Technische Hochschule
Brandenburg
University of
Applied Sciences

Fachbereich Informatik und Medien

MASTERARBEIT

Modellierung und Simulation biologischer Bewegungsapparate zur Veranschaulichung von Evolutionseffekten in einem künstlichen Ökosystem mit Unreal Engine

Vorgelegt von: Allan C. Fodi

am: 15.08.2022

zum

Erlangen des akademischen Grades

MASTER OF SCIENCE

(M.Sc.)

Erstbetreuer: Dipl.-Inform. Ingo Boersch

Zweitbetreuer: Prof. Dr.-Ing. habil. Michael Syrjakow

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit zum Thema

Modellierung und Simulation biologischer Bewegungsapparate zur Veranschaulichung von Evolutionseffekten in einem künstlichen Ökosystem mit Unreal Engine

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Brandenburg/Havel, den 15.08.2022

Unterschrift

Zusammenfassung

Die Evolution hat dazu geführt, dass Lebensformen eine Art der Fortbewegung entwickelt haben, die von der Umgebung, in der sie leben, und von ihrer Anpassungsfähigkeit sowie Kontrolle ihres Körpers abhängt. Im Rahmen dieser Masterarbeit wird ein Konzept für ein Bewegungsapparat vorgestellt welches in Unreal Engine umgesetzt wurde. Dabei werden einzelne Kreaturen mit ihren individuellen Bewegungsapparaten in einer Simulationsumgebung der künstlichen Evolution unterzogen. Es wird auf die Komponenten für die Gestaltung der einzelnen Individuen, sowie der Umgebung eingegangen. Die Funktionsweise der implementierten Evolution, sowie die Zusammensetzung genetischer Operatoren werden näher erläutert und durch Experimentdurchläufe demonstriert. Die Ergebnisse dieser werden im Zusammenhang mit zukünftigen Verbesserungsmöglichkeiten kritisch diskutiert.

Abstract

Evolution has led life forms to develop a way of locomotion that depends on the environment in which they live as well as their adaptability and the control of their body. In this master thesis a concept for a locomotion system is presented which has been implemented in Unreal Engine. Individual creatures with their individual locomotor systems are subjected to artificial evolution in a simulation environment. The components for the design of the single individuals, as well as the environment are discussed. The function of the implemented evolution, as well as the composition of genetic operators, will be explained in more detail and demonstrated by experimental runs. The results of these are critically discussed in the context of future possibilities for improvement.

Inhaltsverzeichnis

1 Einleitung

Für Forscher der künstlichen Evolution sind die Mechanismen der Evolution für einige der dringendsten Rechenprobleme in vielen Bereichen gut geeignet [HHCM97]. Viele Berechnungsprobleme erfordern die Suche nach Lösungen in einer riesigen Anzahl von Möglichkeiten. Solche Suchprobleme können oft von einer geeigneten Nutzung der Parallelität profitieren, bei der viele verschiedene Lösungsmöglichkeiten gleichzeitig untersucht werden. Bei vielen Berechnungsproblemen muss sich ein Computerprogramm an seine Umgebung anpassen, z.B. bei der Robotersteuerung, wo die Simulation in einer variablen Umgebung verläuft, um eine möglichst fehlerfreie Steuerung, auch bei äußeren Veränderungen, zu ermöglichen [Rab16a, HG22]. Andere Problemstellungen erfordern Programme, welche neue Ideen zur deren Bewältigung kreieren [AM06].

Die biologische Evolution ist eine faszinierende Inspirationsquelle für die Lösung solcher Probleme. Sie zeigt uns Strategien auf, die es ermöglichen unter einer riesigen Anzahl von Möglichkeiten nach dieser Art von Lösungen zu suchen. Sie kann somit als eine Methode zur Entwicklung innovativer und kreativer Lösungen für komplexe Probleme gesehen werden. Daraus resultierend können deren Mechanismen als Inspiration für computergestützte Lösungskonzepte solcher Problemstellungen dienen. In der Biologie ist diese enorme Menge an Möglichkeiten, die Masse der möglichen genetischen Erbinformationen [PKWT]. Die Lösungen sind dabei die Organismen mit den am besten geeigneten genetischen Eigenschaften, um in ihrer Umgebung zu überleben und sich fortzupflanzen. Die Eignung von biologischen Individuen in deren Umgebung, hängt jedoch von vielen Faktoren ab, zum Beispiel von anatomischen Eigenschaften, mit welchen sie in der Lage sind für ihre Fortpflanzung zu sorgen und mit anderen Individuen zu konkurrieren. Diese Faktoren, welche auch als Fitness-Kriterien bezeichnet werden, ändern sich mit der Entwicklung der Generationen und sorgen damit für sich ständig ändernden Eigenschaften [Koz92]. Die Suche nach entsprechenden Lösungen in Anbetracht dieses ständigen Wandels von Möglichkeiten, lässt sich als Computerprogramm übersetzen.

Mithilfe einer geeigneten technischen Umgebung zur physikalischen Echtzeitsimulation von starren Körpern, ist man in der Lage das Verhalten von starren Körpern nachzuahmen. Damit lassen sich Kreaturen konstruieren, welche abhängig von ihrer Beschaffenheit in der gegebenen Umgebung eine Evolution durchlaufen können. Mit solch einer Simulation lässt sich der Entwicklungsprozess der natürlichen Evolution demonstrieren.

1.1 Motivation und Zielsetzung

Genetische Algorithmen werden von Forschern seit einiger Zeit zur Nachahmung der darwinistischen Evolution verwendet. Auch die Entwicklung von Kreaturen oder Robotern, die in simulierten Welten leben und evolvieren, hat seit einiger Zeit bei vielen Forschern das Interesse geweckt. Es gibt bereits eine Reihe von 2D-simulierten Implementierungen, in welchen Kreaturen durch die Mechanismen der Evolution ihr Fortbewegungsverhalten entwickeln [Ven05]. Einer der wichtigsten Wegbereiter in diesem interdisziplinären Gebiet ist Karl Sims, mit seiner Entwicklung einer 3D-simulierten Umgebung [Sim94], in welcher künstliche Kreaturen eine Reihe von Fähigkeiten, wie Gehen, Springen oder Schwimmen, während der laufenden Evolution entwickeln.

Der Fokus dieser Arbeit ist die Implementierung eines Bewegungsapparates in der 3D-Grafik-Engine Unreal Engine, mit welcher eine realistische Echtzeitsimulation der Evolution realisiert wird. Dabei wird auf folgende Forschungsfrage eingegangen: Wie kann ein Bewegungsapparat konstruiert werden, mit welchem die Grundvoraussetzung zu einer evolutionären Entwicklung in einer physikalischen Simulationsumgebung gegeben ist?

Ziel der Arbeit ist es evolutionäre Ergebnisse mit der implementierten Bewegungsmechanik zu erreichen, diese zu analysieren und die Ergebnisse kritisch zu diskutieren.

1.2 Abgrenzung

In dieser Arbeit wird nicht auf bereits implementierte Aspekte der KipEvo-Simulation eingegangen [IB21]. Die hier erläuterten Grundlagen unterstützen lediglich das Grundverständnis der im Rahmen der Arbeit vorgenommenen Anpassungen und Entwicklungen.

1.3 Aufbau der Arbeit

Die Arbeit ist in 5 Abschnitte eingeteilt.

Kapitel 2 befasst sich mit der grundlegenden Modellierung von Bewegungsmechaniken. Hierbei wird besonders auf die Aspekte eingegangen, welche aus der klassischen Mechanik stammen. Diese sind in 3D-Engines für die Integration einer realistischen Physik relevant. Zusätzlich wird auf verwandte Gebiete und Arbeiten eingegangen, in welchen verschiedene Modellierungen vorgestellt werden.

Kapitel 3 befasst sich mit der grundlegenden Theorie zur künstlichen Evolution und die der neuronalen Netze.

Kapitel 4 befasst sich mit dem Konzept der Swimbots. Dabei wird die Idee näher erläutert und die Funktion und Struktur dieser verdeutlicht.

Kapitel 5 befasst sich mit der Umsetzung des im vierten Kapitel beschriebenen Konzepts. Dabei wird die Entwicklungsumgebung, sowie die Systemarchitektur und die Implementierung der Komponenten verdeutlicht.

Kapitel 6 beinhaltet die Diskussion der Simulationsexperimente zur Veranschaulichung der Evolutionseffekte. Dabei werden Ergebnisse der insgesamt 4 Simulationsläufe analysiert.

Kapitel 7 diskutiert die Ergebnisse der Arbeit, sowie Probleme die aufgetreten sind. Es werden Ansätze erläutert, die in Zukunft implementiert werden können um Probleme zu lösen oder die Simulation vielseitiger zu gestalten.

2 Modellierung von Bewegungsmechaniken

Computergenerierte Kreaturen sind nicht nur interessante Objekte zum Studieren, sondern können in bestimmten Bereichen der Forschung zur Design-Verbesserung von Bewegungssystemen führen [Kri19]. Sie stellen ein Feld mit weitreichenden Anwendungen dar, von der Robotik bis hin zur Simulation von Ökosystemen, in welchen auf eine faszinierende und unterhaltsame Art die evolutionären Prozesse gezeigt werden können. Da die Grenzen zur Modellierung und dem Entwurf virtueller Kreaturen nicht klar festgelegt sind, gibt es viele Ansätze Bewegung in einer räumlichen Umgebung zu simulieren.

