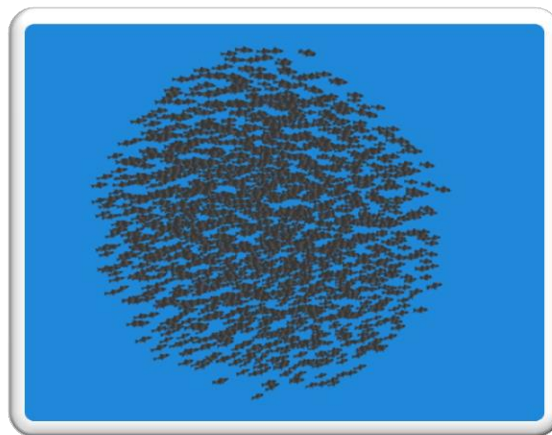
UNIVERSITÄT OSNABRÜCK

**Fachbereich 8: Humanwissenschaften**

# Parallele Simulation und Visualisierung von großen Tierschwärmen mit OpenCL

---

Bachelorarbeit




Oliver Tschesche

Mail: otschesc@uos.de

Matrikelnummer: 939492


Erstprüfender: Prof. Dr. Oliver Vornberger

Zweitprüfender: Prof. Dr. Kai-Uwe Kühnberger

Osnabrück, Juli 2014

# Danksagung

Ich möchte an dieser Stelle allen Leuten danken, die mich bei dieser Arbeit unterstützt haben. Insbesondere meinem Betreuer Hennig Wenke, der mir in allen Belangen geholfen hat und meine Arbeit stets mit sicherem und kritischem Blick betreute.

Auch danke ich meinen Prüfern Prof. Dr. Oliver Vornberger und Prof. Dr. Kai-Uwe Kühnberger, die sich bereit erklärt haben meine Arbeit zu begutachten und diese mit ihrer lehrenden Tätigkeit erst ermöglicht haben.

Zudem danke ich meiner Familie, die mich während meines gesamten Studiums unterstützt hat.

Nicht zuletzt danke ich meiner Freundin, die stets mit Rat und Tat an meiner Seite steht.

# Abstract

Diese Arbeit beschäftigt sich mit der Entwicklung eines Systems zur Simulation und Darstellung von Tierschwärmen. Dabei wird darauf geachtet, dass auch sehr große Schwärme in Echtzeit dargestellt werden können. Das entwickelte System ist selbstverwaltend und kann dynamisch auf äußere Faktoren reagieren. Zur Umsetzung wird hierbei auf eine parallele Berechnung der Simulation per OpenCL gesetzt, die dann mit Hilfe von OpenGL angezeigt wird. Ziel der Arbeit ist es, ein gängiges System zur Schwarmsimulation unter Einbezug vieler Individuen auf Leistungsfähigkeit und Realismus zu testen. Die Auswertung zeigt, dass ein glaubwürdiges Verhalten der Schwarmtiere und eine gute Leistungsfähigkeit erreicht werden.

## Glossar

- **Stigmergie:** Form der Koordination eines dezentralisierten Systems mit vielen Individuen, bei dem nur per Manipulation der Umgebung kommuniziert wird
- **Ameisen-Algorithmus:** Metaheuristisches Verfahren der kombinatorischen Optimierung, das sich an dem Verhalten realer Ameisen orientiert
- **Traveling Salesman Proplem (Rundreiseproblem):** Kombinatorisches Optimierungsproblem, bei dem die kürzeste Route zwischen vielen Orten zu ermitteln ist
- **Boids:** System zur Simulation von Schwarmverhalten
- **N-Body System:** System zur Simulation von Partikelsystemen mit Wechselwirkungen
- **OpenGL:** Programmierschnittstelle für Grafikanwendungen
- **Vertex Shader:** Vom Grafikprozessor ausgeführtes Programm zur Manipulation von Geometrien
- **Fragment Shader:** Vom Grafikprozessor ausgeführtes Programm zur Manipulation von Pixelfarben
- **Wavefront OBJ:** Dateiformat für 3D-Objekte
- **Instancing:** Verfahren zur effizienten Vervielfachung von Objekten
- **OpenCL:** Programmierschnittstelle für parallele Berechnungen
- **OpenCL C:** Programmiersprache für OpenCL
- **Kernel (OpenCL):** Vom Parallelrechner ausgeführtes Programm zur parallelen Berechnung
- **Work-Item / Work-Unit:** Einzelne Recheneinheit eines Parallelrechners
- **Workgroup / tile:** Verbund mehrerer Work-Items; **tile** wird häufig bei Veranschaulichung der Aufteilung mehrerer Workgroups genutzt
- **Host:** Von der CPU ausgeführtes Hauptprogramm; verwaltet **Devices**
- **Device:** Recheneinheit zur Ausführung paralleler Programme
- **Buffer-Objekt:** Temporärer, eindimensionaler Datenspeicher
- **CLMem-Objekt:** OpenCL-Datenformat für Buffer-Objekte
- **CLContext:** Abstrakter Container zur Definierung der Umgebung des Parallelrechners
- **Command Queue:** Objekt zur Regelung der **Host-Device-**Kommunikation

# Inhaltsverzeichnis

1 Einleitung ..... 1

2 Bisherige Arbeiten mit Schwarmintelligenz ..... 2

3 Umsetzung ..... 3

3.1 Handlungsvorschriften ..... 4

3.1.1 Zusatzvorschriften ..... 5

3.1.2 Fressfeind ..... 5

3.2 Simulation ..... 6

3.2.1 N-Body ..... 6

3.2.2 OpenCL ..... 7

3.3 Visualisierung ..... 10

3.4 Gesamtablauf ..... 12

4 Implementation ..... 14

4.1 Simulation ..... 14

4.1.1 Kernelstruktur ..... 14

4.2 Visualisierung ..... 25

4.2.1 Laden der Modelle ..... 25

4.2.2 Ausrichten der Modelle ..... 26

4.3 OpenCL-OpenGL Datenaustausch ..... 29

5 Auswertung ..... 30

5.1 Optische Evaluierung ..... 30

5.2 Parametereinstellungen ..... 37

5.3 Performanz ..... 37

5.4 Generalisierbarkeit ..... 39

6 Fazit ..... 43

7 Erweiterungen ..... 43

8 Quellenverzeichnis ..... 45

9 Abbildungsverzeichnis ..... 47

# 1 Einleitung

Die Tierwelt zeigt eine Vielzahl von Verhaltensweisen und Überlebensstrategien, die sowohl faszinierend, als auch raffiniert sind. Eine dieser Verhaltensweisen ist das Auftreten von Tierschwärmen in der Natur. Diese Schwärme bieten unter anderem Schutz vor Fressfeinden, erleichterte Nahrungssuche und erleichterte Bewegung. Dabei können sich Schwärme aus mehreren Milliarden Individuen bilden[1]. Obwohl dadurch äußerst komplexe Konstrukte entstehen, verwaltet sich der ganze Schwarm, ohne eine übergeordnete Entität, von selbst. Dies ist möglich, indem sich jedes einzelne Tier an gewisse Handlungsvorschriften hält. Folgen alle Tiere denselben Vorschriften, entsteht ohne viel Aufwand für das einzelne Tier eine große, komplexe Formation.

Für diese Arbeit wurde ein System entwickelt, das selbst große Tierschwärme in Echtzeit berechnen und darstellen kann. Es werden die Vorgehensweise und Fallstricke beschrieben, die während der Entwicklung auftreten können.

Das hier entwickelte Programm hat dabei den Anspruch, ohne viel Aufwand, auf möglichst viele verschiedene Tierarten anwendbar zu sein. Als Beispiel dienen dazu je ein Fisch- und ein Vogelschwarm. Diese Schwärme funktionieren vom Grundsatz her gleich, unterscheiden sich in der Natur jedoch deutlich voneinander. Im Folgenden wird aufgezeigt, wie dieses System in der Lage ist durch nur minimale Änderungen den Wechsel der Tierarten zu verwirklichen. Somit zeigt das System eine hohe Generalisierbarkeit und Wiederverwertbarkeit, sodass die Schwarmsimulation auch in andere Anwendungen gut integrierbar ist.

Da die Berechnung vieler Individuen in Echtzeit sehr rechenaufwändig sein kann, liegt ein Schwerpunkt dieser Arbeit auf der parallelen Berechnung der Simulation. Dies wird durch die Auslagerung der Berechnungen der einzelnen Individuen auf die Grafikkarte realisiert, da diese darauf ausgelegt ist viele möglichst gleichartige Berechnungen sehr schnell durchzuführen. Damit ist dieser Ansatz optimal für diese Simulation.

# 2 Bisherige Arbeiten mit Schwarmintelligenz

Tiere haben schon häufig als Vorbild für die Wissenschaft gedient. Dabei haben Tierschwärme bereits vermehrt das Interesse von Forschern geweckt, sodass in diesem Bereich auch einige innovative Ansätze für den Umgang in unserer menschlichen Kultur zu finden sind:

So beschrieb bereits 1959 der französische Zoologe Pierre-Paul Grassé das Konzept der **Stigmergie,** als er Termitenschwärme beobachtete[2]. Er kam zu dem Schluss, dass die Termiten nicht direkt, sondern per Modifikation ihrer Umgebung, miteinander kommunizieren. Dies hat den Vorteil nicht mit jedem anderen Individuum einzeln kommunizieren zu müssen, sondern seine Information an alle weitergeben zu können, die später damit arbeiten müssen. In der Technik hat sich dieses System auch für den Menschen durchgesetzt, wie zum Beispiel die Wiki-Technologie[3] zeigt. Hier erstellt eine Person eine Internetseite und andere Nutzer editieren und erweitern dann das Ergebnis, ohne sich jemals direkt mit dem Ersteller ausgetauscht haben zu müssen. Dadurch ist es möglich, dass viele Individuen mit wenig oder keinem Verwaltungsaufwand gemeinsam an der gleichen Sache arbeiten können.

Ein darauf aufbauendes und prominenteres Beispiel ist der 1991 von Marco Dorigo entwickelte **Ameisen-Algorithmus**[4]. Dieser Algorithmus basiert auf dem Verhalten von Ameisen während der Futtersuche. Dabei legen sie bestimmte Duftstoffe aus, die von anderen Ameisen wahrgenommen werden können. Die Tiere legen dabei also Wege, die vom Nest zu dem Futterplatz führen. Andere Ameisen wiederum nehmen diese Spur wahr, folgen ihr und erneuern dabei den Weg selbst. Da kürzere Wege schneller bewältigt und danach erneut genutzt werden, wird die Geruchsspur auf diesen kurzen Wegen besonders intensiv und somit wiederum von noch mehr Ameisen benutzt. Dadurch entwickeln sich regelrechte Ameisenstraßen, die den effizientesten Weg vom Nest zum Futterplatz darstellen. Der Ameisen-Algorithmus nimmt sich dieses Prozedere zum Vorbild, um damit unter anderem das berühmte **Traveling-Salesman** Problem zu lösen. Der Algorithmus findet zwar nicht immer die optimale Lösung, hat jedoch eine sehr gute Trefferquote und findet in jedem Fall zumindest ein gutes Ergebnis.

Auch auf Menschen selbst kann man Schwarmmodelle anwenden. So entwickelten Moussaïd et al.[5] 2011 ein System, um die Bewegungen von Menschen in Menschenmassen vorauszuahnen. Sie gingen dabei von zwei einfachen Grundsätzen aus: Erstens bewegen sich Menschen, mit Einbezug von Hindernissen, möglichst direkt zu ihrem Ziel, um Umwege zu vermeiden. Zweitens halten sie einen ausreichend großen Abstand zum Vordermann ein, der es ihnen ermöglicht rechtzeitig auszuweichen, sollte dies nötig sein. Mit diesen beiden Grundsätzen war es den Autoren möglich ein sehr präzises Modell zu erzeugen. Dadurch

kann zum Beispiel bei etwaigen Bauprojekten frühzeitig erkannt werden, ob es zu möglichen Engpässen kommen kann.

Das in dieser Arbeit vorgestellte System baut auf dem von Craig Reynolds 1986 entwickelte **Boids**-System[6] auf, das das Verhalten von Schwarmtieren mittels drei Regeln beschreibt: Kohäsion, Separation und Ausrichtung. Dieses System wird im Folgenden noch näher erklärt. Das Boids-System wurde bereits häufig verwendet und findet sich bereits in vielen Anwendungen und auch Medien, wie beispielsweise in Videospielen (z.B. „Half Life“[7], 1998) oder Kinofilmen (z.B. „Batman Returns“[8], 1992).

Das Boids-System wurde bereits häufig in simplen Anwendungen implementiert. So hat Reynolds selbst auf seiner Homepage eine Online-Applikation[8], die dieses System demonstriert. Auch gibt er eine Liste von Implementationen seines Systems an[9]: So gibt es etwa 3D-Anwendungen von Conrad Parker und Martin Rosvall[10], ebenso wie von Ishihama Yoshiaki[11]. Auch 2D-Anwendungen, wie die von Task Gouda[12] mit kalibrierbaren Einstellungen, lassen sich finden. Diese Anwendungen haben jedoch gemein, dass sie nur eine relativ kleine Anzahl an Tieren verarbeiten können. Giliam de Carpentier entwickelte 2006 seinen „fast boid simulator“[13], bei dem er die Berechnungen auf drei Nachbartiere (die zwei Nächsten und das Entfernteste im Sichtradius) beschränkte, wodurch er eine bessere Performanz erhielt, die jedoch die Genauigkeit der Simulation beeinträchtigte.

# 3 Umsetzung

Zur Realisierung des Schwarmmodells ist das Modell von Reynolds aus mehreren Gründen sinnvoll: Dieses Modell weist neben seiner intuitiven Herangehensweise übersichtliche Handlungsvorschriften auf, nach denen sich die Individuen bewegen. Es lässt viel Spielraum für neue Vorschriften und Kalibrierungen der Parameter, sodass man ein möglichst optimales Ergebnis bekommt. Auch ist es dadurch gut möglich das Modell zu variieren, sodass man ohne große Änderungen schnell zwischen verschiedenen Arten von Schwärmen wechseln kann. Dadurch wird das ganze Modell sehr flexibel und gut modifizierbar.

## 3.1 Handlungsvorschriften

Reynolds‘ Modell baut primär auf drei simplen Handlungsvorschriften auf:

1. **Kohäsion:** Das Konzept der Kohäsion beschreibt den Drang möglichst nahe bei den Nachbarn zu bleiben. Es ergibt sich also eine Bewegungsrichtung zum Mittelpunkt der anderen Individuen im Betrachtungsradius.
2. **Separation:** Die Separation ist die der Kohäsion entgegengesetzte Kraft, die verhindert, dass sich die Individuen zueinander bewegen, um, unter anderem, Kollisionen zu verhindern und Bewegungsfreiheit zu ermöglichen. Somit ergibt sich eine Kraft, die von den anderen Individuen wegführt.
3. **Ausrichtung:** Die Ausrichtung gibt vor, in welche Richtung sich das Individuum orientiert. Dabei wird die eigene Richtung mit denen der Nachbarn verglichen und sich dann daran ausgerichtet. Somit richten sich alle Individuen aneinander aus und bewegen sich in die gleiche Richtung.

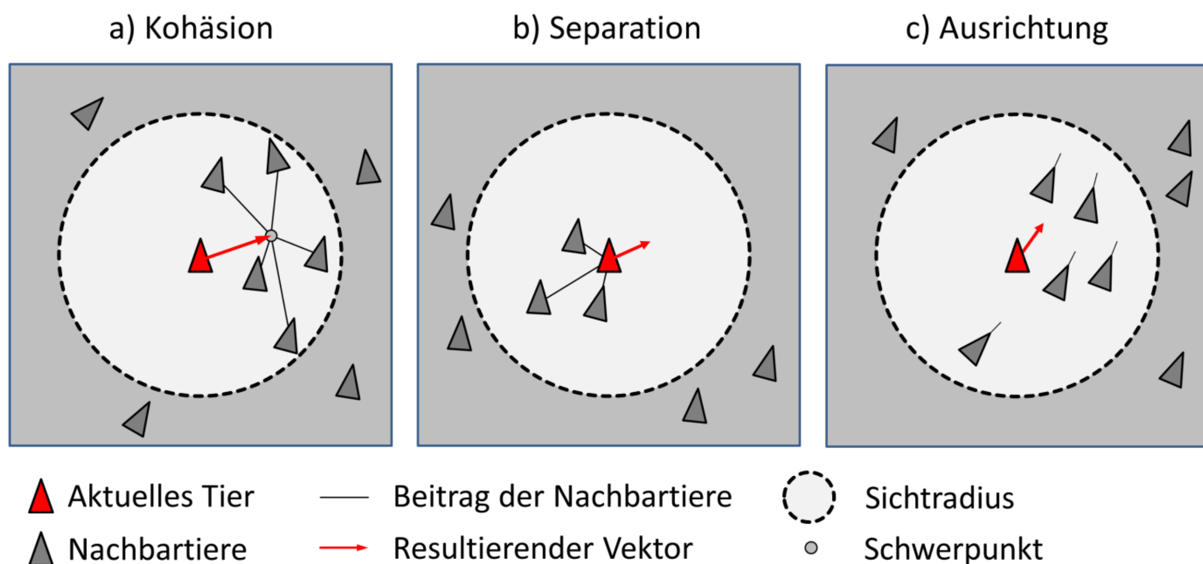



**Abbildung 1: Primäre Handlungsvorschriften des Boids-Systems**

Diese Vorschriften verdeutlichen bereits gut, worauf das einzelne Individuum im Schwarm achten muss, um die Schwarmbildung erst zu ermöglichen. Auch wenn diese Regeln zunächst recht trivial wirken, zeigt sich jedoch schnell, dass dadurch mit sehr geringem Aufwand ein sehr effektives Gebilde zustande kommt. Diese Emergenz macht Schwarmverhalten so erfolgreich und sichert letztendlich das Überleben des einzelnen Individuums.

Zur Berechnung der Bewegungsgeschwindigkeiten wird nicht der gesamte Schwarm betrachtet, sondern nur ein gewisser Teil, welcher sich in unmittelbarer Nähe zum Individuum befindet. Dazu besitzt jedes Tier einen Sichtradius. Nur die Tiere in diesem Radius werden in die Berechnung einbezogen. Der Einfachheit halber wird dabei von der Prämisse ausgegangen, dass die Tiere einen 360° Sichtradius haben, was die Berechnung deutlich reduziert, ohne das Ergebnis spürbar zu verändern.

