Universität Bremen

Fachbereich 3 Mathematik und Informatik

Bachelorarbeit

Informatik

Refactoring einer existierenden Code-Basis zur Simulation von Fisch-Schwärmen und Weiterentwicklung deren Verhaltens

Joscha Cepok <redlotus@tzi.de>
Matrikel-Nr. 2763987

Abgabe:	30. März 2016
Erstprüfer	Herr Prof. Dr. Gabriel Zachmann
Zweitprüfer	Herr Dr. Rene Weller

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Ort, Datum: Unterschrift:

Das Ziel dieser Bachelorarbeit ist, die Codebasis der Fischschwärme aus dem Bachelorprojekt VR CoralReef neu zu strukturieren und zu erweitern, sodass eine einfache Integration von neuen Fischen und Schwärmen mit ihren Verhaltensmustern einfach umsetzbar ist. Dazu soll ein weiteres Modul implementiert werden, welches es ermöglicht die bisher nicht vermiedenen Kollisionen von Fischen mit Objekten zeitkritisch zu verhindern.
Da nach der Performancemessung des aktuellen Kollisionserkennungsverfahrens aufgefallen ist, dass dieses nur für einzelne Objekte verwendbar ist und eine Kollisionsvermeidung nach diesem Verfahren unnatürlich aussieht, wurde ein neues Kollisionsverfahren entwickelt. Dieses Verfahren funktioniert über die Abstrahierung und Zusammenfassung von Objekten über kleine Gruppen von Sphären und Boxen ohne großen Rechenenaufwand frühzeitig Kollisionskurse erkennt und diese durch eine stetige Anpassung der aktuellen Kurse auf Ausweichkurse verhindert. Die Implementierung dieses Verfahrens wurde hierbei generisch gehalten, sodass es über die Simulation hinaus für andere Probleme anwendbar ist.
Die Klasse `Fish` wurde zum Zweck der Integration des neuen Kollisionsverfahrens gewartet und pro Fisch oder Schwarmverhalten verstärkt modularisiert, sodass eine Integration von neuen Verhaltensmustern und neue Schwärme sowie eine Kollisionsvorbeugung vereinfacht möglich sind.
Weiterhin wird das Verfahren beschrieben, wie neue Fische und Schwärme in die Simulation hinzugefügt werden. Diese Beschreibung umfasst zunächst die Implementierungsschritte, welche für die Integration notwendig sind. Neben den Schritten für die Implementierung wird ebenfalls beschrieben, welche Parameter und Skalierung zu welchem Verhalten der Fischschwärme führen.

The aim of this bachelor thesis is to structure and enhance the code basis of the fish swarms of the bachelor project VR CoralReef so that a simple integration of new fish and swarms with their behavioural patterns is easily doable. Additionally, another module shall be created which makes it possible to stop the collision of fish with objects in time.
It has become clear after the performance measurement of the current collision detection procedure that this can only be used for single objects and a collision prevention looks unnatural this way, so a new collision procedure was developed which detects collision courses early via abstracting and summarising objects in small groups of spheres and blocks without big calculation effort and prevents these collisions by adjusting their current course onto evasive courses. The implementation of this procedure was held generically so that it can be used for other problems than the fish simulation.
The category fish was improved to integrate the new collision procedure and stronger modularised per fish or swarm so that an integration of new behavioural patterns and new swarms as well as a kollision prevention are easily possible.
Furthermore, the procedure to add fish into the simulation will be described. This description includes the implementation steps which are necessary for the integration. Besides these steps, which parameters and the scaling of these parameters lead to which behaviour of the fish swarms will be described.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation und Ziele . . . 1

1.2 Eigener Anteil . . . 1

1.3 Überblick . . . 2

1.4 Grundlagen . . . 3

1.5 Glossar . . . 4

2 Verwandte Arbeiten **5**

3 Kollisionserkennung und Ausweichkurse **8**

3.1 Kollisionserkennung . . . 8

3.1.1 Kollisionserkennung für den Beobachter . . . 8

3.1.2 Gegenwärtige Kollisionserkennung . . . 8

3.1.3 Performancemessung . . . 10

3.1.4 Probleme mit Meshkollisionen . . . 12

3.1.5 Parallelisierung . . . 13

3.2 Kollisionsvermeidung . . . 14

3.2.1 Abstrahierung von Meshes . . . 14

3.2.2 Kollisionserkennung durch Abstraktion . . . 14

3.2.3 Erfassung von Kollisionsobjekten . . . 17

3.2.4 Problemstellung bei der Erfassung . . . 18

3.2.5 Ermittlung der Ausweichrichtung . . . 19

4 Architektur und Implementierung 22
4.1 Architektur 22
4.1.1 Klassendiagramm 22
4.1.2 Sequenzdiagramm 24
4.2 Implementierung 26
4.2.1 Erfassung von Kollisionsobjekten 26
4.2.2 Kollisionsvorbeugung 30

5 Refactoring der Klasse Fish 33
5.1 Problemstellung 33
5.2 Modularisierung des Fluchtverhaltens 33
5.3 Entfernung von Attributen 34
5.4 Integration des Ausweichverhaltens 35
5.5 Vereinheitlichung der Bewegungsberechnung 36

6 Schwärme hinzufügen 38
6.1 Betroffene Dateien 38
6.2 Ablauf 39
6.2.1 Mesh erstellen 39
6.2.2 Config erweitern 39
6.2.3 String Konstanten 39
6.2.4 Zusätzliche Attribute definieren 41
6.2.5 Config auslesen 41
6.2.6 Schwarm initialisieren 41
6.2.7 Schwarm aktualisieren 43
6.2.8 Fisch aktualisieren 43
6.3 Verhaltensregeln für Fische im Schwarm 44

7 Fazit 46

8 Ausblick 48

Kapitel 1

Einleitung

In diesem Kapitel wird die Motivation zur Erstellung der Arbeit, die Struktur und der eigene Beitrag zum Erreichen der Ziele erläutert.

1.1 Motivation und Ziele

Im Rahmen dieser Bachelorarbeit soll das Projekt VRCoralReef [VRC] erweitert werden. Das Projekt VRCoralReef ist eine Simulationsumgebung für Korallenriffe, in der ebenfalls Fische und Meerestiere, die auf dem Boden leben, simuliert werden. Das Ziel der Bachelorarbeit ist es, die Codebasis für die Fische und Schwärme zu warten, sodass die Erweiterung des Codes und eine einfache Integration von neuen Fischen und Schwärmen mit ihren Verhaltensmustern möglich ist. Zusätzlich dazu soll die nicht implementierte Kollisionserkennung als eigenes Modul implementiert werden, welches die Fische, bevor sie auf ein Mesh treffen auf einen Ausweichkurs lenkt. Dieses Modul soll so implementiert werden, dass es sich einfach in andere Projekte mit ähnlichen Problemstellungen übertragen lässt.

1.2 Eigener Anteil

Der eigene Beitrag zu der Simulationsumgebung ist die Implementierung und Einbindung eines Moduls, welches die Daten aller Objekte in der Simulation erfasst, für die eine Kollisionsvermeidung mit stattfinden soll. Dieses Modul soll über eine Config Datei sämtliche Parameter über einen Adapter einlesen, welche zur Umsetzung der Kollisionsvermeidung genutzt werden. Des Weiteren ist das Ziel, dass die Vorgänge in den Klassen für die Schwarmlogik mit einem Design gewartet werden, welches einerseits über die Struktur der Methoden Rückschlüsse

über deren Funktionalität bietet, als auch Vorgänge stärker modularisiert. Es ist kein Ziel der Wartung, die Performance zu optimieren. Stellen an denen dies möglich ist. werden ohne diese explizit zu erwähnen verbessert. In dem letzten Abschnitt werden alle Schritte erläutert, wie nach dem Refactoring und der Einbindung des neuen Moduls neue Schwärme in die Simulation hinzugefügt werden.

1.3 Überblick

In dieser Bachelorarbeit wird die Logik der Fische und Fischschwärme ausgebaut, indem der gesamte Code, der mit dieser Logik in Verbindung steht, verbessert wird. Das Ziel ist eine Vereinfachung der Wartung sowie die Integration von neuen Verhaltensmustern zu erreichen. Hierfür werden im Kapitel [2] Veröffentlichungen beleuchtet, welche sich mit der Kollisionserkennung, Kollisionsvorbeugung und mit Implementierungstechniken auseinandersetzen, die für die Umsetzung dieses Projekts relevant sind. Im Kapitel [3] wird daraufhin das gegenwärtige Kollisionserkennungsverfahren in [3.1.2] beschrieben. Das Kapitel [3.1.3] analysiert zunächst die Performance, die der aktuelle Algorithmus im Schnitt braucht, und beurteilt anhand der Messergebnisse die globale Umsetzbarkeit des Verfahrens, auf Basis der Performance. Danach werden diese Erkenntnisse durch weitere Anforderungen an das Zielverhalten in Kapitel [3.1.4] ergänzt, woraufhin in Kapitel [3.2] ein Vorgehen erläutert wird, welches die Kernprobleme löst. Der Algorithmus zu diesem Verfahren, welcher das Problem durch eine abstrakte Darstellung von Meshes löst, wird draraufhin in Kapitel [3.2.2] erklärt. In Kapitel [3.2.3] wird daraufhin die Erfassung von der Meshes und die Integration des Algorithmus in das Korallenriff erläutert. Das folgende Kapitel [3.2.4] befasst sich zum Schluss mit den Problemen und Auswirkungen auf die einzelnen Fische, die mit dem Verfahren ihre Bewegungsrichtung ändern.
In dem Abschnitt [4] wird die Umsetzung der Implementierung der Erfassung der Kollisionsobjekte in der Simulationsumgebung beschrieben. Im ersten Teil 4.1 werden die grundlegende Architektur und die Abläufe für die neuen Klassen beschrieben. In dem zweiten Teil [3.2] wird zunächst geschildert, auf welche Art die Kollisionsobjekte in der Simulationsumgebung eingelesen werden. In dem zweiten Kapitel [4.2.2] wird daraufhin erläutert, wie mit den erfassten Kollisionsobjekten die Kollisionsvorbeugung umgesetzt wird.
Im Abschnitt [5] werden die Maßnahmen beschrieben, welche umgesetzt werden müssen, damit die Klasse `Fish` für eine einfache Integration von neuen Schwärmen und die Integration von dem Ausweichverhalten genutzt werden kann. Dafür wird in

[5.1] zunächst beschrieben, nach welchem Entwurfsmuster die Klasse gewartet wird, sowie welche Maßnahmen getroffen wurden, damit der Code einfacher verständlich ist und wie die neuen Algorithmen integriert werden müssen. In Kapitel [5.4] werden daraufhin die Implementierung und die vereinfachenden Maßnahmen für Schwärme erläutert.
Im letzten Kapitel [6] wird beschrieben, welche Schritte zu durchlaufen sind, damit ein neuer Schwarm integriert werden kann. Der Inhalt dieses Kapitels bezieht sich ausschließlich auf die Implementierung, berücktsichtigt jedoch nicht die Erstellung eines Meshes. Zusätzlich zur Implementierung wird erläutert, welche Skalierung der einzelnen Parameter zu welchem Verhalten führt.
Im Anschluss werden im Abschnitt [7] die Ergebnisse der Integration anhand der Performancemessung nach der Integration und der Wartung erläutert.
Der letzte Abschnitt [8] bewertet daraufhin die Umsetzung der Ziele der Bachelorarbeit kritisch und betrachtet die allgemeine Nutzbarkeit des entwickelten Algorithmus.

1.4 Grundlagen

In diesem Kapitel werden die grundlegenden Kenntnisse über die Architektur des Projektes im Ausgangszustand und im finalen Zustand erläutert, die notwendig sind, um den Inhalt der Bachelorarbeit zu verstehen. In der folgenden Abbildung [1.1] werden sämtliche Klassen und Beziehungen aufgelistet, die für den Inhalt der Bachelorarbeit relevant sind.
Es sind 5 Klassen aus 3 Packages betroffen. Die Klasse `GameState` wurde mit aufgelistet, da sämtliche Events in der Simulation dort ausgelöst werden, jedoch ist diese Klasse von dem Refactoring ausgeschlossen. Die blauen Klassen `FishManager, SwarmManager, Swarm, Fish` und `CollisionManager` sind alle direkt von dem Refactoring betroffen und werden an einigen Stellen verändert. Die gelben Klassen `DodgePointService, DodgePointAdapter` und `DodgeObject` werden in einem neuen Package hinzugefügt und dienen zur Realisierung der Integration eines Moduls, welches für die Anpassung der Fischkurse implementiert wird.
Sämtliche Messungen wurden mit einem Intel(R) Core(TM) i5-2450M CPU @ 2.50GHz(4CPUs), 2.5GHz Prozessor und einer Intel(R) HD Graphix 3000 Grafikkarte durchgeführt.

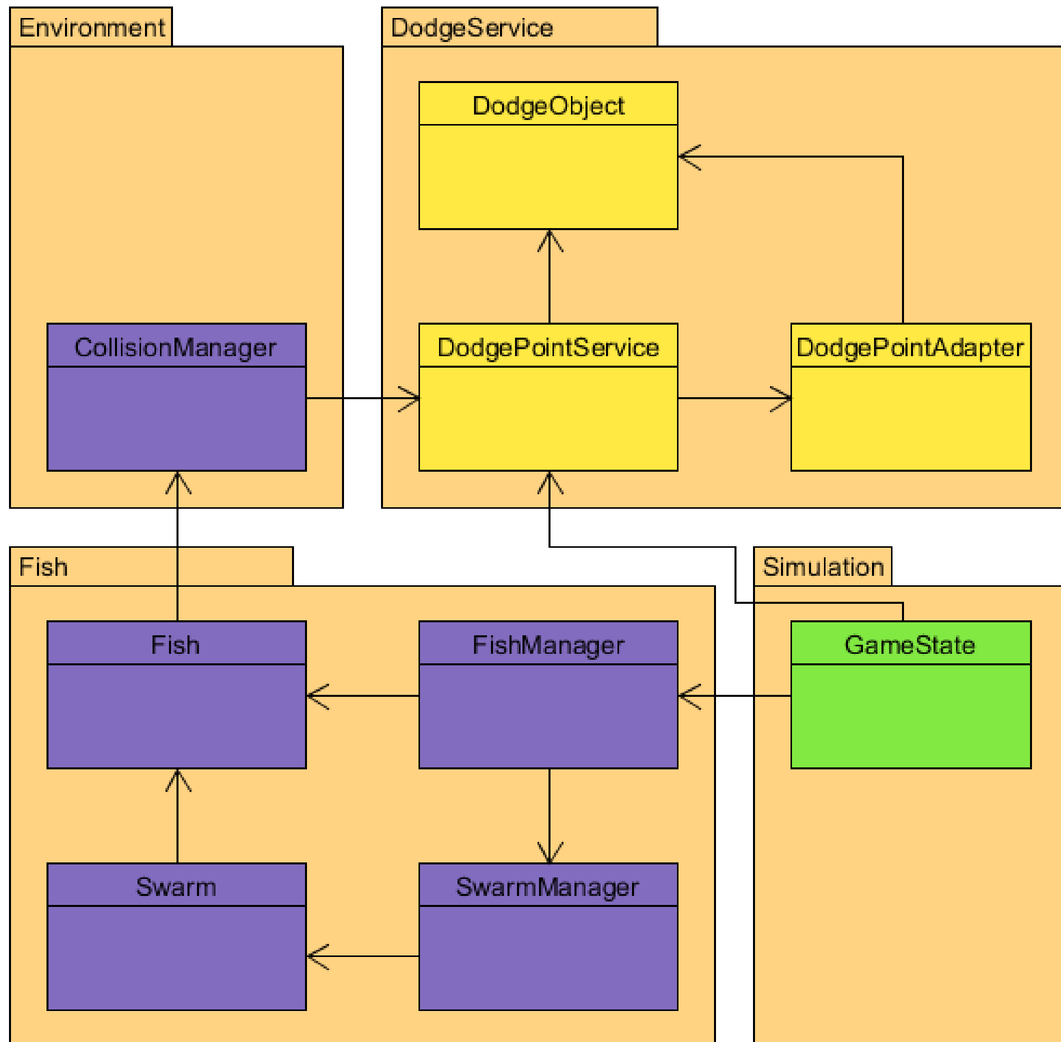

Abbildung 1.1: Grundlagen der Architektur

1.5 Glossar

Mesh	Ein Polygonnetz zur Modellierung von 3D Objekten
Raycast	Ein Verfahren um Objekte die auf einer Geraden liegen zu ermitteln
FPS	Frames per second, Anzahl der Bilder pro Sekunde
Framerate	Die Zeit zwischen 2 Frames
CQS	Command-Query-Separation
CQRS	Command-Query-Responsibility-Segregation

Kapitel 2

Verwandte Arbeiten

Es gibt für diese Problemstellung bereits einige interessante Arbeiten, die sich einerseits mit der Problemstellung Kollisionserkennung auseinandersetzen, als auch das Thema Kollisionsvermeidung behandeln. Ergänzend hierzu gibt es Arbeiten, die sich mit der abstrakten Darstellung von Objekten als z. B. Sphären auseinandersetzen. Im folgenden Kapitel werden die Grundlagen dieser Ansätze näher beleuchtet sowie ein Fazit für die Implementation getroffen.

