Hochschule Konstanz
Fakultät Informatik
GPU-Programmierung

Eingereicht von
Tom Georgi
Matrikelnummer 294854
Tom.Georgi@htwg-konstanz.de

und

Moritz von der Heiden
Matrikelnummer 295349
Moritz.VonderHeiden@htwg-konstanz.de

G

P

Simulation eines Fischschwarm mittels CUDA und OpenGL

U

Konstanz, 04.07.2021

Ehrenwörtliche Erklärung

Hiermit erklären wir, *Tom Georgi, geboren am 04.06.1998 in Konstanz* und *Moritz von der Heiden, geboren am 07.07.1996 in Bad Säckingen*, dass wir

(1) die Dokumentation mit dem Titel

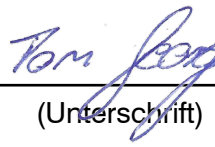
Simulation eines Fischschwarm mittels CUDA und OpenGL

unter Anleitung von Marco Fehrenbach selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt haben;

(2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet haben;

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 04.07.2021



(Unterschrift)

Konstanz, 04.07.2021

(Unterschrift)

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

GPU	Graphic Processing Unit
OpenGL	Open Graphics Library
CUDA	Compute Unified Device Architecture
VBO	Vertex Buffer Object
VAO	Vertex Array Object
FPS	Frames per Second
CPU	Central Processing Unit

1

Einleitung

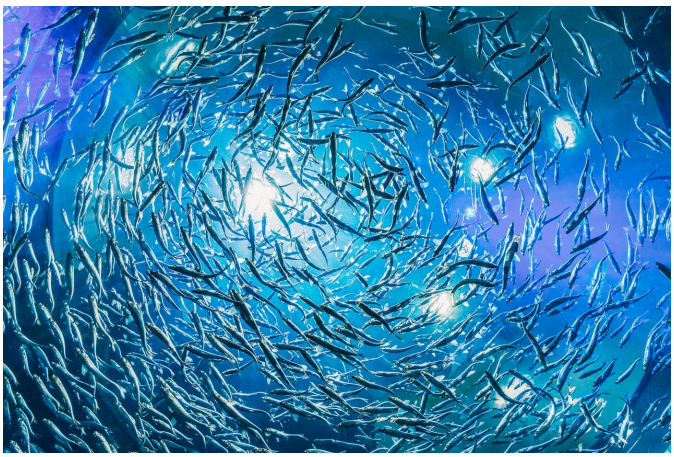


Abbildung 1.1: [Fis]

1.1 Motivation

Das Wort “Schwarm” bezeichnet eine “größere Anzahl sich [ungeordnet,] durcheinanderwimmelnd zusammen fortbewegender gleichartiger Tiere” [Def]. Hierzu zählt auch ein Fischschwarm.

Ein Fischschwarm bleibt eng beieinander, wobei jeder einzelne Fisch sich leicht in unterschiedliche Richtungen mit minimal unterscheidenden Geschwindigkeiten fortbewegt. Geleitet wird dieser durch die Futtersuche, durch Strömungen aber auch durch Fluchtreaktionen vor Jägern. Diese Verhaltensart kann leicht simuliert werden. Dazu wird das Schwarmverhalten auf ein Partikelsystem abgebildet und simuliert. Um den Anforderung eines Schwarms zu genügen, muss jeder Partikel einzeln und am besten parallel berechnet werden. Diese hochgradige Art der Parallelisierung eignet sich perfekt für die Graphic Processing Unit (GPU).

1.2 Zielsetzung

Ziel dieses Projekts ist es, einen Fischschwarm mittels eines Partikelsystems zu simulieren. Dabei soll zum Rendern der einzelnen Partikel die API Open Graphics Library (OpenGL) und für die Berechnung Compute Unified Device Architecture (CUDA) von Nvidia zum Einsatz kommen. Folgende Ziele sind definiert:

- Implementierung in 3D
- Schwarm folgt einer gemeinsamen Richtung (Trackpoint)
- Einzelne Partikel halten Abstand zueinander
- Implementierung eines Haifisches
- Schwarm versucht Haifisch auszuweichen

1.3 Aufbau der Arbeit

In den nächsten Kapiteln werden die benötigten Grundlagen zum Verständnis der Arbeit erläutert. Kapitel 3 zeigt die Implementierung des Partikelsystems. Dabei wird in diesem Kapitel genauer auf die Projektstruktur eingegangen. Des Weiteren soll das Kapitel die gemeinsame Verwendung von CUDA und OpenGL zeigen, um eine optimale Performance zu erreichen. Anschließend werden die gesammelten Ergebnisse evaluiert. Abschließend folgt noch ein Fazit mit Ausblick.