### 3.1.1 Zusatzvorschriften

Zusätzlich zu den drei primären Handlungsvorschriften werden diverse „kleinere" Regeln eingeführt, die die Simulation glaubwürdiger machen und das Modell stabilisieren. Dabei halten sich diese an möglichst realistische Vorgaben, die die physikalischen Möglichkeiten und Grenzen der Tiere darstellen sollen:

1. **Minimalgeschwindigkeit:** Da sowohl Vögel eine minimale Geschwindigkeit zum Fliegen brauchen und auch Fische sich bewegen müssen, damit die Kiemenatmung einsetzt, wird für jedes Tier eine Geschwindigkeit definiert, die die Tiere mindestens einhalten müssen.
2. **Maximalgeschwindigkeit:** Jedes Individuum besitzt eine Geschwindigkeit, die es maximal erreichen kann, wodurch sowohl die physischen als auch physikalischen Grenzen simuliert werden.
3. **Maximaldrehgeschwindigkeit:** Analog zur Maximalgeschwindigkeit gibt es Grenzen, wie schnell sich ein Tier drehen kann. Da sich in der Simulation sehr plötzliche Richtungsänderungen ergeben können, gibt es auch hier eine Begrenzung, damit die Tiere sich nicht zu schnell drehen.

### 3.1.2 Fressfeind

Um das Schwarmverhalten nicht nur im Normalfall, sondern auch unter Einbeziehung äußerer Umstände, zu simulieren, wird zusätzlich ein optional zuschaltbarer Fressfeind, im Folgenden auch als „Jäger" bezeichnet, implementiert. Dieser weist selbst nur ein rudimentäres Verhalten auf und soll lediglich das Verhalten des Schwarms variieren.

Der Jäger bewegt sich in einer Kreisbahn grob um den Schwarm herum. Durch die stetige Bewegung des Schwarms kreuzen sich die Wege der beiden Parteien regelmäßig. Tiere, die sich in einem gewissen Umkreis des Jägers befinden, verändern dabei ihr Verhalten. Es ergibt sich eine zusätzliche Bewegungsrichtung, die vom Jäger wegführt und somit den Fluchtinstinkt simuliert. Außerdem verändert die Anwesenheit des Jägers die Gewichtung

der anderen Handlungsvorschriften. So werden die Kohäsion und die Ausrichtung verstärkt, während die Separation abgeschwächt wird. Dies soll das Zusammenrottungsverhalten im Angesicht einer akuten Bedrohung verstärken. Auf Grund dieser Bedrohungssituation werden auch zusätzlich die Geschwindigkeiten der Individuen über die Maximalgeschwindigkeit hinaus erhöht.

## 3.2 Simulation

Die Grundlage für die Berechnungen des Schwarms bietet das sogenannte **N-Body System**[14], welches im Folgenden noch genauer erklärt wird. Dieses System bietet maximale Genauigkeit und somit Authentizität, auch wenn dies mit einigem Rechenaufwand verbunden ist. Eine alternative Herangehensweise an die Simulation wäre, den Schwarm als einzelnes Objekt zu definieren und als Ganzes bewegen und auf äußere Einflüsse reagieren zu lassen. Das würde den Rechenaufwand deutlich reduzieren, jedoch nicht adäquat die innere Mechanik eines Schwarms simulieren, sondern nur den Schein eines Schwarmgebildes erzeugen. Für Anwendungen bei denen Schwärme nur ein kleiner Teil der Welt sind, mag dies genügen. Um komplexere Wechselwirkungen unter den einzelnen Individuen und damit das Schwarmverhalten als Ganzes zu simulieren, ist es jedoch nicht geeignet.

### 3.2.1 N-Body

Das wohl weitverbreitetste System für die Simulation von vielen Entitäten mit Wechselwirkungen ist das N-Body System. Es findet vor allem Anwendung in Systemen, bei denen sehr viele identisch handelnde Objekte simultan berechnet werden müssen. Bei dem N-Body System handelt es sich um ein System, bestehend aus vielen einzelnen Körpern („Bodies“), denen eine gemeinsame Datenstruktur zugrunde liegt. Die Anzahl der Körper ist dabei sehr variabel, erreicht aber bei dem N-Body System auf Grund der hohen Komplexität ($O(n^2)$) bei sehr vielen Körpern schnell seine Grenzen.

Die gemeinsame Datenstruktur der Körper hilft jedoch dabei, einen möglichst hohen Parallelisierungsgrad zu erreichen, da alle Körper parallel zueinander berechnet werden können und somit den Rechenaufwand minimieren. Zudem kann man dadurch relativ simple Handlungsvorschriften definieren, die dann für jedes Objekt einzeln ausgeführt werden. Damit lassen sich komplexe Systeme mit verhältnismäßig einfachen und intuitiven Anweisungen definieren und ausführen.

Die Berechnung der Simulation wird dabei mit Hilfe der Geschwindigkeits- und Richtungsvektoren der Körper und deren Positionen realisiert. In dieser Arbeit werden dazu nur die Daten der Körper (bzw. Tiere) in die Berechnung mit einbezogen, die sich in einem gewissen Radius zu dem zu berechnenden Körper befinden. Die Daten werden den Handlungsvorschriften entsprechend, wie folgt in den Berechnungen verwendet:

1. Zur Berechnung der Kohäsion werden die Positionen (dargestellt durch den Mittelpunkt der Körper) der benachbarten Tiere ermittelt und dann der Schwerpunkt berechnet. Daraus wird mit Hilfe der eigenen Position ein Richtungsvektor berechnet, der zum Schwerpunkt der benachbarten Tiere zeigt. Auch wird in diesem Schritt der Mittelpunkt des Systems mit einberechnet, um den Schwarm in der Mitte des Systems zu halten und ihn nicht „davonschwimmen“ zu lassen.
2. Die Separation wird berechnet, indem die Vektoren von den Nachbarn zu dem aktuellen Tier berechnet werden. Diese werden aufsummiert und ergeben einen Gesamtvektor, der den größtmöglichen Abstand zu den Nachbarn gewährleistet.
3. Zu Berechnung der Ausrichtung werden die Richtungsvektoren der Nachbarn gemittelt und dann der Durchschnitt mit dem eigenen Richtungsvektor gebildet. Dadurch ergibt sich ein Vektor, der in sich an den Vektoren der Nachbarn orientiert.
4. Ist der Jäger präsent, wird zudem der Vektor vom Jäger zum aktuellen Tier berechnet.

Diese Vektoren werden unterschiedlich gewichtet und aufsummiert (wie später noch detaillierter erläutert wird). Danach wird der dadurch entstandene Vektor bei Bedarf entsprechend der Maximal- und Minimalgeschwindigkeit gestreckt oder gestaucht. Auch wird überprüft, ob der Winkel zwischen dem neuen und dem alten Richtungsvektor zu groß ist und eventuell dementsprechend noch einmal angepasst werden muss, um die Maximaldrehgeschwindigkeit einzuhalten. Dadurch ergibt sich ein Gesamtrichtungsvektor in dessen Richtung sich das Tier dreht und fortbewegt. Diese Berechnungen finden in jedem Simulationsschritt statt und werden parallel für jedes Tier einzeln ausgeführt.

### 3.2.2 OpenCL

Zur Berechnung des Systems in Echtzeit werden die Berechnungen parallel auf der Grafikkarte durchgeführt. Zu diesem Zweck wird die freie Schnittstelle **OpenCL**[15] (Open Computing Language) der **Khronos Group**[16] verwendet. OpenCL ist eine universelle Schnittstelle, um den Grafikprozessor eines PCs auch außerhalb von Grafikanwendungen zu

nutzen. Somit kann man Berechnungen auf die Grafikkarte auslagern, was den Vorteil mit sich bringt, die hoch parallele Arbeitsweise von Grafikprozessoren zu nutzen. In Verbindung mit dem N-Body System lässt sich dadurch sehr effizient auch eine große Anzahl von Individuen in Echtzeit simulieren.

2007 veröffentlichte **Nvidia**[17] die Schnittstelle **CUDA**[18], die es Entwicklern ermöglicht ihre Grafikprozessoren auch für andere Berechnungen zu nutzen, um von deren parallelen Arbeitsweisen zu profitieren. Da CUDA jedoch nur für Nvidia Grafikprozessoren funktioniert, begann daraufhin die Firma **Apple**[19] an einer offenen Schnittstelle zu arbeiten, die für alle möglichen Parallelrechner implementierbar sein soll. Diese Idee wurde dann an Mitglieder der Khronos Group weitergegeben und mit deren Hilfe weiterentwickelt. Die Khronos Group veröffentlichte dann die erste OpenCL-Version 2008 und hatte sich damit zum Ziel gesetzt, eine freie Schnittstelle für Parallelrechner zu etablieren, die für alle Systeme mit mehreren, parallel arbeitenden Recheneinheiten anwendbar ist. OpenCL bietet Hardware-Herstellern somit die Möglichkeit, ihre Geräte für Programmierer ansprechbar zu machen. Dazu muss der Hersteller das OpenCL-Interface implementieren, womit Anwender dieses nutzen können. Dabei gibt es das Interface für alle gängigen Programmiersprachen, sodass OpenCL eine universelle Schnittstelle darstellt, um parallele Berechnungen auf entsprechender Hardware durchzuführen.

Für OpenCL müssen sogenannte **Kernel** definiert werden, die all die Berechnungen enthalten, die parallel auf der Grafikkarte ausgeführt werden sollen. Es wird dann für jedes einzelne Tier eine Kernelinstanz, also eine Ausführung des Kernels mit den Werten eines spezifischen Tieres, erstellt, in der dann die Berechnungen für das Individuum stattfinden. Dazu werden die Daten vom Hauptprogramm in einzelne Buffer-Objekte gespeichert, die dann die Daten von allen Tieren in Form von langen Arrays enthalten. So gibt es ein Objekt, in dem die Positionen aller Tiere hintereinander gespeichert sind und eines, in dem alle Richtungsvektoren gespeichert sind. Um Schreibe-Lese-Konflikte zu vermeiden, gibt es diese Objekte zudem in zweifacher Ausführung: eines für die Positionen und Richtungen des letzten Simulationsschrittes zum Lesen der Daten und eines zum Schreiben für die neuen Daten.

Um die Berechnungen effizienter zu gestalten, kann bei OpenCL zusätzlich auf sogenannte **Workgroups** zurückgegriffen werden. Da Lesezugriffe der Grafikkarte auf die Daten bei einer sehr großen Anzahl von Individuen sehr rechenintensiv ausfallen können, ist es ratsam die Lesezugriffe zu reduzieren. Objekte innerhalb einer Workgroup, sogenannte **Work-Items** oder auch **Work-Units**, teilen sich einen gemeinsamen Speicher, auf den alle Objekte dieser Workgroup zugreifen können. So kann jedes Objekt der Workgroup einen Teil der Gesamtdaten lesen, in den lokalen Speicher schreiben und somit den anderen Objekten zur Verfügung stellen. Dadurch wird der Aufwand zum Laden der Daten für das einzelne

Work-Item um die Anzahl der in der Workgroup befindlichen Work-Items reduziert. Da in dieser Simulation immer alle anderen Individuen geladen werden müssen, ist diese Methode sehr nützlich.

Das nachfolgende Diagramm (Abbildung 2) verdeutlicht dabei, wie das Speichermodell von OpenCL aussieht. Der **Host** stellt dabei das Hauptprogramm dar und wird somit von der CPU berechnet und hat seinen eigenen Speicher (wie beispielsweise den RAM). Das Hauptprogramm liefert dem **Device**, also dem Parallelrechner (in den meisten Fällen eine Grafikkarte), die Daten, die zunächst in dem globalen Speicher abgelegt werden. Auf diesen können nun alle Work-Units zugreifen. Da Lesezugriffe auf diesen Speicher jedoch sehr langsam sind, sollten diese minimiert werden. Dazu kann von den lokalen Speichern der Workgroups Gebrauch gemacht werden. Alle Work-Items in einer Workgroup haben schnellen Zugriff auf diesen gemeinsamen Speicher. Somit kann jedes Work-Item einen Teil der Daten laden und in den lokalen Speicher schreiben. Damit haben dann alle Work-Items der Workgroup Zugriff auf die Daten, ohne den gesamten Datensatz selbst geladen haben zu müssen. Zusätzlich hat jede Work-Unit noch ihren eigenen privaten Speicher, der nur von ihr selbst genutzt werden kann und den schnellsten Zugriff erlaubt.

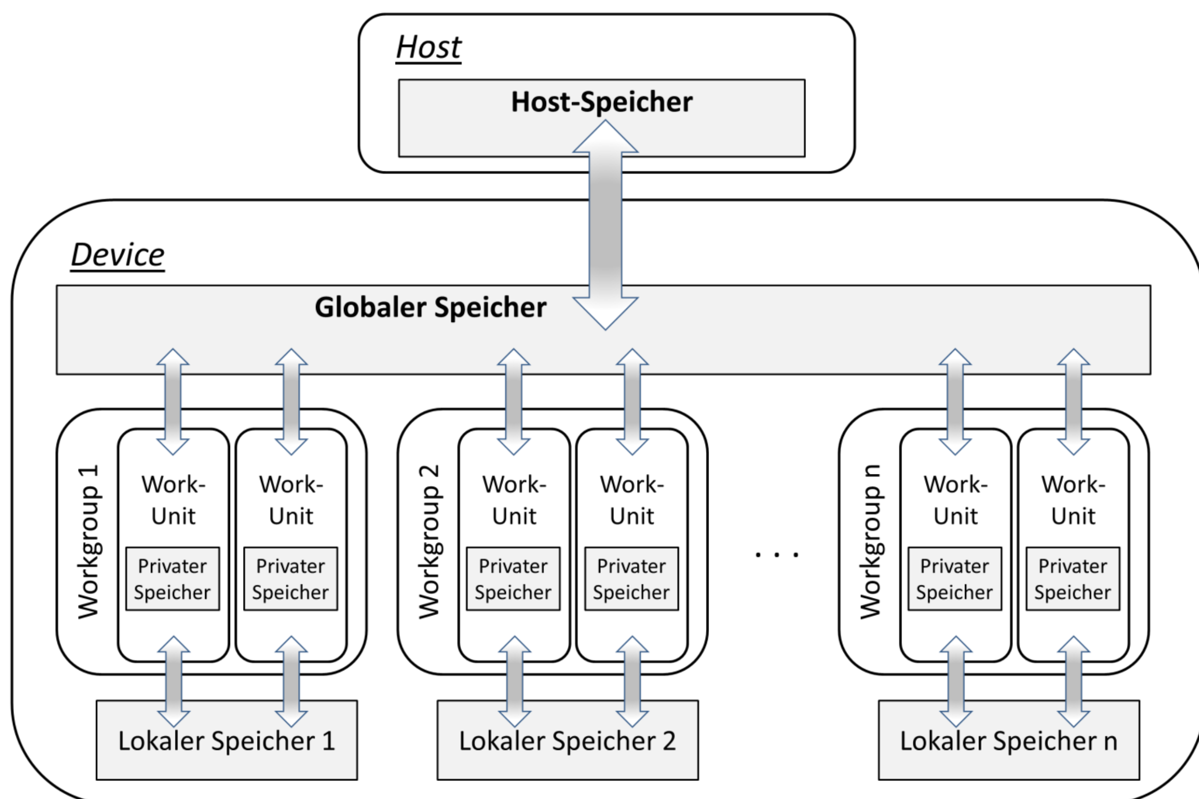



**Abbildung 2: Speichermodell von OpenCL**

## 3.3 Visualisierung

Zur Visualisierung des Schwarms wird eine 3D-Anzeige verwendet, die auf **OpenGL**[20] (Open Graphics Library) basiert. So wie OpenCL, wird auch OpenGL von der Khronos Group entwickelt und ist eine offene Grafikbibliothek zur Erstellung eigener Grafikanwendungen.

Die Simulation findet in einem 3-Dimensionalem Raum statt, in dem sich die Tiere frei bewegen können. Daher brauchen die Tiere neben ihrer Position im Raum auch einen Körper. Dieser wird durch ein Drahtgittergeflecht dargestellt, welches mit einer Textur überzogen wird, um somit ein möglichst gutes, aber rechenkostensparendes Modell des Tieres zu bekommen. Die in dieser Simulation verwendeten Modelle (Abbildungen 3 und 4) und Texturen sind von Hobby-Designern frei zur Verfügung gestellt worden und erhältlich über die Seite *„tf3dm.com“* [21].

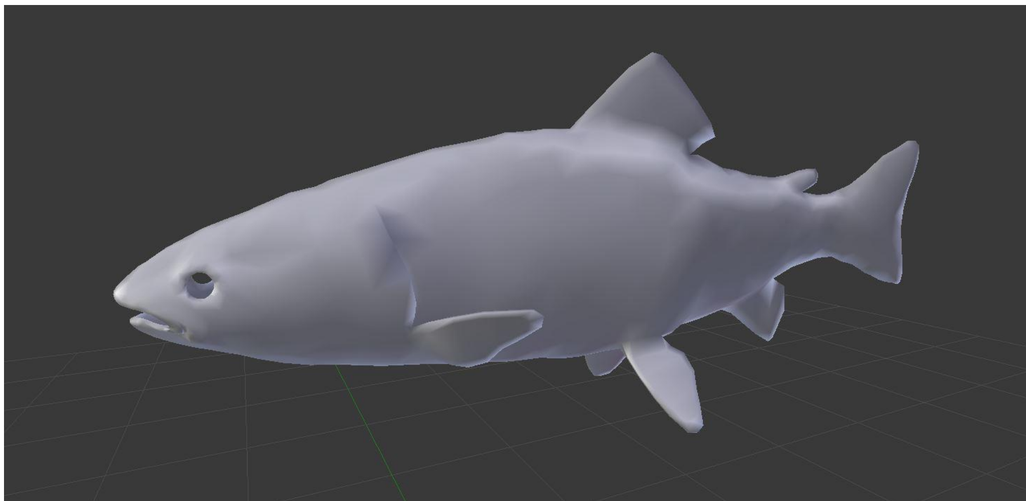


**Abbildung 3: Verwendetes Fischmodell[22]**

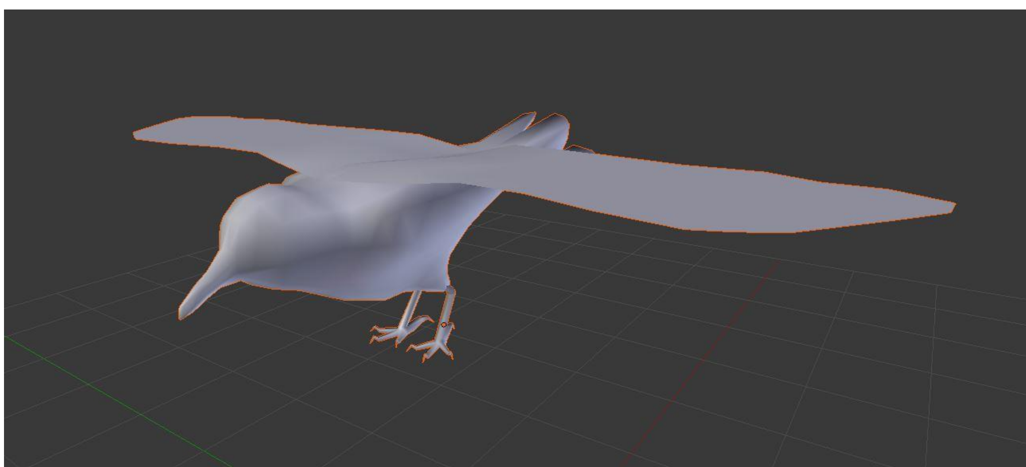


**Abbildung 4: Verwendetes Vogelmodell[23]**

Der Weg von den Modelldaten zum fertigen Modell läuft in vier Schritten ab (Abbildung 5). Diese werden von sogenannten **Shadern** auf der Grafikkarte berechnet. Da viele dieser Schritte unabhängig voneinander sind, kann hier eine hohe Parallelität erreicht werden. Dies ist auch der Grund, warum Grafikkarten überhaupt erst auf eine parallele Arbeitsweise ausgelegt wurden. Zunächst liegen die Modelldaten in Ursprungsvektor-Form vor. An den Stellen, auf die die Vektoren verweisen, werden dann später Punkte erzeugt.