Ein Vorgehen ist das ProtoSphere Verfahren [WZ10b], ein weiteres Verfahren ist das Inner Sphere Trees for Proximity and Penetreation Queries Verfahren [WZ10a] von Rene Weller und Gabriel Zachmann. Beide Verfahren basieren auf der Idee, Objekte als Sammlung von nicht überlappenden Sphären darzustellen. Zusätzlich dazu wird bei dem Inner Sphere Tree Verfahren [WZ10a] die zeitkritische Kollisionserkennung weiter beleuchtet. Ziel dieses Verfahrens ist es, die Oberfläche von Objekten möglichst präzise durch Sphären wiederzugeben. In der Bachelorarbeit werden Meshes ebenfalls in Sphären und andere mathematische Körper zerlegt. Dieses Verfahren lässt sich für die Transformierung von Meshes in Sammlung von Sphären jedoch nur begrenzt gut anwenden, da das in der Bachelorarbeit verwendete Verfahren zur Umsetzung möglichst wenige möglichst große Sphären benötigt, um effizient und ohne Nebeneffekte zu funktionieren.

Ein weiteres Vorgehen ist das Point Cloud Collision Detection Verfahren von Jan Klein und Gabriel Zachmann [ZK04]. Mit diesem Verfahren werden Punktgruppen zu groben Sphären zusammengefasst und innerhalb dieser Sphären iterativ weitere Gruppen gebildet. Das Verfahren Meshes in Gruppen von Sphären und andere mathematische Körper zu zerlegen, ist ein essentieller Bestandteil des erarbeiteten Verfahrens, um den Aufwand für die Kollisionsvermeidung zu reduzieren.

Zur Kollisionsvermeidung gibt es ein Paper von M.R. Hafner D. Cunningham L. Camitiniti D.Del Vecchio [HCCV11]. In diesem Paper ist der Schwerpunkt eine zeitkritische Kollisionserkennung und Vorbeugung durch Kurvenanpassung. Das Ziel dieses Verfahrens ist es, das Objekt auf bestimmten Routen an den Kollisionsobjekten vorbei zu navigieren. Für die Kollisionsvermeidung mit Fischen ist dies jedoch deutlich vereinfacht, da das Verfahren auf einer 2D-Ebene mit Straßen ausgelegt ist. In dem Korallenriff gibt es im dreidimensionalen Bereich jedoch deutlich mehr Ausweichrouten, ohne fest vorgegebene Bedingungen beachten zu müssen.

In dem Buch Implementing Domain Driven Design [Ver13] werden Vorgehen zur Implementierung von Domänen erläutert, welche als Module gut in Projekte integrierbar sind. Die einzelnen Klassen der Domänen werden nach Regeln implementiert, die ihnen per Defintion bestimmte Steuermöglichglichkeiten und Zustände zusprechen, wodurch unerwartetes Verhalten und Nebenläufigkeit vermieden wird.
Die beiden wichtigsten Regeln sind die Verfahren CQS und CQRS. CQS steht für command-querry seperation: Die Idee hinter diesem Entwurfsmuster ist, über den Rückgabewert der Methode Rückschlüsse über den Inhalt der Methode zu erhalten. Unterschieden wird hierbei zwischen void und nicht-void Methoden. Alle Methoden, die den Rückgabewert void haben, verändern den Zustand des Objektes, für das sie implementiert wurden. Alle nicht-void Methoden berechnen mit den Attributen des Objektes Parameter, verändern aber den Zustand des Objektes selbst nicht. Das Refactoring des Moduls, welches für die Schwarmlogik zuständig ist, wird im Wesentlichen die Struktur der Klassen gemäß des CQS und CQRS Prinzips umstrukturieren.

In dem Buch Design Patterns Elements of Reusable Object-Orientet Software [GHJV95] werden verschiedene Designpattern beschrieben, welche in bestimmten Situationen zur Lösung von neuen Problemen beitragen. Das aus diesem Buch verwendete Pattern ist Abstract Factory. Dieses Entwurfsmuster dient vor allem zur einfachen Einlesung von verschiedenen Objekttypen. In der Arbeit wird dieses Entwurfsmuster nicht direkt implementiert, jedoch ist das Einleseverfahren durch dieses Entwurfsmuster stark inspiriert worden.

In dem Buch Curves and Surfaces for Computer Graphics [Sol06] werden verschiedene Algorithmen erläutert, wie Oberflächen und deren Normalen mathematisch dargestellt werden. Aus diesem Buch wurden einige Formeln für die Darstellung der Oberflächen von mathematischen Körpern übernommen und so abgeändert, dass

eine Prüfung, ob sich Koordinaten innerhalb des Körpers befinden, stattfinden kann.

In dem Paper Improved Collision detection and Response [Fau03] von Kasper Fauerby werden die grundlegenden Möglichkeiten, wie man Kollisionen vermeiden kann, beschrieben. Das implementierte Verhalten für die Kollisionsvermeidung der Fische wurde durch die hier beschriebenen Algorithmen inspiriert. Der Algorithmus funktioniert über die Neuberechnung der Position. Für die Problemstellung Fische den Objekten ausweichen zu lassen, wurde das Verfahren so angepasst, dass die Fische ihre Kurse ändern, um optisch auffällige Effekte zu vermeiden.

In dem Buch Game Programming Gems [Dic06] von Michael Dickheiser wird ein Verfahren beschrieben, in dem komplexe Objekte mit einfachen mathematischen Körpern umgeben werden, um diese Berechnungseinfach darzustellen. Dazu wird dort im Ausblick erläutert diese Erkennung für weitere Operationen zu nutzen. In dieser Bachelorarbeit wurde das Verfahren, Objekte durch mathematische Körper darzustellen, stark von diesem Gedanken inspiriert.

Das Paper Collision Avoidance for Multiple Agent Systems [CSMOS03] von Dong Eui Chang, Shawn C. Shadden, Jerrold E. Marsden und Reza Olfati-Saber befasst sich mit Schwarmverhalten und Kollisionsvermeidungsverfahren. Das in diesem Projekt implementierte Verfahren wurde stark durch die in diesem Paper erwähnten Algorithmen inspiriert und auf die dreidimensionale Simulationsumgebung angepasst.

In dem Buch Java Modeling In Color With UML [CLL99] werden gängige UML Diagramme und die Hervorhebung und Trennung von Zusammenhängen durch farbliche Kennzeichnung erläutert.

Kapitel 3

Kollisionserkennung und Ausweichkurse

3.1 Kollisionserkennung

3.1.1 Kollisionserkennung für den Beobachter

In der Simulation des Bachelorprojekts VR CoralReef gab es zum Zeitpunkt des Abschlusses noch keine Implementation für das Ausweichverhalten von Fischen, wenn diese auf Meshes trafen. In dem Projekt wurde bereits ein Algorithmus implementiert, welcher eine verwandte Problemstellung gelöst hat, die Kollisionen von Spieler und Meshes zu erkennen und zu verhindern. Dieser Algorithmus hat die Position des Spielers, sobald dieser auf ein Mesh traf, pro Frame angepasst, sodass er an dem Mesh vorbeiglitt. In diesem Kapitel wird zunächst der aktuelle Kollisionserkennungsalgorithmus analysiert und überprüft, ob dieser für die geplante Implementierung Ausweichkurse für die Fische zu ermitteln verwendbar ist. Mit der Auswertung der Analyse wird daraufhin ein geeigneter Ansatz in einer Klasse entwickelt, welcher die Problemstellung der Kollisionsvermeidung geeignet löst. Neben der Lösung für die reine Kollisionsvermeidung mit Meshes werden ebenfalls weitere Attribute, die ebenfalls einen Ausweichkurs erzwingen, in diese Klasse übernommen, um das Ausweichverhalten in der Klasse `Fish` weiter modularisieren zu können.

3.1.2 Gegenwärtige Kollisionserkennung

Die aktuelle Implementierung des Kollisionsalgorithmusses benötigt für die Berechnung der neuen Attribute die alte Position, die neue Position und die Distanz zu dem Mesh, die eingehalten werden soll. Der Rückgabewert gibt an, ob eine Positionsänderung stattfand. Die Positionsveränderung findet in paralleler Richtung zu

dem Polygon des Meshes statt, welches bei der Kollisionserkennung getroffen wurde. Dadurch lässt der Algorithmus den Spieler an dem Mesh, das er trifft, vorbeigleiten. Der Algorithmus wurde primär dazu entwickelt, Kollisionen des Beobachters mit Meshes zu vermeiden, die im `CollisionManager` registriert wurden. In diesem Abschnitt wird der Algorithmus auf sein Vorgehen und seine Voraussetzungen analysiert. Für diesen Algorithmus wird daraufhin eine Performancemessung durchgeführt. Als Fazit wird auf Basis dieser Daten eine Entscheidung getroffen, ob dieser Algorithmus durch geeignetes Refactoring für das Kollisionsproblem mit den Fischen anwendbar ist, oder ob ein anderer Ansatz besser geeignet wäre.
Zunächst wird der Ablauf des Algorithmus genau untersucht, da jedes Detail einen Einfluss auf die Performance und Umsetzbarkeit haben kann.
Die Implementierung des gegenwärtigen Algorithmus erfolgt wie in der Abbildung [3.1] beschrieben:

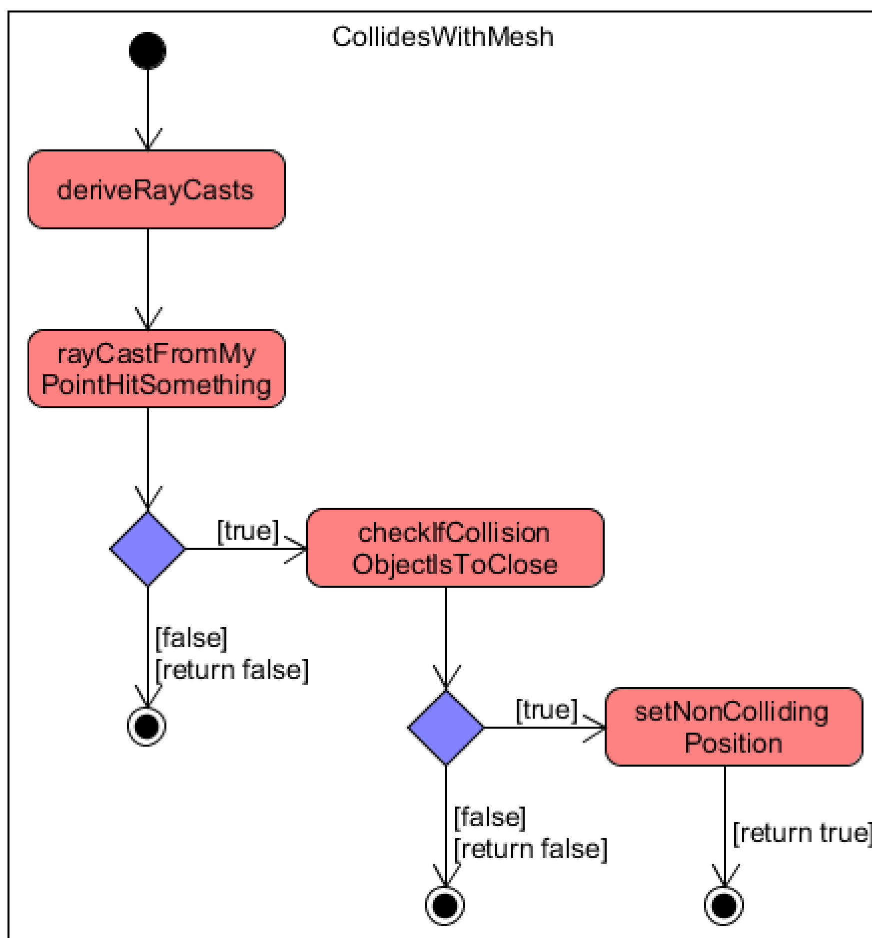

Abbildung 3.1: Ablauf der Funktion collidesWithMesh(..)
Parameter: alte Position, neue Position, minimale Distanz.

Um alle relevanten Meshes zu erfassen, werden neun Raycasts in verschiedenen Winkeln kreisförmig nach vorn durchgeführt. Dies ist notwendig, um nicht nur alle Meshes, die direkt auf dem aktuellen Kurs vor der Position sind, für das die Kollisionsüberprüfung durchgeführt wird, zu erfassen. Die Erfassung über mehrere Raycasts verhindert, dass ein Mesh für die Kollisionserkennung nicht erfasst wird und dieses für den Spieler optisch auffällig gestriffen wird.
Für alle getroffenen Meshes wird zuerst überprüft, ob diese nah genug für eine Kollision an der aktuellen Position sind. Falls dies der Fall ist, wird für das Mesh jedes Polygon berechnet und für dieses überprüft, ob der aktuelle Kurs dieses durchstoßen würde. Ist dies der Fall, wird die weitere Kollisionsberechnung für dieses Polygon durchgeführt. Sollte kein Polygon getroffen werden, so ist das Mesh nicht relevant und das nächste wird überprüft.
Wird ein Polygon ermittelt, so wird die Position des Spielers mit Hilfe der Normalen des Polygons auf eine nicht kollidierende parallele Position zu dem Polygon gesetzt und true als Rückgabewert geliefert, damit die Methode, die die Funktion nutzt, die erzwungene Positionsänderung erkennt.
Der aktuelle Kollisionsberechnungsalgorithmus ist somit stark von performanceintensiven Raycasts abhängig, die unterschiedlich viele Meshes erfassen können, die zum Teil trivial sein können. Zusätzlich dazu benötigt der Algorithmus in jedem Fall die Meshes der Objekte, die erfasst wurden, um Kollisionen zu berechnen. Dies sorgt für einen stetigen Minimalaufwand pro Fisch durch die Raycasts, welche durch die stark unterschiedliche Anzahl der Polygone pro Mesh stark variieren können. Dieser Algorithmus besitzt somit einen relativ großen Mindestaufwand, welcher durch die Anzahl der getroffenen Meshes enorm ansteigt und variiert. Somit variiert die Dauer für den Algorithmus, je nachdem in welcher Situation sich der Fisch befindet, enorm.

3.1.3 Performancemessung

Um herauszufinden, ob die Performance der Kollisionserkennung nach dem gegenwärtigen Verfahren in irgendeiner Form durch Refactoring nutzbar ist, wird ebenfalls gemessen wie hoch der durchschnittliche Aufwand und die durchschnittliche Abweichung von diesem ist, um eine Entscheidung treffen zu können. Die Berechnung des Aufwands wird an dieser Stelle abstrakt gehalten. Somit werden hier Raycasts, die von den Hintergrundprozessen sehr teuer sind, mit Polygonen verglichen, welche für die Berechnung nicht viel Performance benötigen. Dazu werden die Submeshes mit dem Aufwand O(1) vereinfacht, wobei diese ebenfalls unterschiedlich komplex ausfallen können. Der ermittelte Aufwand stellt somit den Minimalaufwand dar. Wenn alle Operationen die gleiche Zeit benötigen würden, jedoch einige Operationen stark herunter skaliert wurden, ist der tatsächliche

erwartete Aufwand erheblich größer.