2

Grundlagen

Um später die Abschnitte in Kapitel 3 besser nachvollziehen zu können, wird in diesem Kapitel genauer auf die Struktur des Programms eingegangen. Außerdem wird die Idee hinter dem Schwarmalgorithmus erläutert.

2.1 Programmstruktur

Das Programm ist so aufgebaut, dass eine Klasse alle benötigten Ressourcen speichert und vereint. Benötigte Informationen können so einfach und schnell an Shader [1] und Kernel weitergeleitet werden. Um einen besseren Überblick zu bekommen, ist die Programmstruktur in Abb. 2.1 bildlich dargestellt.

Die Klasse *Renderer* ist dabei erkenntlich das Herzstück. Sie vereint die einzelnen Ressourcen an einem Ort. So können CUDA Funktionen aufgerufen und benötigter Speicher einfach allokiert werden, ohne direkt auf die API zugreifen zu müssen. Dabei läuft zusätzlich eine Fehlerüberprüfung durch. Um nicht direkt auf die OpenGL API zuzugreifen, existieren einige Hilfsklassen, um vermeintlich unerwartete Fehler zu vermeiden. So kann sowohl ein VBO, als auch ein Vertex Array Object (VAO) schnell und einfach erstellt und modifiziert werden. Des Weiteren kümmert sich der *Renderer* um das Vereinen von CUDA und OpenGL. Mehr dazu in Kapitel 3. Die angelegten Ressourcen werden währen dem Updaten der Informationen an den Kernel weitergeleitet und dort neu berechnet. Diese Daten stehen dem Shader später zum Rendern zur Verfügung.

[1] Software-Module zum berechnen von Render-Effekten (z.B. Schattierungen, Farbe)