Die Vektoren können in sogenannten **Vertex-Shadern** verändert werden, die vom Programmierer selbst geschrieben werden müssen. Im einfachsten Fall gibt man einfach die Vektordaten weiter in den nächsten Verarbeitungsschritt. Dies reicht in den meisten Anwendungen jedoch nicht aus und so kann man hier noch die Modelle manipulieren, indem man die Vektoren dreht, streckt oder staucht. Dazu macht man sich mathematische Umformungen von Vektoren zunutze, indem man Umwandlungsmatrizen auf die Vektoren anwendet. So kann zum Beispiel eine Rotationsmatrix die Punkte um den Mittelpunkt des kartesischen Koordinatensystems drehen, was bei Anwendung auf alle Punkte zur Folge hat, dass sich auch die Modelle drehen. Sofern sich das Zentrum des Modells im Ursprung des Systems befindet, dreht sich dabei das Modell ohne sich von der Stelle zu bewegen, wodurch eine Ausrichtung in eine bestimmte Richtung möglich ist. Mittels Translationsmatrizen hingegen kann man die Modelle im Raum bewegen, sodass aus Modellkoordinaten Weltkoordinaten werden. Vereint man diese beiden Matrizen, kann ein realistisches Bewegungsmuster simuliert werden.

Die modifizierten Vektoren werden nun im nächsten Schritt in Punkte umgewandelt, die dann mittels Triangulation in ein Gittermodell verwandelt werden. Diese Schritte sind von außen nur konfigurierbar, laufen ansonsten aber nach einem vordefinierten Muster ab. Jedes so entstandene Dreieck kann anschließend in dem **Fragment-Shader** eingefärbt werden, welcher wieder selbst programmiert werden muss. Mittels Texturkoordinaten, die vorher jedem der Punkte zugewiesen wurden, kann nun eine 2D-Textur um das 3D-Modell „gelegt werden“, damit die Oberfläche des Modells die gewünschte Optik erhält. Somit wird aus den vorhandenen Daten ein komplettes Modell erzeugt.

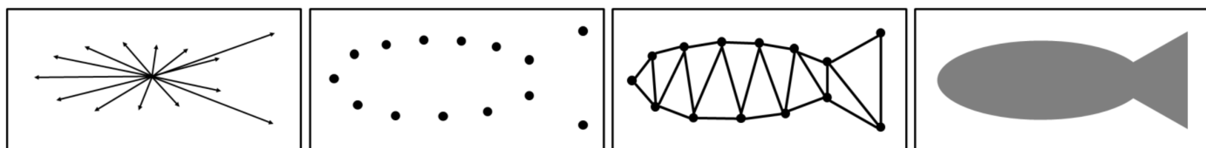


**Abbildung 5: Entstehungsweg eines Modells:**

**(v.l.n.r.) Vektormodell, Koordinatenmodell, Drahtgittermodell, Finalmodell**

## 3.4 Gesamtablauf

Zur Verdeutlichung des Programmablaufs wird dieser im Folgenden näher beschrieben. Zunächst initialisiert das Hauptprogramm die Umgebungsvariablen: Die OpenCL- und OpenGL-Umgebungen werden definiert und konfiguriert. Dann werden die Daten-Buffer erstellt und mit Initialdaten gefüllt. Dabei wird auch ein Buffer mit den Modelldaten, die vorher aus einer separaten Datei gelesen wurden (dies wird später noch näher erläutert), befüllt und an die Vertex-Shader gegeben. Auch die Texturen werden geladen und an die Fragment-Shader weitergereicht.

Nach der Initialisierung kann der eigentliche Programmablauf gestartet werden. Dabei werden die Positions- und Geschwindigkeits-Buffer zunächst an die OpenCL-Umgebung übergeben. Dort wird dann für jedes Tier (also jedem Objekt des N-Body Systems) eine Kernelinstanz erzeugt. Jede Instanz erhält dabei vom Hauptprogramm einen Wert zur Zeitmessung und die Position des Jägers, sofern dieser aktiviert ist. Jede Kernel-Instanz berechnet dann einen Simulationsschritt für je ein Tier nach folgendem Ablauf (wobei hier Geschwindigkeit auch für die Bewegungsrichtung steht):

- Lade Positionen der anderen Tiere
- Berechne Einfluss der Nachbartiere auf das eigene Tier
- Modifiziere den Einfluss, falls der Jäger in der Nähe ist
- Verrechne den Einfluss der Nachbartiere mit der letzten eigenen Geschwindigkeit
- Füge den direkten Einfluss des Jägers, falls präsent, der Geschwindigkeit hinzu
- Verrechne neue Geschwindigkeit mit alter Position
- Schreibe neue Geschwindigkeit und Position

Diese Berechnungen finden in jedem Simulationsschritt und für alle Kernel-Instanzen parallel statt. Nachdem die neuen Werte in die Daten-Buffer geschrieben wurden, werden diese nun an die OpenGL-Umgebung weitergegeben. Dort werden für alle Datenpunkte je eine Vertex-Shader-Instanz erstellt, die ebenfalls parallel berechnet werden können. Der Vertex-Shader arbeitet nach folgendem Prinzip:

- Ermittle Rotationswinkel aus dem Geschwindigkeitsvektor (einer pro Modell)
- Erstelle aus dem Rotationswinkel Rotationsmatrizen
- Wende Rotationsmatrizen auf Modell-Daten an
- Verschiebe Modell-Daten um die Vektoren aus dem Positions-Buffer (einer pro Modell)
- Reiche neue Positionen weiter

Die neuen Positionen werden dann, wie vorher beschrieben, an die nächsten Verarbeitungsschritte weitergereicht, um aus ihnen fertige Modelle mit Textur zu erzeugen. Anschließend wird daraus das Gesamtbild erzeugt und zur Anzeige an den Bildschirm geschickt.

Danach werden die Daten-Buffer zur Berechnung des nächsten Simulationsschrittes wieder an OpenCL gereicht. Der beschriebene Simulationsablauf wird in dem folgenden Pseudo-Code (Abbildung 6) und der dazugehörigen Grafik (Abbildung 7) veranschaulicht. Dabei werden die Tiere nur durch ihre Position und Geschwindigkeit (bzw. Richtung) dargestellt und alle Berechnungen finden mit diesen beiden Attributen statt.

```
while(Simulation_in_Progress)

    for(each animal) in parallel do
        for(each other_animal)

            load(other_animal)
            if(in_range(animal, other_animal))
              influence ← calculate_influence(other_animal, animal)
            end if
        end for
        influence ← update_Influence(hunter, animal)
        update_velocity(animal.velocity, influence)
        update_position(animal.position, animal.velocity)
    end for

    for(each animal) in parallel do
        angle ← calculate_rotation_angle(animal.velocity)
        matrix ← calculate_rotation_matrix(angle)
        rotate_animal(animal.model, matrix)
        shift_animal(animal.model, animal.position)
        draw(animal)
    end for
end while
```

**Abbildung 6: Pseudo-Code des Programmablaufs**

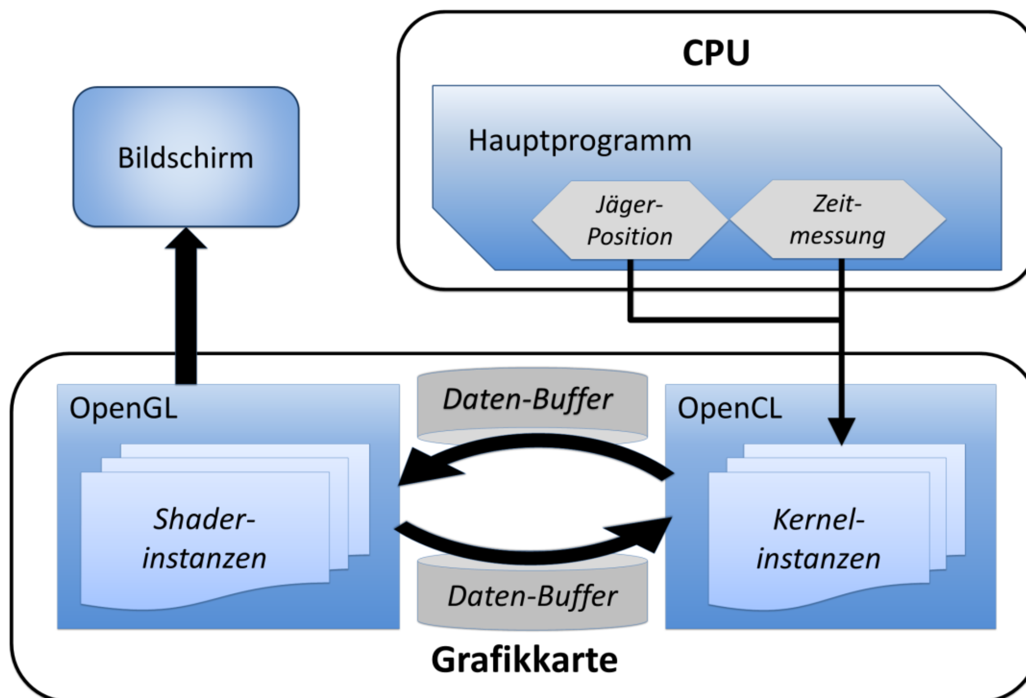



**Abbildung 7: Datenfluss während der Programmausführung**

# 4 Implementation

Im Folgenden wird nun erläutert, wie das vorgestellte System implementiert wurde. Dabei wird lediglich auf die relevanten Aspekte des Programms eingegangen, da eine komplette Beschreibung der Implementation den Umfang dieser Arbeit übersteigen würde.

## 4.1 Simulation

### 4.1.1 Kernelstruktur

Den Kern der Simulation stellt der Kernel dar, der für die Berechnungen jedes einzelnen Tieres verantwortlich ist. Er beinhaltet die allgemeinen Rechenvorschriften, die dann für jedes Individuum interpretiert werden müssen. Zum besseren Verständnis wird im Folgenden grob die Struktur des Kernels, wie er in dieser Implementierung vorkommt, beschrieben. Der Kernel wird dabei in der OpenCL eigenen Syntax **OpenCL C** verfasst, die auf der Programmiersprache **C** (ISO C99) basiert.

Ein OpenCL-Kernel muss stets mit dem Schlüsselwort **kernel** beginnen. Dadurch kann dieser per OpenCL kompiliert und später auf der Grafikkarte ausgeführt werden. Der Kernel ist dabei wie eine Funktion aufgebaut (Abbildung 8). Der Rückgabewert ist immer **void**, da

diese Funktion extern, wie in diesem Fall zum Beispiel auf der Grafikkarte, berechnet wird. Der Datenaustausch mit dem restlichen Programm erfolgt über die **Buffer-Objekte**, die dem Kernel als Argument übergeben werden. Diese werden mit dem Schlüsselwort **global** deklariert, wodurch alle Work-Items auf diesen Datensatz zugreifen können. Der Datentyp **float4*** gibt dabei an, dass es sich um ein Array handelt, bei dem jeder Eintrag ein Vektor der Länge „vier“ und des Datentyps **float** ist. Zur Optimierung des Codes und zur Vermeidung von Konflikten werden dabei die Positions- und Geschwindigkeits-Buffer in zwei Ausführungen übergeben, wobei von einem gelesen (old_) und auf den anderen geschrieben (new_) wird. Der Positions-Buffer (_points) beinhaltet die Positionen in xyz-Koordinaten und der Geschwindigkeits-Buffer (_velos) die Bewegungsrichtung, dargestellt als Vektor, in xyz-Koordinaten. Der jeweils vierte Wert der Vektoren ist ein Platzhalter und kann für andere Werte genutzt werden, wie zum Beispiel einem Wert, der die Größe des Tiers darstellt, wodurch auch unterschiedlich große Tiere im Schwarm realisiert werden könnten. In dieser Version ist der Wert (noch) leer, jedoch können Daten des Typs float4* deutlich besser von der Grafikkarte verarbeitet werden als float3*, weshalb eine Reduzierung des Datentyps unerwartetes Verhalten zeigen könnte. Die Position des Jägers (hunter_pos) wird ebenfalls in xyz-Koordinaten angegeben, jedoch wird diese nur gelesen und kann somit vom Hauptprogramm in jedem Simulationsschritt als einfacher float-Wert übergeben werden. Die Variable **dt** ist ein Wert, der vom Hauptprogramm gemessen wird und beinhaltet die Zeit, die zwischen den Simulationsschritten vergangen ist, was später noch von Relevanz ist.

```
// Kernelfunktion
kernel void fish_sim(

// Pointer auf die Daten-Arrays
global float4* old_points,
global float4* new_points,
global float4* old_velos,
global float4* new_velos,

// Zeitschritt
constant float dt,

// Position des Jägers
constant float hunter_pos_x,
constant float hunter_pos_y,
constant float hunter_pos_z)
{
// Kernel-Code; wird im folgenden Beschrieben
[...]
}
```

**Abbildung 8: Kernelfunktion**

Zu Beginn des Kernels werden zunächst die eigenen Werte geladen (Abbildung 9). Für jedes Tier in der Simulation wird eine Instanz des Kernels erzeugt, die später parallel mit den anderen berechnet werden kann. Damit die Instanzen voneinander unterscheidbar werden, bekommt jede eine eigene Identifikationsnummer (ID). Diese ist einzigartig und kann mit dem Befehl **get_global_id()** abgefragt werden. Das Argument der Funktion gibt dabei die Dimension an, da auch mehrdimensionale Indizierungsmodelle möglich sind. Bei dieser Implementierung genügt jedoch eine Dimension. Die globale ID stellt nun auch die Position der Daten des aktuellen Tieres innerhalb des Buffer-Objekts dar. So kann man dann per ID die Werte aus dem globalen Speicher lesen und lokal zwischenspeichern.

```
// Ermittele globale ID dieser Work-Unit
int myId  = get_global_id(0);

// Ermittele eigene Position
float4 myPos = old_points[myId];

// Ermittele eigene Geschwindigkeit
float4 myV   = old_velos[myId];
```

**Abbildung 9: Laden der eigenen Werte**

Wie bereits erwähnt, werden zum effizienteren Lesen der Daten mehrere Workgroups verwendet, die einen gemeinsamen Speicher nutzen können. Damit die Kernelinstanz weiß, wie die im Hauptprogramm definierte Beschaffenheit der Workgroups ist, werden die folgenden Aufrufe benötigt (Abbildung 10):

- **get_local_id()** bestimmt die ID des aktuellen Tieres innerhalb der Workgroup. Die IDs sind innerhalb der Workgroup einzigartig, wiederholen sich jedoch in den anderen.
- **get_local_size()** ermittelt die Größe der Workgroup. Diese ist standardmäßig auf 256 gesetzt, da dieser Wert bei allen Simulationsgrößen die beste Leistung auf dem getesteten System erzielt hat.
- **get_num_groups()** gibt an, wie viele Workgroups es gibt. Die Argumente der Funktionen dienen wieder der, hier nicht vorhandenen, Dimensionierung.

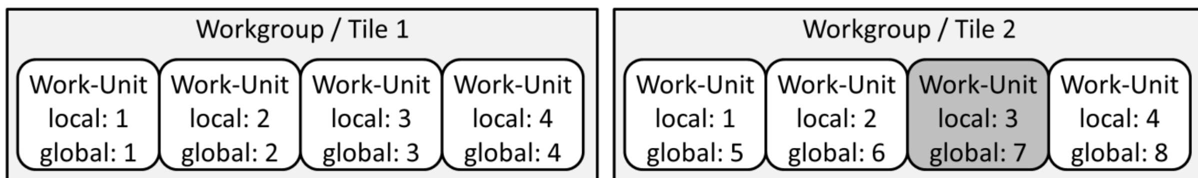
```
// Ermittele lokale ID dieser Work-Unit
int localid   = get_local_id(0);

// Ermittele Workgroup-Size
int tileSize  = get_local_size(0);

// Ermittele Anzahl der Workgroups
int tileCnt   = get_num_groups(0);
```

**Abbildung 10: Laden der Workgroup-Daten**

Die folgende Abbildung (Abbildung 11) verdeutlicht, was die oben vorgestellten Befehle zurückliefern. Der Datensatz besteht in dem Beispiel aus acht Einträgen, aufgeteilt auf zwei Workgroups. Hier beziehen sich die Aufrufe auf das dritte Work-Item der zweiten Workgroup. Es besitzt somit die globale ID „7" (4+3) und die lokale ID „3".




```
get_global_id() =   7     // Globale ID; Position im Daten-Buffer
get_local_id() =    3     // Lokale ID; Position innerhalb der Workgroup
get_local_size() =  4     // Tile-Size; Größe der Workgroup
get_num_groups =    2     // Anzahl der Workgroups
```

**Abbildung 11: Veranschaulichung der Befehle**

Da innerhalb der folgenden Schleifen die Daten von den Tieren geladen werden sollen, die sich innerhalb des Sichtradius' des aktuellen Fisches befinden, um daraus die Hilfswerte für die Berechnungen der Handlungsvorschriften zu erstellen, werden nun schon die entsprechenden Vektoren erstellt (Abbildung 12). Die **seperation_dir** ist dabei für die Berechnung der Separation erforderlich, das **center_of_mass,** also der Schwerpunkt, für die Kohäsion und die **alignment_dir** für die Ausrichtung.

```
// Initialisiere Vektoren für Handlungsvorschriften
float3 seperation_dir = (float3)(0.0f, 0.0f, 0.0f);
float3 center_of_mass = (float3)(0.0f, 0.0f, 0.0f);
float3 alignment_dir = (float3)(0.0f, 0.0f, 0.0f);
```

**Abbildung 12: Initialisierung der Vektoren für die Berechnungen der Handlungsvorschriften**

Um den effektiven Datenaustausch innerhalb der Workgroup nutzen zu können, muss zunächst der lokale Speicher deklariert werden (Abbildung 13). Dies wird über das Schlüsselwort **local** erreicht, welches dem OpenCL-Compiler signalisiert, dass die folgende Datenstruktur innerhalb der Workgroup sichtbar sein soll. Die **LOCAL_MEM_SIZE** ist dabei eine vorher festgelegte Konstante, die hier den Wert 256 beträgt, was der Größe der Workgroup entspricht, da in jeder der folgenden Iterationsschleifen eine andere Workgroup geladen und ausgewertet wird. Somit kann im nächsten Schritt der lokale Speicher mit der nächsten Workgroup überschrieben werden.

```
// Lokaler gemeinsamer Speicher
local float4 sharedMem[LOCAL_MEM_SIZE];
```

**Abbildung 13: Deklarierung lokalen Speichers**

Die folgenden Zeilen beschreiben, wie die Work-Items das gemeinsame Laden der Daten bewerkstelligen (Abbildung 14): Zunächst wird dazu eine Schleife angelegt, die über alle Workgroups (hier auch **tiles** genannt) iteriert. Danach greift jedes Mitglied der Workgroup auf die Stelle im globalen Positionsdaten-Buffer zu, die der aktuellen Workgroup (tileSize*tile) plus der eigenen Position innerhalb der eigenen Workgroup (+localid) entspricht. Da alle Workgroups gleich groß sind, wird somit in jedem Iterationsschritt ein Teildatensatz in Größe einer kompletten Workgroup in den lokalen Speicher geladen. Dabei lädt jedes Work-Item nur ein Tier pro Teildatensatz. Durch den lokalen Speicher kann dann aber auf den gesamten Teildatensatz zugegriffen werden.