Für die Berechnung des durchschnittlichen Aufwands wurde zunächst eine Messung mit dem Spieler durchgeführt, welcher über die FolgenOption dem kohärenten Schwarm drei Minuten lang folgte. Der kohärente Schwarm wurde als Observierungsobjekt gewählt, da davon auszugehen ist, dass dieser den Großteil der Performance für die Kollisionsberechnung benötigt und somit ausschlaggebend für die Validierung des gegenwärtigen Algorithmus ist. Die gemessenen Daten sind eine Stichprobe, die Rückschlüsse über die Auslastung angibt, die der kohärente Schwarm verursachen würde. Die Oberservierungsdauer von drei Minuten, in denen der Schwarm mehrere Male durch die gesamte Simulationsumgebung geschwommen ist, reicht aus, um grobe Messausreißer zu vermeiden. Es können jedoch können andere Messungen zu stark abweichenden Ergebnissen führen, welche jedoch von der Größenordnung vergleichbar bleiben.

Die Messung ergab folgende Werte:
Anzahl der Raycasts: 1
Anzahl der getroffenen Meshes aller Raycasts im Durschnitt: 3
Anzahl der Submeshes pro Mesh im Durchschnitt: 5
Anzahl der Polygone pro Mesh im Durchschnitt: 3112.5

Tabelle 3.1: Anteil getroffener Meshes

Anzahl Polygone	250	350	1200	2000	3400	9000
Prozentualer Anteil	5	20	10	30	15	20

Standardabweichung der Polygone pro Mesh:

$$s = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}} = 3101.31 \tag{3.1}$$

In dem Korallenriff werden mehr als 2000 Fische simuliert. Fasst man die Schwärme in sinnvolle Gruppen zusammen, für die eine gemeinsame Kollisionsberechnung durchgeführt wird, so liegt die Zahl der nötigen Kollisionsberechnungen für alle sinnvoll zusammengefassten Gruppen und allen individuellen Fische bei ca. 250 pro Frame. Dies ergibt, wenn alle durchschnittlichen Faktoren miteinander multipliziert werden, 11.625.000 nötige Berechnungen pro Frame. Wird die Kollisionsberechnung auf eine Berechnung pro Gruppe pro Sekunde mit ca. 60 FPS heruntergebrochen, so ergibt sich weiterhin pro Frame ein durchschnittlicher Aufwand O(200.000). Diese Zahl kann jedoch stark variieren, da die Meshes 1/12 bis 3 mal so viele Polygone

besitzen wie die durchschnittliche Anzahl von Polygonen, die für diese Berechnung genutzt wurden. Im Zusammenhang mit der Standardabweichung bekräftigt dies die Annahme, dass es zu starken FPS-Schwankungen kommen kann, welche von dem Beobachter aktiv und störend wahrgenommen werden können. Zusätzlich dazu ist davon auszugehen, dass der tatsächliche Aufwand weit über dem durchschnittlich geschätzten Aufwand liegt, was die FPS zusätzlich stark reduzieren würde.
Berechnet der Algorithmus das Verhalten der Fische nur einmal pro Sekunde, ist davon auszugehen, dass aufgrund der großen Intervalle zwischen den Kollisionsberechnungen Fische so nah an ein Mesh herankommen, dass kein Ausweichmanöver mehr möglich ist, welches den Kurs nicht unnatürlich beeinflussen würde.
Der Algorithmus der Kollisionserkennung ist somit weiterhin komplett ungeeignet, da dieser zu stark von Raycasts und Meshes abhängt und sich von dieser Abhängigkeit nicht entkoppeln lässt. Dazu variiert die Performancenutzung der Algorithmen erheblich, je nachdem welcher Fall eintritt und ist somit für die Beurteilung nach Durchschittsfällen ebenfalls ungeeignet.

3.1.4 Probleme mit Meshkollisionen

Das gegenwärtige Kollisionserkennungsverfahren hat bereits bei der Kollisionsberechnung für den Spieler in ungünstigen Positionen zu kurzzeitigen FPS-Einbrüchen von 5-10% geführt, da die Algorithmen für die Berechnung sehr viel Aufwand benötigen.
Es gab bereits während des Bachelorprojekts einen Versuch, diesen Algorithmus auf einzelne Fische anzuwenden. Dieser scheiterte zunächst an der Synchronisation von Threads, da der Algorithmus auf Berechnungen im Hauptthread ausgelegt ist. Durch die Verwendung von Methoden, welche nicht synchronisiert wurden, war es im schlimmsten Fall möglich, dass die Simulation komplett abgestürzte ist. Der Ansatz, diese Methode zu synchronisieren, hat dieses Problem auf Kosten der Performance zunächst gelöst. Diese Lösung führte jedoch dazu, dass wenn 50 Fische im Riff geschwommen sind, schon die Kollisionserkennung von verhältnismäßig sehr wenigen Fischen dazu führte, dass die Framerate teilweise mehr als eine Sekunde betrug. Ein weiteres Problem ergab die Kollision selbst, da die Fische an den Meshes langsam vorbeigeschoben wurden, was sehr unnatürlich aussah.
Dieser Ansatz wurde daraufhin verworfen, da dieser, wie bereits im Kapitel *Performancemessung* [3.1.3] beschrieben, das Problem, Fische, mit Objekten nicht mehr kollidieren zu lassen, nicht löste. Zudem haben Änderungen in der Implementierung dazu geführt, dass durch die zusätzliche Synchronisation allgemeine Performanceeinbußen und FPS-Einbrüche feststellbar waren, welche zu keinem akzeptablen Verhalten für die Fische führte.

Als Fazit aus dem Ablauf und den Messungen ergibt sich, dass ein Algorithmus, der von Raycasts und Meshes abhängt, für die Problemstellung der Kollisionsvermeidung ungeeignet ist. Die Algorithmen wurden ursprünglich implementiert, um den Beobachter an Kollisionen zu hindern, indem er parallel zu dem getroffenen Polygon an dem Mesh vorbeigeschoben wurde. Dies ist jedoch keine akzeptable Lösung für Fische, da diese den Meshes frühzeitig ausweichen müssen, um eine Kollision dieser Art zu verhindern. Dazu ist der gegenwärtige Algorithmus stark von der Anzahl der Polygone der Meshes sowie den Raycasts abhängig, was zu einer enormen Varianz des Aufwands führt.
Ein Algorithmus, der dieses Problem löst, hat zwei wesentliche Anforderungen. Die erste Anforderung ist, dass der Aufwand pro Frame für jeden Fisch unabhängig von seiner Situation nahezu konstant bleibt, um starke Schwankungen in der Framerate zu verhindern. Zusätzlich dazu sollte der Aufwand so gering wie möglich sein. Die zweite Anforderung ist, dass der Algorithmus zur Kollisionsvermeidung durch eine Kurvenanpassung implementiert ist, da eine Kollisionserkennung das Problem, wann ein Fisch auf ein Mesh stößt, zwar erkennt, jedoch mit der aktuell ungeeigneten Umsetzung, bei einer Kollision den Fisch an dem Mesh vorbeizuschieben, nicht löst. Eine Möglichkeit dieses Problem zu lösen wird in [HCCV11] beschrieben. Für die Simulation wäre das Problem enorm vereinfacht, da der eigentliche Algorithmus auf eine xz-Ebene mit Einschränkungen ausgelegt ist, während in der Simulation der xyz-Raum zur Verfügung steht. Die zusätzliche Koordinatenachse vervielfacht die Möglichkeiten einem Objekt auszuweichen, da neben der Ausweichrichtung auf der Ebene zusätzlich die Möglichkeit besteht, in die Höhe oder Tiefe auszuweichen.

3.1.5 Parallelisierung

Neben der algorithmischen Lösung für das Problem, muss die Kollisionsvermeidung auf geeignete Art in das Projekt integriert werden. Eines der Kernprobleme war die Synchronisation zwischen den Threads wie in *Sequenzdiagramm* [4.1.2] beschrieben. Die Objekte welche an der Berechnung beteiligt waren Zustände hatten, was zu Nebenläufigkeit und Abstürzen führen konnte. Dieses Problem lässt sich lösen, indem sämtliche Algorithmen sequentiell auf dem gleichen Thread stattfinden, wodurch Zustände keinen Einfluss mehr hätten. Dies würde jedoch die Framerate negativ beeinflussen. Ein weiterer Ansatz, wäre wie in [Ver13] beschrieben die Klasse, welche für die Berechnung des Ausweichverhaltens implementiert wird, nach dem CQS Prinzip so zu modellieren, dass sie als reine Datenquelle für Rechnungen dient und keine Zustände besitzt. Dieser Ansatz würde, sofern genügend Kerne vorhanden sind es möglich machen, die gesamte Ausweichlogik in einen weiteren Thread auszulagern,

um weitere Performance durch die dann mögliche Parallelisierung zu sparen.

3.2 Kollisionsvermeidung

3.2.1 Abstrahierung von Meshes

Es gibt für diese Problemstellung bereits einige interessante Arbeiten, die sich mit der Problemstellung Kollisionserkennung auseinandersetzen, indem die Meshes, die sich bewegen, als Sphären dargestellt werden und diese parallel zu den Polygonen verschoben werden [Fau03]. Dazu wird auch das Thema Kollisionsvermeidung behandelt, indem die Kurse angepasst werden [HCCV11]. Ergänzend hierzu gibt es Arbeiten, die sich mit der abstrakten Darstellung von Meshes als z.B. Sphären auseinandersetzen [WZ10b] [ZK04]. Dieser Ansatz ergibt Sinn, da ein hoch aufgelöstes Mesh mit vielen Polygonen von der groben Struktur ebenfalls als Gruppe von Sphären erfasst werden könnte, womit die groben Eigenschaften des Meshes erhalten bleiben, während die Komplexität der Darstellung deutlich geringer ist.

3.2.2 Kollisionserkennung durch Abstraktion

Viele Kollisionserkennungsverfahren nutzen keine genaue Darstellung der Meshes, um diese zu erkennen. Ein Ansatz ist, die Meshes in viele Sphären zu zerlegen, mit denen die Oberfläche darstellbar ist.
Die abstrakte Darstellung von Meshes führt zu einem Präzisionsverlust der Darstellung der Oberfläche, wodurch bei der Kollisionsberechnung Fehler entstehen. Diese waren jedoch je nachdem wie viel Präzision eingebüßt wurde in der Regel nicht höher als 10%.

Für die Problemstellung in der Simulation wäre eine direkte Kollision jedoch weiterhin ein Problem, da diese optisch stark auffallen würde. Aus diesem Grund muss das verwendete Verfahren so funktionieren, dass es selbst mit 10% Fehler weiterhin Fische ausweichen lässt, ohne dass diese zu nah an die Meshes heranzukommen.
Ein geeigneter Ansatz wäre, eine für den Beobachter unsichtbare Sphäre um Meshes herum zu bilden, die so groß ist, dass ein Fisch schon lange, bevor er mit einem Mesh kollidiert, auf einen Ausweichkurs geführt wird. Diese sehr grobe Darstellung wäre fehlertolerant, solange die Hülle der Sphäre so weit von dem Mesh entfernt ist, dass auch ein Fisch, der bereits in dieses eingedrungen ist, durch einen Ausweichkurs weiterhin eine Kollision mit dem Mesh vermeidet. Je nach Form des

Meshes könnte man es wie in [RW13] beschrieben ebenfalls in beliebig viele Sphären oder zusätzlich auch Boxen aufteilen. Der Rechenaufwand pro Mesh würde von Tausende Polygonen auf die Kollision mit wenigen Sphären und Boxen reduziert werden.
Nach dem Point Cloud Collision Detection [ZK04] Verfahren werden diese mathematischen Körper zusätzlich über Gruppen zusammengefasst, um den Rechenaufwand weiter zu reduzieren. Die Gruppierung würde, je nachdem wie sich das Mesh geeignet erfassen lässt, sämtliche Sphären und Boxen, mit denen das Mesh zusammengefasst wurde, mit einer großen Sphäre oder einer Box zusammenfassen. Dies würde den Aufwand für den Fall, dass ein Fisch sich nicht in Kollisionsgefahr mit einem Mesh befindet, den Aufwand pro Objekt auf O(1) reduzieren. dieser Aufwand würde, sobald ein Objekt in Kollisionsnähe ist sich maximal um die Anzahl der erfassten Sphären und Boxen erhöhen.
Die Erfassung eines Mesh würde innerhalb der Simulation somit über eine Gruppe von Sphären und Boxen, die unter einer großen Sphäre oder Box, die das gesamte Mesh und alle Teilsphären und Boxen umgibt zusammenfassen. Dies erfordert pro Frame unter der Berücksichtigung von dem Spieler, den Prädatoren und den Grenzen der Simulationsumgebung einen Aufwand O(a+b+c+d+7). Dabei enspricht a der Anzahl der statischen Meshes, b der Anzahl der Prädatoren, c entspricht der Anzahl der dynamisch platzierten Steine im Riff, d der Anzahl der statischen Korallen und 7 enthält die Überprüfung ob der Fisch die Simulationsumgebung in irgendeiner Richtung verlässt oder in der Nähe des Spielers ist. Die Parameter a,b,c und d sollten jedoch manuell über die Config deaktivierbar sein. Besonders sollte an dieser Stelle auf die Meshes hinter dem Parameter c und d geachtet werden. Diese Objekte fordern verhältnismäßig viel Performance, da sich diese auf dem Boden befinden und somit verhältnismäßig selten für eine Kollision in Frage kommen. In Kombination mit der Situation, dass die meisten Steine in der Umgebung von Pflanzen sind, würde eine Kollision in den seltensten Fällen auffallen und verliert somit noch mehr optische Relevanz. Da jedoch die Parameter c und d einen direkten Einfluss auf die Minimalperformance haben, die der Algorithmus braucht, ergibt es Sinn, diese nur optional auf leistungsstarken Rechnern zu aktivieren.

Zum Zeitpunkt als das Projekt abgeschlossen wurde, befanden sich fünf statische Objekte in der Simulation, durch die die Fische regelmäßig hindurch geschwommen sind. Der Aufwand O(a) würde entsprechend 5 sein. Zu diesem Zeitpunkt befand sich nur ein Prädator in der Simulation, wodurch sich der gesamte Aufwand für die Kollisionserkennung mit allen Objekten auf einen Aufwand O(a+b+7) = O(13) pro Fisch steigern würde. Die Summanden c und d waren zum Zeitpunkt des Abschlusses mit den Zahlen 50 und 20 belegt, wodurch der Aufwand für die Kollisionserkennung

von O(13) auf O(83) um den Faktor 6 steigen würde. Die Anzahl der Boxen und Sphärengruppen variiert je nach Körper zwischen 4 und 8, somit wäre der Gesamtaufwand für die Kollisionsberechnung mit allen Objekten zwischen O(13+8)=O(21) oder im schlimmsten Fall O(83+8)=O(91). Der Aufwand ist im Vergleich zum aktuellen Kollisionsverfahren, welches bei einer Kollision mit dem Mesh mit den wenigsten Polygonen O(250), deutlich performanter. Der durchschnittliche Aufwand wäre jedoch mit O(9.270) und O(135.000) im Vergleich zu den Messungen aus dem Kapitel *Performancemessung* [3.1.3] hundertfach so groß, bzw. die Berechnung für einen Fisch würde mehr Performance brauchen als alle Fische über das Kollisionsvermeidungsverfahren, wenn die Parameter c und d deaktiviert sind.
Das Verfahren arbeitet mit Meshes, die nach folgendem Prinzip abstrahiert wurden:

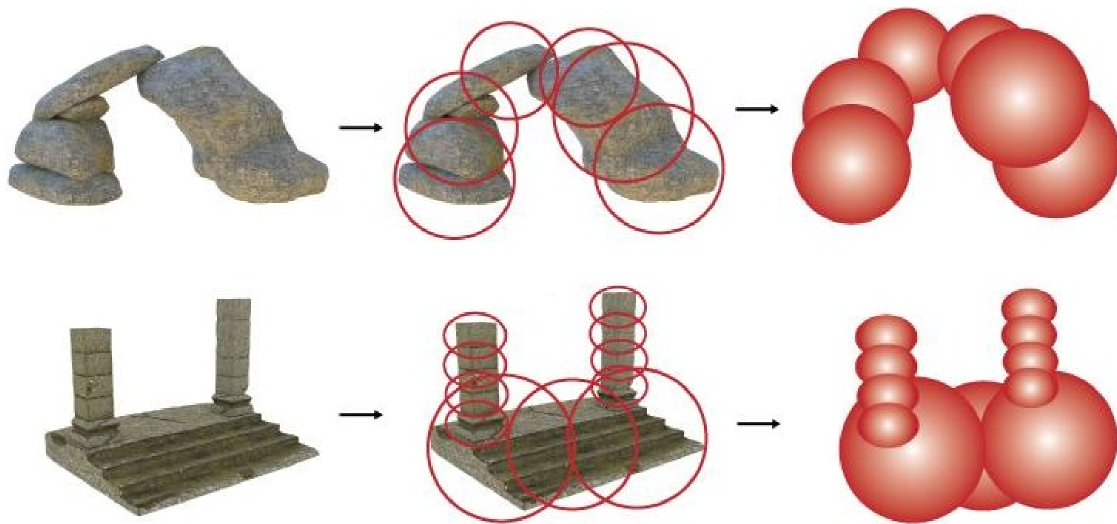

Abbildung 3.2: Abstrahierung von Meshes
Meshes werden durch Sphären abstrahiert, die das Mesh komplett einschließen.

Zunächst muss für das Mesh die Entscheidung getroffen werden, wie es am besten in welche mathematischen Körper zu zerlegen ist. Daraufhin müssen pro mathematischen Körper die nötigen Parameter erfasst werden. Für die Sphären sind der Mittelpunkt und ein beliebiger Außenpunkt erforderlich, um das Zentrum und den Radius zu bestimmen. Für Boxen sind 2 quer gegenüberliegende Punkte erforderlich, mit denen die gesamte Box aufgespannt wird. Beide Punkte sollten so gewählt sein, dass jeweils das x,y und z Attribut des Startpunktes von dem Zahlenwert unter denen der entsprechenden Attribute der Endkoordinate liegen. Alle erfassten Körper sollten so gewählt sein, dass das Zentrum dieser Körper innerhalb des Meshes liegt, um Nebeneffekte zu vermeiden. Zuletzt muss für das Mesh noch ein umfassender mathematischer Körper erfasst werden um ein durch die Gruppierung nach [ZK04] für zusätzliche Performanceeinsparung zu ermöglichen. Die minimale Entfernung der Meshes von den mathematischen Körpern sollte wie in *Problemstellung bei der*

Erfassung [3.2.4] erläutert so gewählt sein, dass die Fische durchschnittlich eine Sekunde brauchen, um das Mesh tatsächlich zu erreichen.
Alle Meshes müssen in der Simulation erfasst und über die `Config` nach dem in dem Kapitel *Erfassung von Kollisionsobjekten* [4.2.1] erläuterten Verfahren in das Framework eingelesen werden.

3.2.3 Erfassung von Kollisionsobjekten

Damit Fische dazu in der Lage sind Ausweichkurse anzunehmen, müssen sämtliche Meshes, für die die Fische eine Kollisionsvermeidung durchführen sollen, geeignet erfasst sein. Mit dem Protophere Verfahren [WZ10b] oder Fast Sphere Packing Verfahren [RW13] wäre es möglich, alle Meshes als Gruppe von Sphären darzustellen, mit denen die Oberfläche abgebildet wird. Für die Kollisionsvermeidung ist jedoch keine genaue Erfassung aller Sphären notwendig, da Fische sobald sie in Reichweite dieser Sphäre kommen, bereits auf Ausweichkurs nach [HCCV11] gehen sollen. Dies verhindert, wie in *Ermittlung des Ausweichrichtung* [3.2.5] berechnet wird, mit großer Sicherheit die Kollision.
Für runde Meshes wäre es somit möglich, diese als nur eine Sphäre zu erfassen, wodurch der Aufwand diese als Kollisionsobjekt darzustellen von der Komplexität von mindestens 250 Polygon und einem Raycast auf eine Distanzberechnung zu einer Sphäre oder einer Box reduziert wird. Für komplexere Meshes ist dies ebenfalls möglich. Diese können durch Gruppen mathematischer Körper wie Boxen und Sphären ebenfalls in ihrer Form abstrakt dargestellt werden.
Sphären und Boxen sind für diese Art von Kollisionserkennung besonders gut geeignet, da der Aufwand für eine Position zu erkennen, ob sie sich innerhalb dieser befindet, sehr gering ist. Für Sphären muss jeweils das Zentrum und der Radius erfasst werden. Für die Kontrolle, ob eine Position sich innerhalb einer Sphäre befindet, muss die quadratische Distanz von der zu kontrollierenden Position mit dem quadratischen Radius der Sphäre verglichen werden. Für die Box muss überprüft werden, ob sich alle Koordinaten der Position zwischen dem Anfang und Ende der Eckpunkte der Box befinden. Beide Operationen, wie in dem Kapitel *Kollisionsvorbeugung* [4.2.2] erläutert, verbrauchen wenig Performance und sind somit gut geeignet.

Um einen Vergleich möglich zu machen, müssen alle Meshes, mit denen Kollisionen möglich sein sollen, in der Klasse `DodgePointService` zusammengefasst werden. Die relevanten Objekte sind: statische Strukturen, statische Steine, statische Korallen, die Prädatoren und der Spieler. Als statisch werden alle Meshes bezeichnet, die bei der Initialisierung entstehen und nicht über Events in der Simulation nachträglich

verändert werden. Um ein effizientes Erkennungsverfahren zu implementieren, werden alle Objektgruppen in unterschiedlichen Listen erfasst. Diese werden über ihre Relevanz priorisiert und sequentiell wie in dem Kapitel *Kollisionsvorbeugung* [4.2.2] erläutert wird, durchlaufen. Da, sobald eine Kollisionsgefahr erkannt wird, das Kollisionsvermeidungsverfahren die Ausweichrichtung direkt als Rückgabewert liefert, werden somit selten relevante aber performanceintensive Listen häufig ausgelassen. Darüber hinaus wird Performance gewonnen, indem für Bodenobjekte zunächst abgefragt wird, ob der Fisch in relevanter Nähe zum Boden ist, bevor die Listen für die Kollisionsvermeidung abgearbeitet werden.

Zuletzt muss für die Klasse `DodgePointService` an einer geeigneten Stelle im Projekt eingebunden werden. Der `CollisionManager` war bisher die Stelle, an der alle Funktionen gesammelt wurden, welche für Kollisionen nötig waren. Entsprechend wird dieser um das Attribut `DodgePointService` erweitert. Die nötigen Funktionen werden in den `CollisionManager` ebenfalls nach dem CQS Muster aus [Ver13] hinzugefügt. Dies ermöglicht es, die fehlende Erkennung für die Kollision mit der Oberfläche und dem Untergrund, die nur über den `CollisionManager` zu ermitteln ist, ebenfalls zu berücksichtigen.

3.2.4 Problemstellung bei der Erfassung

Für die Erfassung aller Meshes ist es notwendig, diese als Sammlung von Sphären und Boxen zu abstrahieren. Diese Abstrahierung wäre nach dem Protosphere Verfahren [WZ10b] über Algorithmen möglich. Dieser Ansatz würde die Meshes in eine sehr große Sammlung von Sphären zerlegen, welche initial viel Performance zum Bestimmen dieser benötigen würden. Diese präzise Erfassung durch viele kleine Sphären würde jedoch für das Kollisionsvermeidungsverfahren ungeeignet sein, da der Aufwand pro Sphäre um 1 steigt und durch die Größe der Sphären der Fisch erst zu spät den Kurs anpassen würde. Damit das Kollisionsvermeidungsverfahren korrekt funktioniert, ist eine Erfassung über große Sphären notwendig die frühzeitig eine Kollisionsgefahr erkennen. Eine präzise Erfassung über kleine Sphären, die nur die Oberfläche darstellen, würde zusätzlich zu einem Problem führen, wenn ein Fisch zufällig ein Kollisionsobjekt durchschwimmt und zukünftig innerhalb des Meshes gefangen wäre. Dieser Effekt entsteht, da eine Darstellung über die Oberfläche der Meshes, den Fisch innerhalb des Meshes durch die Kollisionsvermeidung einsperren würde. Dies disqualifiziert alle Algorithmen, die keine großen Sphären innerhalb des Meshes bestimmen. Über das Fast Sphere Packing Verfahren [RW13] wäre eine große Sphäre innerhalb des Meshes bestimmbar, welche diesem Problem entgegen-

wirkt, da eine möglichst große Sphäre im Mesh ermittelt wird. Dadurch, dass das Zentrum dieser Sphäre innerhalb des Meshes liegt, kann ein Fisch nicht innerhalb eines Meshes gefangen werden und würde in jedem Fall wieder einen Kurs nach draußen wählen.
Wird ein Mesh über eine Gruppe von solchen Objekten wie in der Abbildung [3.2] erfasst, so würde ein Fisch, der in ein Mesh hineingerät, wieder herausgedrückt werden. Ein Ansatz ist, alle Sphären und Boxen manuell zu erfassen, mit denen die Meshes in der Simulation dargestellt werden. Da einige Meshes nicht nur natürlich abgerundet sind sondern auch künstliche Objekte die mitunter aus Boxen bestehen, sind sowohl Sphären als auch Boxen für die Erfassung geeignet. Der Algorithmus erfordert somit einen einmaligen großen Messaufwand im Korallenriff, um alle Meshes geeignet als eine Mischung aus Sphären und Boxen darzustellen, um den Rechenaufwand während der Simulation zu minimieren. Da keine große Präzision erforderlich ist, und die Sphären und Boxen relativ groß sein sollten, um keine unerwünschten Effekte herbeizuführen, ist eine grobe Messung ausreichend.
Der Fall, dass ein Fisch mit einem Mesh durch ungünstige Umstände kollidiert, kann mit diesem Verfahren jedoch durch die Art der Erfassung nicht verhindert werden, da die Informationen über das Mesh selbst verloren gehen. Der Algorithmus ignoriert somit jede Kollision und funktioniert allein nach dem Prinzip Fische so schnell auf Ausweichrouten zu bringen, dass diese möglichst nie nah genug an das Mesh herankommen.

3.2.5 Ermittlung der Ausweichrichtung

Die meisten Kollisionsvermeidungsverfahren passen die Position des Objektes parallel zu dem getroffenen Polygon an. Dies ist für den Fall, dass der Beobachter mit einem Mesh kollidiert, in Ordnung, jedoch für den Fall, wenn ein Fisch mit einem Mesh kollidiert, optisch extrem ausfällig und unpassend. Um eine Kollision zu vermeiden, muss der Kurs der Fische frühzeitig angepasst werden, sodass eine Kollision aktiv verhindert wird. Der grundlegende Unterschied zwischen einem Verfahren, was Kollisionen verhindert, und einem Verfahren, was einer Kollision vorbeugt, ist ,dass bei dem Verhindern die Position und bei dem Vorbeugen die Richtung angepasst wird.
Die grundlegende Annahme für die Kursanpassung ist, dass die FPS stetig mehr als 30 betragen. Der Kurs wird dabei pro Frame um die vergangene Zeit pro Frame in Richtung des Ausweichkurses angepasst.

Dies wäre bei einer Framerate von 60 FPS 1.66% und bei einer Framerate von 30 FPS 3.33%. In der folgenden Rechnung wird die Kursanpassung nach einer Sekunde berechnet.

$$\begin{pmatrix} 0.9833 & 0 \\ 0.0167 & 0 \end{pmatrix}^{60} \times \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.365 \\ 0.635 \end{pmatrix} \tag{3.2}$$

$$\begin{pmatrix} 0.9666 & 0 \\ 0.0337 & 0 \end{pmatrix}^{30} \times \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.362 \\ 0.638 \end{pmatrix} \tag{3.3}$$

Aus dieser Rechnung lässt sich erschließen, dass ein Fisch seinen Kurs innerhalb von einer Sekunde um ca. 63% ändern wird. Wird eine Sphäre so gewählt, dass der minimale Radius zum Mesh mindestens so groß wie die Bewegungsgeschwindigkeit des Fisches ist, weicht dieser unter der Annahme, dass er bei einer Kursänderung von 50% nicht mehr kollidieren, würde dem Mesh mit großer Sicherheit innerhalb von einer Sekunde stark genug aus, um eine Kollision sicher zu verhindern. In der folgenden Abbildung [3.3] wird das Ausweichprinzip auf einer 2D-Ebene dargestellt.

Der Kurs des dunkelblauen Fisch gibt an, wie sich der Fisch zunächst bewegen würde. Für die Kollisionsvermeidung wird zunächst pro Frame kontrolliert, ob er sich innerhalb der grünen Sphäre befindet. Sobald er in diese Sphäre eindringt, werden pro Frame weitere Kontrollen für die rote Sphären durchgeführt. Sobald er in die erste rote Sphäre eindringt, fängt der Algorithmus an, seinen Kurs anzupassen. Für die Anpassung ändert sich der hellblaue Kurs so lange, bis dieser von dem Sphärenzentrum weg zeigt und diese verlassen hat. Dieses Verfahren ist analog im dreidimensionalen Raum anwendbar.

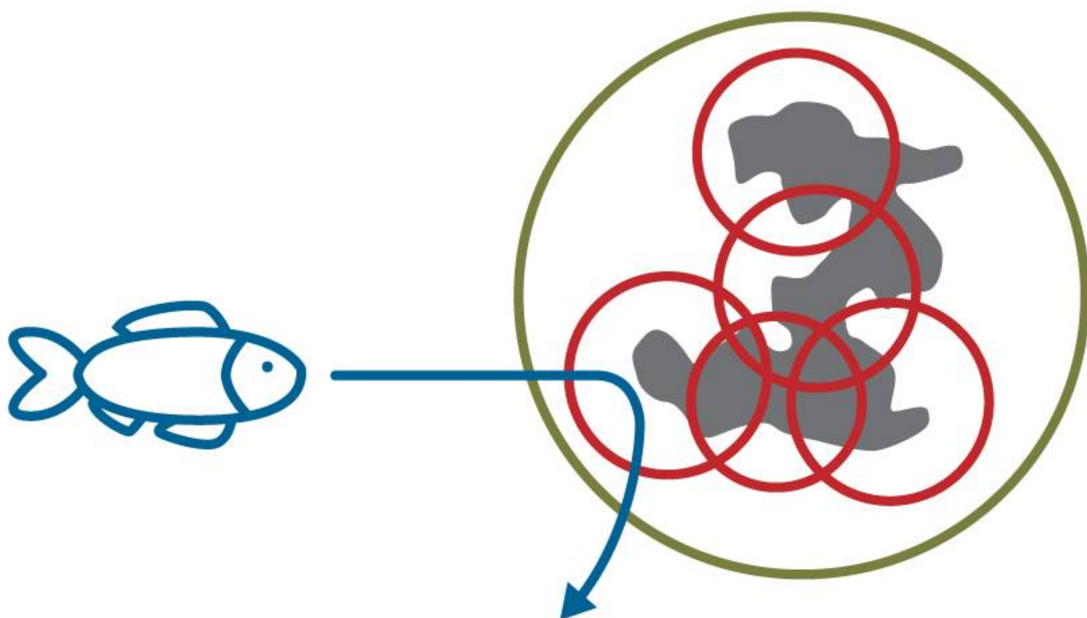

Abbildung 3.3: Anpassung der Bewegungsrichtung.
Fische passen ihren Kurs an, sobald sie die Nähe einer Sphäre kommen, bis sie diese wieder verlassen haben.

Kapitel 4

Architektur und Implementierung

4.1 Architektur

4.1.1 Klassendiagramm

In diesem Kapitel wird die Architektur des Moduls beschrieben, welches für die Schwarmlogik verantwortlich ist. Die Darstellung enthält sämtliche Klassen, die bisher vorhanden sind: `GameState, SwarmManager, FishManager, Swarm, Fish` und `CollisionManager`. Zu den bereits vorhandenen Klassen werden weitere Klassen hinzugefügt, welche zur Realisierung der Bestimmung der Ausweichkurse notwendig sind: `DodgePointAdapter, DodgeService` und `DodgeObject`. Im Klassendiagramm [4.1] wird verdeutlicht, welche Klassen wie gesteuert werden. Die Darstellung der Klassen wurde abstrakt gehalten, und um viele Attribute und Methoden, die für die Integration des Ausweichverhaltens notwendig sind, erleichtert.