2.1 Physikbasierte Modellierung in der Computergrafik

Grundlage der Modellierung von natürlichen Bewegungsmechaniken in einer räumlichen Simulationsumgebung ist ein geeignetes Physikmodell. In der Informatik wird die Umsetzung eines solchen Modells auch als Physik-Engine bezeichnet.

Eine Physik-Engine ist eine Computersoftware, die eine annähernd natürliche Simulation bestimmter physikalischer Systeme wie Starrkörper- (inklusive einer Kollisionserkennung), Weichkörper- und Flüssigkeitsdynamik ermöglicht und in der Computergrafik eingesetzt wird. Die Simulationen erfolgen in Echtzeit und erzeugen damit ein realistisches Abbild physikalischer Naturgesetze. Der Begriff wird auch in Softwaresystemen verwendet, mit welchen physikalische Phänomene oder Hochleistungsrechnungen simuliert werden (wie z.B. partikelbasierte Flüssigkeitsdynamiksimulation [phy]).

Langfristiges Ziel einer physikbasierten Modellierung ist es, Methoden zu entwickeln, um physikalische Systeme von Objekten zu spezifizieren, zu entwerfen und zu kontrollieren. Unter diese Modellierung versteht man eine mathematische Repräsentation eines Objekts, das die physikalischen Eigenschaften, Momente und Energien einem Modell zuordnet, was eine numerische Simulation seines Verhaltens ermöglicht. In der Computergrafik sind dies beispielsweise, gängige Bestandteile der Dynamik (Bewegung durch Kraftauswirkung, Masse und Trägheit) mit starren oder weichen Körpern, sowie durch sogenannte Constraints beschränktem Verhalten.

2.1.1 Softbody Bewegungsdynamik am Beispiel von Würmern

In der Publikation *The Motion Dynamics of Snakes and Worms* [Mil88] befasst sich der Autor Gavin Miller mit der Frage wie biologische Formen für eine natürliche Animation modelliert werden können. In seiner Arbeit beschreibt er ein Masse-Feder Modell in welchem die Fortbewegung von wirbellosen Tieren wie Schlangen und Würmern ermöglicht wird. Dabei unterscheidet Miller zwischen zwei essenziellen Aufbauarten des Körpers.

1. Tiere mit einem starren Skelett (Starrkörperaufbau), bei welchen die einzelnen starren Körperteile durch Gelenke miteinander verbunden sind.
2. Wirbellose Tiere wie Würmer, die sich biegen und dehnen lassen.

Bei der Körperstruktur von Tieren mit starren Körperteilen erfordert es die Berücksichtigung der Dynamik hierarchischer starrer Strukturen. Miller betont, dass die kinematischen und dynamischen Modelle, welche für die Modellierung von wirbellosen Tieren sind, nicht das Problem in Bezug auf Haut und Muskeln behandeln. Außerdem verändern Würmer und Schlangen ihre Form während der Bewegung. Die Bewegung ist außerdem abhängig vom Halt bzw. der Reibung zum Boden. Diese Faktoren machen eine Bewegung mit Softbody-Mechaniken zu einer komplexen Herausforderung.

2.1.2 Begriffe aus der klassischen Mechanik

2.1.2.1 Kinematik und Dynamik

Eine Physik-Engine bietet eine solide und weit verbreitete Nachbildung der Realität in virtuellen Welten. Dabei gibt es aus der traditionellen Mechanik zwei wichtige Teilbereiche, welche essentiell für die Berechnung der Bewegungszustände sind, durch welche eine Animation ermöglicht wird [Rab16b, ISPM20]. Die Kinematik wirkt sich, unabhängig von der Physik (Kraft), auf den Zustand der Figur. Die Dynamik befasst sich mit der Ursache der Bewegung, d.h. mit der Beschreibung der Bewegung von Körpern durch die einwirkenden Kräfte.

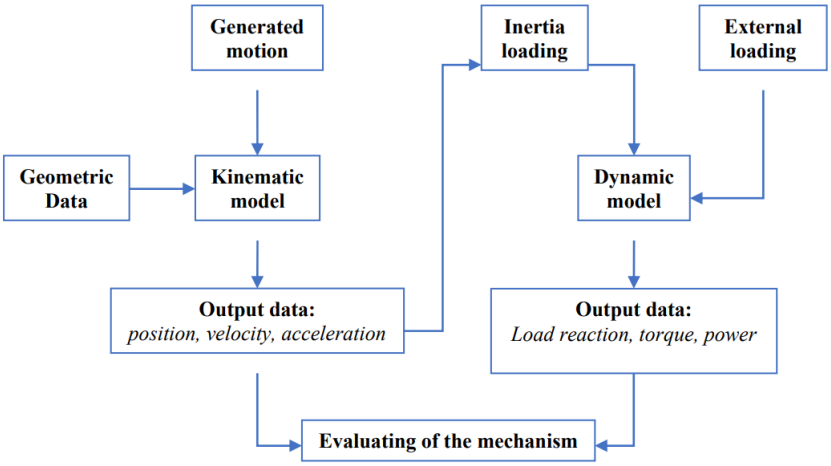


Abbildung 2.1: Visualisierung der Zusammenhänge der Kinematik und Dynamik
So wie sie auch in gängigen Physik-Engines verwendet und berechnet wird. Das kinematische Modell übernimmt Berechnungen der Bewegung ohne Bezug auf Krafteinfluss, während das dynamische Modell die wirkenden Kräfte berücksichtigt. Quelle: [ISPM20]

Abbildung 2.1 verdeutlicht die Zusammenhänge der Kinematik und Dynamik. Dabei beschreiben die geometrischen Daten den zu bewegenden Körper und auch die Kollisionsobjekte, welche den Bewegungszustand beeinflussen. Aus der Trägheitsbelastung und den extern-wirkenden Kräften kann dann die resultierende Bewegung des Objektes ermittelt werden. Das Zusammenspiel beider Teilbereiche beschreibt die in einer physikalischen Umgebung auftretenden Ursachen und den Charakter von Bewegungen. Die newtonschen Gesetze zur Bewegungslehre[VP09] bilden dabei die Grundlage dieser Berechnungen.

2.1.2.2 Inverse Dynamik

Die Inverse Dynamik ist ein inverses Problem. Es beschreibt die Dynamik von Gelenksystemen, also dem Verhalten miteinander verbundener Glieder (wie z.B. bei Robotersystemen). Die Kräfte werden dabei aus der Kinematik der Gelenke abgeleitet, die zu den einzelnen Bewegungen führen. Dabei werden die internen Momente und Kräfte aus Messungen der Bewegung der Glieder und auch die beteiligten externen Kräfte, wie z.B. Widerstände, berechnet.

Es gibt viele Anwendungsmöglichkeiten für die inverse Dynamik. Der traditionelle Anwendungsfall der inversen Dynamik findet man in der Robotik. Hier werden Gelenkkräfte bzw. Drehmomente berechnet, die erforderlich sind, um einen Roboterarm von einer kartesischen Koordinate zu einer anderen zu bewegen. Diese wesentliche Anforderung in der Robotik ist bei einer einfachen Animation nicht wichtig, da dort ein Punkt zu einem anderen bewegt werden kann, ohne physikalische Gesetze (wie z.B. die Trägheit) zu brechen. Bei physikbasierten Simulationen, allerdings, ist dieser Bereich

weiterhin essentiell.

In der Biomechanik wird durch die inverse Dynamik der Drehmoment aller anatomischen Strukturen eines Gelenks erfasst, die notwendig sind, um die beobachteten Bewegungen des Gelenks zu erzeugen. Dazu zählen auch Faktoren wie Muskeln und Bänder. Diese Drehmomente können dann verwendet werden, um die Menge an mechanischer Arbeit zu berechnen, die von diesem Drehmoment geleistet wird. Jedes Drehmoment kann positive Arbeit leisten, um die Geschwindigkeit und die Höhe des Körpers zu erhöhen, oder negative Arbeit leisten, um diese entsprechend zu verringern. Die erforderlichen Bewegungsgleichungen basieren auf der Newtonschen Mechanik.

2.1.2.3 Inverse Kinematik

Die inverse Kinematik ist auch, wie die inverse Dynamik, ein inverses Problem. Anders als bei der inversen Dynamik wird allerdings nach einem statischen Satz von Gelenkwinkeln gefragt, wodurch die Position eines bestimmten Punktes (oder mehrerer Punkte) eines Mehrkörpermodells gegeben ist. Sie wird bei der Synthese menschlicher Bewegungen eingesetzt, insbesondere bei der Entwicklung von Videospielen (z.B. der Animationen)[uni, unr]. Eine weitere Anwendung findet sich in der Robotik, wo die Gelenkwinkel eines Arms anhand der gewünschten Position des Endeffektors (das letzte Element einer kinematischen Kette von Gelenken) berechnet werden müssen [KB06].

2.1.2.4 Starre Körper

In der klassischen Mechanik bezeichnet man als einen starren Körper einen idealisierten Körper, welcher sich nicht verformen lässt. Die Unverformtheit ist dahingehend so definiert, dass die Abstände zwischen zwei Punkten innerhalb des Körpers bei einwirkenden Kräften gleich bleibt; man spricht dabei auch von einer durchgehenden Verteilung der Masse.

2.1.2.5 Beschränkungen

In der Hamilton-Mechanik ist eine Beschränkung eine Beziehung zwischen den Koordinaten und Impulsen, die ohne Verwendung der Bewegungsgleichungen gilt. Wenn die Bewegung eines Körpers durch eine oder mehrere Bedingungen eingeschränkt wird, verringert sich die Zahl der unabhängigen Möglichkeiten, den Körper frei zu bewegen. Die Begrenzungen oder Einschränkungen der Bewegung des Systems werden als Beschränkung (engl. Constraint) bezeichnet.