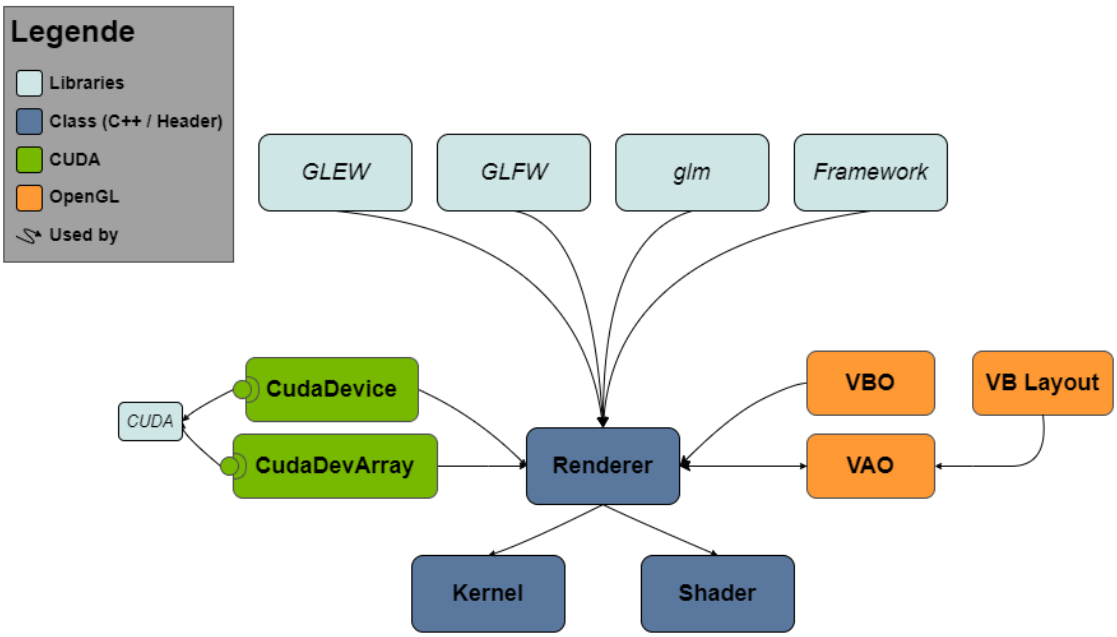


Abbildung 2.1: Programm Struktur

2.2 Schwarm Algorithmus Idee

Das Schwarmverhalten wird nahezu komplett auf der GPU berechnet. Vom Host-System werden im wesentlichen nur die Position des Schwarmzentrums und des Hais berechnet. Das Schwarmzentrum ist ein Punkt, der nicht gerendert wird, unabhängig von der Position der Fische berechnet wird und dem sich die Fische versuchen anzunähern. Es bewegt sich auf einer vorgegebenen Bahn von Wegpunkt zu Wegpunkt, wobei beim Erreichen des letzten Wegpunkts als nächstes wieder der Erste angesteuert wird. Der Hai wird ebenso unabhängig von den Fischen berechnet, jedoch als großer hellgrauer Punkt angezeigt. Die Fische versuchen vor diesem wegzuschwimmen, wenn er ihnen zu nahe kommt. Dieser Fluchttrieb ist stärker gewichtet, als der Drang der Fische sich dem Schwarmzentrum zu nähern, wodurch ein geschickter Hai einzelne Fische kurzzeitig aus dem Schwarm treiben kann. Sobald ein Fisch in Reichweite des Hais ist wird er gefressen und verschwindet.

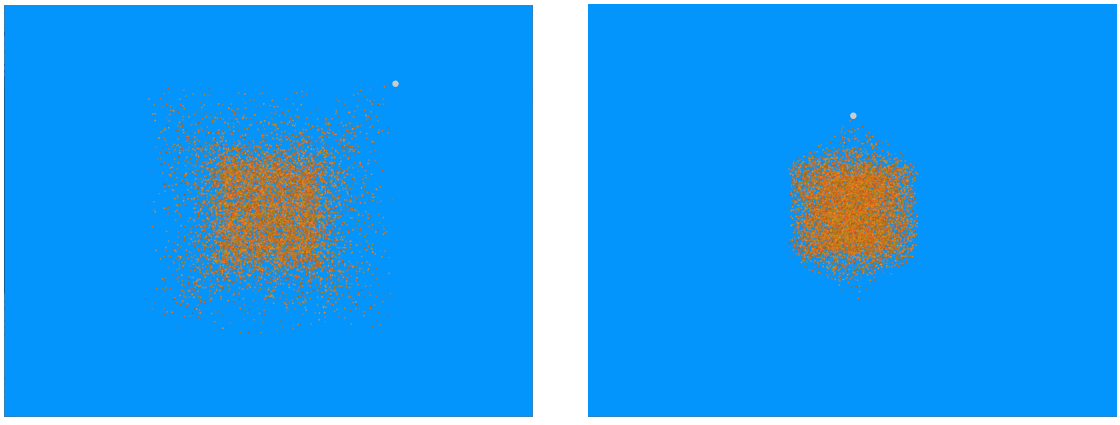
Neben diesen beiden äußeren Einflüssen wirken die Fische auch aufeinander ein, indem sie versuchen einen bestimmten Mindestabstand zueinander einzuhalten. Da dieser Drang stärker gewichtet ist als der des Annäherns an das Schwarmzentrum, entsteht eine wuselnde Sphäre an Fischen um das Schwarmzentrum herum. Die Szene wirkt durch abrupte Richtungswechsel sowie das Jagen des Hais besonders lebendig.

3

Implementierung

3.1 Erstellung der Partikel

Um die Partikel rendern zu lassen, benötigt die OpenGL Render-Pipeline Positionen und Farben. Dafür schreibt der Host zufällige Werte in einem festgelegten Intervall in ein Array, welches auf dem Host-System liegt. Die Farbwerte liegen in einem zweiten Array ebenso auf dem Host. Da die Daten nun zu rendern sind, bzw. auch zu modifizieren sind, müssen diese vom Host auf die GPU mittels zwei erstellten VBOs geschrieben werden. Es sind zwei VBOs vonnöten, da die Farbe ab dem Zeitpunkt der ersten Wertzuweisung konstant bleibt, die Position sich jedoch verändert. So findet das Positions-Array nicht nur in der Render-Pipeline Anwendung, sondern auch im CUDA-Kernel. Das Farb-Array hat nur Anwendung in der Render-Pipeline. In der Abb. 3.1 sind die Ergebnisse eines Random Spawns zu sehen.



(a) Frontale Ansicht.

(b) Rechte Ansicht.

Abbildung 3.1: Random Spawn von 10000 Fischen als Würfel.

3.2 Verwendung von OpenGL in CUDA

Wie schon im vorherigen Abschnitt erwähnt, finden die Positionen auch im CUDA-Kernel Anwendung. Um die veränderten Werte später zu sehen, müssen diese nun neu gerendert werden. Dafür muss der VBO neue Daten bekommen. Eine Möglichkeit die Daten an den VBO zu übergeben wäre diese neu in den VBO zu kopieren, was massive Performance Einbrüche nach sich ziehen würde. Da die Daten schon auf der GPU liegen, können sie auch direkt auf der GPU verändert werden. Um dies zu ermöglichen, muss der zuvor erstellte VBO mit CUDA registriert werden, was den Vorteil gibt, den VBO direkt im Kernel zu verändern. Um Zugriff auf den VBO zu bekommen, kann dieser mit CUDA-Funktionen gemapped werden [Noa]. Algorithmus 3.1 soll die Prozedur etwas veranschaulichen. Alle CUDA-Funktionen zur Verwendung von VBOs liegen in der *CudaDevice*-Klasse.

Algorithmus 3.1: VBO Verwendung

```
1  begin:
2  device.mapResource()
3  vboPtr ← device.getMappedPointer()
4  modify(vboPtr)
5  device.unmapResource()
6  end
```

3.3 Schwarm Algorithmus Implementierung

Das Verhalten der Fische wird in einem einzelnen Kernel berechnet, in dem ein selbst erdachter Algorithmus läuft und ist in folgende sequenzielle Teilaufgaben gegliedert:
Von Hai gefressen werden → Hai ausweichen → nächsten Fisch finden → Abstand zu nächstem Fisch Halten → zu Schwarm zurückkehren → langsamer werden
Das Schwarmverhalten ist also Teil des gleichen Kernels, in dem auch die Reaktion auf den Hai berechnet wird. Diese wird aber extra im Kapitel 3.4 behandelt. An dieser Stelle wird nur das Verhalten der Fische erklärt. Dem Kernel wird folgende Liste an Argumenten übergeben:

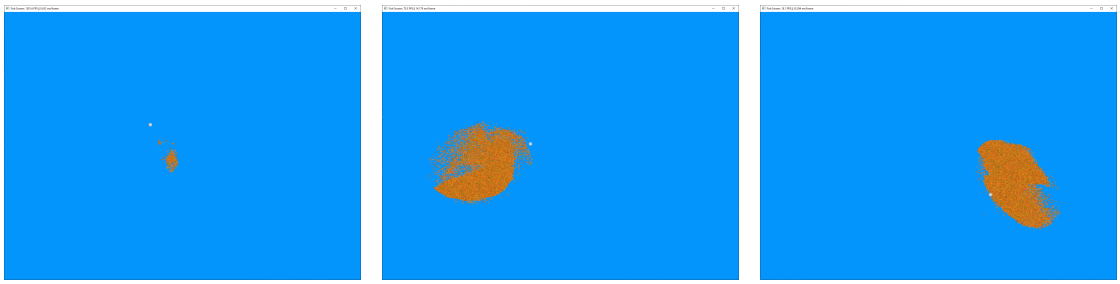
- Pointer auf Array mit Positionen aller Fische
- Pointer auf Array mit Statuswerten aller Fische, welche aus einem Geschwindigkeitsvektor sowie einem zufälligen konstanten Wert pro Fisch bestehen
- Anzahl an Fischen
- Maximalgeschwindigkeit der Fische, die von einer globalen Geschwindigkeitskonstanten abhängt, an der sich auch Schwarmzentrum und Hai orientieren

- Position des Schwarmzentrums
- Position des Hais

Innerhalb des Kernels wird zunächst berechnet für welchen Fisch dieser Thread zuständig ist. Das ist aufgrund der Berechnung des Grids relativ einfach über folgende Formel möglich:

$$Fischindex = Blockindex \cdot Blockdimesion + Threadindex$$

Im nächsten Schritt wird der Fisch ausfindig gemacht, der diesem am nächsten ist, also den kleinsten euklidischen Abstand hat. Das geschieht, indem in einer Schleife über alle Fische iteriert wird, was von allem was im Kernel geschieht mit Abstand den stärksten negativen Einfluss auf die Performance hat. Eine deutliche Verbesserung wäre zu erwarten, wenn diese Berechnung in einen eigenen Kernel ausgelagert und über Reduktion der nächste Fisch berechnet würde. Das sollte in diesem Projekt eigentlich auch so umgesetzt werden, aus Zeitgründen war es aber nicht mehr möglich. Ohne dieses Feature sind problemlos mehrere Millionen Partikel möglich, so dauert die Berechnung eines Frames bei 1000 Fischen bereits ca. 33 ms, was zu etwa 30 Frames per Second (FPS) führt. Siehe dazu Abb. 3.2 und 3.3.