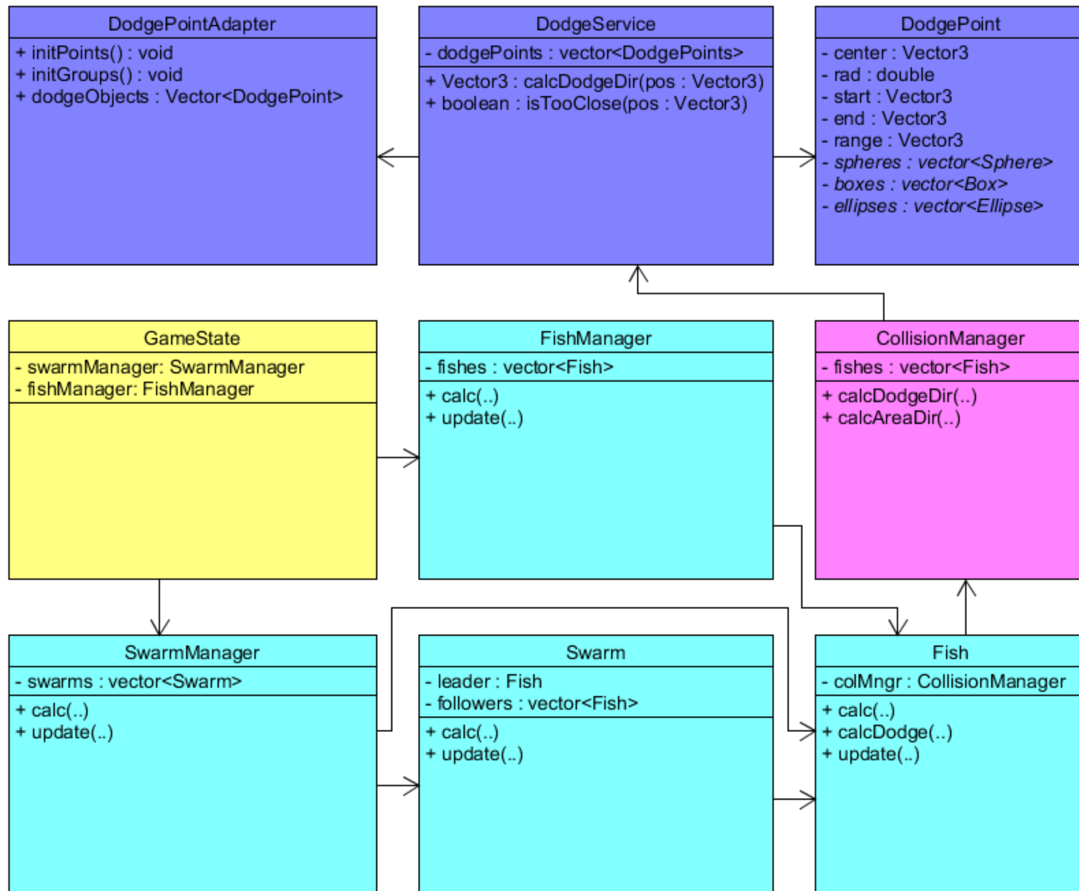

Abbildung 4.1: Klassendiagramm
dunkelblau: Realisierung der Ausweichlogik über DodgeObjects
magenta: Realisierung der Ausweichlogik über Terraineigenschaften
gelb: Steuert sämtliche Manager im gesamten Framework
cyan: Realisierung der Fischlogik und der Schwarmlogik

Das Klassendiagramm stellt die Hierarchie dar, nach der die Klassen gesteuert werden und lässt die Initialisierungsprozesse aus. Die Klasse `GameState`, welche beim Start des Frameworks die Klassen `CollisionManager` und `DodgeService` initialisiert, hat aus diesem Grund keine direkte Verbindung zu diesen. In der Klasse `GameState` laufen zwei Threads parallel, ein Thread ist für sämtliche `calc(..)` Prozesse verantwortlich, der andere für sämtliche `update(..)` Prozesse. Der Unterschied zwischen den beiden Threads ist, dass der `calc(..)` Thread für sämtliche Berechnungen zuständig ist, und der `update(..)` Thread für die visuelle Darstellung. Der `SwarmManager` und `FishManager` werden einmal pro Frame von der Klasse `GameState` aufgerufen und aktualisieren alle mit ihnen verbundenen Objekte. Der `FishManager` steuert dabei sämtliche Objekte der Instanz `Fish` direkt an und aktualisiert ihre Richtung und Position. Der `SwarmManager` führt diese Aktualisierung für sämtliche Prädatoren durch und aktualisiert ebenfalls sämtliche Schwärme.

In der Klasse `Swarm` werden sämtliche Berechnungen durchgeführt, die für alle Fische pro Schwarm relevant sind, wie z.B. die allgemeine Schwarmbewegung und das Schwarmzentrum. Zusätzlich aktualisiert er sämtliche Fische. In der Klasse `Fish` wird pro Frame einmal die Bewegungsrichtung über `calc(..)` und die Ausweichrichtung über `calcDodge(..)` neu errechnet. Die Berechnung findet über den CollisionManager statt, der zunächst für die in der Klasse bekannten Parameter Kontrollen durchführt, und gegebenfalls die Klasse `DodgeService` aufruft, um über alle `DodgeObject` Objekte, die in dieser erfasst wurden, zu kontrollieren, ob für diese eine Ausweichrichtung notwendig ist.
Der `DodgePointAdapter` dient dem `DodgePointService` dazu, um alle relevanten Objekte als `DodgeObject` einzulesen.

4.1.2 Sequenzdiagramm

In diesem Kapitel wird ergänzend zu dem Kapitel *Architektur* [4.1] beschrieben, welche Abläufe pro Frame sequentiell stattfinden. In der Abbildung [4.2] wird der Ablauf der beiden Threads beschrieben, welche für die Schwarmlogik notwendig sind. Die Abläufe durch die Klasse `FishManager` wurden an dieser Stelle nicht berücksichtigt, da diese von der Komplexität im Verhältnis zu den erfassten Abläufen trivial und analog verständlich sind.
Die Methodennamen, die den Ablauf beschreiben, sind abstrakt gewählt. Die Methode `calc` durch die Klasse `Swarm` steht stellvertretend für alle schwarmspezifischen `calc-Methoden`, was ebenfalls für abstrakte Aufrufe wie calcdir(..) durch den `Fish` an den `CollisionManager` gilt.
Beide Threads laufen, sofern sie nicht durch die in `GameState` geregelte Synchronisation aufgehalten werden, parallel. Eine Terminierung der Threads findet nur bei dem Beenden der Simulation statt.

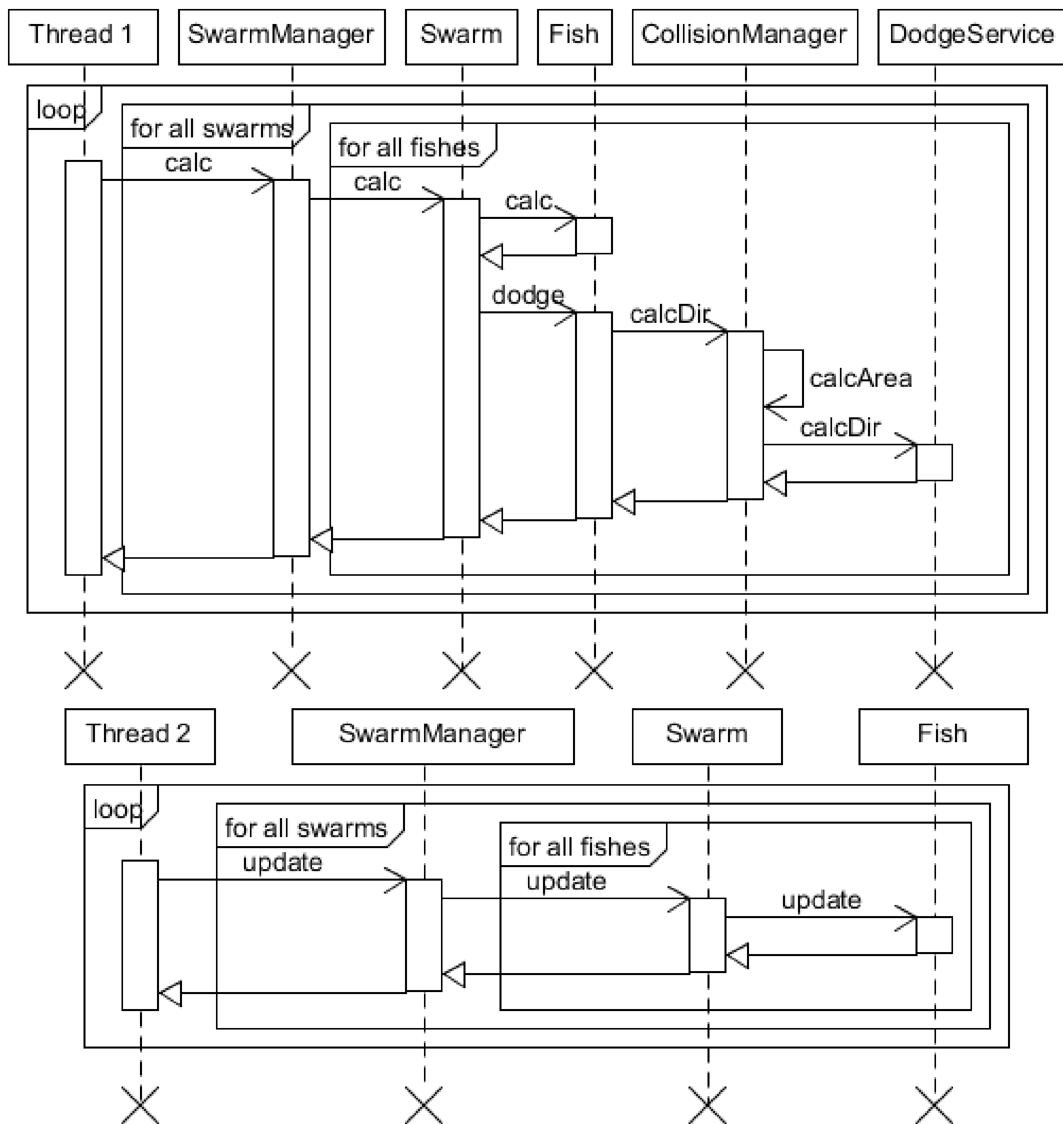

Abbildung 4.2: Sequenzdiagramm
Thread 1: Berechnung der Bewegungen
Thread 2: Aktualisierung des Meshes

4.2 Implementierung

4.2.1 Erfassung von Kollisionsobjekten

Die Implementierung des Kollisionsvermeidungsverfahren verfolgt das Ziel, Objekte einfach in die Algorithmen zu integrieren und dynamisch konfigurierbar zu sein. Für die einfache Einbindung von Objekten werden Schnittstellen zur Verfügung stehen, die sowohl Sphären als auch Boxen im einzelnen als Kollisionsobjekte erfassen, sowie Schnittstellen, die eine Gruppe von Boxen und Sphären als Untergruppe einer großen Sphäre oder Box erfassen. Die Aufteilung in Boxen und Sphären ermöglicht es, die meisten für die Simulation relevanten Objekte grob darzustellen. Die `Config` ermöglicht es, Objekte in der Simulation über manuelle Messungen zu erfassen. Es ist jedoch ebenfalls möglich, Objekte durch das Protosphere Verfahren [WZ10b] oder das Fast Sphere Packing Verfahren [RW13] die Attribute für der Erfassung der Meshes durch Sphären zu ermitteln. Für die Wartung soll es möglich sein, über die Config Objektgruppen zu erfassen, sowie dort die Einstellungen vorzunehmen, welche Objekte für die Kollisionsvorbeugung relevant sein sollen. Sollten die Kollisionsobjekte in der `Config` korrekt erfasst sein, so werden sie über folgende Algorithmen eingelesen:

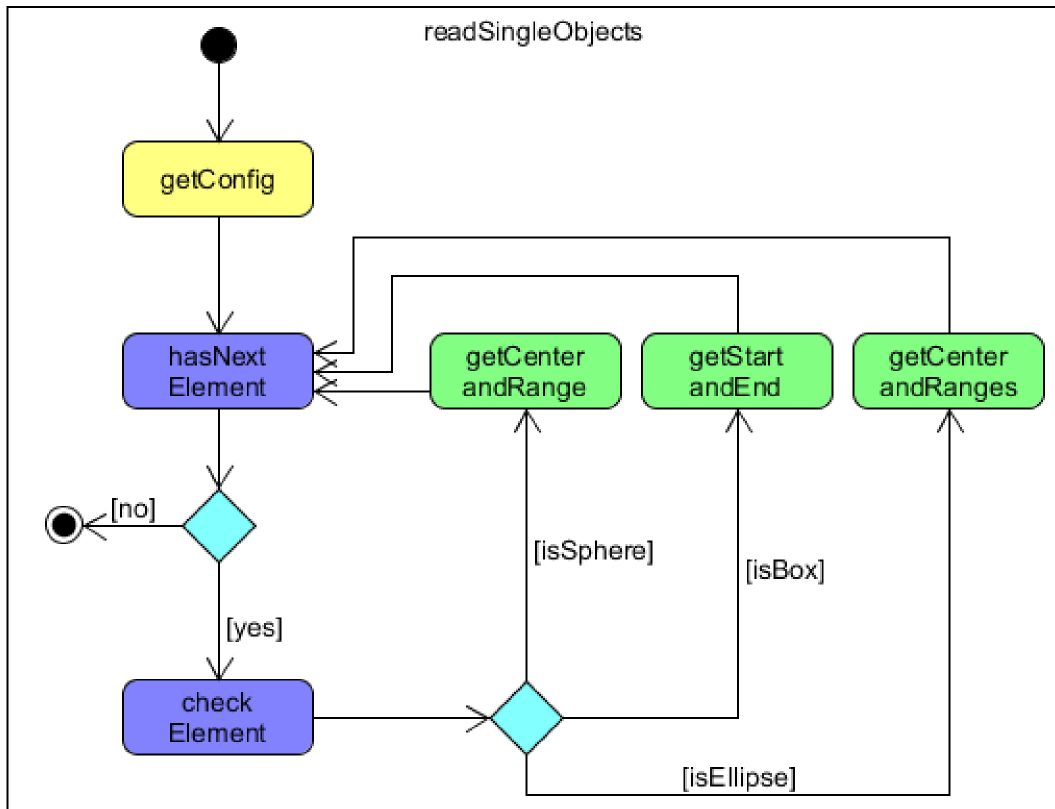

Abbildung 4.3: Auslesen einzelner Objekte aus der Config
gelb: Config laden
blau: Nach weiteren Elementen suchen
grün: Elemente aus der Config in das Framework einlesen

Mit diesem Algorithmus werden aus der `Config` sämtliche Boxen, Sphären und Ellipsen ausgelesen, welche zu keiner Gruppe gehören und als individuelle `DodgeObjects` in der Simulationsumgebung ausgewertet werden. Sphären werden in der `Config` als einzelne Objekte abgespeichert und als vierdimensionale Vektoren ausgewertet. Die ersten drei Parameter geben hier das Zentrum der Sphäre an, der vierte Parameter legt fest, welchen Radius die Sphäre hat. Boxen werden in der `Config` paarweise als dreidimensionale Vektoren erfasst. Der erste Vektor gibt dabei den Anfang der Box an und der zweite das Ende. Die Vektoren legen die Grenzen für die Box fest. Ellipsen werden über zwei dreidimensionale Vektoren erfasst. Der erste Vektor gibt dabei das Zentrum der Ellipse an, der zweite die x,y,z-Reichweite.
Gruppen von Boxen und Sphären werden ebenfalls aus der `Config` ausgelesen, der Algorithmus dafür folgt folgenden Ablauf:.

Die `Config` wird solange nach Gruppen durchsucht, bis keine mehr gefunden werden. Wird eine Gruppe gefunden, so wird zunächst überprüft, ob es sich um eine Gruppe in einer Sphäre oder einer Box handelt. Je nachdem, welcher Fall eingetreten ist, werden die Grenzen der Sphäre bzw. die der Box analog zum Verfahren, wie einzelne Sphären und Boxen erfasst werden, ausgelesen. Im nächsten Schritt läuft eine Schleife so lange, bis alle Sphären und Boxen, die zu der Gruppe gehören, ausgelesen wurden. Die erfassten Sphären und Boxen werden dabei in einer Liste gespeichert. Wenn für die gefundene Gruppe in der `Config` keine weiteren Boxen und Sphären gefunden werden, so wird im letzten Schritt der Schleife die aktuelle Gruppe mit der umgebenden Sphäre oder Box und den Listen der Untergruppen als ein `DodgeObject` abgespeichert.
Diese Implementierung ermöglicht es, Sphären und Boxen für Gruppen durcheinander anzugeben, wobei es weiterhin notwendig ist, Boxen paarweise als dreidimensionale Vektoren anzugeben. Bei den Boxen muss darauf geachtet werden, zwei gegenüberliegende Koordinaten so zu erfassen, dass die Startpunkte jeweils kleinere Werte als die Endpunkte haben.

