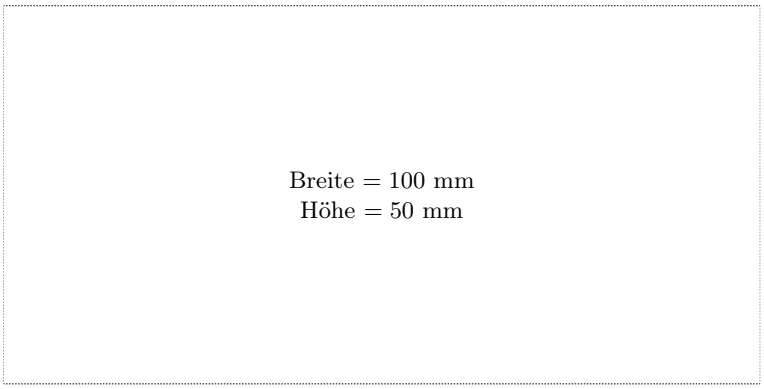
[32] Seifert, A.: *Ein dezentraler Ansatz für computergesteuerte Einheiten in DeltaStrike.* Diplomarbeit, Digitale Medien; FH Oberösterreich – Fakultät für Informatik, Kommunikation und Medien, Hagenberg, Austria, 2008.

[33] *Supreme Commander 2 – Flowfield.* Online Video, 2010. http://www.youtube.com/watch?v=jA2epda-RkM.

[34] Tozour, P.: *The Evolution of Game AI.* In: Rabin, S. (Hrsg.): *AI Game Programming Wisdom*, Kap. 1.1, S. 3–15. Charles River Media, Inc., Hingham, MA, USA, 2002.

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —