

Julian Hopp

**Ist es möglich eine realistische Simulation eines, durch Anwender-Interaktion beeinflussbares, Schwarmverhaltens in der Unity-Engine performant zu gestalten?**

GPD415

SAE Hamburg

Abgabe: 14.03.2016

Betreuender Fachlehrer: Fabio Anthony

(Wortanzahl)

# 1.1 Einleitung

"Schwarm: größere Anzahl sich [ungeordnet,] durcheinander wimmelnd zusammen fortbewegender gleichartiger Tiere, Menschen" \* (Duden - Die deutsche Rechtschreibung, von Dudenredaktion)

"Allgemein bezeichnet man einen Schwarm als einen Zusammenschluss von Tieren in großer Zahl, ohne soziale Bindung oder Hierarchie." \*

(<http://felixmetzfacharbeit.de/schwarmverhalten.html>)

" Der wohl wichtigste und bemerkenswerteste Punkt des Schwarmes ist die fehlende Hierarchie. Ein Schwarm besitzt keinen Anführer. Jedes Mitglied des Schwarmes ist absolut gleichrangig." \* (<http://felixmetzfacharbeit.de/schwarmverhalten.html>)

## 1.2 Themen Definition

In dieser Arbeit werden die drei Grundlegenden "Steering Behaviours" begriffe von Craig Reynold beschrieben, "Separation", "Alignment" und "Cohesion", um eine natürliche Animation von Tierschwärmen in Unity zu simulieren. Anschließend wird in der Unity-Engine getestet ob dies Performant abläuft unter Anwender-Interaktion, desweiteren als Impact benannt.

## 1.3 These

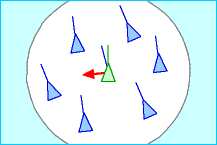
These: Ist es möglich eine realistische Simulation eines, durch Anwender-Interaktion beeinflussbares, Schwarmverhaltens in der Unity-Engine performant zu gestalten?

-(veraltet)In dieser Facharbeit geht es darum, die Anwender-Interaktion dazulegen, welche bei einem Bombardements im 2D Bereiches auftritt und die Reaktion des Schwarmes auf dieses. Der Schwerpunkt liegt darauf, dies Performant zu gestalten.

# 2. 1 Grundlagen des Flockings

Schwarmverhalten beschreibt die Bewegung eines einzelnen Individuums, auch Boid oder autonomer Agent genannt, innerhalb eines Schwarms, sowie das daraus resultierende Verhalten aller Boids als Schwarm. (Buckland, 2005)

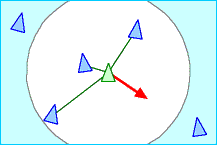
## 2.2 Alignment



(Abbildung 1, Alignment) \*

Das Alignment ist dafür zuständig, dass die Boids sich in eine vorgegebene Richtung und an eine vorgegebene Geschwindigkeit orientieren und somit die Positionsänderung vorgenommen wird.

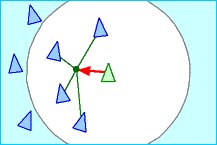
## 2.3 Separation



(Abbildung 2, Separation) \*

Die Separation sorgt dafür, dass die Boids untereinander einen vorgegebenen Abstand einhalten. Dadurch kann die Separation genutzt werden, eine Kollisionserkennung nicht notwendig zu machen, um so Performance zu sparen (siehe Abb.2).