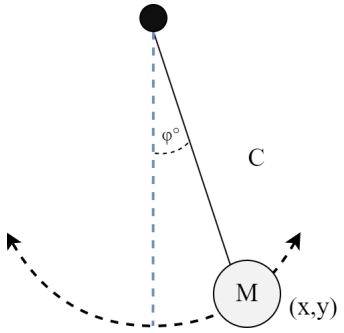


Abbildung 2.2: Einfache Visualisierung einer Beschränkung am Beispiel eines Pendels.

In der vorliegenden Abbildung ist ein Pendel zu sehen. Bei einem einfachen Pendel ist der Radius des Drehwinkels oder die Länge des Befestigungsdrahtes fest vorgegeben. Bei einer Beschränkung sind genau diese beiden Aspekte erforderlich um die beschränkte Bewegung zu berechnen. Der Freiheitsgrad gibt dabei an, welche Achsenabschnitte bei einer Beschränkung berücksichtigt werden. Ist die Befestigung des Pendels somit auf die Bewegung in den (x,y) Abschnitten begrenzt, so muss der z-Abschnitt räumlich nicht betrachtet werden. Generell spricht man von linearen und rotierenden Beschränkungen.

2.2 Verwandte Gebiete und Arbeiten

2.2.1 Evolutionäre Robotik

Die evolutionäre Robotik ähnelt der simulierten Evolution virtueller Kreaturen. Allerdings befasst sich dieser Bereich mit einem evolutionären Prozess, basierend auf einem genetischen Algorithmus, um Steuersysteme zu bewerten. Die Steuersysteme sind durch künstliche Genotypen definiert, die ihrerseits über viele Generationen hinweg miteinander gekreuzt werden. Die fittesten Individuen (d.h. mit effizientesten Steuersystemen) erzeugen am ehesten Nachkommen. Die Fitness wird mithilfe eines Bewertungskriteriums am Verhalten des Roboters gemessen. Die evolutionäre Robotik wird prinzipiell zur Entwicklung von Steuerungssystemen für Roboter eingesetzt. Eher selten angewandt, kann die evolutionäre Robotik aber auch zur Erzeugung von Körperplänen (Modelle für physische Körper) und zur gleichzeitigen Koevolution von Steuersystemen und Körperplänen eingesetzt werden. [LP00]

2.2.2 Evolving Creatures - Karl Sims

Die Arbeit von Karl Sims war wegweisend im Bereich der evolutionären Simulation virtueller Kreaturen. In seiner physikbasierten Umgebung entwickeln seine Kreaturen verschiedene Geh-,

Schwimm- und Sprungverhalten. Die aus verbundenen Blöcken bestehenden Körper, werden durch einfache neuronale Netzwerke gesteuert, welche auf Grundlage der Wahrnehmungen der Gliedmaßen unterschiedliche Drehmomente an den Gelenken erzeugen, wodurch realistische Bewegungen ermöglicht werden. Sims hat als Fitnessfaktor die Bodengeschwindigkeit der Kreaturen ausgewählt. Durch den Prozess der Evolution haben sich interessante Gestalten entwickelt. Wenn sie simuliert wurden, fielen einige Kreaturen um, wobei sie ihre anfängliche potenzielle Energie nutzten, um eine hohe Geschwindigkeit zu erreichen. Einige vollführten sogar Purzelbäume, um ihre horizontale Geschwindigkeit zu erhöhen. Die Körperstruktur, bestehend aus verbundenen Körperblöcken wurde als Inspiration für die strukturelle Körperanatomie der Swimbots genutzt.

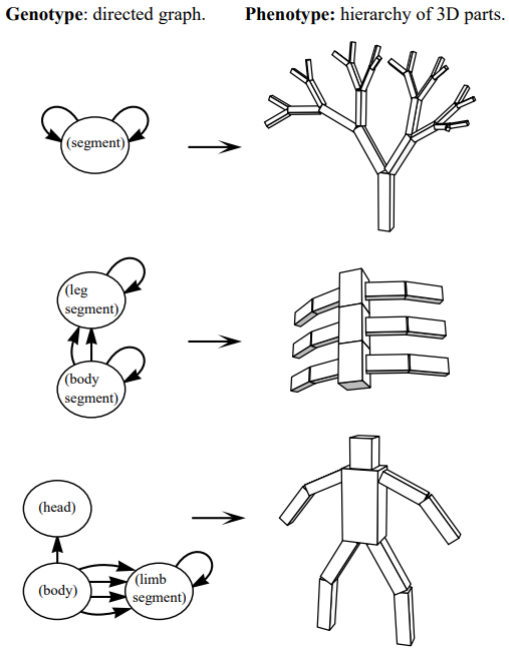


Abbildung 2.3: Morphologie von Sims Kreaturen in *Evolving Creatures* Quelle: [Sim94]

3 Theorie

3.1 Künstliche Evolution

Das Forschungsgebiet der künstliche Evolution befasst sich mit künstlichen Lebenssystemen, welche inspiriert durch die natürliche Evolution, anhand mathematischer Modelle simuliert werden [SK92]. Obwohl das künstliche Leben grundsätzlich sowohl auf die Ursprünge der Biologie als auch auf ihre Zukunft ausgerichtet ist, erfordern der Umfang und Komplexität dieses Forschungsbereiches eine Verknüpfung vieler Disziplinen. Dieses breit angelegte Forschungsgebiet umfasst die Möglichkeit, lebensechtes Verhalten in ungewohnten Umgebungen zu entdecken und neue und ungewohnte Lebensformen zu schaffen.

3.1.1 Evolutionäre Algorithmen

Die Evolution ist die Veränderung von Merkmalen einer Art über mehrere Generationen hinweg und beruht auf dem Prozess der natürlichen Selektion. Sie beschreibt das historische Auftreten von Veränderungen einer Population von Lebewesen. Die natürliche Selektion hingegen beschreibt den Ausleseprozess der am besten geeigneter Individuen in einer spezifizierten Umwelt. Evolution ist daher nur im Kontext ganzer Populationen sinnvoll - für Individuen ist sie nicht definiert.

Die Evolution bietet unzählige Beispiele für erstaunlich komplexe Lösungen für die Herausforderungen des Lebens [GG01]. Sie ist nicht nur auf die biologische Welt beschränkt sondern kann in jeder Umgebung stattfinden, wo die genetischen Operatoren der Replikation, Variation oder Selektion aufeinandertreffen.

Folglich, kann eine Evolution künstlich als Computerprogramm simuliert werden, um entweder evolutionäre Experimente durchzuführen oder technische Herausforderungen durch gerichtete Evolution zu lösen. Ähnlich wie bei der biologischen Evolution bringen künstliche Experimente oft überraschende und kreative Ergebnisse hervor [WH22, PJ08a, LFLL$^+$21].

Das Gebiet der künstlichen Evolution hat seine Anfänge in den 1960er Jahren mit den Arbeiten von Ingo Rechenberg (Evolutionsstrategien, kurz ES - [DJFS97]), John Holland (genetische Algorithmen,

kurz GA - [Hol94]) und L.J. Fogel (evolutionäre Programmierung, kurz EP - [DJFS97]). Holland war an der Anpassung in natürlichen Systemen interessiert, Fogel und Rechenberg mehr an der Nutzung von evolutionären Algorithmen zur Optimierung. Aus algorithmischer Sicht gibt es zwar wichtige Unterschiede zwischen diesen drei Verfahrenstypen, welche von Schwefel und Bäck in ihrer Publikation zur evolutionären Optimierung verdeutlicht worden sind. [BS93].

3.1.1.1 Exkurs Blind Watchmaker

The Blind Watchmaker ist ein Algorithmus von Richard Dawkins aus seinem im Jahr 1986 stammendem Buch [Daw03],in dem eine exemplarische Umsetzung zur Theorie der Evolution durch natürliche Selektion beschrieben wird.

Der Ansatz von Dawkins, beschreibt eine einfache und lehrreiche künstliche Evolution sogenannter biomorpher Strukturen. Diese werden von Dawkins als zweidimensionaler Gebilde konzipiert, bestehend aus einer Verkettung von Strichen. Die Morphologie dieser, wird durch künstliche Mutation verändert [Daw03].

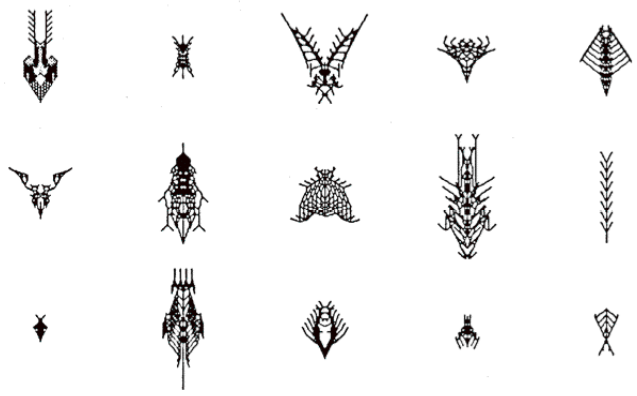


Abbildung 3.1: Biomorphe Gebilde aus Dawkins Blind Watchmaker.
Quelle: [Bar07]

Die künstliche Evolution geht immer von einem Genom aus, d. h. das zu lösende Problem muss in einem Genotyp kodiert sein. Das *Problem* in Dawkins Umsetzung ist lediglich die Suche einer ansehnlichen biomorphen Struktur. Die Selektion findet dabei nämlich durch den Nutzer statt, der nach jeder Generation entscheidet welche der Genotypen selektiert werden. Es handelt sich bei der von Dawkins konzipierten Evolution um eine zielgerichtete Evolution. Die Länge sowie Winkel der einzelnen Striche für die Darstellung der Struktur, sind im Genotyp numerisch kodiert (4 Gleitkommazahlen jeweils für die Länge und Winkel der Striche) und werden durch den Mutationsprozess verändert. Der nach jeder Generation weitergegebene mutierte Genotyp wird in den Phänotyp, also der sichtbaren Struktur mit den Linien, umgewandelt. Nach Durchlauf einiger Generationen ergeben

sich vielseitige Strukturen, wie in der Abbildung 3.1 sichtbar.

3.1.2 Zyklus Künstliche Evolution

Die künstliche Evolution wird in der Regel in Form eines evolutionären oder genetischen Algorithmus (GA) umgesetzt. Ein GA ist ein Suchverfahren, das die drei wesentlichen Elemente der darwinistischen Evolution - Replikation, Mutation und Selektion - umsetzt. In der Regel wirkt das Verfahren auf eine Population von Lösungskandidaten ein, die nach einem Fitnesskriterium bewertet werden.

Populationen, oder auch Genpools genannt, entwickeln sich, indem sich die Genfrequenzen ändern. Die Variation in Populationen wird von den im Genpool der Population vorhandenen Genen bestimmt, die durch die Mutation verändert werden. Die genetische Zusammensetzung einer Population kann auch durch zufällige Ereignisse wie eine Gendrift oder die gemeinsame Vererbung von Genen (genetisches Trampen) beeinflusst werden.

Die Evolution funktioniert immer mit Populationen von Individuen. In der Natur sind dies Lebewesen, in der künstlichen Evolution sind es oft Problemlösungen. Jedes Individuum trägt eine Beschreibung einiger seiner Merkmale (Farbe, Körpergröße). Diese Beschreibung wird als Genom bezeichnet. Der Genotyp hingegen beschreibt die Menge aller Genome. Er wird verwendet, um den Unterschied zwischen der genetischen Anlage und dem endgültigen Organismus, dem Phänotyp, auszudrücken. Das Genom besteht aus einer Reihe von Genen, wobei ein Gen lediglich ein Merkmal beschreibt. Die Werte aller Gene eines Individuums sind vorzeitig festgelegt und ändern sich nicht im Laufe der Lebenszeit eines Individuums.

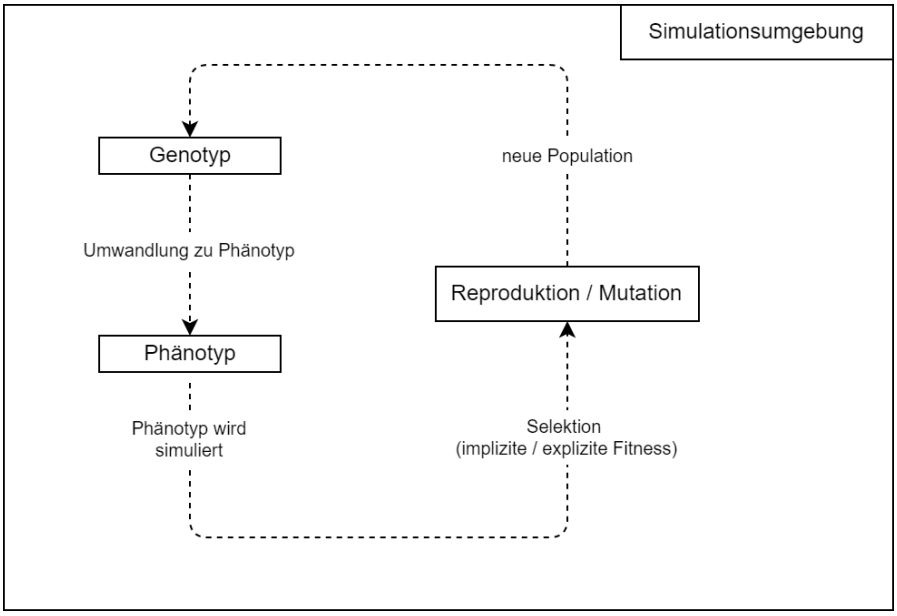


Abbildung 3.2: Grundlegende Veranschaulichung des Evolutionszyklus.
Der Genotyp wird in den Phänotyp umgewandelt. Der Phänotyp wird in der festgelegten Umgebung simuliert und entsprechend der ausgewählten Methode der Selektion unterworfen. Die Phase der Reproduktion erzeugt die selektierten Nachkommen und unterwirft sie einer Mutation. Der Zyklus wiederholt sich.

Die natürliche Selektion wirkt sich auf den Phänotyp bzw. die physischen Merkmale eines Individuums aus. Der Phänotyp wird durch die genetische Beschreibung eines Individuums (Genotyp) und seiner Umwelt bestimmt. Einige Merkmale werden von nur einem einzigen Gen bestimmt, andere Merkmale wiederum werden jedoch durch das Zusammenspiel vieler Gene beeinflusst. Eine Variation in einem der vielen Gene, die zu einem Merkmal beitragen, kann sich nur geringfügig auf den Phänotyp auswirken.

3.1.3 Genetische Algorithmen

Im Gegensatz zu den anderen Ansätzen der evolutionären Algorithmen, war Hollands ursprüngliches Ziel nicht Algorithmen zu entwickeln, um direkt konkretisierte Probleme zu lösen, sondern das Phänomen der Anpassung, wie es in der Natur vorkommt, formal zu untersuchen und Wege zu entwickeln, um die Mechanismen der natürlichen Anpassung in Computersysteme zu übertragen.

Holland stellte in seiner Publikation den genetischen Algorithmus als eine Abstraktion der biologischen Evolution vor und gab einen theoretischen Rahmen für die Anpassung unter genetischen Algorithmen an [Hol94].

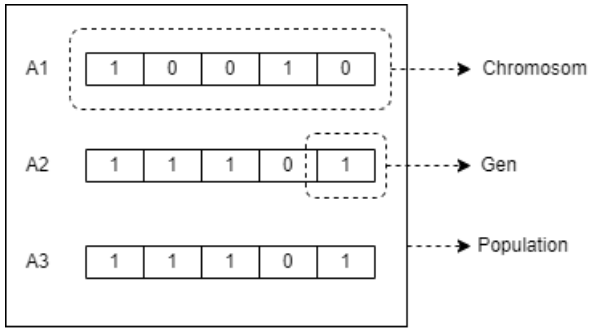


Abbildung 3.3: Darstellung der Begrifflichkeiten am Beispiel einer Population als Bitzeichenfolge kodiert. Angelehnt an [Mal20]

Seine Methode ist es, von einer Population von *Chromosomen* (welche im Computer als Reihe von Bits dargestellt ist) zu einer neuen Population zu gelangen, indem eine Art *natürliche Selektion* zusammen mit den genetischen Operatoren der Kreuzung, bzw. der Rekombination, und der Mutation.

Jedes Chromosom besteht aus *Genen* (als Bits oder Zahlen dargestellt), wobei jedes Gen ein Allel beschreibt. Der Selektionsoperator wählt diejenigen Chromosomen in der Population aus, die sich fortpflanzen dürfen, und im Durchschnitt erzeugen die fitteren Chromosomen mehr Nachkommen als die weniger fitten. Bei der Kreuzung werden Teile von zwei Chromosomen ausgetauscht, was in etwa der biologischen Rekombination zwischen zwei einchromosomalen Organismen entspricht. Die Mutation verändert hierbei durch Zufall die Allel-Werte im Chromosom.

3.1.4 Mutation

Die Mutation ist in der genetischen Programmierung ein Operator, mit welcher die genetische Vielfalt von Generationen einer Population gewährleistet wird. Das Grundprinzip dieser Abstraktion, stammend aus der natürlichen Mutation, funktioniert ähnlich.

Dabei wird prinzipiell auf Basis einer Wahrscheinlichkeitsentscheidung p_m, ein Teil der genetischen Sequenz verändert [Koz92]. Ein häufig vorkommendes Beispiel ist die Bit-Umdrehung, bei der in einer genetischen Bit-Sequenz ein spezielles Bit abhängig von einer Zufallsvariable, gekippt wird. Dieses Verfahren basiert auf der biologischen Punktmutation, in welcher einzelne Nukleinbasen modifiziert werden.

Für die Mutation gibt es allerdings verschiedene Ansätze, welche alle jeweils maßgeblich vom Anwendungsfall und der Struktur des Genotyps abhängen. Der Zweck bei allen Mutationsarten ist jedoch der selbe: Die Erzeugung von Vielfalt in Populationen. Ähnlichkeiten werden dadurch vermieden. Das führt auch dazu, dass die meisten genetische Algorithmen bei der Generierung

der nächsten Generation nicht nur die fittesten der Population nehmen, sondern eine zufällige oder teils-zufällige Auswahl mit einer Gewichtung zugunsten der fittesten treffen [Koz92].

3.1.5 Genotyp zu Phänotyp

Um vorhersagen zu können, wie sich eine Population an eine bestimmte Umweltveränderung anpasst, muss man verstehen, wie genetische Veränderungen entstehen, wie sie sich als phänotypische Veränderungen manifestieren und wie lebensfähig die daraus resultierenden Phänotypen im Kontext bestimmter Umgebungen sein werden. W. Johannsen, der im Bereich der Genetik besonders die Kausalität zwischen genetischem Erbmaterial und der Erscheinung durch den Phänotyp untersucht hat [Joh11], unterschied zwischen den Auswirkungen, welche genetisches Erbmaterial und der sich daraus bildenden Phänotyp auf die Evolution haben. Diese Beziehung ist keine Eins-zu-Eins Beziehung [Joh11]. Die Komplexität dieser Beziehung wurde durch Pere Alberch als Genotype-Phenotyp-Map (GP-Map) eingeführt [NRTAJ19]. Durch diese wird die Abhängigkeit von Genotyp und Phänotyp modelliert. Die Identifizierung der physikalischen Mechanismen der Vererbung gaben einen gewissen Einblick in die Art und Weise, wie genetische Faktoren den Phänotyp beeinflussen, konnten jedoch nicht viel von dem Prozess aufklären.

In der quantitativen Genetik und verwandten Bereichen wird versucht mithilfe von GP-Mapping, die Gene zu identifizieren, die mit Phänotypen assoziiert sind. Darüber hinaus werden statistische Verfahren angewendet, um messbare phanötypische Variationen zu ermitteln. Diese Korrelationen bieten einen Einblick in das GP-Mapping als eine Art *Black Box*-Prozess, indem sie die Genotypen mit den Phänotypen in Verbindung bringen. Bei evolutionären Algorithmen, welche Phänotypen in einer Umgebung simulieren, ist die Auswahl einer geeigneten Genotyp-Phänotyp-Beziehung von hoher Relevanz für die Aussagekraft eines Genotyps, und damit verbundenem Erfolg der Evolution. Allerdings geben statistische Korrelationen nur wenig Aufschluss über die Mechanismen, durch die sich genetische Veränderungen als phänotypische Veränderungen manifestieren [NRTAJ19].

3.1.6 Fitness

Bei der künstlichen Evolution zur Lösung praktischer Probleme wird in der Regel eine Fitnessfunktion ausgewählt, die das gewünschte Ziel der Suche widerspiegelt. Solche Fitnessfunktionen sind oft einfache quantitative Maße, die intuitiv die entscheidenden Merkmale eines erfolgreichen Ergebnisses zu erfassen scheinen. Diese Maße sind ein Kernpunkt von evolutionären Algorithmen, da sie wie ein Trichter die Suche steuern [Koz92]. Die Fortpflanzung ist auf Individuen mit hoher Fitness ausgerichtet, in der Hoffnung, dass kommende Generationen weitere Fitnessverbesserun-

gen hervorbringen. Dieser Ansatz ähnelt dem Prozess der Tierzucht und stützt sich auf dieselben evolutionären Prinzipien für seinen Erfolg.

Als einfaches Beispiel einer Fitness könnte man die reellwertige eindimensionale Funktion maximieren wollen

$$f(x) = 2x + sin(4x), \qquad 0 < x < \pi \tag{3.1}$$

Hier sind die Lösungsvorschläge Werte von x, die als eine Reihe von Bit-Zeichen kodiert werden können um die reelle Zahl darzustellen. Die Fitnessberechnung übersetzt eine gegebene Bitzeichenfolge x in eine reelle Zahl y und wertet dann die Funktion an diesem Wert. Die Fitness einer Zeichenkette ist der Funktionswert an diesem Punkt.

Es gibt jedoch auch Ansätze in denen eine Spezifikation der Fitness, also dem gewünschten Ziels der Evolution, nicht durch eine Fitnessfunktion ausgedrückt wird. Diese Art von Fitness wird auch als *Implizite Fitness* bezeichnet. Diese ist ein Ansatz in evolutionären Berechnungen für Probleme, bei denen die Fitness eines Individuums als seine Erfolgsrate über eine Anzahl von Versuchen gegen eine Sammlung von Erfolgs-/Misserfolgstests bestimmt wird, statt durch konkretisierte Bewertungsfunktionen.

3.2 Neuronale Netze

Ein neuronales Netz ist eine zusammenhängende Anordnung von sogenannten Perzeptronen (Neuronen), die dem Modell des menschlichen Gehirns nachgestaltet ist. Die Verarbeitungsfähigkeit des Netzes ist in Gewichtungen gespeichert, die durch einen Anpassungsprozess bzw. Lernprozess, üblicherweise mithilfe von Trainingsdaten, neu berechnet werden.

3.2.1 Das Neuronen-Modell

Neuronen sind die Hauptbestandteile eines künstlichen neuronalen Netzes.

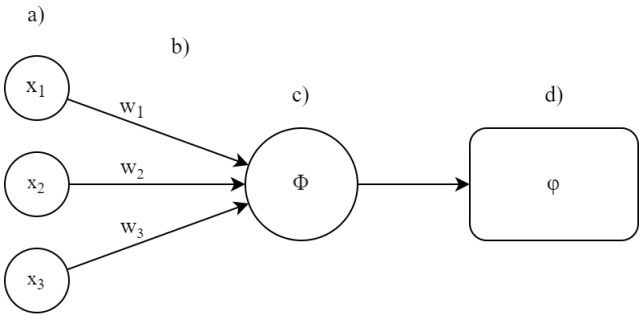


Abbildung 3.4: Modell eines Neurons mit den vier Grundelementen.
a) bildet die Eingangswerte ab. b) deutet auf die Gewichtungen der einzelnen Neuronen zu. c) ist die Übertragungsfunktion, welche die Eingangswerte und Gewichtungen zusammenführt. d) ist die Aktivierungsfunktion, durch welche bestimmt wird ob das Neuron gefeuert wird oder nicht.

Ein Neuron enthält eine Anzahl von Eingangswerten. Bei diesen kann es sich um initiale Eingangswerte oder Ausgaben vorstehender Neuronen handeln. In 3.4 abgebildet, sieht man drei eingehende Eingangswerte für das abgebildete Neuron. Die Eingänge werden durch separate Gewichtungen priorisiert. Das bedeutet, dass die Eingabewerte, abhängig von der zugehörigen Gewichtung einen unterschiedlichen Effekt auf das nächste Neuron haben können. Eine Übertragungsfunktion sorgt für die Aufsummierung der Gewichtungen und der eingehenden Neuronen-Werte. Diese ist mathematisch definiert als

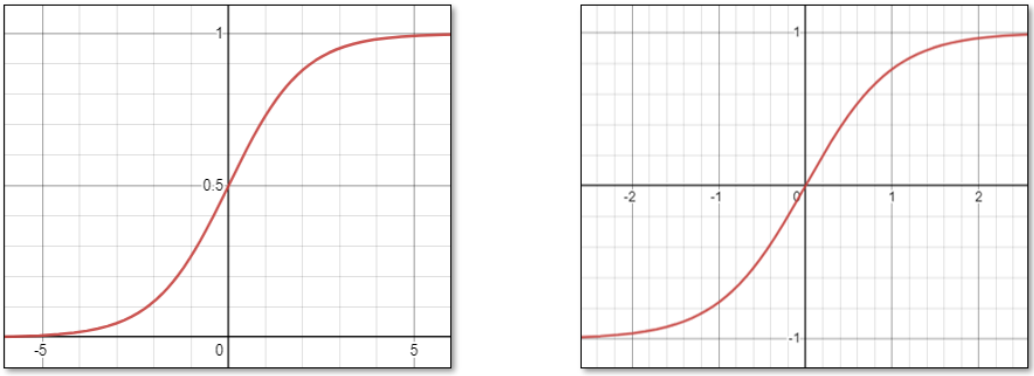
$$\phi(x,w) = \sum_{i=0}^{m} = x_i w_{ik} \tag{3.2}$$

Dabei steht das x_i für den Wert des i-ten Eingangswertes. Das w_{ik} beschreibt den k-ten Gewichtungswert zu gegebenem Neuron an der i-ten Stelle.

3.2.2 Aktivierungsfunktionen

Das menschliche Gehirn versucht jederzeit die eingehenden Informationen zu kategorisieren. Das Ergebnis dieses Prozesses entscheidet welche Entscheidungen *aktiviert* werden. Im Neuronen-Modell bildet man dies durch eine Aktivierungsfunktion ab. Die Trennung spielt eine Schlüsselrolle für die ordnungsgemäße Funktion eines neuronalen Netzes, da sie sicherstellt, dass es aus den nützlichen Informationen lernt und nicht bei der Analyse des nicht nützlichen Teils stecken bleibt.

Es gibt eine Reihe verschiedener Funktionen die gängig als Aktivierungsfunktionen für neuronale Netze genutzt werden. Je nach dem Anwendungsfall eignen sich die ein oder andere Funktion. Gängige Funktionen sind Sigmoid, Tanh und ReLu [Sha21].



(a) Sigmoid (b) TanH

Abbildung 3.5: Aktivierungsfunktionen Sigmoid und Tanh

Sigmoid - Die sigmoide Aktivierungsfunktion wird auch als logistische Funktion bezeichnet. Sie wird unter anderem auch bei der logistischen Regression angewandt [For] und ist wie folgt definiert

$$f(x) = \frac{1}{1+e^{-x}} \tag{3.3}$$

Die Funktion nimmt einen beliebigen reellen Wert als Eingabe an und gibt Werte im Bereich von 0 bis 1 aus. Je größer die Eingabe, desto näher liegt der Ausgabewert bei 1, je kleiner die Eingabe , desto näher liegt die Ausgabe bei 0.

TanH - Der hyperbolische Tangens (kurz *TanH*), ist der sigmoidalen Funktion ähnlich. Auch diese nimmt einen beliebigen reellen Wert als Eingabe an und gibt jedoch anders als bei der Sigmoid Funktion, Werte im Bereich von -1 bis 1 aus. Dementsprechend sind auch hier die positiven Eingabewerte näher am Ausgabewert 1 und die negativen Eingabewerte am Ausgabewert -1. Sie ist wie folgt definiert.

$$f(x) = \tanh x = \frac{e^{x} e^{-x}}{e^{x} + e^{-x}} \tag{3.4}$$

Bias - Üblicherweise fügt man dem Berechnungsprozess noch eine Konstante hinzu, wodurch das Ergebnis der Aktivierungsfunktion verschoben wird, diese wird auch Bias genannt. Eine einfachere Art dies zu verstehen, wäre es mit dem Vergleich zu einer linearen Funktion, z.B. durch $y = ax + b$, wobei b die Verschiebung beschreibt.

Damit lässt sich eine Linie nach oben und unten verlegen, um die Vorhersage besser an die Daten anzupassen. Ohne die Konstante *b* durchläuft die Linie immer den Ursprung, und man erhält eine schlechtere Anpassung.

3.2.3 Schichten und Topologien

Ein einfaches neuronales Netz, oft auch als Feed-Forward-Netz bezeichnet, besteht aus drei grundlegenden Typen von Schichten. Die erste Schicht bildet die sogenannte Eingangsschicht, welche eine Reihe von Eingabeneuronen enthält. Die rohen Eingabedaten werden in dieser Schicht durch die Neuronen in das Netz eingespeist. Der zweite Schichttyp wird als versteckte Schicht (engl. *hidden layer*) bezeichnet. Sie liegt zwischen der Eingabe- und der Ausgabeschicht. Versteckte neuronale Netzschichten können auf viele verschiedene Arten aufgebaut werden und unterschiedlich beliebig oft wiederholt werden. Die Ausgabeschicht ist die letzte Schicht und gibt die Ergebnisse des Netzwerks aus.

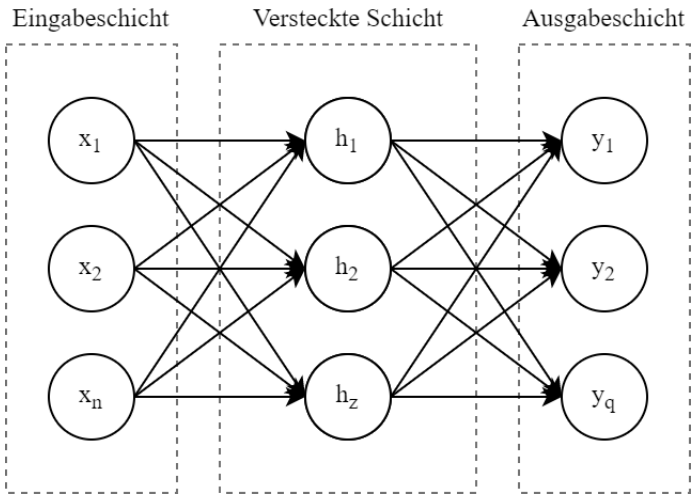


Abbildung 3.6: Topologie eines einfachen Feed-Forward Netzes mit einer versteckten Schicht (engl. hidden layer).

3.2.4 Neuroevolution

Neuronale Netze haben sich in den vergangenen Jahren als erfolgreiche Systeme zur Bewältigung von Aufgaben gezeigt, wie z.B. bei Klassifizierungsproblemen. Dies ist zwei maßgeblich relevanten Faktoren zu verdanken: dem Zuwachs der Datenmengen (Big-Data) und der zunehmenden Computerperformance [Gal20]. Die Entwicklung solcher Netze erfordert jedoch Fachwissen und Erfahrung. Neuroevolution zielt darauf ab, diesen schwierigen und oft zeitraubenden Prozess durch die Anwendung evolutionärer Techniken zu lösen. Vereinfacht ausgedrückt ist die Neuroevolution ein Teilbereich der künstlichen Intelligenz, bei dem versucht wird, einen Evolutionsprozess in Gang zu setzen, der dem unserer Gehirne ähnelt. Es wird versucht, die Mittel zur Entwicklung neuronaler Netze durch evolutionäre Algorithmen zu entwickeln.

4 Konzept - Swimbots

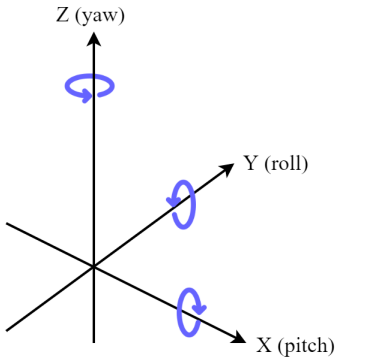
Die Namensgebung für die Swimbots hat seinen Ursprung in der Anwendung Genepool von Jeffrey Ventrella [Ven05]. Diese handelt um eine 2D-Simulation, in der eine Population von virtuellen Organismen (welche Venetrella als Swimbots bezeichnet) Schwimmfähigkeiten entwickeln. Diese können Kriterien für die Partnerwahl festlegen, um geeignete genetische Eigenschaften für zukünftige Generationen zu sichern. Es bilden sich mit der Zeit sogenannte lokale Genpools (nach der Definition von Dawkins [Daw03]), die jeweils um Partnerwahl und Nahrung konkurrieren, da Energie ein essenzieller Bestandteil der Fortpflanzung ist.

Anders als in Ventrellas Arbeit, sind die Swimbots in der KipEvo Simulation von der Gestaltung her an die Arbeit von Karl Sims angelehnt.

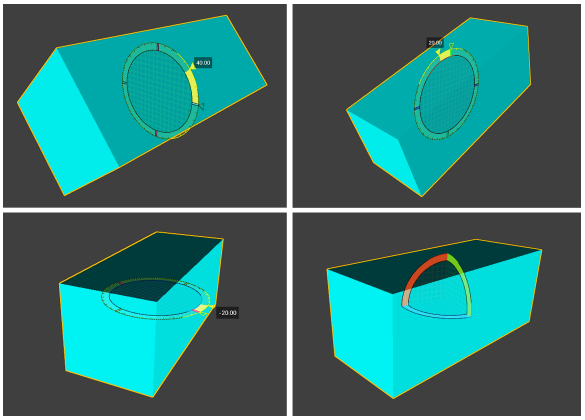
Die Grundidee ist die Erzeugung von virtuellen Kreaturen, welche in einer physikbasierten Umgebung leben und evolvieren. Diese Kreaturen sind in der Lage sich zu bewegen, sich zu reproduzieren und Nahrung einzusammeln. Durch die Informationen aus äußerer und innerer Sensorik, welche der Kreaturen naturgemäß gegeben ist, wird das Bewegungsverhalten ständig neu ermittelt. Die Anpassungsfähigkeit der Kreaturen an die Umgebung, erweist sich implizit durch das Überleben oder Sterben der Kreaturen bzw. einer Population von Kreaturen.

4.1 Die Mechanik der Körper

Die Swimbots sind als gelenkbasierte Körper konzipiert. Dabei bestehen die einzelnen Gliedmaßen aus starren quaderförmigen Körpern. Diese sind durch Gelenke miteinander verbunden. Jedes Gelenk verbindet zwei Glieder miteinander, jedoch kann jedes Glied mehrere Gelenke haben, an denen weitere Glieder verbunden sind. Durch die Gelenke sind eingeschränkte Bewegungen relativ zueinander möglich.



(a) Euler Winkel im Koordinatensystem



(b) Einzelne Rotationen am Mesh durchgeführt

Abbildung 4.1: Rotation im 3D-Raum

Damit im dreidimensionalen Raum eine Drehung der Gelenke an den Gliedern möglich wird, muss das Gelenk als Rotationsgelenk implementiert werden. D.h. die Rotation um alle drei Achsenabschnitte muss berechnet werden. Der dazugehörige Winkel wird auch als Euler Winkel bezeichnet (in Grafik-Engines auch unter *roll-yaw-pitch angle* bekannt). Die Zusammensetzung der verschiedenen Glieder erzeugt eine Körperstruktur, wie in Abbildung 4.3 zu sehen. Die Bewegung wird durch Drehmomente an den Gliedern erzeugt. Diese unterliegen den in den Gelenken spezifizierten Rotationsbeschränkungen.

4.2 Morphologie

4.2.1 Genotyp und Phänotyp

Wie bereits erwähnt bestehen die Swimbots aus Körperteilen, die durch Gelenke miteinander verbunden sind, und bilden so gelenkige Körper. Der Genotyp der Kreaturen gibt Anweisungen darüber wie die Kreaturen gebaut bzw. erzeugt werden, sowie zur deren Verhaltensweise. Daher muss der Genotyp so viel Information wie möglich über die Form und Verhaltensbeschaffenheit haben. Das bedeutet, dass der Genotyp unter anderem die Form und Größe der Körper, die Topologie (d.h. wie die Körper miteinander verbunden sind), die Freiheitsgrade der Gelenke, die Position und Ausrichtung der Gelenke sowie die Gelenkkraftregler des Bewegungsverhaltens und interne Lebensmerkmale (Informationen zur Fortpflanzung, Energiegehalt und weitere) beinhaltet.

Die Informationen für die Ausprägung und Verhaltensweise der Swimbots lassen sich in drei Teilbereiche aufteilen. Der **Körper**-Teil speichert die Körperstruktur, bzw. in welcher Art und Weise einzelne Körperteile miteinander verbunden sind, sowie die Beschaffenheit der Glieder. Die **Basiseigenschaften** beschreiben die unveränderlichen Eigenschaften, welche Einfluss auf mehrere

Faktoren nehmen. Die **genexpressiven Eigenschaften** sind jene Eigenschaften, welche aus den Basiseigenschaften gebildet werden.