(a) 4000 Fische → 0,6ms/frame (b) 1.000.000 Fische → 14ms/frame (c) 5.000.000 Fische → 55ms/frame

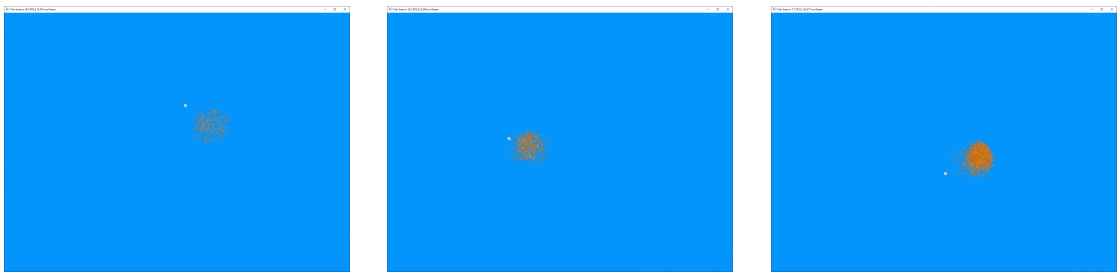
Abbildung 3.2: Simulation läuft mit vielen Fischen, ohne dass diese zueinander Abstand halten.

Falls die Distanz zum nächsten Fisch unterhalb eines festgelegten Werts liegt, wird der Geschwindigkeitsvektor V wie folgt in die entgegengesetzte Richtung beeinflusst:

$$V = V - nn \cdot v\_max \cdot \vec{a}\_max \cdot 0.7$$

Dabei ist nn der normierte Distanzvektor zum nächsten Fisch, v_max die erwähnte Maximalgeschwindigkeit multipliziert mit dem Zufallswert und $\vec{a}\_max$ die maximale Beschleunigung. Der Faktor 0.7 dient der Gewichtung im Vergleich zu anderen Einflüssen auf das Verhalten wie vor dem Hai zu fliehen oder zum Schwarm zurück zu kehren. Jede Richtungsänderung wird nach diesem Schema berechnet, wodurch eine intuitive

Interpretation dieser Gewichtung möglich ist: Das Abstand halten hat 70% des maximal möglichen Einflusses auf die Richtung und ist damit relativ hoch gewichtet. Außerdem wird $\vec{a}\_max$ halbiert, um den Einfluss der folgenden Richtungsänderungen abzuschwächen.



(a) 4000 Fische → 17ms/frame (b) 1.000.000 Fische → 33ms/frame (c) 5.000.000 Fische → 130ms/frame

Abbildung 3.3: Simulation läuft mit deutlich weniger Fischen, wenn diese zueinander Abstand halten.

Falls der Abstand des Fisches zum Schwarmzentrum zu groß ist, wird wie oben der Geschwindigkeitsvektor neu berechnet, nur mit dem normierten Vektor zum Schwarmzentrum sc statt nn, einer anderen Gewichtung und einer Addition statt Subtraktion, da der Fisch sich diesmal dem Ziel nähern und nicht vor ihm weg schwimmen soll.

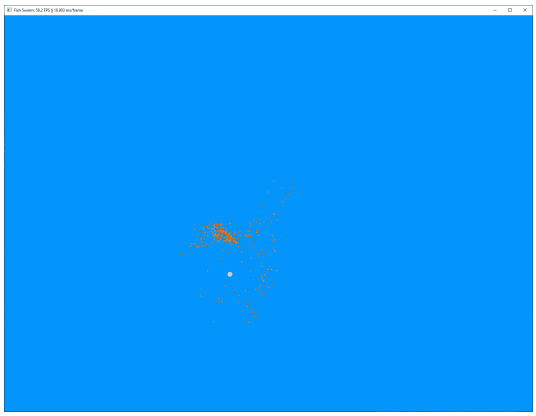
$$V = V + sc \cdot v\_max \cdot \vec{a}\_max \cdot 0.4$$

Die Gewichtungen werden bewusst nicht als Variable angelegt, da sie einmal richtig eingestellt zu sinnvollem Verhalten führen und es nicht sinnvoll ist diese danach noch zu ändern. Im Gegensatz dazu sind Werte wie der Abstand, den die Fische zueinander halten am Anfang der Datei definiert, um sie komfortabel ändern zu können. Einige Unterschiedliche Einstellungen sind in Abb. 3.4 zu sehen.