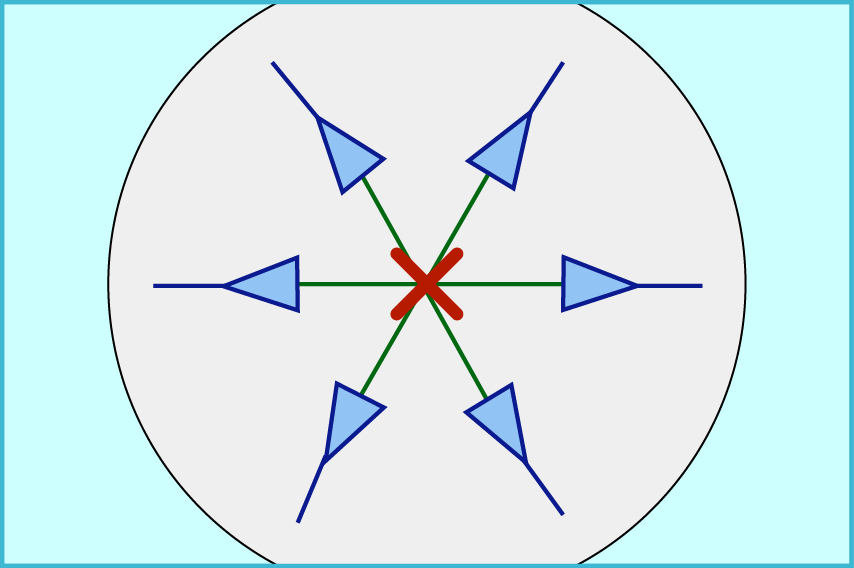
## 2.4 Cohesion



(Abbildung 3, Cohesion) \*

Die Cohesion wird verwendet, um zu verhindern, dass die Boids sich zu weit voneinander entfernen, so dass der Schwarm keine Boids verliert. Die Verbindung von Cohesion und Separation wirkt sich somit auf den Abstand zwischen den Boids aus und die Geschwindigkeit der Distanzveränderung zwischen den Boids im Schwarm (siehe Abb.3).

## 2.5 Impact



(Abbildung 4, selbst erstellt\*)

Die Impact Simulation ist dafür verantwortlich, um Interaktion mit dem Schwarm realistisch zu gestalten. Dies würde unter anderem bei einem Haiangriff auf einen Fischschwarm zu finden sein (siehe Abb.4).

Für die Berechnung des Impacts hilft beispielsweise das Unity interne *NavMesh*, die Unity *Transform.LookAt* Funktion und *Transform.forward* Funktion, auf welche in Kapitel 4.1 noch genauer eingegangen wird.

# 3 Methodik

Bevor der Schwarm implementiert und die Performance getestet werden kann, muss untersucht werden, wie ein Schwarm auf einen Impact reagieren würde. In der Tierwelt ist dies besonders gut zu beobachten. Wenn zum Beispiel in einem Fischschwarm ein Fremdkörper, wie ein Hai oder ein Stein auf der Wasseroberfläche, auftaucht, zersprengt sich der Schwarm sofort. Er braucht dann einige Zeit, bis er sich wieder zusammen fügt.

"All the boids can be moving in one direction at one moment, and then the next moment the tip of the flock formation can turn and the rest of the flock will follow as a wave of turning boids propagates through the flock. Reynolds’ implementation is leaderless in that no one boid actually leads the flock" (AI for Game Developers, von David M Bourg und Glenn Seemann, Kapitel 4) \*

## 3.1 Engine Auswahl

Es wird die Unity Engine (Version 5.3.2) benutzt, da Unity durch diverse Möglichkeiten dem Benutzer das Implementieren erleichtert. Besonders erwähnenswert ist der Profiler, mit dem sich die Performance permanent über ein extra Fenster kontrollieren lässt \*(http://docs.unity3d.com/Manual/ProfilerPhysics.html).

Desweiteren bietet Unity eine zuverlässige und offizielle Dokumentation\* ([http://docs.unity3d.com](http://docs.unity3d.com/))

Ein weiterer wichtiger Punkt ist die kostenfreie Nutzung von Unity 3D für unkommerzielle Projekte.

## 3.2 Programmiersprache

Um die Performance des Schwarms unter Anwender-Interaktion vergleichen und testen zu können, muss diese erst implementiert werden. Durch die eben beschriebene Engine-Auswahl, gibt es nun die Programmiersprachen Java, C#, UnityScript und Boo zum Implementieren des Schwarmverhaltens. Aus diversen Gründen wurde hier die Programmiersprache C# gewählt. Unter anderem weil C# fortlaufend weiterentwickelt wird,

## 3.3 Entwicklungsumgebung

Um die C# Bibliotheken und Anwendungen zu erstellen, wird in der Theorie lediglich ein Compiler und ein Texteditor benötigt. Da eine professionelle Entwicklungsumgebung jedoch das Entwicklungstempo beschleunigt und den Komfort erhöht, wurde aus verschiedenen Gründen die Microsoft Visual Studio Community 2015 Edition gewählt. Diese Entwicklungsumgebung wird ab Unity 5.2 kostenfrei mitgeliefert \* (<http://blogs.unity3d.com/2015/09/08/unity-5-2-easy-access-to-unity-services/>). Desweiteren wird diese fortlaufend aktualisiert.

Zusätzlich wird das Plug-In ReSharper genutzt, um die Naming Convention zu beschleunigen und zu vereinfachen.

## 3.4 Versionsverwaltung

Als professionelles Versionsmanagement mit Cloud Speicherung wurde Github gewählt, um das Schwarmverhalten auf verschiedenen Rechnern nicht nur zu synchronisieren, sondern auch testen zu können. Somit ist sichergestellt, dass jederzeit eine funktionsfähige Version besteht und diese, falls benötigt, nur noch aktualisiert werden muss. Als grafische Benutzeroberfläche zur Bedienung des Versionskontrollsystems wurde SourceTree genutzt.

# 4 Durchführung

Um die Performance des Algorithmus testen zu können, muss dieser Algorithmus vorerst in Unity Implementiert werden.

Hierfür bietet Unity viel Unterstützung, beispielsweise das *NavMesh* und den *NavMesh* Agent. Das *NavMesh* bietet unter anderem die Möglichkeit, eine bereits vorhandene Oberfläche für den *NavMesh* Agent begehbar zu gestalten. Dies ist gerade im 2D Bereich sehr von Vorteil. Die *NavMesh* Agent Klasse besitzt die Funktion *Destination*, die es dem Agent erlaubt, zum Zielobjekt zu laufen.

Auch nimmt Unity dem Benutzer eine Vielzahl an mathematischen Berechnungen ab. Besonders wichtig hierbei sind die Vektor-Klassen, vor allem im 3D Bereich. -\* (Warum ist es vor allem im 3D Bereich wichtig/gut? Wenn möglich -> Zitat)

Mit der Funktion *velocity* wird die Bewegungsgeschwindigkeit ermittelt. Diese Geschwindigkeit kann zur Überprüfung der erlaubten Maximalgeschwindigkeit des Boids genutzt werden. Die *velocity* wird gegen die vorher initialisierte Maximalgeschwindigkeit geprüft und falls überschritten, zurück auf die diese gedrosselt. Zum Drosseln muss die Funktion *Velocity* mathematisch mit der in der Funktion vorhandenen Funktion *normalized* normalisiert werden. Anschließend wird diese wieder, unter Berücksichtigung der *deltaTime*, mit der Maximalgeschwindigkeit multipliziert. Auf die *deltaTime* wird anschließend noch genauer eingegangen.

Noch zu überprüfen im Bezug auf die Geschwindigkeit, ist die ebenfalls vorher initialisierte Mindestgeschwindigkeit. Hierbei wird ganz ähnlich, wie im Absatz zuvor, zuerst überprüft, ob die Mindestgeschwindigkeit unterschritten wurde. Falls dies eintrifft, wird diese auf die Mindestgeschwindigkeit angehoben. Dazu wird wieder die Magnitude durch das mathematische Normalisieren ermittelt und mit der Mindestgeschwindigkeit multipliziert.

Das Unity Keyword *Time* ist ein Interface, welches genutzt wird, um Information bezüglich der Zeit zu erhalten. Die Funktion *deltaTime* dieses Interfaces sollte in Unity in der Regel mit allem multipliziert werden, dass einmal pro Frame etwas Addiert oder Subtrahiert, um die mathematische Berechnung Frameunabhängig macht. "Use this function to make your game frame rate independent." \* (http://docs.unity3d.com/ScriptReference/Time-deltaTime.html)

Gerne genutzt im Bereich der *Time*-Klasse, ist die *Coroutine*. Die *Coroutine* Funktion ermöglicht es, die Ausführung im Skript solange zu verzögern, bis die Bedingung der *Coroutine* erfüllt ist. Diese Bedingung muss als *Yield* der *WaitForSeconds*-Funktion deklariert werden.

Falls die Interaktion des Schwarmes es benötigt, ist es möglich, über die *LookAt* Funktion die Boids in Richtung des Impacts gucken zu lassen. Das ermöglicht es, anschließend mit dem negativen lokalen X Wert der *Transform.forward* Funktion, die Bewegung einer Flucht zu simulieren. Ebenfalls ist es möglich mit der *LookAt* Funktion, die Kamera Dynamisch auf den Schwarmmittelpunkt zu richten.

Ein weiteres nützliches Tool im bezug auf die Performance ist das *Level of Detail*. Hierbei handelt es sich um eine Technik, die es ermöglicht Performance zu sparen indem Tries reduziert werden. Je weiter das Objekt von der aktuell Aktivierten Kamera entfernt ist, um so niedriger wird das *Level of Detail* gesetzt und umso mehr Triangels werden reduziert. (indirektes Zitat: http://docs.unity3d.com/Manual/LevelOfDetail.html)

Auf Abbildung 5 ist das Objekt mit voller Triangel Anzahl zu sehen (LOD 0). Je Höher die LOD, des so niedriger die Triangel Anzahl.



(Abbildung 5, http://docs.unity3d.com/Manual/LevelOfDetail.html \*)



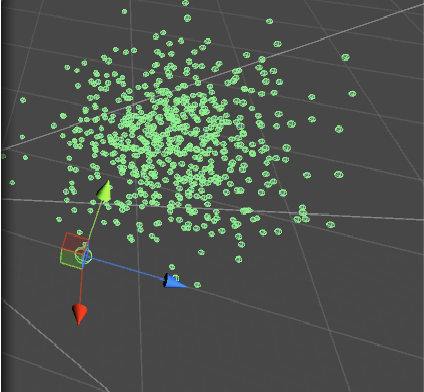
(Abbildung 6, http://docs.unity3d.com/Manual/LevelOfDetail.html \*)

Bei Abbildung 6 wurden viele Triangels eingespart, deren Details bei einer weiter Distanzierten Kamera nichtmehr sichtbar wären.

Desweiteren kann bei einem hohen LOD der Collider deaktiviert werden, was sehr zugute der Performance sein kann.

## 4.1 Praxis Test

Ein Optischer Test zeigt, ob die Implementierung des Schwarm-Algorithmus erfolgreich zum gewünschten Ziel geführt hat.



(Abbildung 7, selbst erstellt, Unity Scene \*)

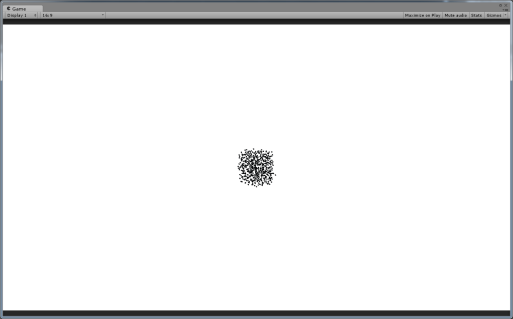
Auf Abbildung 7 zu sehen ist der Versuch der Implementierung eines möglichst realen Schwarms, mit einer Schwarmgröße von 600 Boids im 3D Bereich. Jeder Boid besitzt einen Collider Component, einen Rigidbody Component, einen Mesh Filter Component, einen Mesh Renderer Component, einen Shader Component und einen Transform Component.



(Abbildung 8, selbst erstellt, Profiler \*)

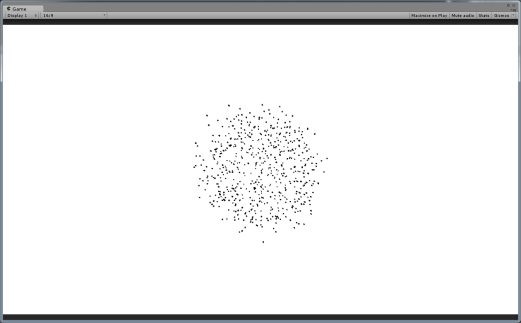
Abbildung 8 zeigt die 600 aktiven Rigidbodys der vorherigen Abbildung und deren Kontakten per Frame. Wie auf dem Diagramm zu sehen ist, gibt es durchschnittlich 10 bis 60 Kontakte unter allen Boids pro Frame. Wenn der Schwarm sich konstant in eine gleichbleibende Richtung bewegt, kommt es zu den zu sehenden durchschnittlichen 10 Kontakten, bei raschen Bewegungsänderungen erhöht sich der Kontakt per Frame auf 60.