Durch die maximale Energie wird der maximal möglich Energiegehalt festgelegt. Kreaturen können nicht mehr Energie zunehmen als die festgelegte Energiegrenze zulässt. Die Mutation soll auf einzelne Eigenschaften angewendet werden. Es findet eine durch die Mutationswahrscheinlichkeit abhängige Mutation statt, analog zur natürlichen Mutation in der durch genotypische Gegebenheiten oder äußere Effekte die Mutationswahrscheinlichkeit beeinflusst wird. Diese gibt sozusagen die Mutationsanfälligkeit an. Die Kopfgröße soll die Basisdimensionen des Kopfes spezifizieren, welche für die Nahrungsaufnahme entscheidend sind. Die beiden Sichteigenschaften beeinflussen die Fähigkeit der Swimbots Nahrung wahrzunehmen.

Es gibt genotypisch abhängige Variablen die nach dem Mutationsprozess erzeugt werden, sogenannte genexpressive Eigenschaften, welche sich aus den Basiseigenschaften ableiten lassen. Zu den genexpressiven Eigenschaften zählen jene, die in Tabelle 4.2 abgebildet sind.

Die Farbe soll als visuelle Unterscheidung der Swimbots dienen, sodass Arten zusammen mit der Körperstruktur voneinander unterschieden werden können. Allerdings kann es natürlich passieren, dass die für die Farbbildung relevanten Variablen so stark mutieren, dass die Farbe sich, trotz der körperähnlichen Struktur des Elternteils, stark verändert, ähnlich wie beim Albinismus (wo durch ein Gendefekt ein Mangel des Farbpigments Melanin entsteht).

4.2.2 Körperbaum

Die Körperstruktur ist wie bereits einleitend erwähnt, als ein Baum von Gliedern aufgebaut. Jedes der Glieder enthält Informationen über die kinematisch geometrischen Beschaffenheit des starren Körpers (Geometrie des Körpers, Rotationsgelenk), sowie der angeknüfpten Glieder. Ein solcher Körperbaum fängt immer mit dem Kopf an, welcher dann eine beliebige Anzahl an Kinderknoten enthalten kann. Der Typ Kopf erhält, wir bereits in der Tabelle 4.1 abgebildet, das Zeichen *h* (head). Für alle weiteren Glieder wird der Typ als Zeichen *b* (bone) festgelegt.

Gen-Typ	Eigenschaft	Beschreibung
Basis-eigenschaften	max. Energie	Die maximale Energie eines Swimbots ist festgelegt als eine Zahl $E_{max} \in \{x \mid 0 < x < 1000,\ x \in R\}$
	Mutations-wahrsch.	Wahrscheinlichkeit mit der eine Eigenschaft im Laufe der Rekombination mutiert wird. Sie ist definiert als $x \in \{x \mid 0 < x < 1,\ x \in R\}$
	Kopfgröße	Der Multiplikationsfaktor zur Größe des Kopfes (Länge/Breite identisch) als Faktor $x \in \{x \mid 1 \leq x \leq 2.5,\ x \in R\}$
	Sichtreichweite	Die Sichtreichweite des peripheren Sichtsensors als Zahl $x \in \{x \mid 100 \leq x \leq 1000,\ x \in R\}$
Körper	ID	Gibt eine, innerhalb der Kreatur, eindeutige ID an, bestehend aus einer Zahl $x \in \{x \mid 1 \leq x \leq N\}$
	Typ	Gibt den Typ der Gliedmaße als Zeichen an. Der Typ unterscheidet zwischen Kopf-Knoten und normalen Knoten. Dieser ist als Zeichen $x \in \{x \mid x \in \{h, b\}\}$
	Mesh	Beschreibt die geometrische Beschaffenheit der Gliedmaße, sowie den Anschluss-Socket. Die geometrische Beschaffenheit der Gliedmaße ist in drei Werten bezüglich G_L, G_B und G_H beschrieben (Länge/Breite/Höhe). $G_L \in \{x \mid 1 \leq x \leq 2,\ x \in R\}$ $G_B \in \{x \mid 1 \leq x \leq 1.5,\ x \in R\}$ $G_H \in \{x \mid 1 \leq x \leq 1.5,\ x \in R\}$ Der Anschluss-Socket G_{Socket} gibt an, auf welcher Körperseite des Elternknoten, die Gliedmaße durch ein Gelenk angeknüpft wird. S = {Top,Bottom,Left,Right,Front,Back} $G_{Socket} \in \{x \mid x \in S\}$
	Anwinkelung	Die Anwinkelung zum Eltern-Körperteil gibt den Rotationswinkel als Euler-Winkel an. Jede der drei Achsenabschnitte α, β, γ (Euler Winkel) können einen Wertebereich von [-90,90] Grad annehmen.
	Kinder-Körperteile	Es enthält die Menge der unterliegenden Körperteile.

Tabelle 4.1: Gesamte Genotypstruktur mit den internen Faktoren sowie den Körpervariablen

Eigenschaft	**Beschreibung**
Farbe	Die Farbe des Swimbots wird im RGB Farbraum dargestellt. Dafür erhalten die additiven Grundfarbenwerte F_{Red}, F_{Green} und F_{Blue} jeweils einen Wert x $\in \{x \mid 0 \leq x \leq 1,\ x \in R\}$
Zeugungs-schwelle	Die Zeugungsschwelle gibt an ab welchem Energiegehalt Nachkommen erzeugt werden. Sie wird aus einem Bruchteil der maximalen Energie ermittelt und ist eine Zahl x$\in \{x \mid 0 \leq x \leq 1,\ x \in R\}$
Energieübergabe	Der Anteil an Energie der an die Nachkommen übergeben wird. Es ist ein Bruchteil der maximalen Energie. $E_{next} \in x \mid 0 \leq x \leq$E$_{max}$, x $\in R\}$
Sichtwinkel	Gibt die Winkelbreite des peripheren Sichtsensors als eine Zahl x $\in \{x \mid 10 < x < 360,\ x \in R\}an.$

Tabelle 4.2: Genexpressive Eigenschaften welche aus dem Genotyp ableitbar sind.

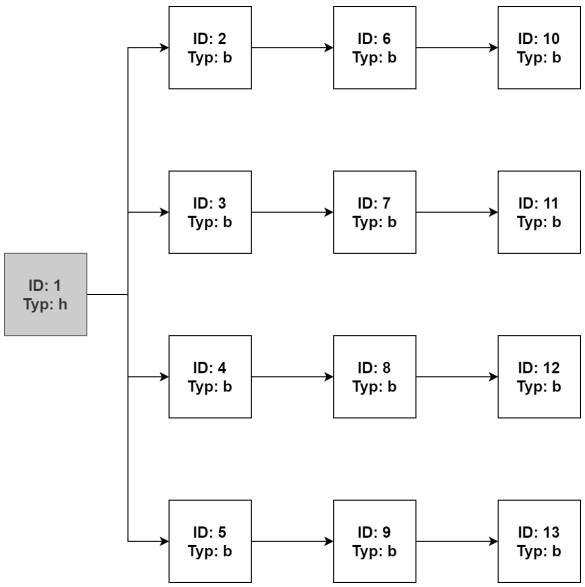


Abbildung 4.2: Körperbaum welcher die Verknüpfungen der Körperteile graphisch visualisiert. Die dazuegehörigen Werte für die einzelnen Glieder sind in Abbildung B.1 hinterlegt.

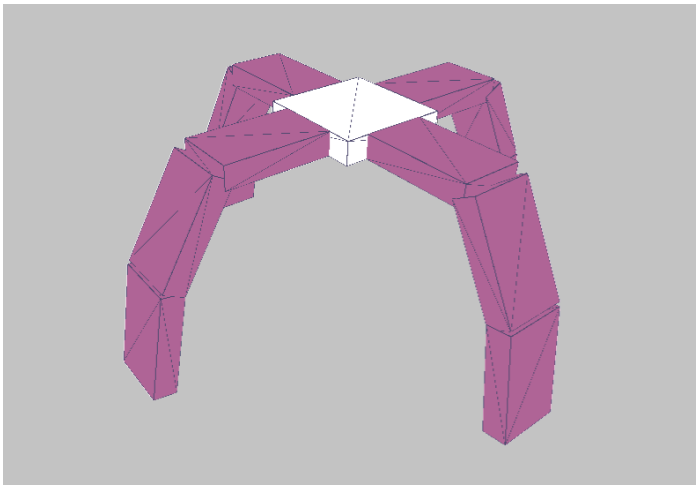


Abbildung 4.3: 3D-Visualisierung des Körpers mit dem Körperbaum aus der Abbildung 4.2

Abbildung 4.3 zeigt ein Beispiel für eine vierbeinige Kreatur, welche entsprechend dem Körperbaum aus der Abbildung 4.2 erzeugt wurde. Die einzelnen Knotenpunkte, welche bei der 3D-Visualisierung als physische Körperglieder übersetzt werden, enthalten jeweils die in der Tabelle 4.1 beschriebenen Körperinformationen.