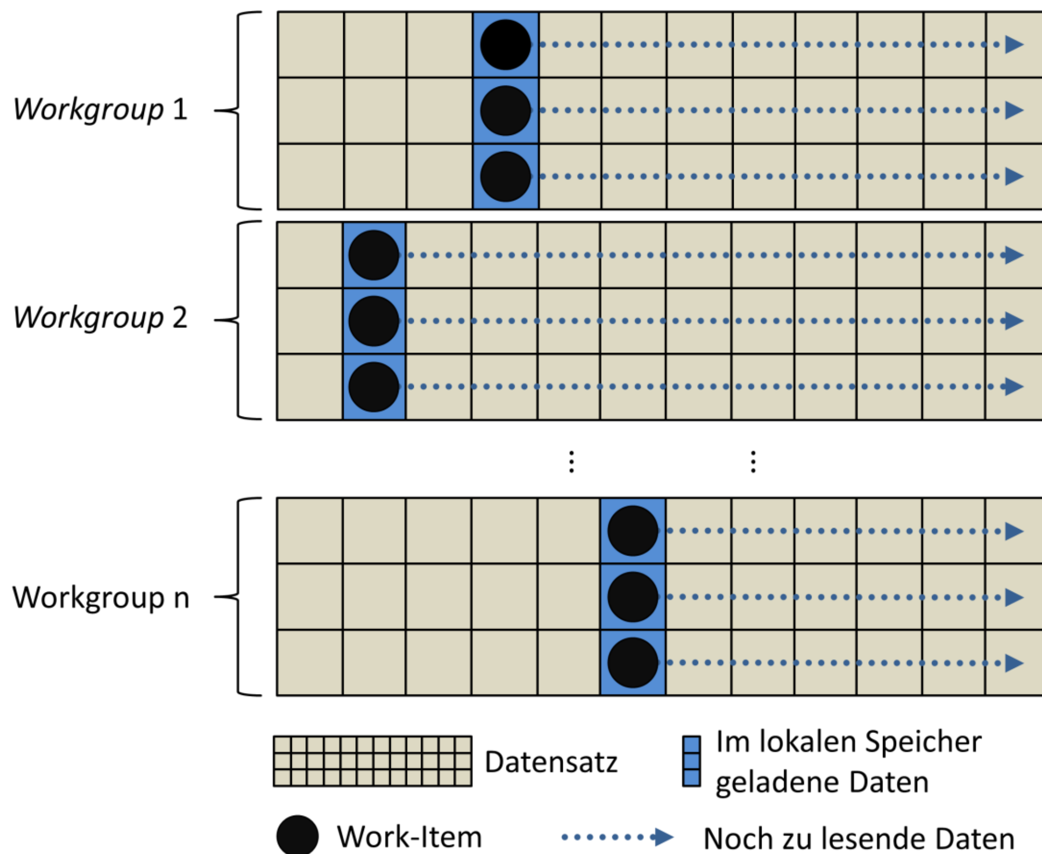



**Abbildung 14: Daten-Ladeschema**

OpenCL-Work-Items arbeiten meist asynchron. Damit im Folgenden auch sichergestellt ist, dass bereits der ganze Teilabschnitt der Daten geladen ist (Abbildung 15), muss die Workgroup synchronisiert werden. Dies kann mit dem **barrier()** Befehl erreicht werden. Dieser Befehl kann nur die Work-Items innerhalb einer Workgroup synchronisieren. Eine Synchronisation über mehrere Workgroups ist nicht möglich. Das Argument des Befehls bezieht sich dabei auf den Speicher, der synchronisiert werden soll. **CLK_LOCAL_MEM_FENCE** ist eine Konstante, mit der der lokale Speicher adressiert werden kann. Somit blockiert dieser Befehl den Zugriff auf den lokalen Speicher, bis alle Work-Items dieser Workgroup diese Stelle erreicht haben. Damit ist sichergestellt, dass der komplette Teildatensatz geladen ist und danach von allen Work-Items gelesen werden kann. Bevor der nächste Abschnitt geladen werden kann, muss noch einmal der barrier-Befehl aufgerufen werden, um sicherzustellen, dass alle Work-Items mit dem Lesen des alten Abschnitts fertig sind, da der Speicher im nächsten Schritt neu belegt wird und ansonsten die Gefahr besteht, dass Daten überschrieben werden, die noch gelesen werden müssen. Dadurch könnte es passieren, dass nicht alle Tiere erfasst werden.

```
// Für alle Workgroups:
for(int tile = 0; tile < tileCnt; ++tile)  {

  // Speichere Koordinaten anderer Tiere in den lokalen Speicher
  sharedMem[localid] = old_points[tileSize * tile + localid];

  // Synchronisiere Work-Units
  barrier(CLK_LOCAL_MEM_FENCE);

  // Für alle Tiere in der Workgroup:
  for(int j = 0; j < tileSize; ++j) {

  //[...] -> Abbildung 16

  }

  // Synchronisiere Work-Units
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

**Abbildung 15: Laden der Tiere**

Da nun die ganze Workgroup geladen ist, kann überprüft werden, ob eines der Tiere innerhalb des Sichtradius' des Tieres, für das das aktuelle Work-Item zuständig ist, liegt (Abbildung 16) und somit für spätere Berechnungen relevant wird. Also wird über alle Einträge des lokalen Speichers, in dem jetzt die Positionen der anderen Tiere gespeichert sind, iteriert. Mit Hilfe eines zuvor definierten Radius' (**COHESION_RAD**) und der bereits integrierten **distance()**-Funktion kann nun überprüft werden, ob der Abstand zu einem anderen Tier so klein ist, dass sich dieses innerhalb des Radius' befindet. In diesem Fall wird von dem Tier auch noch die Geschwindigkeit geladen. An dieser Stelle ist noch anzumerken, dass die eigenen Daten auch geladen werden, was die späteren Berechnungen jedoch nicht negativ beeinflusst, da dies über die Parametereinstellungen aufgefangen werden kann. Das Herausfiltern der eigenen Daten würde hingegen die Berechnungen zusätzlich verlangsamen.

```
// Für alle Tiere in der Workgroup:
for(int j = 0; j < tileSize; ++j) {

  // Lade Positionsdaten anderer Tiere aus dem lokalen Speicher
  float4 otherPos = sharedMem[j];

  // Falls im Radius
  if(distance(myPos.xyz, otherPos.xyz) < COHESION_RAD) {

    // Lade Geschwindigkeit des anderen Tieres
    float4 otherV = old_velos[tileSize * tile + j];

    //[...] -> Abbildung 17
}
```

**Abbildung 16: Laden der Daten von Tieren innerhalb des Sichtradius'**

Zur Berechnung der sich aus den Handlungsvorschriften (Abbildung 1) ergebenden Kräfte müssen zunächst die Nachbarn ausgewertet werden (Abbildung 17). Die Separation ergibt sich aus den kombinierten Vektoren von den Nachbarn zum aktuell betrachteten Individuum. Einen solchen Vektor erhält man, indem man die Position des Nachbarn von der eigenen subtrahiert. Diese Vektoren werden dann nur noch aufsummiert, um eine von den Nachbarn zur eigenen Position gerichtete Kraft zu bekommen. In dieser Implementation werden die Vektoren in jedem Iterationsschritt durch die Nachbarn sequentiell aufsummiert.

Für die Berechnung der Kohäsion wird zunächst der Schwerpunkt der Nachbarn benötigt. Dieser ergibt sich, indem man den Mittelwert der Positionen bestimmt. Dies wird auch hier wieder sequentiell errechnet. Da auch der initiale, leere Vektor mit in diese Berechnung einbezogen wird, erreicht man, dass der Schwarm beim Mittelpunkt des Systems bleibt.

Die Berechnung der Ausrichtung erfolgt, indem man die Geschwindigkeitsvektoren der Nachbarn mittelt (auch hier wieder sequentiell). Dabei werden die Vektoren nicht normalisiert (hier also nicht auf die Länge „eins" gebracht), wodurch erreicht wird, dass zusätzlich zur Ausrichtung auch eine Geschwindigkeitsangleichung stattfindet, was dem Schwarmverhalten mehr Realitätsnähe gibt.

```
  // Berechnung der Separation
  seperation_dir = seperation_dir + (myPos-positions[i]);

  // Berechnung des Schwerpunktes
  center_of_mass = (center_of_mass + positions[i]) / 2.0f;

  // Berechnung der Ausrichtung
  alignment_dir = (alignment_dir + velocities[i]) / 2.0f;
```

**Abbildung 17: Vorberechnungen für die Handlungsvorschriften mit Hilfe der Nachbartiere**

Nach Abschluss der vorangegangen Schleifen ist nun die Vorarbeit erledigt und die Berechnungen für die neue Geschwindigkeit des Individuums können beginnen. Um das Verhalten der Tiere zu modifizieren wenn der Jäger in der Nähe ist, werden zunächst die konstant definierten Dämpfungswerte für die einzelnen Kräfte in Variablen übertragen. Falls der Jäger sich nun ausreichend dem Individuum genähert hat, werden die Dämpfungswerte modifiziert (Abbildung 18), um ein dichteres Schwarmgefüge zu erzeugen. Der „Sichtradius" des Jägers (**HUNTER_RAD)** entspricht hierbei dem fünffachen des normalen Sichtradius' und ist der Radius, ab dem die Tiere aktiv vor dem Jäger flüchten (dies wird im Folgenden noch erläutert).

```
float3 hunterPos = (float3)(hunter_pos_x, hunter_pos_y, hunter_pos_z);

float alignment_damp = ALIGNMENT_DAMP;
float seperation_damp = SEPERATION_DAMP;
float cohesion_damp = COHESION_DAMP;

// Modifiziere Verhalten, wenn Jäger in der Nähe
if(distance(myPos.xyz, hunterPos.xyz) < HUNTER_RAD * GRAND_DISTANCE) {
  alignment_damp = alignment_damp * HUNTER_ALIGNMENT_REDUCTION;
  seperation_damp = seperation_damp * HUNTER_SEPERATION_REDUCTION;
  cohesion_damp = cohesion_damp * HUNTER_COHESION_ENFORCEMENT;
}
```

**Abbildung 18: Modifikation von Umgebungsvariablen durch den Jäger**

Zur Berechnung der neuen Geschwindigkeit (Abbildung 19) wird zunächst die alte Geschwindigkeit als Ausgangswert genutzt. Zu dieser addieren sich nun die durch die Parametereinstellungen modifizierten Werte der vorherigen Berechnungen. Dabei muss bei der Kohäsion noch der Vektor von dem Tier zum Schwerpunkt und bei der Ausrichtung

noch die Differenz zwischen der letzten Richtung des Tieres und der Richtung, in die sich die Nachbarn bewegen, berechnet werden.

```
float4 myNewV = myV;

// Separation
myNewV.xyz = myNewV.xyz + seperation_dir * seperation_damp;

// Kohäsion
myNewV.xyz = myNewV.xyz + (center_of_mass - myPos) * cohesion_damp;

// Ausrichtung
myNewV.xyz = myNewV.xyz + (alignment_dir - myV) * alignment_damp;
```

**Abbildung 19: Berechnung der Gesamtkraft**

Die Reaktion auf den Jäger verstärkt sich, je näher dieser dem Individuum kommt. Werden bei einer großen Distanz, wie oben beschrieben, nur die Dämpfungsvariablen modifiziert, wirkt sich die direkte Nähe des Jägers stärker aus. Zunächst wird die Maximalgeschwindigkeit nur etwas erhöht und eine leichte vom Jäger weg gerichtete Kraft hinzugefügt. Bei sehr geringem Abstand wird die Geschwindigkeit deutlich erhöht und die zum Jäger entgegengesetzte Bewegungsrichtung viel stärker gewichtet. Dadurch verstärkt sich die Reaktion auf den Jäger mit schwindender Distanz zum Tier (Abbildung 20).

```
float max_speed = MAX_SPEED;

// Modifiziere Dämpfungsvariablen, wenn Jäger in der Nähe:
if(distance(myPos.xyz, hunterPos.xyz) < HUNTER_RAD){
  myNewV.xyz = myNewV.xyz + (myPos.xyz-hunterPos.xyz) * HUNTER_DAMP;
  max_speed = max_speed * HIGH_SPEEDUP;

} else if(distance(myPos.xyz, hunterPos.xyz) < HUNTER_RAD * BIG_DISTANCE) {
  myNewV.xyz = myNewV.xyz + (myPos.xyz-hunterPos.xyz) * REDUCED_HUNTER_DAMP;
  max_speed = max_speed * SMALL_SPEEDUP;
}
```

**Abbildung 20: Direkte Reaktion auf Jäger**

Zur Umsetzung der Minimal- und Maximalgeschwindigkeit der Tiere wird überprüft, welche Länge der bisher entstandene Geschwindigkeitsvektor hat. Ist dieser zu kurz oder zu

lang, wird er zunächst normalisiert und dann entsprechend der Grenzwerte gestaucht oder gestreckt. Mit Hilfe des Skalarprodukts wird abschließend noch überprüft, ob sich der neue Vektor zu stark vom alten unterscheidet, also das Skalarprodukt kleiner als der zuvor festgelegte Grenzwert ist. Ist dies der Fall, wird er so lange sukzessiv an den alten Vektor angepasst, bis die Differenz klein genug ist. Dadurch werden zu starke und somit unrealistische Bewegungsänderungen unterbunden (Abbildung 21).

```
// Geschwindigkeitsbegrenzung
if(length(myNewV.xyz) > max_speed) {
  myNewV.xyz = normalize(myNewV.xyz) * max_speed;
}

// Geschwindigkeitsbegrenzung
if(length(myNewV.xyz) < MIN_SPEED) {
  myNewV.xyz = normalize(myNewV.xyz) * MIN_SPEED;
}

// Verhindere zu starke Drehungen
while(dot(normalize(myNewV),normalize(myV)) < MAX_ROATION) {
 myNewV = (myNewV + myV) / 2.0f;
}
```

**Abbildung 21: Umsetzung der „kleineren" Regeln**

Zuletzt werden noch die Daten in die neuen Daten-Buffer geschrieben (Abbildung 22). Die Geschwindigkeit wird dabei lediglich erneuert, die Position noch um den Geschwindigkeitsvektor verschoben. Letzterer wird zudem noch mit der Zeitvariablen modifiziert, damit die Simulation unabhängig von der Rechengeschwindigkeit läuft.

```
// Schreibe neue Geschwindigkeit
new_velos[myId] = myNewV;

// Schreibe neue Position
new_points[myId] = myPos + (float4)(new_velos[myId].xyz,0) * dt;
```

**Abbildung 22: Schreiben der Daten**

## 4.2 Visualisierung

### 4.2.1 Laden der Modelle

Die zuvor heruntergeladenen Modelle müssen zunächst in Grafikprogrammen geladen werden, um eine Datenstruktur zu bekommen, die später gut ausgelesen werden kann. Dazu wird das Free-Ware Programm **Blender**[24] genutzt. Nach dem Laden der Modelle, werden sie dann passend skaliert und im **Wavefront OBJ**[25]-Format exportiert. Dieses Format wird verwendet, da dieses gut verständlich und somit auch gut maschinell lesbar ist. Das OBJ-Format ist im Prinzip eine Textdatei, die in mehrere Blöcke aufgeteilt ist (Abbildung 23). Die verschiedenen Blöcke beschreiben dabei verschiedene Eigenschaften. Jeder Block ist in Zeilen aufgeteilt, wobei jede Zeile die Daten eines Punktes des Gittermodells beschreibt. Der erste Block gibt die Positionen der Drahtgitterknotenpunkte in Modell-Koordinaten an, der zweite Block die Texturkoordinaten, anhand derer die 2D-Textur auf das 3D-Modell übertragen werden kann, und der dritte Block die Normalen der Punkte, die man für einige Berechnungen der Darstellung benötigt (Beleuchtung, Sichtbarkeit der Punkte, etc.). Der vierte und letzte Block beinhaltet die sogenannten **faces**, eine Art Definition, welche Punkte der vorherigen Blöcke zusammengehören. Dabei ergeben immer drei Punkte ein Dreieck, woraus später das Drahtgittermodell (Abbildungen 24 und 25) aufgebaut ist.