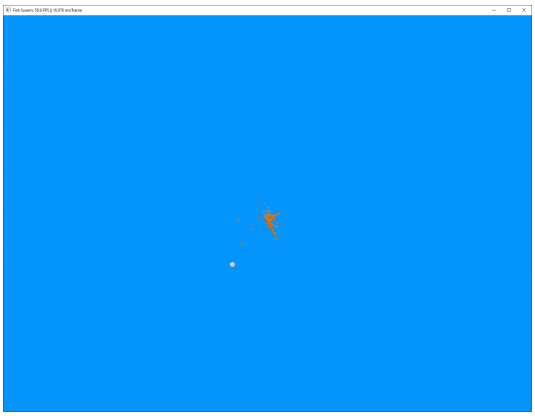
Sollte der Fisch schneller sein als 75% seiner maximalen Geschwindigkeit wird er um 4% langsamer. Das hat den Effekt, dass die Fische nicht immer so schnell schwimmen, wie sie können, sondern dies nur tun, wenn sie beispielsweise vor dem Hai fliehen. Da dies die letzte Änderung am Geschwindigkeitsvektor ist, wird dieser anschließend auf die Position addiert und die beiden Werte werden in die entsprechenden Arrays geschrieben.

3.3.1 Berechnung des Grids

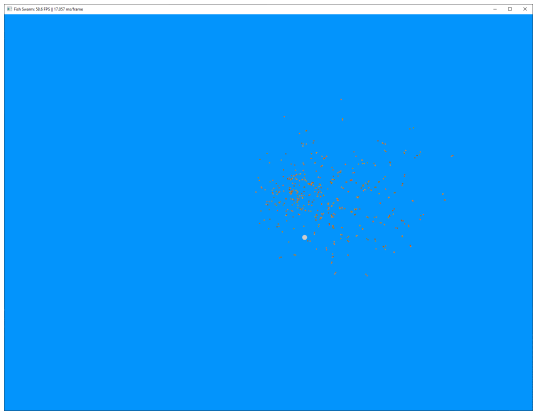
Auf Grundlage der Anzahl an Fischen wird automatisch eine Aufteilung in eindimensionale Blöcke und Threads berechnet. Dies geschieht mit Hilfe zweier Funktionen, die aus den CUDA Samples entnommen sind: `computeGridSize` und `iDivUp` [Shi21].



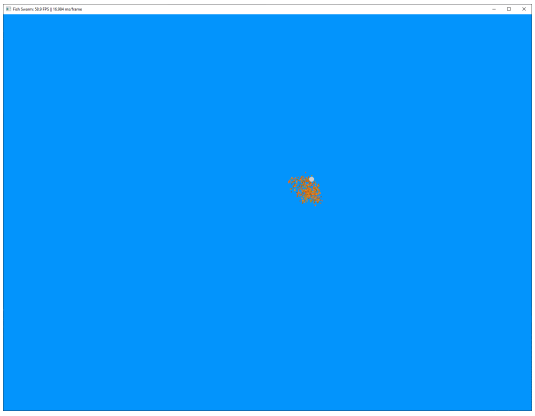
(a) Fische weichen Hai sehr weit aus, wodurch Schwarm auseinander getrieben wird.



(b) Fische halten nur wenig Abstand zueinander.



(c) Fische bilden einen Schwarm mit geringer Dichte, da erlaubte Distanz zum Zentrum hoch eingestellt ist.



(d) Fische bilden einen Schwarm mit hoher Dichte, da erlaubte Distanz zum Zentrum niedrig eingestellt ist.

Abbildung 3.4: Verschiedene Einstellungen von Parametern.

Diese berechnen mit einer gegebenen Blockgröße die Anzahl an Blöcken, die nötig ist, um alle Partikel berechnen zu können. Um die optimale Auslastung der GPU zu erreichen, ist ein naiver Ansatz die maximale Anzahl an Threads pro Block zu verwenden, die auf der GPU möglich ist. Dadurch soll der maximale Grad an Parallelisierung und Auslastung erreicht werden. Allerdings resultiert dieser Ansatz zumeist und vor allem bei rechenintensiven Kerneln nicht in der maximal möglichen Performance - so auch in diesem Programm. In einer Präsentation von Nvidia [Vol] werden die Gründe dafür ausgeführt.

Der dort beschriebene Effekt tritt auch in dieser Schwarm Simulation auf. Die beste Performance wird bei einer Blockgröße von 128 bzw. 256 erreicht. Dieser Ergebnisse stammen von der Ausführung auf einer GTX 3060ti, auf einer GTX 1080 kann das

gleiche Verhalten beobachtet werden. In diesem Projekt ist eine Blockgröße von 256 definiert, da bei vielen Partikeln so vermutlich eine bessere Auslastung möglich wäre.