Abbildung 4, aus dem Kapitel 2.5 Impact, stellt die Anwender-Interaktion dar. Der Schwarm simuliert auf Kommando ein Szenario, wie beispielsweise einen Fischschwarm, der von einem Hai angegriffen wird. Dieses Kommando kann unter anderem ein Maus-Klick in den Schwarm sein, oder das Auslösen eines Hotkeys.



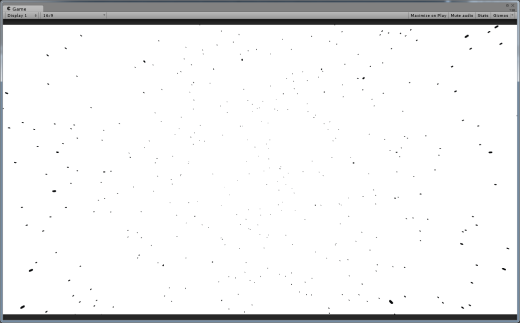
(Abbildung 9, selbst erstellt, Unity Game Scene \*)

Auf Abbildung 9 bleibt der Schwarm noch zusammen, da noch keine Interaktion passiert ist. Dies ist die Ausgangsposition mit 10 bis 60 Kontakten pro Frame.



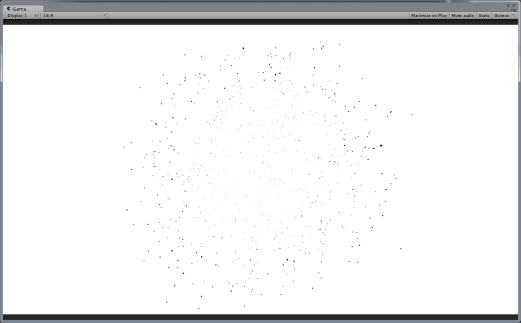
(Abbildung 10, selbst erstellt, Unity Game Scene \*)

Die Anwender-Interaktion passiert: Der Schwarm entfernt sich vom Eintrittsort des Impacts (siehe Abb. 10). Ab hier gibt es keine Boid Kontakte pro Frame, bis hin zu Abbildung 13, da sich die Boids auch untereinander voneinander entfernen.



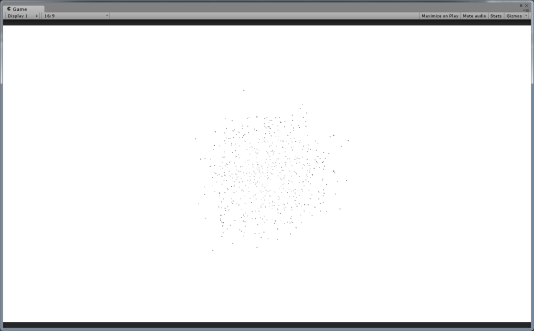
(Abbildung 11, selbst erstellt, Unity Game Scene \*)

Die individuellen Boids haben ihren maximalen Fluchtpunkt erreicht. Dies kann auf verschiedene Weisen passieren, zum Beispiel, wenn die Boids eine Abstandsberechnung zum Impact haben, oder über eine *Coroutine* (siehe Abb.11).



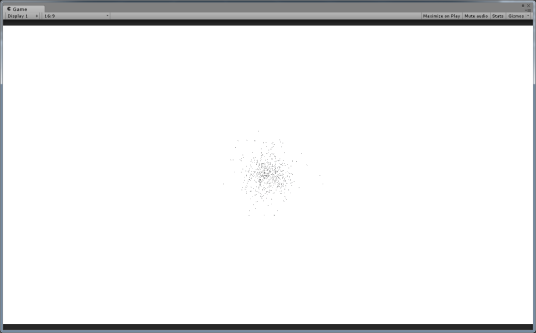
(Abbildung 12, selbst erstellt, Unity Game Scene \*)

Die Boids bewegen sich individuell erneut Richtung Schwarm-Mittelpunkt (siehe Abb. 12).



(Abbildung 13, selbst erstellt, Unity Game Scene \*)

Der Mittelpunkt des Schwarms wurde von den Boids fast erreicht (siehe Abb. 13). Dadurch kommt es zu einer Bewegungsdrosslung, womit die Kontakte zwischen den Boids wieder möglich sind.