```
v 0.086690 -0.239472 -0.251515
v 0.130660 -0.265988 -0.224774
v 0.111793 -0.282284 -0.215125
⋮
v -1.800794 0.154997 0.015237

vt 0.260167 0.436027
vt 0.251828 0.403085
vt 0.246906 0.417482
⋮
vt 0.372833 0.645851

vn 0.185705 -0.532182 -0.825983
vn 0.184942 -0.506729 -0.842006
vn -0.379864 -0.792810 -0.476577
⋮
vn 0.069887 0.269112 0.960540

f 1/1/1 2/2/2 3/3/3
f 4/4/4 5/5/5 6/6/6
f 7/7/7 1/1/1 3/3/3
⋮
```

**Abbildung 23: Ausschnitt aus der Fischmodell.obj-Datei**

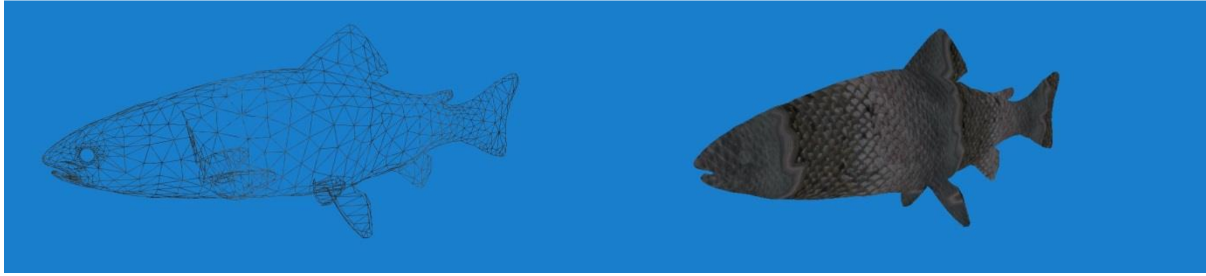


**Abbildung 24: Fisch als Drahtgittermodell (links) und mit Textur[26] in der fertigen Simulation (rechts)**

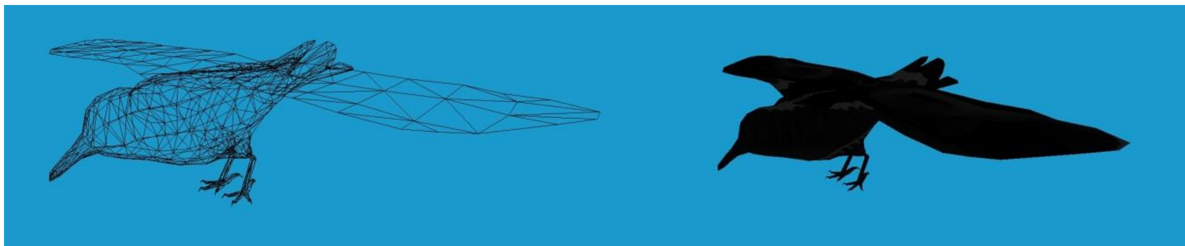


**Abbildung 25: Vogel als Drahtgittermodell (links) und mit Textur[27] in der fertigen Simulation (rechts)**

Die geladenen Modelldaten können dann um die Koordinaten, die man aus der OpenCL-Berechnung bekommt, „aufgebaut" werden. Dabei bilden die OpenCL-Koordinaten die Mittelpunkte der Tiere, von denen aus anschließend per Modell-Koordinaten, die man aus der OBJ-Datei gelesen hat, die Tiermodelle erstellt werden.

### 4.2.2 Ausrichten der Modelle

Damit die Tiere in die richtige Richtung zeigen, müssen die Modelle zunächst noch gedreht werden. Da die Daten nur auf der Grafikkarte vorliegen, muss die Drehrichtung aus dem Geschwindigkeitsvektor, der in dem OpenCL-Kernel ermittelt wird, berechnet werden. Initial zeigen die Modelle genau in x-Richtung des Koordinatensystems. Die Drehung der Tiere findet dabei über zwei Achsen statt: Die Drehung um die z-Achse sorgt dabei für die Auf- und Ab-Bewegungen, die Drehung um die y-Achse bewirkt die Richtungsänderung. Eine Drehung um die x-Achse hätte zur Folge, dass sich die Tiere um die eigene Längsachse „rollen" würden, was sie in der Natur normalerweise nicht machen.

Um die Modelle zu drehen, muss man die einzelnen Datenpunkte um den Mittelpunkt des Modells drehen. Da die Datenpunkte zunächst in Modellkoordinaten vorliegen, genügt eine einfache Drehung um den Mittelpunkt, ohne dass die Datenpunkte erst dorthin verschoben werden müssen. Die Drehung der Punkte funktioniert über Rotationsmatrizen, die für jede

Rotationsachse ein festes Schema haben, in das nur noch der Winkel eingetragen werden muss, um den gedreht werden soll. Da die Winkel nicht explizit vorliegen, müssen die Richtungsvektoren zur Berechnung herangezogen werden. Zur Bestimmung des z-Winkels wird folgende Formel benutzt:

$$\boldsymbol{zWinkel} = \frac{\overrightarrow{\boldsymbol{xyz}} \cdot \begin{pmatrix} \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \end{pmatrix}}{|\overrightarrow{\boldsymbol{xyz}}|}$$

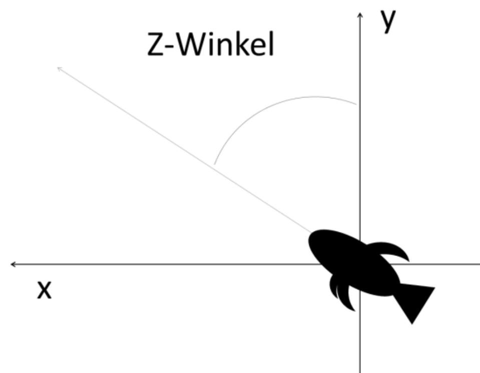



**Abbildung 26: Winkel zur Drehung um die z-Achse (Seitenansicht)**

Dabei bezeichnet $\overrightarrow{\boldsymbol{xyz}}$ den nicht normalisierten Richtungsvektor und $\begin{pmatrix} \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \end{pmatrix}$ ist ein normalisierter Vektor, der genau in y-Richtung zeigt. Das Skalarprodukt dieser beiden Vektoren ergibt, geteilt durch die Länge des Richtungsvektors, den Winkel zwischen den beiden Vektoren und somit den Winkel, um den um die z-Achse gedreht werden muss (Abbildung 26).

Die Berechnung des y-Winkels wird mit der folgenden Formel realisiert:

$$yWinkel = acos\left(\frac{\vec{x}}{|\overrightarrow{xz}|}\right)$$

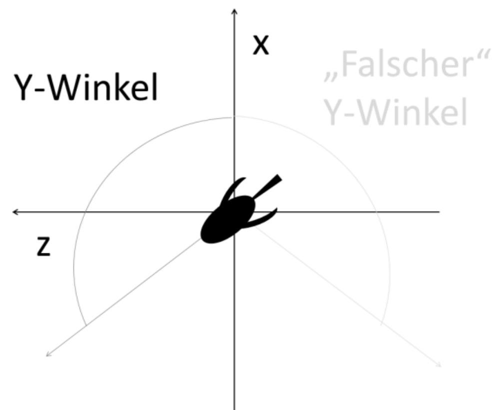



**Abbildung 27: Winkel zur Drehung um die y-Achse (Draufsicht)**

Hierbei ist $\vec{x}$ die x-Komponente des Richtungsvektors und $\overrightarrow{xz}$ die xz-Komponente. Mit Hilfe des Kosinussatzes kann somit der Winkel zur Drehung um die y-Achse ermittelt werden. Da dieser Winkel jedoch nur den Winkel zwischen den Geraden darstellt, liegt dieser immer zwischen 0° und 180° und sagt somit nichts darüber aus, ob es sich um eine Links- oder Rechtsdrehung handelt, wodurch auch ein „falscher“ Winkel zustande kommen kann (Abbildung 27). Daher wird noch überprüft, ob die z-Komponente des Winkels positiv oder negativ ist, um bei Bedarf den Winkel mit „-1“ zu multiplizieren, wodurch eine Drehung in die andere Richtung erfolgt.

Anschließend werden die Winkel in die entsprechenden Rotationsmatrizen eingefügt und miteinander multipliziert. Dadurch ergibt sich die Gesamtdrehung für die Datenpunkte des Modells. Somit kann diese Matrix auf die Modellkoordinaten angewendet werden, um das Modell als Ganzes zu drehen. Danach werden die Punkte noch um die Positionsdaten verschoben, wodurch die Modelle dann auch an der richtigen Position in der Welt angezeigt werden.

Um den Rechenaufwand bei der Anzeige zu reduzieren, wird hierbei das sogenannte **Instancing**-Verfahren angewandt. Das Instancing kann angewendet werden, wenn viele, identische Modelle angezeigt werden sollen. Dafür werden mehrere Instanzen von ein und demselben Modell erzeugt, welche dann unterschiedlich transformiert werden. In diesem Fall werden auf das gleiche Objekt verschiedene Rotationen und Translationen angewendet. Da hierbei viele, sehr ähnliche Berechnungen stattfinden, können sie per Instancing effizient von der Grafikkarte parallel berechnet werden.

## 4.3 OpenCL-OpenGL Datenaustausch

Da sowohl die Simulationsberechnung, als auch die Grafikanzeige auf der Grafikkarte stattfinden und die Grafikanzeige die Daten der Simulation benötigt, muss der Datenaustausch gelingen, ohne das System zu sehr zu verlangsamen. Die Daten müssen also auf der Grafikkarte von beiden Fraktionen genutzt werden, da der Umweg über CPU und Arbeitsspeicher deutlich rechenintensiver ausfallen würde. Die Kombination aus OpenGL und OpenCL wird genutzt, da die beiden Systeme diesen Datenaustausch unterstützen.

Damit dieser Austausch funktioniert, muss zunächst ein OpenGL-Daten-Buffer erstellt werden, in dem die Daten initial vorliegen. Es werden also zunächst zwei FloatBuffer (Buffer-Objekte des Datentyps *float*) erstellt: einer für die Positionen und einer für die Geschwindigkeiten. Diese werden mit den Initialdaten gefüllt. In diesem Fall sind die Positionen zunächst in einer Würfelstruktur angeordnet, die als Startformation der Tiere dient. Die Initialgeschwindigkeit ist dabei ein Vektor der Länge „eins" in x-Richtung. Da die Buffer Daten enthalten, die später auf viele Instanzen des gleichen Objekts angewendet werden, werden diese Buffer als sogenannte **Instance Buffer** definiert, die das bereits genannte Instancing unterstützen.

Damit diese Buffer nun auch von OpenCL genutzt werden können, müssen sie zunächst in **CLMem**-Objekte konvertiert werden. Dies ist über den Befehl **clCreateFromGLBuffer()** möglich, der aus dem Buffer und dem **CLContext** ein CLMem-Objekt erzeugt. Der CLContext wird während der Einrichtung der OpenCL-Umgebung erstellt und beinhaltet die Informationen des Systems (**Device**), auf dem später der Kernel rechnen soll, was in diesem Fall die Grafikkarte ist. Da der CLContext auch die Information zur OpenCL-Implementierung, hier ist dies die Nvidia-Implementierung, beinhaltet, muss dieser mit übergeben werden, damit ein CLMem-Objekt erzeugt werden kann, das auch von der Grafikkarte nutzbar ist.