3.4 Implementierung eines Hais

Der Hai wird als großer grauer Punkt dargestellt. Das wird erreicht, indem eigene Vertex Buffer für ihn angelegt werden. Die Position des Hais wird im Gegensatz zu den Fischen auf der Central Processing Unit (CPU) berechnet, da es nur einen Hai geben kann und eine Berechnung auf der GPU daher keinen Sinn ergibt. Er teilt sich mit den Fischen die Eigenschaft sich am Schwarmzentrum zu orientieren, allerdings schwimmt er ohne sonstige externe Einflüsse darauf zu. Der Hai macht also nicht gezielte Jagt auf einzelne Fische, was auch nicht zielführend wäre, da sich die Fische in einem Schwarm befinden, um sich vor genau dieser Art von Angriffen zu schützen.

3.4.1 Schwarmverfolgung

Der Hai kann sich in zwei Zuständen befinden. Entweder ist seine Distanz vom Schwarmzentrum größer oder kleiner als ein definierter Wert. Ist er weiter weg, so bewegt er sich wieder auf den Schwarm zu, ist er näher dran, schwimmt er geradeaus, bis er wieder weiter weg ist. Dabei erhöht er seine Geschwindigkeit, um seine Chancen auf einen Fang zu erhöhen. Es handelt sich also um einen sehr simplen Algorithmus, der vermeintlich Verbesserungspotential hat, indem ein dritter Zustand, zum Umkreisen des Schwarms, eingeführt wird. Dadurch, dass der Hai sich nicht abrupt umdreht und auf den Schwarm zu schwimmt, sondern dies in einer flüssigen Bewegung passiert und es nur einen Schwellwert für die Distanz gibt, kommt es auch so schon ab und zu zu einem Umkreisen. Nämlich dann, wenn der Hai sich noch nicht genau auf das Schwarmzentrum ausgerichtet hat, aber schon wieder nah genug ist, dass er geradeaus schwimmt.

3.4.2 Fressen eines Fisches

Die Position des Hais wird dem Kernel wie oben beschrieben als Parameter übergeben. Die Berechnung, ob ein Fisch gefressen wurde geschieht auf der GPU, da so in jedem Thread die Entfernung zum Hai effizient und parallel berechnet werden kann. Sollte sich ein Fisch in Bissweite des Hais befinden, so wird der Zufallswert des Fischs, der sonst immer positiv sein muss auf -1 gesetzt, was ihn als gefressen markiert. Dieser Wert wird vom Vertex Shader auf Negativität geprüft und ein illegaler Farbwert an den Fragment Shader weitergegeben. Das erkennt dieser und stellt den entsprechenden Partikel nicht dar. Der Partikel ist also weiterhin vorhanden, er bewegt sich allerdings

nicht mehr und wird nicht mehr dargestellt. Eine mögliche Optimierung wäre den Partikel komplett zu entfernen, allerdings werden mit den Standardparametern nur sehr vereinzelt Fische gefressen, wodurch diese Optimierung nur wenig bringen würde oder die Simulation sehr lange laufen müsste, um einen Effekt erkennbar zu machen. Daher ist diese Optimierung nicht implementiert.

4

Auswertung

4.1 Auswertung mit unterschiedlich vielen Partikeln

Durch die automatische Berechnung der Block- und Thread-Größe existieren nur noch zwei Parameter, die Einfluss auf den Kernel nehmen: die Standard-Blockgröße und die Anzahl der Partikel.

Um zu überprüfen, welche Blockgröße am besten für den Kernel geeignet ist, wurden einige unterschiedliche Größen mit unterschiedlich vielen Partikeln getestet. In Tabelle 4.1 sind die Testergebnisse mit jeweils 1000 und 5000 Partikeln zu betrachten. So scheint auf den ersten Blick die zuvor getroffen Aussage, dass der Best-Case zwischen 128 und 256 Blöcken liegt, falsch zu sein. Es ist gut zu erkennen, dass je größer die Anzahl der Blöcke gewählt wird, desto langsamer der Kernel bei der Berechnung wird. Außerdem ist eindeutig zu erkennen, dass die Anzahl der Partikel eine sehr starke Auswirkung auf die Berechnungszeit haben. Das liegt an der Schleife im Kernel, die jeden Partikel mit allen anderen vergleicht. So kommt ein Kernel auf eine Laufzeit von $O(n)$. Dies auf die Anzahl der Kernel hochgerechnet ergibt das eine Laufzeit von $O(n^2)$.