(Abbildung 14, selbst erstellt, Unity Game Scene \*)

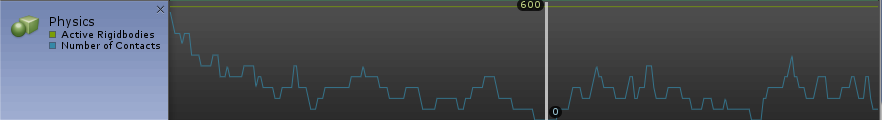
Die Ausgangsposition ist fast erreicht und es pendeln sich wieder die 10 bis 60 Kontakte pro Frame ein. Die Anwender-Interaktion ist somit abgeschlossen (siehe Abb. 14).

-LOD anmachen und beschreiben http://docs.unity3d.com/Manual/LevelOfDetail.html

- Quality settings berücksichtigen http://docs.unity3d.com/Manual/class-QualitySettings.html

-\* weitere Mathe Klassen erwähnen

# 5 Ergebnisse



(Abbildung 15, selbst erstellt, Unity Game Scene \*)

Nach Implementierung des Schwarm-Algorithmus ist zu sehen, dass bei der Anwender-Interaktion die Kollision pro Frame nahezu gleich null ist.

Daher muss Unity intern keine zusätzlichen physikalischen Berechnungen machen, was der Performance zugutekommt.

## 5.1 Auswertung

Bei einer Schwarmgröße von 600 Boids mit den Unity Standard Objekten, gibt es zur Runtime keinerlei Veränderungen. Sichtbar wird dies erst bei höherer Anzahl an Boids. Bei hoher Boid Anzahl kann es beim Projekt Start zu Ladeverzögerungen kommen. Hierbei hilft jedoch das Unity interne *SceneManager.LoadSceneAsync*, oder das *Object Pooling*. Seid Unity 5.3 ersetzt das *SceneManager.LoadSceneAsync* nun das Obsolete *Application.LoadLevelAsync \**(http://docs.unity3d.com/Manual/UpgradeGuide53.html. Mit der *SceneManager.LoadSceneAsync* Funktion ist es möglich, eine Unity Scene asynchron im Hintergrund laden zu lassen.

Beim abrupten Richtungswechsel des Schwarmes ist es möglich, dass es zu vermehrter Kollision kommt. Daher ist es empfehlenswert diesen Richtungswechsel möglichst langsam und geschmeidig zu gestalten.

## 5.2 Tabellen & Diagramme

# 6 Zusammenfassung

## 6.1 Fazit

Inhaltsverzeichnis

[1.1 Einleitung 2](#_Toc445470041)

[1.2 Themen Definition 2](#_Toc445470042)

[1.3 These 2](#_Toc445470043)

[2. 1 Grundlagen des Flockings 2](#_Toc445470044)

[2.2 Alignment 2](#_Toc445470045)

[2.3 Separation 3](#_Toc445470046)

[2.4 Cohesion 3](#_Toc445470047)

[2.5 Impact 3](#_Toc445470048)

[3 Methodik 4](#_Toc445470049)

[3.1 Engine Auswahl 4](#_Toc445470050)

[3.2 Programmiersprache 4](#_Toc445470051)

[3.3 Entwicklungsumgebung 4](#_Toc445470052)

[3.4 Versionsverwaltung 4](#_Toc445470053)

[4 Durchführung 5](#_Toc445470054)

[4.1 Praxis Test 6](#_Toc445470055)

[5 Ergebnisse 9](#_Toc445470056)

[5.1 Auswertung 9](#_Toc445470057)

[5.2 Tabellen & Diagramme 10](#_Toc445470058)

[6 Zusammenfassung 10](#_Toc445470059)

[6.1 Fazit 10](#_Toc445470060)

[Abbildung 1, Abbildung 2, Abbildung 3: http://www.red3d.com/cwr/boids/ by Craig Reynolds](#_Toc445071016)

Literaturverzeichnis:

Abbildungsverzeichnis:

Abbildung 1, Abbildung 2, Abbildung 3: <http://www.red3d.com/cwr/boids/> by [Craig Reynolds](http://www.red3d.com/cwr/index.html)

Abbildung 4 ist selbstgemacht