Die Daten des CLMem Objekts können nun von sowohl OpenGL, als auch OpenCL genutzt werden. Da die weitere Verarbeitung jetzt auf der Grafikkarte stattfindet und nicht mehr direkt vom Hauptprogramm (**Host**) gesteuert wird, muss vorher definiert werden, wer wann auf die Daten zugreifen darf, damit es auch hier nicht zu Schreibe- oder Lesekonflikten kommt. Dazu wird im Zuge der OpenCL-Initialisierung eine **Command Queue** erstellt, die die Kommunikation zwischen dem Host und dem Device regelt. Die „Rechte" für den Zugriff auf den Daten-Buffer liegen bei OpenGL. Damit OpenCL die Daten nutzen kann, müssen zuerst die Zugriffsrechte angefordert werden. Dies passiert mit dem Befehl **clEnqueueAcquireGLObjects()**, der als Argument das CLMem-Objekt und die Command Queue bekommt. Nach diesem Befehl kann der Kernel aufgerufen werden, der einen Simulationsschritt des Schwarms durchführt. Nach diesem Simulationsschritt müssen die

Zugriffsrechte per **clEnqueueReleaseGLObjects()** wieder abgegeben werden, damit OpenGL die Daten zur Anzeige der Tiere nutzen kann (Abbildung 28).

**Abbildung 28: Wechsel der Zugriffsrechte auf die Daten-Buffer**

# 5 Auswertung

## 5.1 Optische Evaluierung

Eine eindeutige Auswertung des Endergebnisses ist in diesem Kontext schwer durchführbar, da das System auf Glaubwürdigkeit und Realismus ausgelegt ist. Somit ist die einzig mögliche Evaluierungsmethode eine optische, sehr subjektive Einschätzung, wie realistisch das Schwarmverhalten wirkt. Daher werden an dieser Stelle die technischen Aspekten und Erfahrungswerte, die während der Entwicklung gemacht wurden, beleuchtet. Generell ist jedoch zu sagen, dass das System darauf ausgelegt ist ein bestmöglichstes Verhalten zu erzielen und letztendlich ein Ergebnis erzeugt wurde, das sich mit vergleichbaren Videoaufnahmen von entsprechenden Schwärmen sehr gut deckt.

Die folgende Bilderstrecke (Abbildungen 29 bis 35) soll dabei verdeutlichen, welche Dynamik der Schwarm (hier die Fischschwarm-Variante) aufweist. Dabei beginnt der Schwarm in der initialen Würfelformation (wobei sich der Ursprung des Systems in der unteren, vorderen, rechten Ecke des Würfels befindet) und versucht anschließend in eine optimale Schwarmformation zu gelangen:

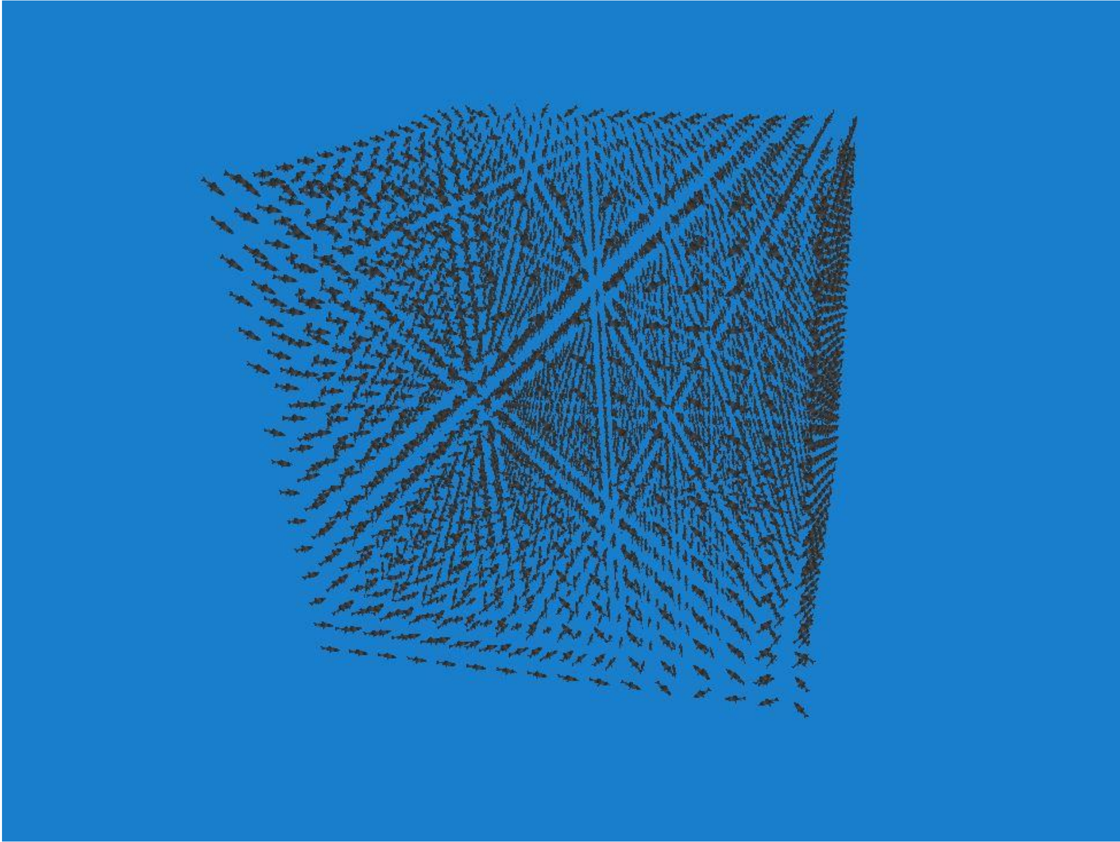


**Abbildung 29: Initialformation des Schwarms**

**Abbildung 30: Aufbrechen der Initialformation**

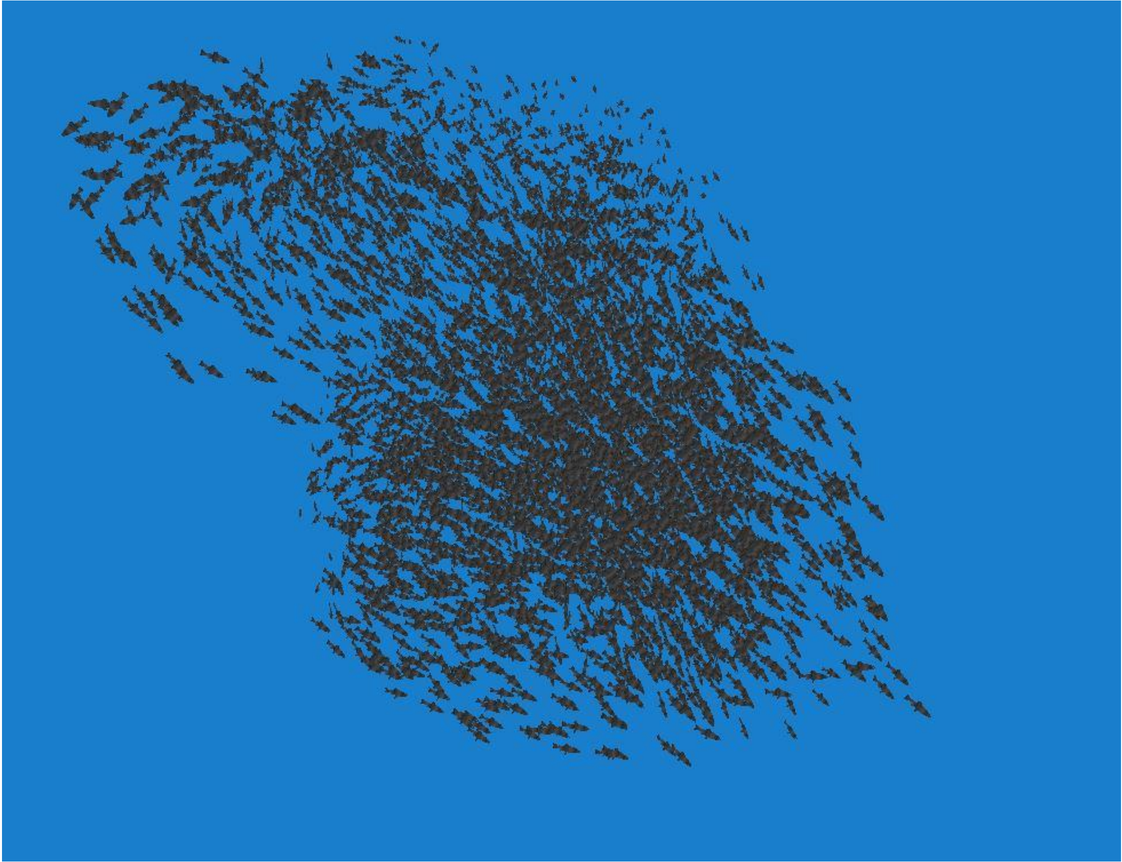


**Abbildung 31: Erste Orientierung zum Schwarmgefüge**

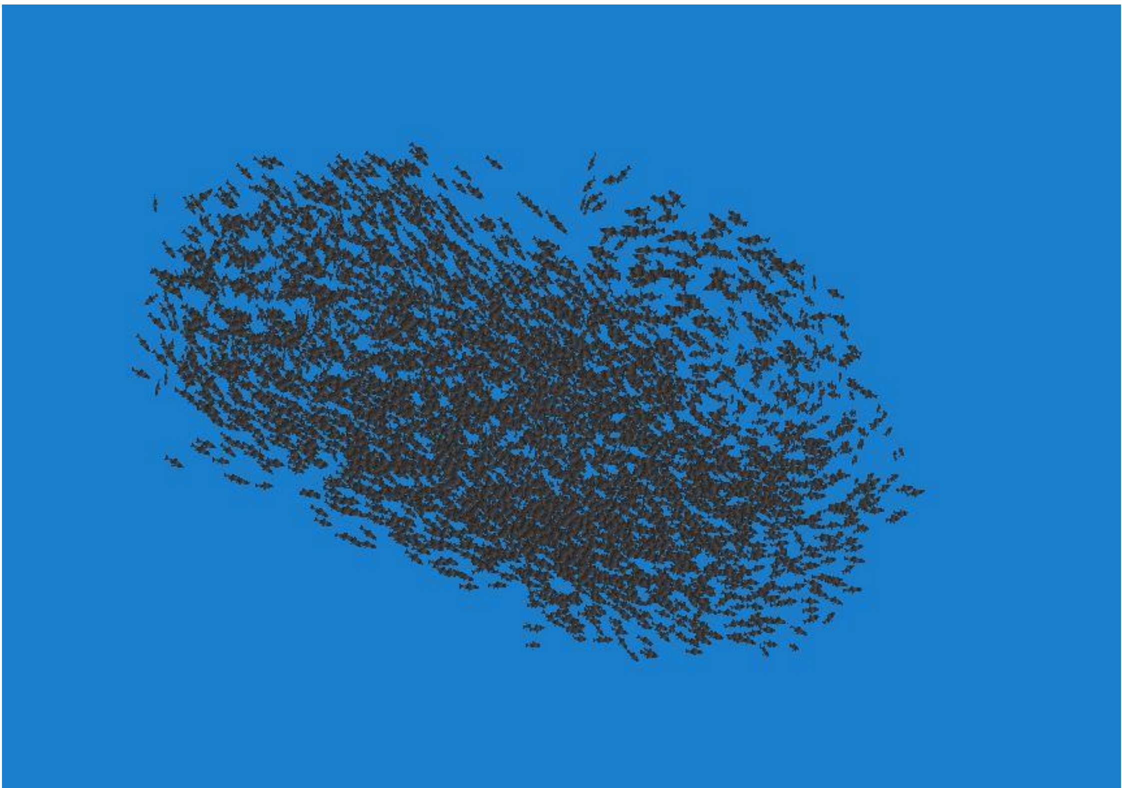


**Abbildung 32: "Umorientierung" zur Mitte**

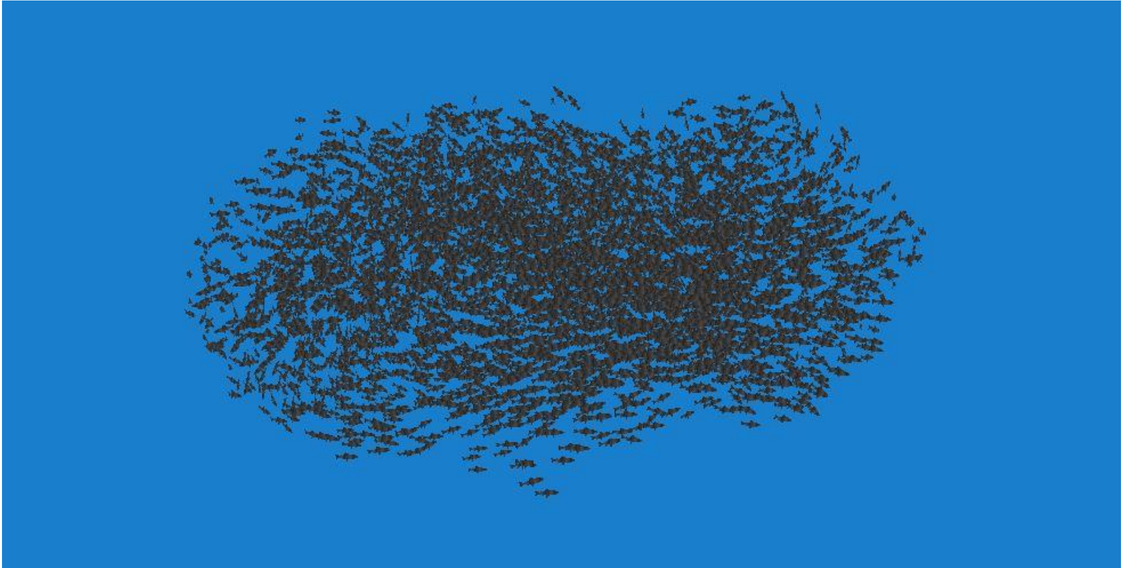


**Abbildung 33: Annähernd optimale Form**

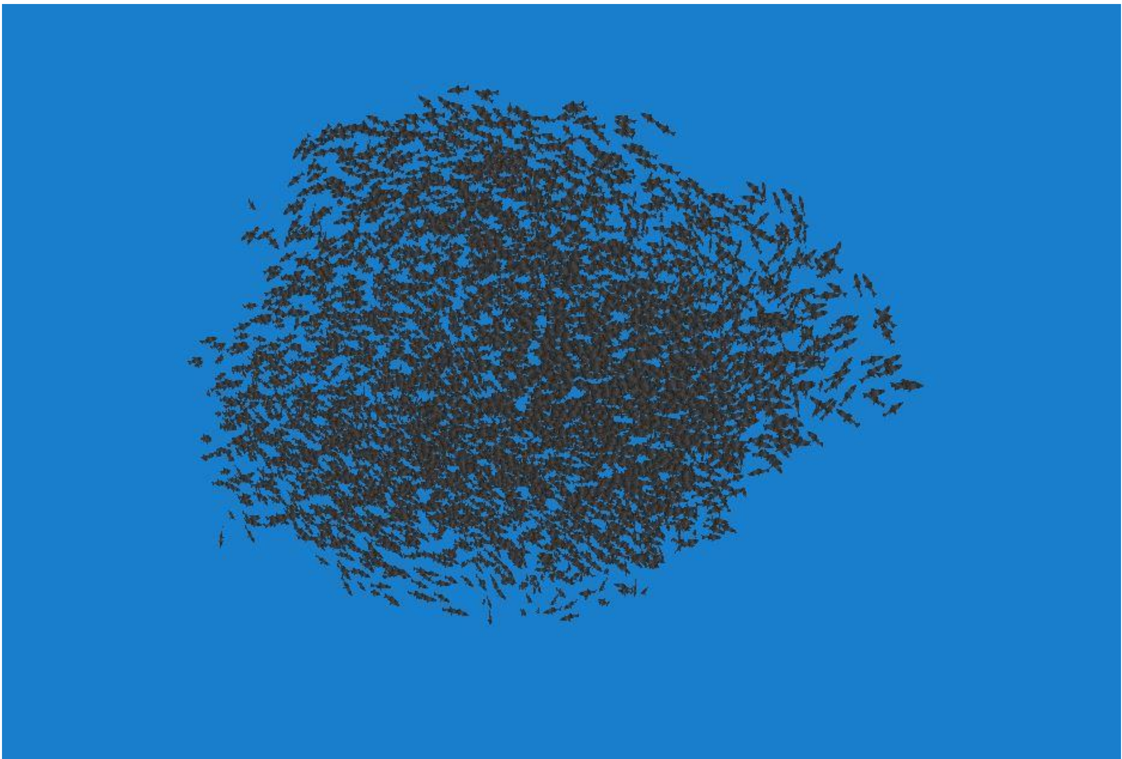


**Abbildung 34: Optimale Form mit fehlender Ausrichtung**

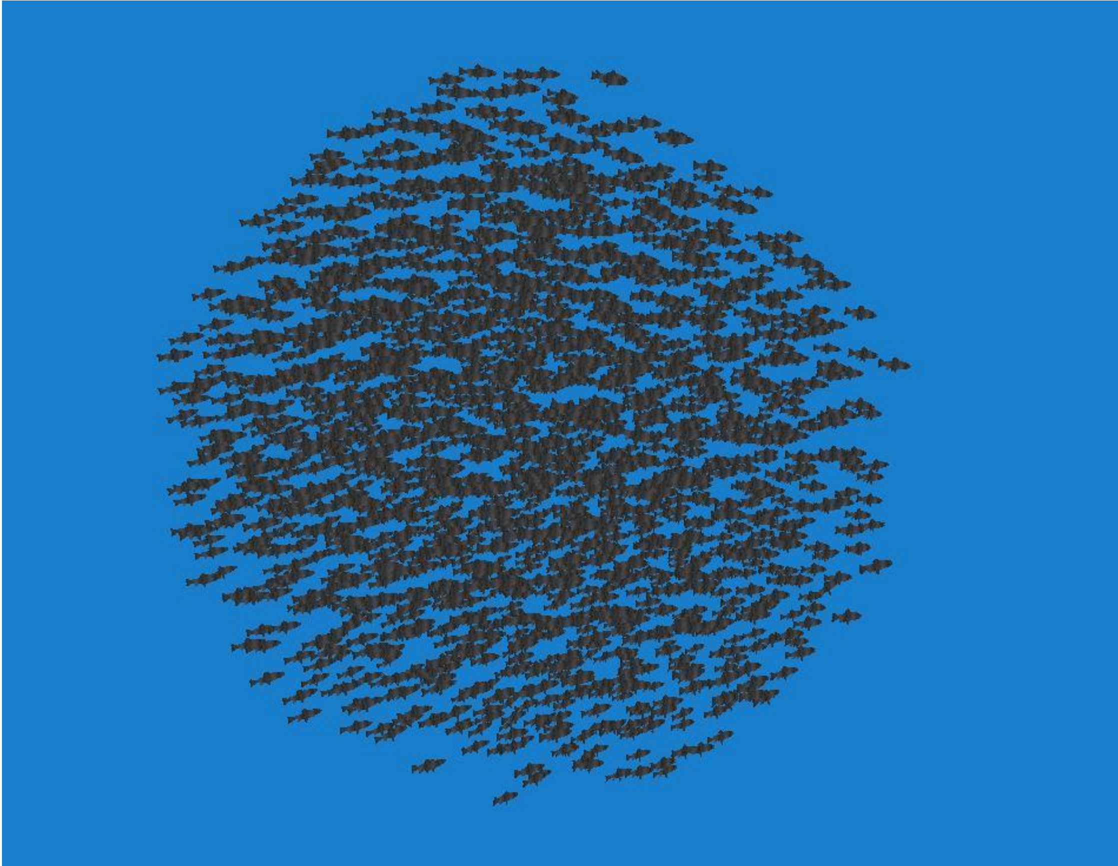


**Abbildung 35: Ausgerichtetes, stabiles, optimales Schwarmgefüge**

**Abbildung 36: Echter Fischschwarm (Vergleichsbild)[28]**

Der Vergleich des „Endzustandes“ der Simulation mit einer vergleichbaren, echten Fotoaufnahme (Abbildung 36) zeigt, dass die typische Schwarmformation sehr gut erreicht wird. Aufnahmen der Animation sind auf der beiliegenden DVD zu finden. Diese Aufnahmen entstanden mit dem Free-Ware Programm **Fraps**[29].

In der aktuellen Version des Programms besitzen die Tiere keine Animationsmodelle, wodurch sie sehr statisch wirken. Das fällt aus der Entfernung auch nicht weiter auf, wirkt von Nahem jedoch unnatürlich. So scheint es weniger glaubwürdig, wenn sich zum Beispiel ein Fisch deutlich schneller als vorher bewegt (etwa um dem Jäger zu entkommen), ohne dabei typische Bewegungsmuster zu zeigen. Dadurch wird die optische Evaluierung teils beeinträchtigt, was die Glaubwürdigkeit des Gesamtsystems herabsetzt. Dies fällt jedoch, wie bereits beschrieben, aus der Ferne nicht weiter auf und würde die Rechenkapazität zusätzlich belasten, wodurch solche Änderungen nur gemacht werden sollten, wenn es wichtig ist, dass der Schwarm auch von Nahem zu sehen sein soll.

Die Reaktion auf die Präsenz des Jägers äußert sich in einem chaotischeren, aber auch kompakteren Gefüge (Abbildung 37). Dies veranschaulicht auch gut, wie sehr sich die Variierung der Parameter auf das Schwarmverhalten auswirkt.

**Abbildung 37: Verhalten des Schwarms bei Anwesenheit des Jägers**

Da sich der Jäger in der jetzigen Version nur in einer Kreisbahn bewegt und nicht auf den Schwarm reagiert, sorgt das Aufeinandertreffen der beiden Parteien für ein nicht adäquates Bild. Der Jäger schwimmt dabei direkt durch den Schwarm und die Tiere stieben wild auseinander (Abbildung 38). In der Natur bewirkt das Schwarmgefüge eigentlich, dass sich der Jäger dem Schwarm gar nicht soweit nähert, da er die Form der einzelnen Tiere nicht wahrnehmen kann und durch das große Gebilde abgeschreckt wird. Dadurch konzentriert er sich vorrangig auf Tiere am Rande des Schwarms, die sich zu weit von selbigem entfernen. Dieses Schwarmverhalten führt wiederum dazu, dass die Tiere sich näher zusammen rotten und dadurch mehr Möglichkeit haben die Schwarmform aufrecht zu erhalten. Da der Jäger in dieser Simulation jedoch nicht davon abgeschreckt wird, können auch die Schwarmtiere nicht adäquat reagieren.

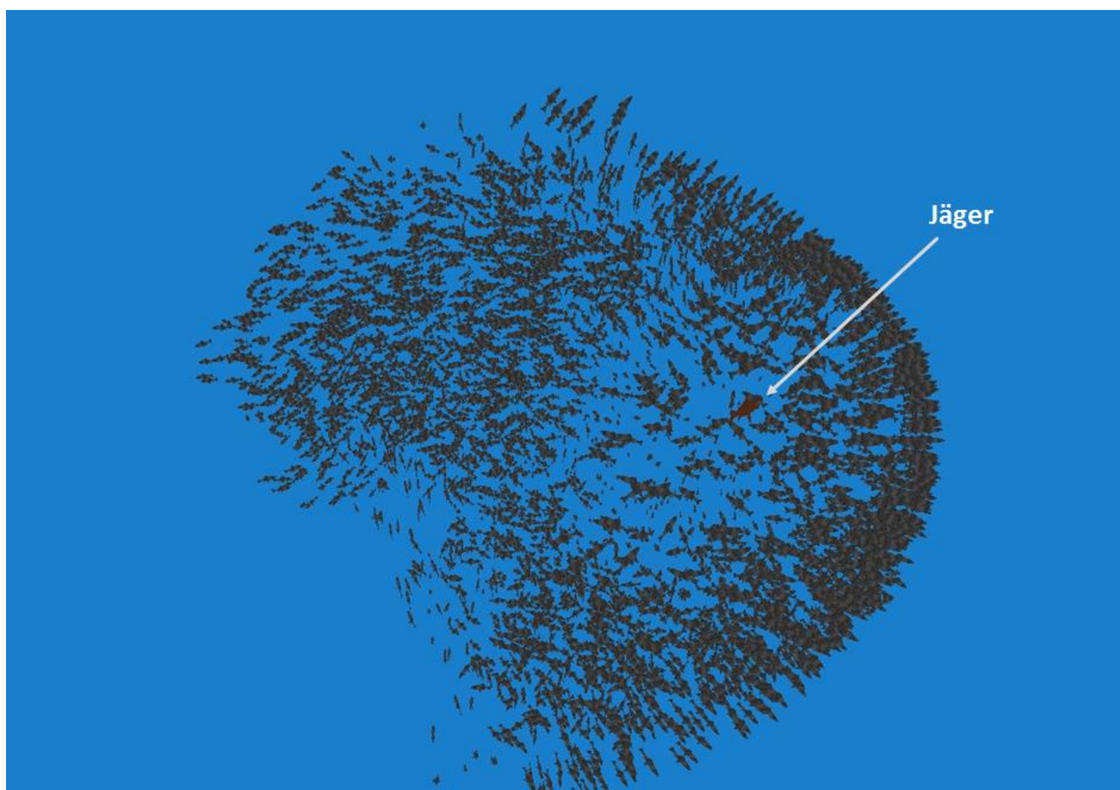



**Abbildung 38: Auseinanderstieben bei geringer Jägerdistanz**

## 5.2 Parametereinstellungen

Dieses System ist, wie alle Partikelsysteme mit Wechselwirkungen, auch stark abhängig von den moderierenden Parametern, die die entstehenden Vektoren abschwächen oder verstärken. Auch Variablen wie Radien und Grenzwerte für die Berechnungen modifizieren das Ergebnis deutlich. Dabei beeinflussen sich die Werte auch untereinander, sodass komplexe Wechselwirkungen entstehen, die die adäquate Einstellung der Parameter stark erschweren. Schon kleinste Änderungen können das gesamte System zerrütten, was dazu führt, dass die Justage der Parameter sehr zeitintensiv ist. Zudem erschwert dies die Fehlersuche während der Implementierung des Systems, da nicht immer ersichtlich ist, ob es sich um Logikfehler oder Parameterfehler handelt. Des Weiteren erfordert jedes nachträgliche Eingreifen in das System eine erneute Justierung.

In dem hier implementierten System hat es sich durch experimentelles Herantasten als ratsam erwiesen, die Vektoren, die bei den Handlungsvorschriften entsprechenden Berechnungen entstehen, auf ein Hundertstel zu reduzieren, da die Kräfte ansonsten deutlich zu stark sind. Zudem sollten die Kohäsionskräfte doppelt so stark wie die Separationskräfte sein und die Ausrichtung fünfmal so stark. Der Sichtradius der Tiere ist etwa so groß, dass bei den vorhandenen Vorschriften etwa 30 Tiere mit in die Berechnungen einbezogen werden. Unterschiedliche Sichtradien für die drei verschiedenen Handlungsvorschriften sind nicht nötig, können jedoch zur besseren Feinabstimmung in Betracht gezogen werden. Wenn der Fressfeind anwesend ist, sollte zusätzlich die Kohäsion um 30% verstärkt werden, wohingegen die Ausrichtung um 80% und Separation um 20% abgeschwächt werden sollten. Somit ergibt sich ein sehr glaubwürdig agierender Schwarm mit stabilen Bewegungen, der immer wieder in eine optimale Form kommt und sich auch von äußeren Einflüssen nicht zu stark beeinträchtigen lässt.