Block Size	Zeit [ms] (1000 Partikel)	Zeit [ms] (5000 Partikel)
32	37,79	193,49
64	38,00	196,73
128	38,65	207,29
256	42,12	208,61
512	64,49	321,81
1024	128,12	636,67

Tabelle 4.1: Berechnungsdauer

Zieht man den Wert der theoretisch möglichen und die tatsächlich erreichte Dichte hinzu, so ist zu erkennen, dass eine Blockgröße von 1024 den höchsten Prozentsatz der tatsächlichen Dichte erreicht. Bei einer Blockgröße von 32 die niedrigste.

Grid Size	Block Size	Threads	Theoretische Dichte [%]	Erreichte Dichte [%]
157	32	5024	33,33	8,61
79	64	5065	66,67	8,67
40	128	5120	83,33	8,80
20	256	5120	83,33	16,63
10	512	5120	66,67	15,92
5	1024	5120	66,67	66,36

Tabelle 4.2: Dichte mit unterschiedlichen Blockgrößen

Um nun das Optimum aus beiden zu bekommen sollte eine Blockgröße von 128 oder 256 gewählt werden. Dort existieren keine großen Zeitunterschiede in der Berechnung und der Prozentsatz der theoretisch maximal erreichbaren Dichte ist dort am höchsten.

5

Fazit

5.1 Rückblick

Rückblickend ist festzuhalten, dass einige Verbesserungsmöglichkeiten bestehen. So erleidet der Kernel massive Performance-Einbußen durch den Versuch Abstand zwischen den einzelnen Partikeln zu halten. Dies liegt an der Schleife im Kernel. Besser lief jedoch das Verschlingen von Fischen durch den Hai. Ein weiterer positiver Aspekt ist die enge Zusammenarbeit zwischen CUDA und OpenGL, die wir erreichen konnten. Die gute Strukturierung des Programms macht das Erweitern schnell und leicht.

Ein großes Problem war dennoch die Zeit. Durch mehrere Neustarts des Projektes konnten einige Dinge nicht zu unserer Zufriedenheit implementiert werden. Dazu zählt auch die zuvor erwähnte Schleife im Kernel, die wir gerne komplett vermieden hätten. So hat sich auch die Anzahl der maximalen Partikel leider stark verringert. Dennoch konnten wir viele positive Einblicke gewinnen, wodurch das Thema *Parallelisierung* in späteren Projekten durchaus von uns mit der GPU gelöst werden kann.

5.2 Ausblick

Durch das Projekt konnten wir viel über die GPU, das Parallelisieren, aber auch die gemeinsame korrekte Verwendung von CPU und GPU lernen. Das Projekt hat jedoch ein paar Verbesserungsmöglichkeiten. Einige Punkte sind:

- Schleifenvermeidung durch einen feingranularen Kernel
- Verwendung des Reduction-Pattern für Abstandhalten
- Schattierung zum Shader hinzufügen

- Texturen für Fische und Hai
- Wasserfilter erstellen
- Physikalisch nahe Strömungen implementieren

Abschließend kann gesagt werden, dass wir viel durch dieses Projekt lernen konnten. Wir haben im Verlauf einige Fehler gemacht und wissen nun wie wir diese vermeiden können. So haben wir auf dem bitteren Weg gelernt, wie es nicht zu machen ist. Dennoch kam aus unserer Sicht ein gelungen Endergebnis heraus!

Literatur

[Fis] *Schule Der Grauen Fische · Kostenloses Stock Foto*. de-DE. URL: https://www.pexels.com/de-de/suche/fischschwarm/ (besucht am 02. 07. 2021).

[Def] *Duden | Schwarm | Rechtschreibung, Bedeutung, Definition, Herkunft*. de. URL: https://www.duden.de/rechtschreibung/Schwarm (besucht am 02. 07. 2021).

[Noa] *NVIDIA/cuda-samples*. original-date: 2018-03-27T17:36:24Z. Juli 2021. URL: https://github.com/NVIDIA/cuda-samples/blob/master/Samples/simpleGL/simpleGL.cu (besucht am 03. 07. 2021).

[Shi21] Koichi Shiraishi. *zchee/cuda-sample*. original-date: 2015-10-06T22:31:02Z. Juni 2021. URL: https://github.com/zchee/cuda-sample/blob/master/5_Simulations/particles/particleSystem_cuda.cu (besucht am 03. 07. 2021).

[Vol] Vasily Volkov. *Better Performance at Lower Occupancy*. URL: https://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf (besucht am 03. 07. 2021).