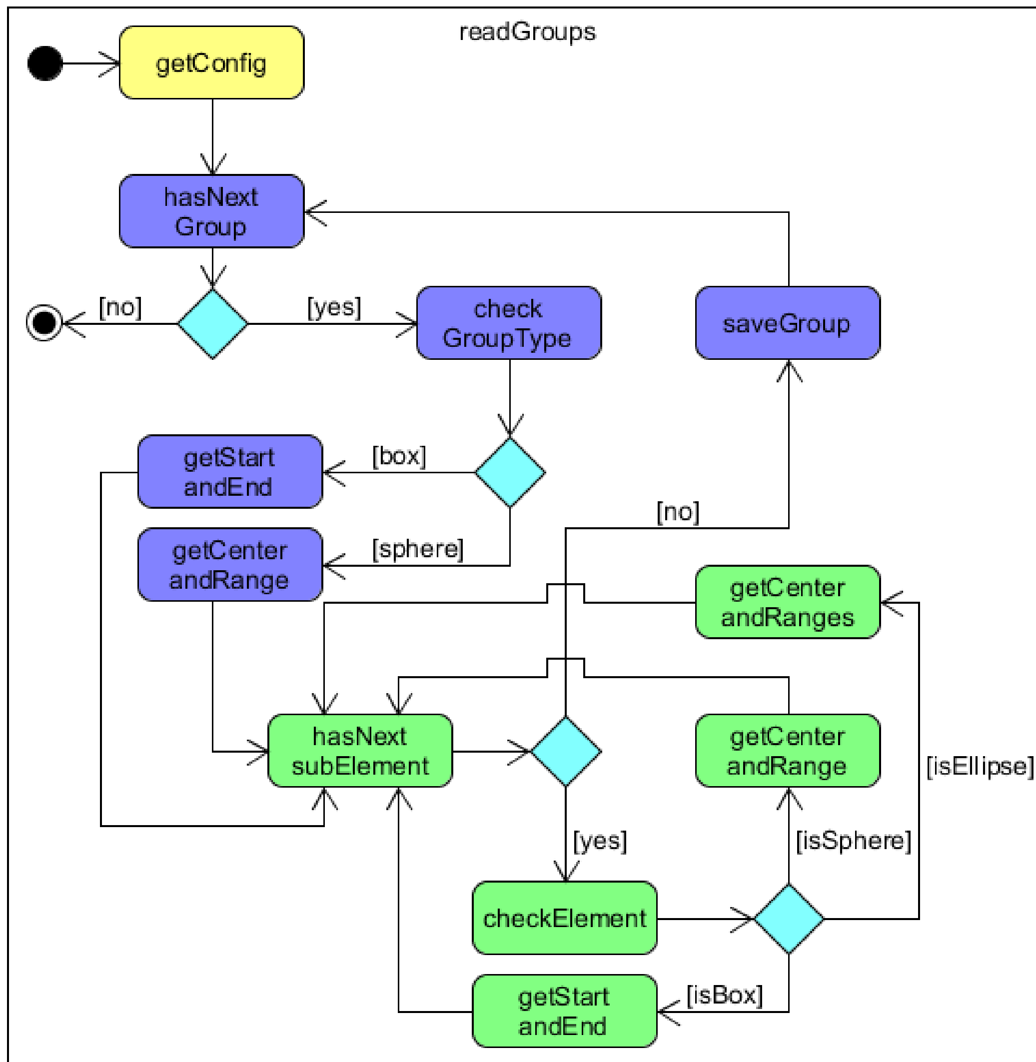

Abbildung 4.4: Auslesen einzelner Objekte aus der Config
gelb: Config laden
blau: Nach weiteren Gruppen suchen
grün: Elemente aus der Config in das Framework einlesen

4.2.2 Kollisionsvorbeugung

Für die Kollisionsvermeidung mit den `DodgeObjects` wird für die übergebende Position zuerst für jeden `DodgeObject` überprüft, ob sie sich in der Nähe von diesem befindet. Der Algorithmus, der die grobe Nähe zu `DodgeObjects` untersucht, unterscheidet, arbeitet nach folgendem Prinzip.

```
bool isTooClose(Vector3 position){
    if (type == BLOCK || type == DODGEBLOCK)
        return (
            position.x > start.x && position.x < end.x &&
            position.y > start.y && position.y < end.y &&
            position.z > start.z && position.z < end.z);
    if (type == SPHERE || type == DODGESPHERE)
        return (
            centre.squaredDistance(position)
           < squaredDistance);
    if (type == ELLIPSE){
        const Vector3 d(position - start);
        const Vector3 abc(end);
        return (pow(d.x / abc.x, 2) + pow(d.y / abc.y, 2)
                    + pow(d.z / abc.z, 2) <= 1.0);
    }
    return false;
}
```

Der Algorithmus arbeitet zunächst für Boxen und Gruppen von Boxen gleich, da die grobe Erkennung sich entweder direkt auf die Box oder der übergeordneten Box bezieht. Ähnlich verhält es sich für die Sphären. Der Vergleich findet über die quadratische Distanz statt, um Performanceverlust durch Wurzeln ziehen zu vermeiden. Der letzte Fall kontrolliert, ob sich der Punkt innerhalb einer Sphäre befindet. Der Algorithmus für die Oberflächenberechnung wurde aus [Dic06] auf eine Berechnung, ob der Punkt innerhalb der Ellipse ist, umgewandelt. Sollte sich die gegebene Position in Kollisionsgefahr befinden, so liefert diese Methode `true` als Ergebnis, sonst `false`.

Sollte die Kontrolle für eine Position, ob diese sich in Kollisionsreichweite befindet, zu einem Erfolg führen, so ist es notwendig, weitere Informationen über die Ausweichrichtung zu erhalten. Die Ausweichrichtung wird mit folgender Methode ermittelt:

```
vector3 getDodgeDir(Vector3 position){
    if (type == SPHERE) return (position-sphereCenter).norm();
    if (type == BLOCK)  return (position-blockCenter).norm();
    if (type == ELLIPSE){
        const Vector3 d(position - start);
        const Vector3 abc(end);
        return Vector3( d.x / pow(abc.x, 2),
                        d.y / pow(abc.y, 2),
                        d.z / pow(abc.z, 2)).norm();
    }
    if (type == DODGESPHERE || type == DODGEBLOCK){
        for (int i = 0; i < spheres.size(); i++)
            if (sphere[i].isTooClose(position))
                return (position - spheres[i].center).norm();
        for (int i = 0; i < blocks.size(); i++){
            if (blocks[i].isTooClose(position))
                return (position - blocks[i].center)).norm();
    return Vector3::ZERO;
}
```

Zu dem Zeitpunkt, zu dem die Methode aufgerufen wird, sollte über die `isTooClose(..)` Methode bereits bekannt sein, ob sich das Objekt in Kollisionsnähe befindet. Somit werden im Fall von `DodgeObjects`, die nur Sphären oder Boxen sind, direkt die Ausweichkurse als Ergebnis geliefert. Sollte es sich um eine Ellipse handeln, so wird die Normale der Ellipse nach der Formel aus [Dic06] ermittelt. Im Fall von Gruppen muss jedoch für jede Untergruppe kontrolliert werden, ob sich ein Punkt in Kollisionsreichweite von diesem befindet und in diesem Fall ebenfalls die Ausweichroute nach dem in Kapitel *Problemstellung bei der Erfassung* [3.2.4] erläuterten Algorithmus berechnet werden. Da dies aber nicht unbedingt immer der Fall sein muss, wenn die Position sich innerhalb des übergeordneten Objekts der Gruppe befindet, muss es als Standardrückgabewert einen Vektor ohne Bewegungsrichtigung geben, damit der Kurs des Fischs nicht weiter verfälscht wird.
Da die Sphären und Boxen die Meshes nur grob erfassen, gibt die Oberfläche der abstrahierten Meshes keinerlei Rückschluss darüber, wie weit die Positionen tatsächlich von den Meshes entfernt sind. Für die Kollisionserkennung mit Sphären wird überprüft, ob die gegebene Position sich innerhalb des Radius um das Zentrum befindet. Eine Skalierung je nach Nähe zum Zentrum findet nicht statt, die Richtung wird jedoch vorher normalisiert. Sollte eine Kollisionsgefahr erkannt worden sein, so wird die Richtung vom Zentrum weg als Ergebnis geliefert. Falls keine Kollision

droht, ist das Ergebnis ein Vektor, der in keine Richtung zeigt.
Für die spätere Kollisionserkennung mit Boxen wird überprüft, ob die gegebene Position sich innerhalb der Box befindet. Sollte dies der Fall sein, wird wie bei der Sphäre als Ergebnis die Richtung geliefert, die vom Zentrum der Box wegzeigt. Diese Richtung wird ebenfalls normalisiert.
In folgender Tabelle [4.1] werden die Messergebnisse für verschiedene Szenarien zusammengefasst. Da die Simulationsdaten für die Korallenaktualisierung zu kurzzeitigen FPS-Einbrüchen führen, wurden diese für die Berechnung ausgeschaltet, um Ausreißer zu verhindern.

Tabelle 4.1: Messwerte der Rechenzeit der Kollisionsvermeidung

Anzahl Fische	Anzahl gemessener Fische	Durchschnittliche Rechenzeit pro Frame in Sek.	Durchschnittliche Rechenzeit Ausweichen in Sek.	Anzahl erfasster Objekte
1126	500	0.01127	0.00453	13
1626	1000	0.01769	0.00725	13
2626	2000	0.02493	0.01625	13

Anhand der Ergebnisse fällt auf, dass die Rechenzeit für die Kollisionserkennung signifikante Anteile von der eigentlichen Rechenzeit pro Frame ausmacht. Jedoch ist diese Rechenzeit im Vergleich zum Kapitel *Probleme mit Meshkollisionen* [3.1.4] akzeptabel, da die Framerate selbst für über 2500 Fische nicht unter 40 FPS eingebrochen ist und für ca. 1600 Fische bei durchschnittlichen 56 FPS lag. Die realistischen Daten für die Messung sind jedoch deutlich höher anzusetzen, da für die Messung ein Zeitfenster gewählt, wurde in dem der signifikante Großteil des Schwarms in keiner Kollisionsreichweite befand, somit also den worst case an Rechenzeit hatte und das Testsystem relativ alte Hardware besitzt.
Ergänzend zu diesen Ergebnissen wurde zusätzlich ein CPU-Sampling mit 1300 Fischen, während die Korallen deaktiviert wurden, vorgenommen. Das Sampling ergab eine absolute Zahl von 42.399 Inclusive Samples, der Anteil für die Methode `calcDodge(..)` ergab dabei 1.0602 Inclusive Samples, was einen Anteil von ca. 25.01% ausmacht. Da diese Zahl jedoch keine Rückschlüsse auf die Performance der Funktion zulässt, ist sie nicht aussagekräftig.

Kapitel 5

Refactoring der Klasse Fish

5.1 Problemstellung

In diesem Kapitel wird beschrieben, welche Refactoring-Maßnahmen für die Klasse `Fish` vorgenommen wurden, um Problemstellungen, die an vielen Stellen unterschiedlich gelöst wurden, einheitlich zu lösen, komplexe Abläufe zusammenzufassen und Parameter aus dem Ablauf zu entfernen, die zu Verwirrung führen und durch einfachere Logik ersetzt werden können.
Ziel des Refactoring ist, dass sämtliche Methoden der Klassen `Fish, Swarm, SwarmManager, CollisionManager und FishManager` aus der Korallenriffsimulation [VRC] nach dem CQR Prinzip [Ver13] funktionieren, um Abläufe zu atomarisieren und die Funktionalität der einzelnen Methoden zu verdeutlichen. Ein zusätzliches Ziel der Atomarisierung ist, die Modularisierung der Klasse `Fish` ebenfalls umzusetzen.

5.2 Modularisierung des Fluchtverhaltens

Das erste Problem ist, dass das Fluchtverhalten für Fische unterschiedlich durchgeführt wird, abhängig davon, ob dieses bei einem einzelnen Fisch oder bei einem Fisch in einem Schwarm angewendet werden soll. Freie Fische weichen den Prädatoren direkt aus und passen ihre Bewegungsrichtungen kaum an. Für Fischschwärme wurden viele unterschiedliche Algorithmen implementiert, damit sich diese möglichst realistisch verhalten. So sind z. B. Clownfische in ihre Anemone geflohen, Fische, die sich in "Boid-Schwärmen" bewegen, fliehen nach Möglichkeit als große Gruppe in eine Richtung und Fische in kohärenten Schwärmen umschwimmen die Gefahr. Da alle Algorithmen für das Fischverhalten nacheinander entwickelt

wurden, als es zum Teil noch kein Ausweichverhalten gab, wurde dieses für jeden Schwarm individuell entwickelt, obwohl die Kernfunktionalität, den Prädatoren auszuweichen, für alle Fische identisch ist. Zusätzlich zu der Extrahierung dieser Algorithmen wurde das Problem, Fische innerhalb der relevanten Simulationsumgebung zu halten, ebenfalls über die Kursanpassungslogik erfasst. Zusammen mit dem neuen Problem der Kollisionsvorbeugung wurden alle Verfahren, die die Kurse von Fischen verändern, an einer Stelle umgesetzt, um die zukünftige Wartbarkeit des Codes sowie die Integration neuer Algorithmen zu vereinfachen. Ausnahmen bilden hierbei die Clownfish-Schwärme, da diese ein alternatives Fluchtverhalten haben.
Alle Algorithmen sollen gemäß CRS [Ver13] in einer neuen Klasse `DodgeManager`, die über die Klasse `CollisionManager` erreicht werden, zusammengefasst werden. Die Schnittstelle hierfür wird in der Klasse `Fish` die Methode `calcDodge(..)` sein, die weiterhin von allen `Fish` und `Swarm` Managern aufgerufen werden muss. Eine Auslagerung in die Methode `update(...)`, welche die zentrale Methode zur Berechnung der Fischpositionen ist, ist nicht ohne Performanceverlust möglich, da diese Methode von dem Hauptthread genutzt wird, welcher für die Renderprozesse zuständig ist und somit eine Berechnung der Ausweichkurse auf diesem Thread zu starken FPS-Einbrüchen führen würde.
Zuletzt wurde das Ausweichverhalten aus allen Algorithmen, die die einzelnen Fische und Schwärme betroffen haben, wo es möglich war, entfernt. Diese Änderung führt dazu, dass die eigentlichen `calc(...)` Methoden in der Klasse `Fish` wieder das Verhalten der Fische und Schwärme umsetzen und keine speziellen Fälle berücksichtigt werden müssen.

5.3 Entfernung von Attributen

Ein weiterer Schritt, den Code einfacher wartbar zu machen, ist den Parameter `mMoveAnimation` aus den Algorithmen für die Bewegungsrichtung der Fische zu entfernen. Der Parameter `mMoveAnimation` löste zu frühen Zeitpunkten im Projekt das Problem, dass Fische durch starke Richtungsänderungen zwischen den Frames stark zappelten. Der Ansatz war, mit dem Parameter `mMoveAnimation` die letzte Bewegungsrichtung zu erhalten und in Kombination mit `mMovement` nur eine mäßige Richtungsänderung auszulösen. Die beiden Parameter wurden statisch miteinander pro Frame verrechnet, ohne die Framerate an sich zu berücksichtigen. Diese Umsetzung soll durch einen neuen Ansatz, das Attribut `mMoveAnimation` aus dem Code zu entfernen, umgesetzt werden und die Kursanpassung über einen

weiteren Parameter in der Signatur zu streichen, welcher angibt, wie viel Zeit seit dem letzten Frame vergangen ist und mit diesem die Richtungsänderung zu skalieren, um das gleiche Verhalten zu erzielen.