## 5.3 Performanz

Auf dem getesteten System (Notebook mit folgender Hardware-Konfiguration: Intel® Core™ i7-4700MQ CPU, 8GB DDR3 RAM, GeForce GTX 770M Grafikkarte) läuft die Simulation wie in den folgenden Abbildungen (Abbildung 39 bis 41) dargestellt. Wie gut zu sehen ist, hängt die Performanz, hier angegeben in Frames per Second (**fps**, also auch Simulationsschritte pro Sekunde), hauptsächlich von der Anzahl der Tiere ab (Abbildung 39). Werden bei circa 200 Tieren noch Werte von über 400fps erreicht, sinkt die Zahl der Simulationsschritte schon bei etwa 1000 Tieren auf die Hälfte. Die Gesamtkurve sinkt dabei mit steigender Schwarmgröße asymptotisch gegen 0fps. Die Simulation läuft noch bis zu einer Größe von circa 8000 Tieren flüssig mit 30fps. Betrachtet man den Durchsatz

(Abbildung 40), sieht man, dass die optimale Arbeitsgröße bei circa 6000 Tieren liegt. Bei einer geringeren Anzahl ist der Aufwand, die OpenCL-Umgebung zu betreiben, höher als der daraus resultierende Nutzen. Folglich ist die Grafikkarte in diesem Fall de facto unterfordert.

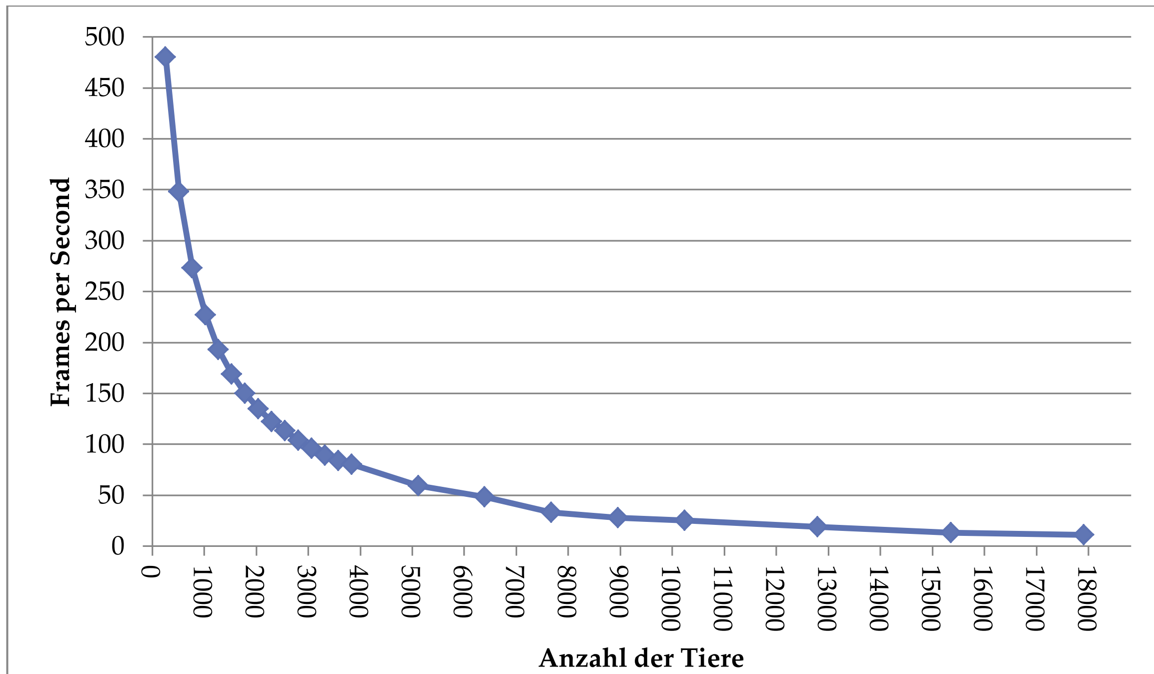



**Abbildung 39: fps in Abhängigkeit der Schwarmgröße (Worksize: 256)**

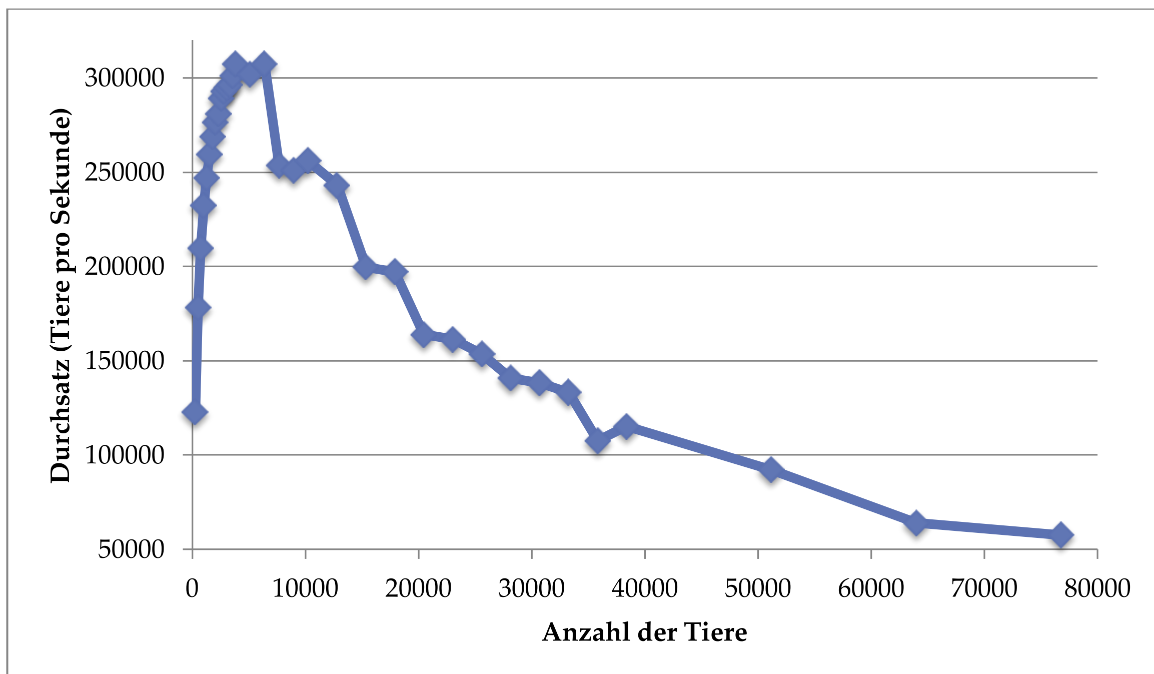



**Abbildung 40: Durchsatz in Abhängigkeit der Schwarmgröße (Worksize: 256)**

Die genaue Größe der Worksize, und der damit einhergehende eingesparte Ladeaufwand der Tiere, wirkt sich nicht allzu sehr auf die Performanz aus (Abbildung 41). Es ist jedoch zu sehen, dass mit steigender Schwarmgröße die Worksize immer mehr Einfluss bekommt. Wirken sich bei einer Schwarmgröße von circa 2000 Tieren Worksize-Größen von über 32 gar nicht aus, sieht man bei etwa 16000 Tieren, dass eine Worksize von 256 optimal ist. Auch bei anderen Schwarmgrößen erzielt der Wert 256 stets eines der besten Ergebnisse.

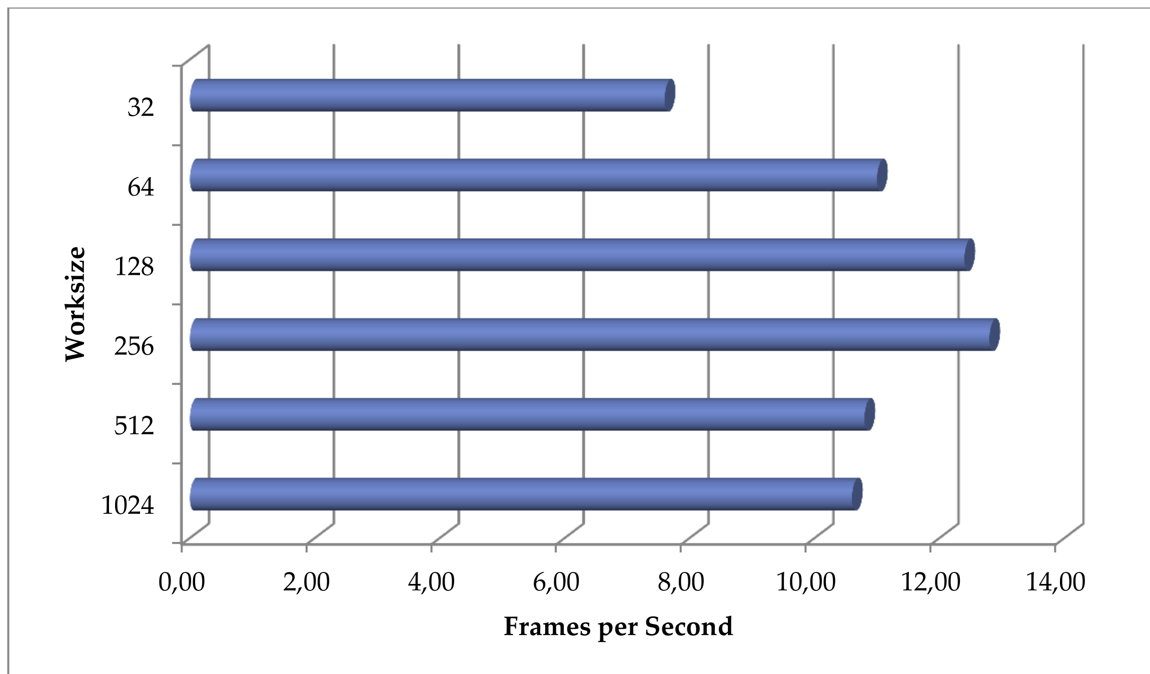



**Abbildung 41: fps in Abhängigkeit der Worksize (#Tiere: 16384)**

Diese Werte sollten bei besserer Hardware-Konfiguration noch etwas steigen, zeigen aber auch jetzt schon, wie gut die Simulation funktioniert. Programme, bei denen ein Tierschwarm nur ein Teil des Gesamtprogramms darstellt, sollten auf eine effiziente Vorsortierung der Tiere setzen, um damit den Rechenaufwand in dieser Hinsicht zu minimieren oder sollten die Anzahl der Tiere reduzieren. Bereits ein Schwarm mit 2000 Tieren stellt meist ein ausreichend großes Gebilde dar.

## 5.4 Generalisierbarkeit

Bei dem hier vorgestellten System wurde großer Wert auf die Generalisierbarkeit und somit Wiederverwendbarkeit gelegt. Daher war der Ansatz, ein System zu entwickeln, dass sich ohne viel Aufwand auf mehrere Schwarmtierarten übertragen lässt. Die Wahl fiel dabei auf Fisch- und Vogelschwärme, da sich diese deutlich in ihrer Umgebung unterscheiden (Wasser

gegenüber Luft), sich aber dennoch beide im 3-Dimensionalen Raum bewegen. Somit sind diese beiden Schwarmarten möglichst gut vergleichbar, ohne sich zu ähnlich zu sein.

Es zeigte sich, dass das System in dieser Hinsicht äußerst flexibel ist. Allein durch die Anpassung von Minimal- und Maximalgeschwindigkeit der Tiere, und natürlich durch den Austausch der Tiermodelle, ergibt sich ein deutlich verändertes Bild, das dem der Vergleichsvideos von echten Schwärmen sehr ähnelt. Dies zeigt wiederum, dass der Grundgedanke, der hinter dem System steht und durch die drei Handlungsvorschriften repräsentiert wird, zu einem Bewegungsmuster führt, das dem echten Handeln von solchen Schwarmtieren sehr nahe kommt. Des Weiteren bekommen auch sehr verschiedene Tierarten, wie Fische und Vögel, durch den Schwarmtrieb eine Gemeinsamkeit, die man ansonsten vielleicht nicht vermutet hätte. Ferner bestätigt dies, wie solche Tiere große, stabile und strukturierte Schwärme bilden können, ohne dabei angeleitet zu werden. Es muss sich lediglich jedes Individuum an gewisse, simple Regeln halten und schon wird ein solches komplexes Gebilde durch emergente Effekte möglich.

So zeigt sich auch bei dem Vogelschwarm ein Bild, das sich weitestgehend mit denen echter Vogelschwärme (Abbildung 46) deckt, wie folgende Bilder (Abbildungen 42 bis 45) verdeutlichen:

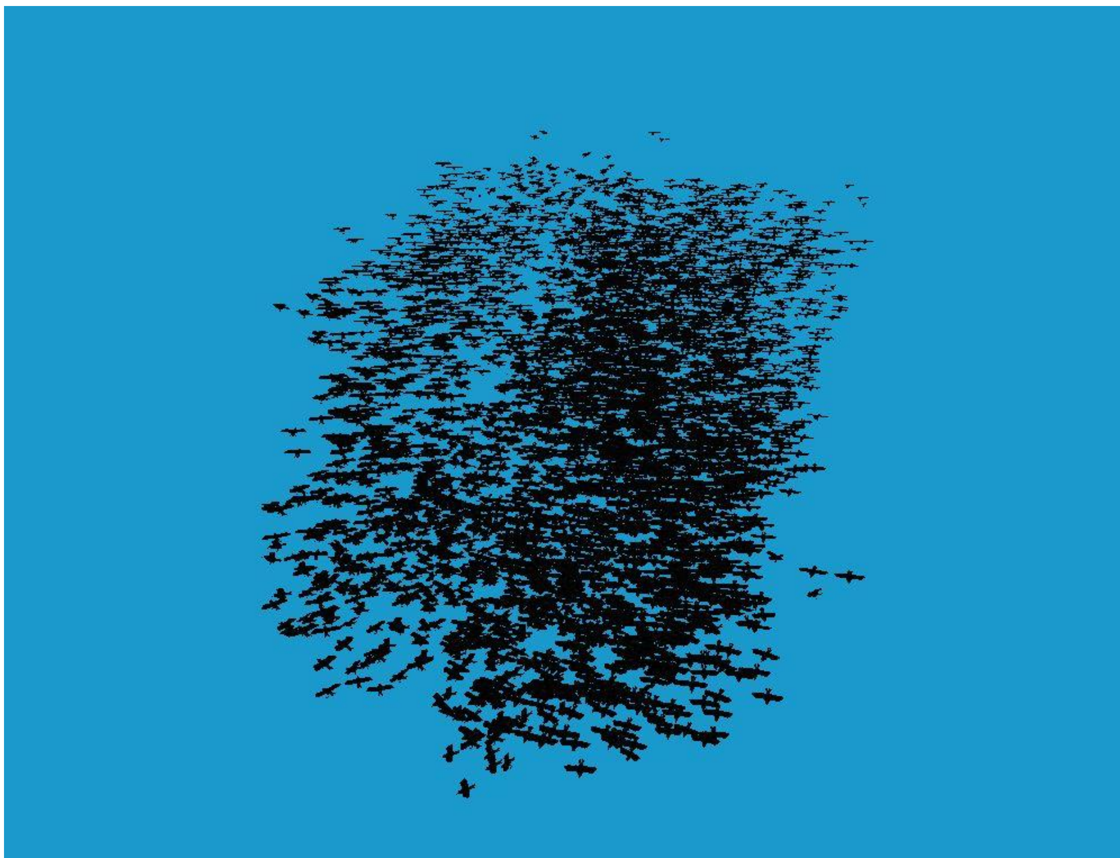


**Abbildung 42: Vogelschwarmimpression I**

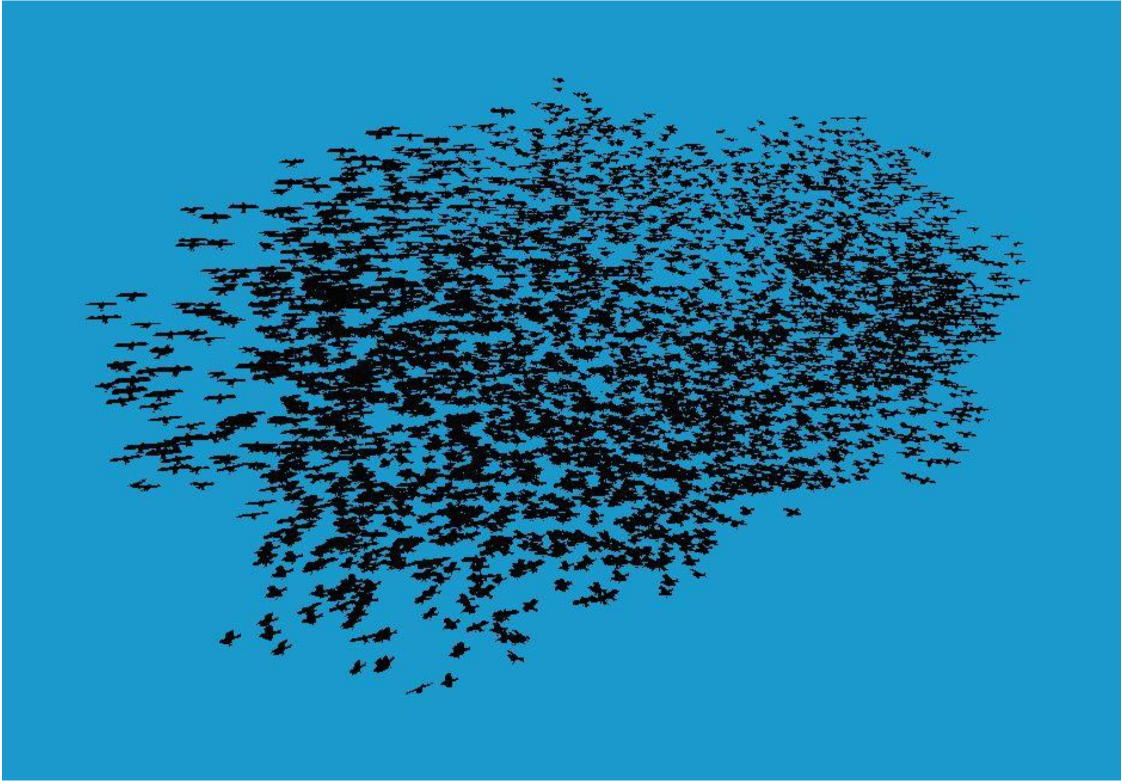


**Abbildung 43: Vogelschwarmimpression II**

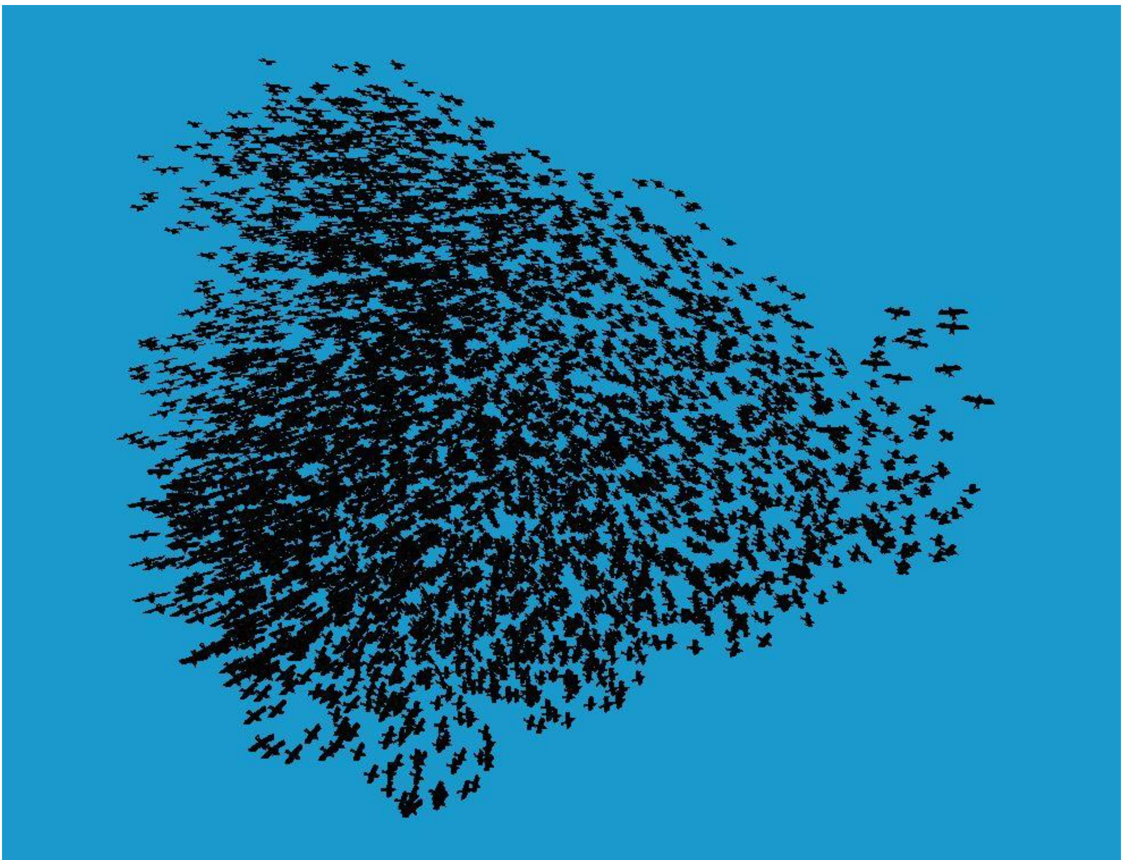


**Abbildung 44: Vogelschwarmimpression III**

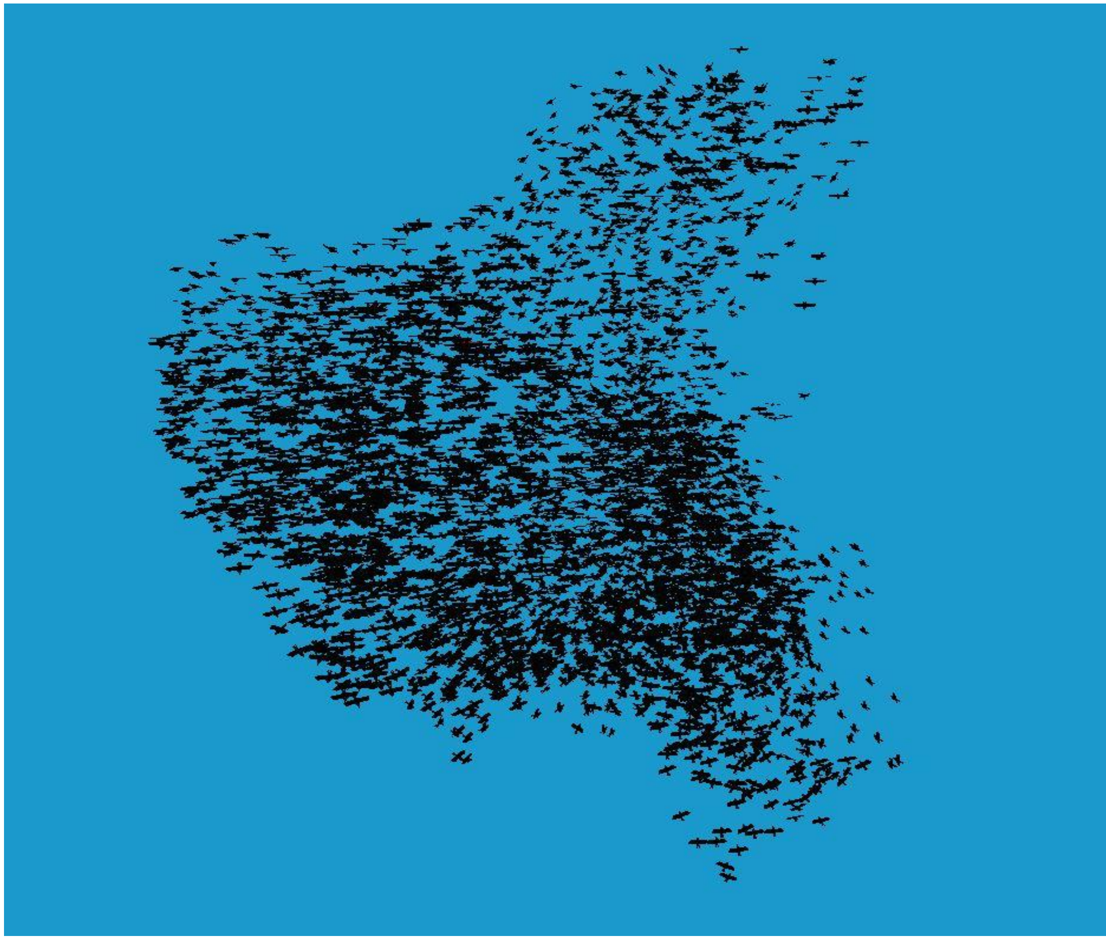


**Abbildung 45: Vogelschwarmimpression IV**

**Abbildung 46: Echter Vogelschwarm (Vergleichsbild)[30]**

# 6 Fazit

Das hier entwickelte System bestätigt, dass sich große, komplexe Schwärme mit verhältnismäßig einfachen Vorschriften und guter Generalisierbarkeit implementieren lassen. Die parallele Herangehensweise an die Berechnungen ermöglicht auch eine große Anzahl an Tieren, was bei anderen Simulationen meist nicht der Fall ist. Mit der richtigen Feinjustierung der Parameter kann ein glaubwürdiges System erstellt werden, welches auch für unterschiedlichste Schwarmtierarten anwendbar ist.

Dabei hängt die optische Glaubwürdigkeit auch immer stark von der Umgebung ab. So sollten auch äußere Einflüsse, wie zum Beispiel Fressfeinde, richtig simuliert werden, um die Reaktion des Schwarms darauf adäquat nachzustellen.

Alles in Allem ist diese Art von System eine gute Methode, komplexe Systeme durch emergente Effekte zu erzeugen und kann in Anwendungen, in denen Schwärme eine Rolle spielen, durchaus verwendet werden, anstatt auf ein vorgefertigtes Gesamtschwarmgerüst zurückzugreifen. Man sollte jedoch darauf achten, dass die Kosten den Nutzen nicht übersteigen. Wird dies berücksichtigt, stellt ein dynamischer Schwarm die deutlich bessere Variante dar.

# 7 Erweiterungen

Das hier vorgestellte System ist lediglich ein erster Schritt in Richtung realistischer Simulation. Es bietet die Basis für Erweiterungen, die den Gesamteindruck eines glaubwürdigen Gefüges vermitteln können. So gibt es noch viele Möglichkeiten das System zu verbessern:

Zunächst ist es sinnvoll, die grundlegende Mechanik noch zu verfeinern. Das Optimieren der Parameter ist dabei ein wichtiges Thema. Es ist jedoch zu beachten, dass dies eine äußerst zeitaufwändige Arbeit ist, dessen Perfektion keine deutliche Verbesserung zu den ersten glaubwürdigen Einstellungen darstellt.

Um noch komplexere und größere Schwärme in Echtzeit zu simulieren, muss zusätzlicher Aufwand betrieben werden, um die Berechnungen effizienter zu gestalten. So wäre es ratsam, die Individuen nach ihrer Position vorzusortieren, um dann anschließend nur die direkte Nachbarschaft zu betrachten, anstatt für jedes andere Individuum zu testen, ob es sich im Sichtradius befindet. Diese Vorsortierung könnte zum Beispiel erreicht werden, indem man eine Würfelgitter-Struktur auf die Welt anwendet und nur die Tiere lädt, die sich