5.4 Integration des Ausweichverhaltens

Das Ausweichverhalten wurde an vielen Stellen realisiert und bezog sich auf verschiedene Aspekte, wie den Spieler, den Prädator, den Boden, die Wasseroberfläche und die Grenzen, in denen sich die Tiere aufhalten sollten. Sämtliche Algorithmen, die das Ausweichverhalten realisiert haben, wurden aus dem aktuellen Code entfernt und stattdessen in einer neuen Methode `calcDodge(..)` modularisiert. Diese Methode nutzt einen `DodgeManager`, der für die aktuelle Position die Kursanpassung, wie im Kapitel *Problemstellung bei der Erfassung* [3.2.4] erläutert, berechnet und diese Richtung als Rückgabewert liefert. Mit dieser Änderung ist es möglich, das gesamte Ausweichverhalten an einer einzigen Stelle zu berechnen.
Der Algorithmus, um den Ausweichkurs zu berechnen läuft nach folgendem Prinzip ab:

```
void calcDodge(double timeSinceLastFrame){
    Vector3 dodgeDir = getDodgeDirection(mCurrentPosition);
    double fr = timeSinceLastFrame;
    if (dodgeDir != Vector3::ZERO)
        mMovement = 1*fr *dodgeDir +(1-1*fr)*mMovement;
    mMovement.normalise();
}
```

Die Ausweichkursberechnung braucht `timeSinceLastFrame` in Sekunden, um die Bewegung zu skalieren. Im ersten Schritt wird die Ausweichrichtung vom `DodgeService` berechnet, diese Richtung ist normalisiert. Sollte diese Richtung ungleich dem Vektor (0,0,0) sein, so muss ein Ausweichkurs berechnet werden. Dieser wird mit der Framerate und der Differenz von einer Sekunde und der Framerate der aktuellen Bewegungsrichtung skaliert. Dies führt zu dem Verhalten wie in dem Kapitel *Problemstellung bei der Erfassung* [3.2.4] beschrieben. Da dieser Vektor nicht mehr die Länge Eins hat und alle Methoden so implementiert sind, dass sie davon ausgehen, dass alle Richtungen bereits normalisiert sind, wird die neue Richtung wieder normalisiert.

5.5 Vereinheitlichung der Bewegungsberechnung

Im letzten Schritt wurden für die Schwärme einheitliche Skalierungen für die Bewegungsanpassung ermittelt. Die Bewegung von Fischen in Schwärmen wurde bisher zustandsbasiert berechnet. Die Zustände hingen davon ab, in welcher Situation sich der Fisch im Verhältnis zum Schwarm befand. Eine Neuberechnung der Skalierungen ist nötig, da der Parameter `mMoveAnimation` in dem Kapitel *Integration des Ausweichverhaltens* [5.4] beschrieben, weggefallen ist und somit die bisherigen Skallerungen nicht mehr valide sind. Darüber hinaus wird nun die Framerate für die Neuberechnung als eigener Faktor berücksichtigt.
Die Zustände waren:
ESCAPE, wenn ein Prädator oder der Spieler in der Nähe waren
JOINING, falls ein Fisch nah an dem Schwarm, aber nicht im Schwarm war
OUTSITE, wenn der Fisch weit abseits den dem Schwarm war
ADAPT, wenn der Fisch im Schwarm ist, aber sich zu nah an anderen Fischen befindet
DEFAULT, falls der Fisch sich im Schwarm befindet.
Die Richtungen, welche in die Berechnungen eingehen, sind folgende:
adaptMov gibt die Richtung an, dass der Fisch den Abstand zu den Nachbarn einhält
toLeader gibt die Richtung zum Anführer es Schwarms an
mMoveAnm gab die letzte Bewegung an, wurde entfernt
mMove gibt die neue Richtung an, in die der Fisch schwimmen soll
mSwarmMove gibt die gewichtete Bewegungsrichtung des Schwarms an
toCenter gibt die Richtung an, die zu dem gewichteten Schwarmzentrum führt.
In der folgenden Tabelle [5.1] werden alle bisherigen Skalierungen für das Beispiel kohärente Schwärme und in der letzten Zeile **RESULT** die neuen Skalierungen der einzelnen Richtungen aufgelistet. Der Faktor `aF` ist der `AdaptFaktor`, der je nachdem, wie viele Fische sich um den einzelnen Fisch herum befinden, zwischen 4 und 13 variiert.

Tabelle 5.1: BewegungsAnpassung

	adapt Mov	to Leader	mMov Anm	m Mov	mSwm Mov	to Center
ESCAPE	0.1	0	0.55	0.2	0.75	0.05
JOINING	0.025	0.03	0.8	0.175	0.066	0.015
OUTSITE	0	0	0.55	0.1	0.1	0.125
ADAPT	0.02	0	0.85	0.15	0.059	0.01
DEFAULT	0.01	0.015	0.9	0.125	0.075	0.05
RESULT	aF*fr	0	0	1-(8+2.5*aF)*fr	(8+aF)*fr	aF/2*fr

Die resultierenden Skalierungen wurden durch manuelle Vermessung im Korallenriff ermittelt, die solange approximiert wurden, bis der kohärente Schwarm wieder sein altes Verhaltensmuster aufwies. Für alle weiteren Schwarmtypen wurden analoge Anpassungen durchgeführt.
Die Reihenfolge für sämtliche Richtungsanpassungsmethoden sollte immer zuerst die `calcBehavior(..)`-Methode und danach die `calcDodge(..)`-Methode in der Klasse `Fish` sein, damit die geringe Skalierung des Ausweichfaktors nicht durch Schwarmberechnungen verloren geht.

Kapitel 6

Schwärme hinzufügen

In diesem Kapitel wird erläutert, welche Schritte notwendig sind, um einen neuen Schwarm mit neuem Verhalten hinzuzufügen.

6.1 Betroffene Dateien

Um einen Schwarm hinzuzufügen, müssen an verschiedenen Stellen im Projekt Methoden, Attribute und Dateien hinzugefügt werden, mit denen das Schwarmverhalten realisiert wird. Die betroffenen Klassen sind:
SwarmManager: Diese Klasse initialisiert alle Schwärme mit ihren Fischen und speichert diese in eine Liste ab, welche zur Aktualisierung der Schwärme pro Frame genutzt wird.
Swarm: Diese Klasse differenziert die unterschiedlichen Schwarmtypen und führt grundlegende Berechnungen für Parameter durch, die die einzelnen Fische für ihr Verhalten benötigen.
Fish: Diese Klasse enthält sämtliche Logik zur Berechnung der Bewegung für das nächste Frame für jeden Fisch und sein jeweiliges Verhalten.
Das betroffene Dokument ist:
AnimalConfig: Enthält die Attribute, wie viele Schwärme der Art es gibt, wie viele Fische im jeweiligen Schwarm vorkommen, sowie weitere relevante dynamische Parameter.
Der Betroffene Ordner ist:
solution media fish: In diesem Ordner müssen sämtliche Dateien abgelegt werden, die für die Darstellung als Mesh und die Animation nötig sind.

6.2 Ablauf

Die folgende abstrahierte Abbildung [6.1] beschreibt sämtliche Schritte, die durchlaufen werden müssen, um einen neuen Schwarmtyp mit seinen Eigenschaften in der Simulationsumgebung hinzuzufügen. Hierbei ist zu beachten, dass der Punkt *Mesh erstellen* [6.2.1], zwar notwendig ist, um eine neue Fischart hinzuzufügen, jedoch wird die Erstellung des Meshs nicht explizit erläutert.

6.2.1 Mesh erstellen

Zuerst muss, um einen neuen Schwarm mit neuen Fischen in die Simulationsumgebung hinzuzufügen zu können, ein Mesh für den Fisch vorhanden werden. Sollte ein bestehendes Mesh gewählt werden, sind an dieser Stelle keine weiteren Schritte notwendig. Sollte jedoch ein neues Mesh in die Simulation eingebunden werden, muss dieses an mehreren Stellen integriert werden.
Die Integration erfolgt zunächst im Dateisystem des Ordners `solution media fish`, dort müssen für den Fisch sämtliche für die Darstellung und Animation relevanten Dateien hinzugefügt werden. Zusätzlich müssen in `Fish.h` und `Fish.cpp` in der Region `DefaultValues` einige String-Konstanten angelegt werden, die das Mesh und die Animation dieses Meshes referenzieren.

6.2.2 Config erweitern

In der `Config` müssen sämtliche relevanten Parameter für die Schwärme angelegt werden. Die Parameter, welche festlegen, wie viele Schwärme und Fische pro Schwarm in der Simulation angezeigt werden, sind zwingend notwendig. Gibt es neben diesen Parametern weitere, welche hinreichend notwendig, aber dynamisch konfigurierbar sein sollen, so müssen diese ebenfalls in der `Config` erfasst werden.
Die `Config` ist unter `solution  media  AnimalConfig.txt` zu finden.

6.2.3 String Konstanten

Um auf die Parameter in der `Config` zugreifen zu können, müssen in den `SwarmManager.h` in der Region Config-Strings die notwendigen String-Konstanten gesetzt werden, welche identisch mit den Attributen aus der `Config` aus dem Kapitel *Config erweitern* [6.2.2] sind, die referenziert werden sollen. Die String-Konstante im `SwarmManager` wird aus dem Präfix der Region, in der sich das Attribut in der `Config` befindet und dem Namen des Attributes, zusammengesetzt.

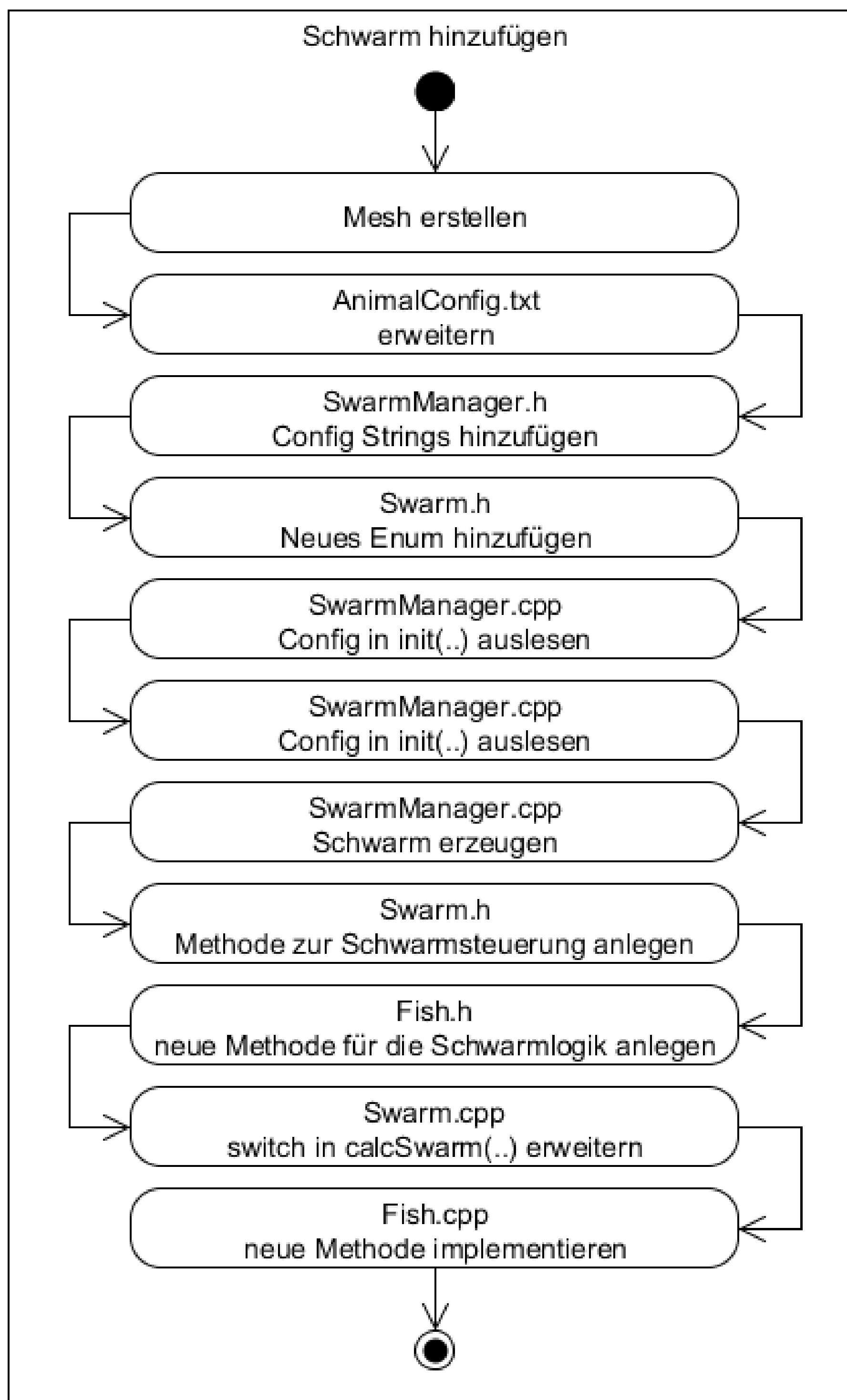

Abbildung 6.1: Schwarm hinzufügen

6.2.4 Zusätzliche Attribute definieren

Das Schwarmverhalten wird im Frameworks durch den Schwarmtyp unterschieden, der als Enum in der Klasse `Swarm` unterschieden wird. Soll der Schwarm ein neues Verhalten aufweisen, muss für dieses Verhalten ein neuer `Enum` Fall hinzugefügt werden. Für den Fall, dass ein Schwarm bereits das Zielverhalten aufweist, ist es ausreichend, für diesen das `Enum` zu wählen, mit dem das Schwarmverhalten realisiert wurde.
Neben dem Schwarmtyp `Enum` existiert ein weiteres `Enum` in der Klasse `Fish`, dieses enthält verschiedene Zustände, in denen sich ein Fisch befinden kann. Zu jedem Zustand wird eine Zahl zugeordnet. Diese Zahl gibt die Geschwindigkeit an, mit der sich ein Fisch fortbewegt, wenn er sich in diesem Zustand befindet. Die Einheit hierfür ist Zentimeter. Fische, deren Bewegungsgeschwindigkeit nicht über die existierenden Zustände abgebildet werden kann, müssen neue Zustände mit den entsprechenden Geschwindigkeiten hinzugefügt werden.
Benötigt das neue Verhaltensmuster weitere Attribute, die bisher in der Klasse `Fish` nicht enthalten sind, müssen diese ebenfalls mit den Methoden, welche diese steuern definiert werden.

6.2.5 Config auslesen

Alle relevanten Attribute für den Schwarm müssen in der Klasse `SwarmManager.cpp` in der Region Attribute-Settings aus der `Config` ausgelesen werden. Dies geschieht über die String-Konstanten, die in der Klasse `SwarmManager` definiert wurden. Diese Attribute müssen in temporären Parametern gespeichert werden, da diese Attribute für die spätere Initialisierung des Schwarms benötigt werden.

6.2.6 Schwarm initialisieren

In der Klasse `SwarmManager` muss eine Region für den neuen Schwarmtyp erstellt werden. In dieser Region wird der Schwarm initialisert. Für die Initialisierung des Schwarms werden die notwendigen und hinreichenden Parameter aus der `Config` benötigt, sowie ein Schwarmtyp, der in dem Kapitel *Zusätzliche Attribute definieren* [6.2.4] angelegt wurde. Für die Initialisierung der Fische sind das Mesh und weitere hinreichende Parameter für die Realisierung der Logik nötig.
Als Erstes muss die Entscheidung getroffen werden, ob der Schwarm einen Anführer braucht, falls ja, muss ein Anführer deklariert und initialisiert werden. Die Größe des Anführers sollte stets auf 0.0 gesetzt werden, da dieser sonst optisch auffallen könnte, weil dieser nur zur Orientierung des Schwarms dient. An dieser Stelle ist je-

doch zu beachten, dass obwohl der Anführer eine Größe von 0.0 besitzt und aufgrund der internen Logik weiterhin ein Mesh und eine Animation braucht. Diese können über die String-Konstanten aus der Klasse `Fish` in der Region `DefaultValues` referenziert werden.
Für die Initialisierung des Schwarms ist der Typ des Schwarms und falls vorhanden der Anführer notwendig.
Sofern der Schwarm so groß ist, dass durch ein Grid ein Performance Gewinn möglich ist, sollte das Attribut `calcByGrid` für diesen auf `true` gesetzt werden. Die Distanzberechnung durch das Grid benötigt für große Schwärme im Schnitt n*log(n) Vergleiche, wobei hier zu berücksichtigen ist, dass das Grid regelmäßig neu erstellt werden muss, bevor Vergleiche stattfinden können. Für kleine Schwärme wäre somit die Pflege des Grids performancelastiger als n*n Vergleiche durchzuführen. Das Grid für die Abstandsberechnung der Fische sollte an dieser Stelle nicht mit dem `Worldgrid` verwechselt werden, welches andere Anwendungsgebiete hat.
Nachdem der Schwarm initialisiert wurde, müssen alle Fische dem Schwarm hinzugefügt werden. Dafür läuft im nächsten Schritt eine Schleife so häufig durch, wie Fische im Schwarm vorhanden sind, um alle Fische zu initialisieren und deren relevante Parameter zu setzen.
Für die Initialisierung der Fische sind zunächst nahezu ausschließlich `Ogre` spezifische Parameter notwendig. Diese Parameter sind der SceneManager, das Mesh und die Animation. Zusätzlich werden weitere Parameter erfasst, wie der Anführer des Schwarms, sofern er benötigt wird, die Skalierung für die Größe des Mesh, der `CollisionManager` und das `Worldgrid`.
Das Mesh und die Animation wird bei der Initialisierung nur über den Pfad referenziert, dieser ist in der Klasse `Fish.cpp` in der Region `DefaultValues` aus dem Kapitel *String Konstanten* [6.2.3] zu finden. Sollte das Mesh größer oder kleiner als die Fische in der Simulationsumgebung sein, so ist es möglich, dieses nachträglich zu skalieren. Sollte die Größe bereits stimmen, so muss an der Stelle im Konstruktor eine Eins übergeben werden. Als Umrechnungsfaktor gilt, dass eine Einheit in der Simulationsumgebung einem Zentimeter entspricht.
Nachdem der Fisch initialisiert wurde, müssen noch weitere Attribute gesetzt werden. Dies sind die Startposition, die ID im Schwarm und die Skalierung von der Bewegungsgeschwindigkeit, falls diese variabel sein soll.
Wenn alle Attribute gesetzt sind, muss dieser über die Methode `joinSwarm(Fish*)` hinzugefügt werden, damit dieser mit seinem Verhalten in jedem Frame aktualisiert wird.
Der letzte Schritt, nachdem der Schwarm mit allen Parametern erzeugt wurde, ist die Instanz von diesem in den Vektor `mSwarms` hinzuzufügen, damit dieser bei der Aktualisierung berücksichtigt wird.

6.2.7 Schwarm aktualisieren

Für die Aktualisierung des Schwarms gibt es zwei Methoden in der Klasse `Swarm`. Sobald ein Schwarm im Vector `mSwarms` in dem `SwarmManager` registriert ist, werden beide Methoden jeweils einmal pro Frame aufgerufen. Die Methoden sind `updateSwarm(..)`, welche vom Hauptthread genutzt wird und `calcSwarm(..)`, welche vom Nebenthread genutzt wird. Die Methode `updateSwarm(..)` aktualisiert sowohl die Position als auch den Animationszustand für jeden Fisch im Schwarm. An dieser Methode sind in der Regel keine Änderungen vorzunehmen.
Der zweite Methode `calcSwarm(..)` enthält einen switch-Block für alle Schwarmtypen. In diesen Block muss, sofern ein neuer Schwarmtyp hinzugefügt wurde, dieser ebenfalls hinzugefügt werden und eine entsprechende `calc(..)` Methode für den Schwarm mit den nötigen Parametern in die Klasse `Swarm.h` und `Swarm.cpp` implementiert werden. Die Hauptaufgabe dieser Methode ist es, für alle Fische relevante Parameter auszurechnen, welche auf der Ebene der Klasse `Fish` nicht mehr erfasst werden können, da die Logik der Klasse `Fish` Informationen über den Schwarm enthält, in dem sich die individuellen Fische befinden. Die wichtigste Berechnung ist die Abstandsberechnung zwischen den einzelnen Fischen, mit dem Ziel für jeden Fisch die Position des zu ihm nächsten Fischs zu ermitteln, damit jeder den korrekten Abstand einhalten kann. Da die Abstandsberechnung sehr performanceintensiv ist, sollte diese nur in bestimmten Zeitintervallen und nicht in jedem Frame stattfinden. Sind ebenfalls weitere Berechnungen für die Fische nötig, die zur Berechnung Daten brauchen, die nur über die Klasse `Swarm` referenzierbar sind, so müssen diese ebenfalls an dieser Stelle berechnet werden. Des Weiteren sollte für einen flüssigen Ablauf in der Simulation pro Frame die Methoden `calcMoveForMyBehavoir(..), calcMove(..) und calcDodge(..)`, die in dem Kapitel *Integration des Ausweichverhaltens* [5.4] erläutert wurde, aufgerufen werden, da diese das Verhalten, die Position und Ausweichrichtung anpassen.

6.2.8 Fisch aktualisieren

Sollten die Fische des neuen Schwarmes ein neues Verhaltensmuster realisieren, so ist es notwendig, für dieses Muster eine neue Methode in der Klasse `Fish` anzulegen. Die Methode `calcMoveForMyBehavoir(..)` muss sämtliche Logik für das Verhaltensmuster der Fische im entsprechenden Schwarm enthalten.

6.3 Verhaltensregeln für Fische im Schwarm

Die Implementierung der Methode in `Fish.cpp` kann, je nachdem wie ein Schwarm sich verhalten soll, sehr zeitaufwändig sein, da die Gewichtung der Parameter jeweils unterschiedliche Einflüsse auf Form und Verhalten des Schwarms haben. In diesem Kapitel wird erläutert, welche Parameter welchen Einfluss auf das Schwarmverhalten haben.

Schwarmzentrum: Das Schwarmzentrum ist das gewichtete Zentrum aller Positionen der Fische im Schwarm und dient vor allem dazu, dass Fische sich sammeln können. Je nachdem wie stark der Parameter gewichtet ist, sammeln sich die Fische stärker an diesem Punkt oder nicht. Typische Probleme mit zu starker Gewichtung sind, dass der Schwarm in der Mitte eine Art Taille hat und die Fische sich an dieser Stelle verklumpen. Sollte dies der Fall sein, so sollte die Gewichtung für diese Richtung reduziert werden. Sollten die Fische Probleme haben, sich zu seinem Schwarm zu sammeln, muss die Gewichtung dieser Richtung erhöht werden.

Anführer: Der Anführer gibt dem Schwarm die Richtung vor. Diese Richtung ist vor allem wichtig, damit der Schwarm kohärent in eine Richtung schwimmt und z. B. im Fall einer Richtungsänderung an dem Rand seitlich nicht weiter schwimmt. Je nachdem, wie einheitlich der Schwarm hintereinander in die gleiche Richtung schwimmen soll, ist die Gewichtung dieses Parameters zu wählen.

Allgemeine Schwarmbewegung: Diese Richtung gibt die durchschnittliche Bewegung des Schwarm wieder. Dieser Parameter ist wichtig, damit die Fische bei einer Kursänderung die gleiche Richtung beibehalten. Sollte bei einer Kursänderung Chaos ausbrechen, ist dieser Parameter zu schwach gewichtet.

Anpassung an den nächsten Fisch: Diese Richtung ist wichtig, damit die Fische die Mindestdistanz zu ihren Nachbarn aufrecht erhalten, oder, falls dieser zu weit weg ist, sich diesem nähern. Sollten die Fische beim Anpassen stark zappeln, ist dieser Parameter zu stark gewichtet, sollten die Fische zu nah aneinander schwimmen, ist diese Richtung zu schwach gewichtet.

Geschwindigkeitsfaktor: Dieser Parameter wird in `SwarmManager.cpp` gesetzt und gibt die Abweichung von der gegebenen Geschwindigkeit an. Je nachdem, ob der Schwarm lang sein soll, sollte dieser Faktor stark variieren. Sollte der Schwarm eher sphärenförmig aussehen, so sollte dieser Faktor kaum von der

Ausgangsgeschwindigkeit abweichen.

Des Weiteren ist wichtig, dass alle Richtungen normalisiert in die Gewichtung eingehen, sowie Richtungen, die in Konkurrenz stehen, richtig gewichtet werden, wie z. B. die Richtung zum Zentrum und die Anpassungsrichtung. Sollte die Zentrumsrichtung eine höhere Gewichtung als die Anpassungsrichtung haben, so entstehen Klumpen von Fischen, die sich nicht voneinander wegbewegen können.

Kapitel 7

Fazit

In diesem Kapitel werden die Ergebnisse der Bachelorarbeit zusammengefasst. Dies sind folgende Themen: Die Analyse des gegenwärtigen Algorithmus zur Kollisionsvermeidung, die Implementierung eines neuen Moduls für die Kollisionsvermeidung, das Refactoring der Klassen, die für die Simulation der Fische und Schwärme verantwortlich sind und die Beschreibung der Integration von neuen Schwärmen mit ihren Verhaltensmustern in die Simulation.
Die Analyse hat ergeben, dass der gegenwärtige Algorithmus zur Kollisionserkennung in sämtlichen Aspekten für eine Kollisionsvermeidung nicht in Frage kommt. Die Hauptgründe waren die enorme Rechenzeit und die starke Standardabweichung, die zu sehr instabilen Framerates führen würden. Zusätzlich dazu erfolgte die Kollisionsvermeidung durch eine parallele Verschiebung zum Mesh, was extrem unrealistisch auf den Betrachter wirken würde. Um eine Kollision zu verhindern, muss der Kurs frühzeitig angepasst werden. Algorithmen, die die Position anpassen, verhindern die Kollision zu spät und sind optisch auffällig.
Das implementierte Modul, mit dem die Kollisionsvermeidung umgesetzt wurde, hat durch die Abstraktion von Meshes durch Sphären, Boxen und Ellipsen und stetige Anpassung der Ausweichkurse sämtliche Probleme gelöst, sodass Fische sämtlichen erfassten Meshes auf optisch natürlich aussehenden Pfaden ausgewichen sind. Die Rechenzeit lässt für ein relativ altes System für ca. 2000 Fische eine Framerate von ca. 56 FPS zu. Das Modul wurde so implementiert, dass Sphären und Boxen, einzeln als auch gruppiert, eingelesen werden können. Im gegenwärtigen Zustand müssen jedoch alle Objekte manuell gemessen und über eine `Config` erfasst werden. Dieser Ansatz ist für den Zweck, wenige große Meshes in der Simulationsumgebung zu erfassen, ausreichend, jedoch für Projekte in denen deutlich mehr Objekte erfasst werden müssen, deutlich zu aufwändig und würde einen automatisierten Erfassungsalgorithmus benötigen.
Die Klasse `Fish` sowie sämtliche Klassen, die für die Schwarmlogik zuständig waren,

wurden nach dem CQS-Design gewartet, um dem Programmierer mehr Rückschlüsse über den Inhalt der Funktionen zu geben. Aus der Klasse `Fish` wurde ein Parameter, der die Bewegungsanpassung von Fischen enorm kompliziert gemacht hat, erleichtert und mit einer Logik ersetzt, die die vergangene Zeit zwischen den Frames als Skalierungsfaktor genutzt hat. Diese Änderung erzwang eine Anpassung aller Methoden, die für die Bewegungsanpassungen der Fische zuständig waren. Zusätzlich sind einige Methoden durch die Entfernung des Parameters redundant geworden und weggefallen.
Im letzten Abschnitt wurde erläutert, welche Schritte zu durchlaufen sind, um einen neuen Schwarmtyp in die Simulation hinzuzufügen. Dieses Kapitel hat sich mit sämtlichen Methoden befasst, welche angepasst oder hinzugefügt werden müssen. Der Aspekt, wie ein Mesh erstellt wird, wurde dabei nicht berücksichtigt. Für die Skalierung der Parameter, die für die Implementierung der Schwarmlogik notwendig sind, wurde erläutert, wie sich diese auf die Schwärme auswirken, jedoch wurde, da jeder Schwarmtyp eine eigene Logik erfordert, keine explizite Skalierung vorgeschlagen.
Zusammenfassend wurden sämtliche Klassen für die Schwarmlogik gewartet und erweitert, sodass eine einfache Integration von neuen Schwärmen möglich ist und sämtliche Fische ohne großen Aufwand Objekten ausweichen. Der Aspekt, wie Meshes optimal zerlegt werden, wurde durch den großen damit verbundenen Aufwand einen geeigneten Algorithmus zu implementieren und der Gefahr, dass dieser die Anforderungen nicht generisch erfüllt, absichtlich ausgelassen.

Kapitel 8

Ausblick

In dieser Bachelorarbeit wurde ein Modul geschaffen, welches für sehr viele Objekte eine Kollisionsvermeidung ermöglicht und das Problem, wie die Daten erfasst werden, mittels eines Adapters gelöst. Der Adapter liest aktuell aus einer `Config` Daten ein, welche für die Kollisionsvermeidung relevant sind, jedoch ist eine automatisierte Einlesung von Daten noch nicht implementiert. Ein Schritt für die Zukunft wäre, einen Algorithmus zu implementieren, der sämtliche Meshes in mathematische Körper zerlegt, welche für eine Kollisionsvermeidung gut geeignet sind. Ein weiterer Schritt im Performance zu sparen, wäre, die Objekte rekursiv zu erfassen, sodass der Aufwand für den gesamten Ablauf so minimal wie möglich ist.
Ein weiterer Aspekt, um den das Modul erweitert werden kann, ist, weitere mathematische Körper wie Prismen und Pyramiden zu erfassen, womit es möglich wäre, ein Mesh noch optimaler zu zerlegen. Diese Möglichkeiten würde den Algorithmus, der das Mesh zerlegt, von der Komplexität deutlich erhöhen, jedoch wäre der Aufwand zur Laufzeit gering und die Präzision deutlich besser.

Literaturverzeichnis

[CLL99] Coad, Peter ; Lefebvre, Eric ; Luca, Jeff de: *Java Modeling In Color With UML*. 1. Aufl. Prentice Hall PTR, 1999. – ISBN 978–0130115102

[CSMOS03] Chang, Dong E. ; Shadden, Shawn C. ; Marsden, Jerrold E. ; Olfati-Saber, Reza: *Collision Avoidance for Multiple Agent Systems*. Version: 2003. http://shaddenlab.berkeley.edu/uploads/changetal03.pdf, Abruf: 13.01.2016. – 2–4 S.

[Dic06] Dickheiser, Michael: *Game Programming Gems 6*. 2006. – 442–447 S. – ISBN 1–58450–450–1

[Fau03] Fauerby, Kasper: *Improved Collision detection and Response*. http://www.peroxide.dk/papers/collision/collision.pdf. Version: 2003, Abruf: 13.01.2016

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns Elements of Reusable Object-Orientet Software*. 1995. – 87–95 S. – ISBN 0–201–63361–2

[HCCV11] Hafner, M. R. ; Cunningham, D. ; Caminiti, L. ; Vecchio, D. D.: *Automated Vehicle-to-Vehicle Collision Avoidance at Intersections*. 2011. – 2–7 S.

[RW13] Rene Weller, Stefan Gruthe Jörn T. Gabriel Zachmann Z. Gabriel Zachmann: *Fast Sphere Packings with Adaptive Grids on the GPU*. Version: 2013. http://www.gcc.tu-darmstadt.de/media/gcc/papers/Teuber-2013-GI.pdf, Abruf: 18.12.2015. – 3–9 S.

[Sol06] Solomon, David: *Curves and Surfaces for Computer Graphics*. 2006. – 46 S. – ISBN 0–387–24196–5

[Ver13] Vernon, Vaughn: *Implementing Domain-Driven Design -*. 1. Aufl. Amsterdam : Addison-Wesley, 2013. – ISBN 978–0–133–03988–7

[VRC] *VR-Coralreef.* http://cgvr.cs.uni-bremen.de/teaching/studentprojects/vrcoralreef/, Abruf: 2015-11-01

[WZ10a] Weller, Rene ; Zachmann, Gabriel: *Inner Sphere Trees for Proximity and Penetration Queries.* Version: 2010. http://www.roboticsproceedings.org/rss05/p10.pdf, Abruf: 18.12.2015. – 3–6 S.

[WZ10b] Weller, Rene ; Zachmann, Gabriel: *ProtoSphere: A GPU-Assisted Prototype Guided Sphere Packing Algorithm for Arbitrary Objects.* Version: 2010. http://cgvr.cs.uni-bremen.de/papers/siggraph_asia2010/ProtoSphereSiggraph.pdf, Abruf: 18.12.2015. – 1–2 S.

[ZK04] Zachmann, Gabriel ; Klein, Jan: *Point Cloud Collision Detection.* Version: 2004. http://zach.in.tu-clausthal.de/papers/pointcoll_eg2004_electr.pdf, Abruf: 18.12.2015. – 2–4 S.