in der eigenen oder den benachbarten Zellen befinden. Somit wird sich die Anzahl der Berechnungen, gerade bei noch größeren Schwärmen, deutlich reduzieren.

Um die Glaubwürdigkeit der Tiere weiter zu erhöhen, sollten noch Animationen für die Tiermodelle hinzugefügt werden. Dadurch wirkt das Tier deutlich realistischer und die Illusion eines echten Schwarms wird verbessert.

Da ein Schwarm meist nicht ausschließlicher Bestandteil einer Welt ist, sind zusätzliche äußere Einflüsse auf den Schwarm unabdingbar. Dabei sind viele Möglichkeiten denkbar: Angefangen bei Strömungs-, beziehungsweise Windeinflüssen, über Hindernisse wie Steine, Bäume oder Boden, bis hin zu anderen Lebewesen, wie zum Beispiel Fressfeinde, oder auch Unterschiede innerhalb des Schwarms, wie etwa Jungtiere gegenüber adulten Tieren. Bei allen Einflüssen sollte aber auch auf ein realistisches Verhalten dieser geachtet werden, da der Schwarm ansonsten nicht wie erwartet reagiert. In dieser Simulation wäre es noch besser, wenn etwa der Jäger auch wie ein solcher agieren und ein komplexeres Verhalten zeigen würde.

Letztlich sollte das System noch in eine größere Anwendung eingegliedert werden, da die Beobachtung eines Schwarms allein höchstens für sehr spezielle biologische Forschungszwecke sinnvoll ist. Man könnte sich aber zum Beispiel ein Spiel vorstellen, in dem man als „Leitfisch“ versucht andere Fische in den eigenen Schwarm aufzunehmen, damit dieser stetig wächst, während man sich vor Fressfeinden schützen muss, indem man den Schwarm rechtzeitig zusammenrottet oder zu großen Feinden aus dem Weg geht. Auch als Teilelement, wie etwa in einer Tauchsimulation, in der man die Unterwasserwelt erkundet, könnte ein solcher Schwarm als faszinierendes, interaktives Gebilde eingebaut werden.

# 8 Quellenverzeichnis

[1] RADAKOV, Dmitriĭ Viktorovich. Schooling in the ecology of fish. 1973.

[2] GRASSÉ, Plerre-P. La reconstruction du nid et les coordinations interindividuelles chezBellicositermes natalensis etCubitermes sp. la théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs. *Insectes sociaux*, 1959, 6. Jg., Nr. 1, S. 41-80.

[3] http://www.wikiservice.at/dse/wiki.cgi?WikiTechnologie

[4] DORIGO, Marco. Optimization, learning and natural algorithms. *Ph. D. Thesis, Politecnico di Milano, Italy*, 1992.

[5] MOUSSAÏD, Mehdi; HELBING, Dirk; THERAULAZ, Guy. How simple rules determine pedestrian behavior and crowd disasters. *Proceedings of the National Academy of Sciences*, 2011, 108. Jg., Nr. 17, S. 6884-6888.

[6] REYNOLDS, Craig W. Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, 1987, 21. Jg., Nr. 4, S. 25-34.

[7] DAVISON, Andrew. *Killer game programming in Java*. " O'Reilly Media, Inc.", 2005, Chapter 22.

[8] http://www.red3d.com/cwr/boids/

[9] http://www.red3d.com/cwr/boids/applet/

[10] http://cmol.nbi.dk/models/boids/boids.html

[11] http://www.bekkoame.ne.jp/~ishmnn/java/boid.html

[12] http://hp.vector.co.jp/authors/VA009508/Java/Boids/boids.html

[13] http://www.decarpentier.nl/boids

[14] http://www.cs.cmu.edu/~scandal/alg/nbody.html

[15] http://www.khronos.org/opencl/

[16] http://www.khronos.org/

[17] http://www.nvidia.de/page/home.html

[18] http://www.nvidia.de/object/cuda-parallel-computing-de.html

[19] https://www.apple.com/de/

[20] http://www.opengl.org/

[21] http://tf3dm.com/

[22] http://tf3dm.com/3d-model/trout-19979.html

[23] http://tf3dm.com/3d-model/crow-50181.html

[24] http://www.blender.org/

[25] http://www.martinreddy.net/gfx/3d/OBJ.spec

[26] http://www.bryanlikestofish.com/2011/12/fish-skins-first-installment.html

[27] http://tf3dm.com/3d-model/crow-14515.html

[28] http://www.aschihaas.ch/serien/serie_santamaria/santamaria_03.php

[29] http://www.fraps.com/

[30] http://www.fotocommunity.de/pc/pc/display/24001056

*Stand der Internetquellen 20.06.2014*

# 9 Abbildungsverzeichnis


Abbildung 1: Primäre Handlungsvorschriften des Boids-Systems ..... 4
Abbildung 2: Speichermodell von OpenCL ..... 9
Abbildung 3: Verwendetes Fischmodell[22] ..... 10
Abbildung 4: Verwendetes Vogelmodell[23] ..... 10
Abbildung 5: Entstehungsweg eines Modells: ..... 11
Abbildung 6: Pseudo-Code des Programmablaufs ..... 13
Abbildung 7: Datenfluss während der Programmausführung ..... 14
Abbildung 8: Kernelfunktion ..... 15
Abbildung 9: Laden der eigenen Werte ..... 16
Abbildung 10: Laden der Workgroup-Daten ..... 17
Abbildung 11: Veranschaulichung der Befehle ..... 17
Abbildung 12: Initialisierung der Vektoren für die Berechnungen der Handlungsvorschriften ..... 18
Abbildung 13: Deklarierung lokalen Speichers ..... 18
Abbildung 14: Daten-Ladeschema ..... 19
Abbildung 15: Laden der Tiere ..... 20
Abbildung 16: Laden der Daten von Tieren innerhalb des Sichtradius‘ ..... 21
Abbildung 17: Vorberechnungen für die Handlungsvorschriften mit Hilfe der Nachbartiere ..... 22
Abbildung 18: Modifikation von Umgebungsvariablen durch den Jäger ..... 22
Abbildung 19: Berechnung der Gesamtkraft ..... 23
Abbildung 20: Direkte Reaktion auf Jäger ..... 23
Abbildung 21: Umsetzung der „kleineren“ Regeln ..... 24
Abbildung 22: Schreiben der Daten ..... 24
Abbildung 23: Ausschnitt aus der Fischmodell.obj-Datei ..... 25
Abbildung 24: Fisch als Drahtgittermodell (links) und mit Textur[26] in der fertigen Simulation (rechts) ..... 26
Abbildung 25: Vogel als Drahtgittermodell (links) und mit Textur[27] in der fertigen Simulation (rechts) ..... 26
Abbildung 26: Winkel zur Drehung um die z-Achse (Seitenansicht) ..... 27
Abbildung 27: Winkel zur Drehung um die y-Achse (Draufsicht) ..... 28
Abbildung 28: Wechsel der Zugriffsrechte auf die Daten-Buffer ..... 30
Abbildung 29: Initialformation des Schwarms ..... 31
Abbildung 30: Aufbrechen der Initialformation ..... 31
Abbildung 31: Erste Orientierung zum Schwarmgefüge ..... 32
Abbildung 32: "Umorientierung" zur Mitte ..... 32
Abbildung 33: Annähernd optimale Form ..... 33
Abbildung 34: Optimale Form mit fehlender Ausrichtung ..... 33
Abbildung 35: Ausgerichtetes, stabiles, optimales Schwarmgefüge ..... 34
Abbildung 36: Echter Fischschwarm (Vergleichsbild)[28] ..... 34
Abbildung 37: Verhalten des Schwarms bei Anwesenheit des Jägers ..... 35
Abbildung 38: Auseinanderstieben bei geringer Jägerdistanz ..... 36

Abbildung 39: fps in Abhängigkeit der Schwarmgröße (Worksize: 256) ........ 38
Abbildung 40: Durchsatz in Abhängigkeit der Schwarmgröße (Worksize: 256) ........ 38
Abbildung 41: fps in Abhängigkeit der Worksize (#Tiere: 16384) ........ 39
Abbildung 42: Vogelschwarmimpression I ........ 40
Abbildung 43: Vogelschwarmimpression II ........ 41
Abbildung 44: Vogelschwarmimpression III ........ 41
Abbildung 45: Vogelschwarmimpression IV ........ 42
Abbildung 46: Echter Vogelschwarm (Vergleichsbild)[30] ........ 42

**Selbstständigkeitserklärung**

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Zuhilfenahme der angegebenen Quellen erstellt habe. Dabei habe ich diese Arbeit, weder als Ganzes noch in Teilen, für den Erhalt eines Abschlusses an dieser oder einer anderen Universität eingereicht.

______________________________________

*Oliver Tschesche*

Osnabrück, den 01.07.2014