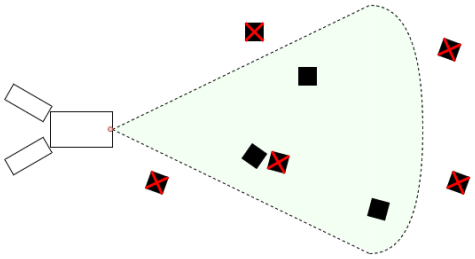
4.3 Sensorik

Damit die Swimbots Informationen aus der Umgebung und dem Inneren (z.B. Tastsensorik, Kraftpotentiale) richtig aufnehmen können, ist eine entsprechende Sensorik erforderlich. Die Auswahl der sensorischen Elemente sind von entscheidender Bedeutung, da sie Verhaltensweise des Swimbots beeinflussen. Die ausgewählten Sensoren müssen daher Umwelteinflüsse wahrnehmen, sodass bestimmte Reaktionen ausgelöst werden können, welche das Überleben der Swimbots gewährleisten.

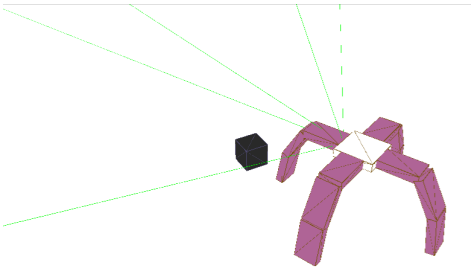
4.3.1 Sichtsensorik

In einer dreidimensionalen Welt mit physischen Komponenten gibt es eine Reihe von Umgebungsinformationen, welche man verarbeiten kann. Dazu zählen zunächst diejenigen Informationen, die durch visuelle Sensoren wahrgenommen werden. Im menschlichen Auge gibt es zwei Wahrnehmungsarten der Sicht. Unter fovealer Sicht versteht man die direkte Zentrierung der Sicht auf einen bestimmten Bereich, nicht breiter angewinkelt als 2 Grad. In diesem wird der Fokus auf genau ein Zentrum gesetzt, wodurch dieser durch das Auge scharf wahrgenommen wird [zop]. Die periphere Sicht beschreibt hingegen die Sicht aus einem breiteren Augenwinkel und nimmt hauptsächlich Hinweisreize auf. Für die Swimbots ist eine periphere Sichtsensorik konzipiert, die auf bestimmte

Objekte reagiert. Dabei wird die Zentrierung auf einen Punkt, wie in der fovealen Wahrnehmung, nicht beachtet. Abbildung 4.4 verdeutlicht das Konzept dieser Sichtsensorik.



(a) Peripherer Sichtwinkel aus einer Top-Shot Perspektive. Objekte die sich außerhalb des Sichtbereiches befinden oder durch ein anderes Objekt verdeckt sind, werden nicht wahrgenommen



(b) Periphere Sichtsensorik in einer 3D-Umgebung. Die dünnen grünen Linien sind Grenzlinien

Abbildung 4.4

4.3.2 Berührungssensorik

Die Berührungs- oder Tastsensorik soll dazu dienen Informationen darüber zu liefern, wie viele Glieder des Körpers den Boden oder andere Objekte berühren.

4.3.3 Kraftsensorik

Der Kraftsensor ist ein interner Sensor, welcher die wirkenden Kräfte aufnimmt. Damit wird das aktuell genutzte Kraftpotential aller Glieder betrachtet.

4.4 Verhaltenssteuerung

Die Verhaltenssteuerung der Kreaturen kontrolliert die Drehmomente, die an den Gelenken ausgeübt werden. Die Größe der Kraft, die auf ein solches Gelenk ausgeübt wird, hängt von den Beschaffenheiten der Gliedmaßen ab. Die Lebewesen können ihre Körperteile in alle Richtungen bewegen, die ihre Morphologie erlaubt. Die Aufgabe der Steuerung besteht darin, die richtigen Kräfte für die jeweilige Aufgabe bereitzustellen. Das System zur Verhaltenssteuerung wird zusammen mit der Morphologie der Kreaturen evolviert. Daraus folgt, dass der Genotyp Informationen enthält, die für den Aufbau der Steuerungsnetzwerke relevant sind. Diese Information unterliegen ebenso der Mutation, wie die genotypischen Informationen zur Beschreibung der Morphologie der Kreaturen. Das neuronale Netz wird jedoch separat als Objekt gespeichert und ist nicht in der Genotyp-Struktur aus Abbildung 4.1 enthalten. Das Steuerungssystem ist als neuronales Feed-Forward-Netz konzipiert. Jedes Neuron

(siehe Abbildung 3.4) im Netz hat eine Anzahl von Eingängen, welche ein Abbild des aktuellen Zustands sind. Dazu gehören z.B. sensorische Informationen. Welche Eingangsinformationen bei der Umsetzung implementiert wurden, wird im Kapitel 5 erläutert.

Die Anzahl der Ausgabeneuronen hängt von der Anzahl der Glieder q ab. Jede Gliedmaße muss für die Drehmomentänderung drei Kraftpotentiale, entsprechend dem Euler-Winkel, erhalten. Daher werden die Ausgabeneuronen in Tripeln dem Drehmoment einer Gliedmaße zugeordnet. D.h. die Gesamtanzahl der Ausgabeneuronen für jede Kreatur ist $q * 3$.

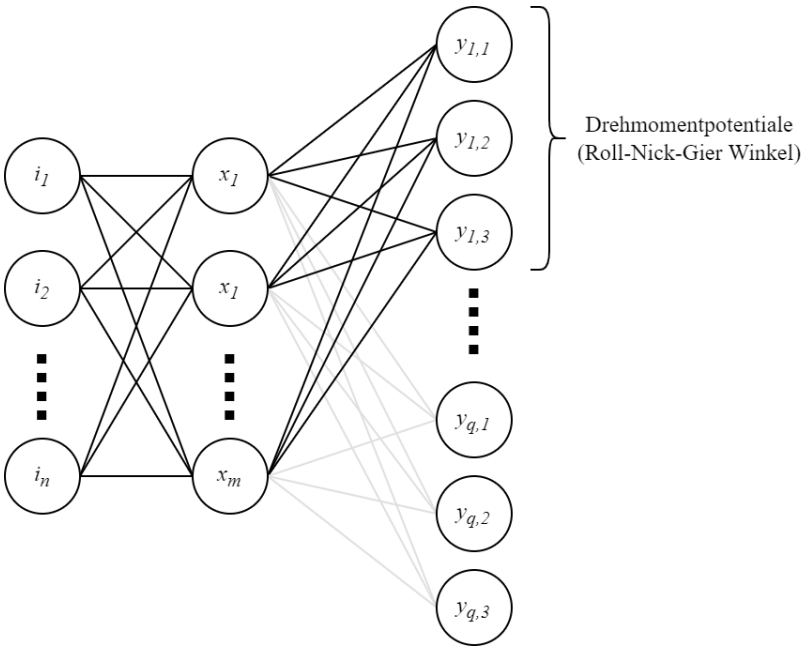


Abbildung 4.5: Abbildung des neuronalen Netzes zum Konzept.
Wobei i die Eingabe-Neuronen beschreibt. Es gibt eine versteckte Schicht, welche durch die Neuronen x gekennzeichnet ist. Als y sind die Neuronen der Ausgabeschicht gekennzeichnet, wobei q für die Anzahl der Glieder steht.

4.5 Mutation

Eine einzige Mutation kann eine große Wirkung haben, aber in vielen Fällen beruht der evolutionäre Wandel auf der Anhäufung vieler Mutationen mit kleinen Auswirkungen. Die Auswirkungen von Mutationen können je nach Kontext oder Ort nützlich, schädlich oder neutral sein. Die meisten nicht-neutralen Mutationen sind schädlich. Im Allgemeinen gilt: Je mehr einzelne Eigenschaften (biologisch gesehen die Basenpaare) von einer Mutation betroffen sind, desto größer ist die Wirkung der Mutation und desto größer ist die Wahrscheinlichkeit, dass die Mutation schädlich ist.

Die Mutation, welche im Laufe der Evolution im Rekombinationsprozess der Swimbots zustande kommt, wirkt sich auf die einzelnen genotypischen Eigenschaften aus. Die am häufigsten erschei-

nende Art der Mutation in genetischen Algorithmen ist die Bit-String Methode, welche bereits in den Grundlagen der künstlichen Evolution eingeleitet wurde. Die Eigenschaften im Genotyp der Swimbots haben allerdings nicht einheitliche Datenformate und können deshalb nicht als Bit-String dargestellt werden. Ein Schema für die Genotyp-Beschreibung kann verschiedenste Werte-Typen besitzen, z.B. Zeichen oder begrenzte Gleitkommazahlen (für die Angabe von Min und Max Werten). Daher eignet sich die Bit-String Methode nicht für die Beschreibung der Swimbot-Mutationen. Die Alternative dazu, wäre der Ansatz von *Real Coded GA's* oder RGA's (dt. Echt-kodierte genetische Algorithmen). Bei RGA's werden die Entscheidungsvariablen oder genotypischen Eigenschaften direkt (ohne Kodierung zu binärer Form) verwendet, um die Bedeutung dieser in speziellen Rekombinations- und Mutationsoperatoren miteinfließen zu lassen. Die Mutation wird daher für alle Eigenschaften individuell begrenzt und durch unterschiedliche Mutationsfunktionen, abhängig von Typ bzw. Bedeutung der Eigenschaft, abgebildet (ähnlich wie bei einer Mehrfach-Optimierung, allerdings nur mit einem Ausgabeziel).

$$M_{float}(v,p,min,max) = \begin{cases} |X_v| * v + v, & \text{für } v < min \text{ und } p > |X_p| \\ -|X_v| * v + v, & \text{für } v > max \text{ und } p > |X_p| \\ X_v * v + v, & \text{für } p > |X_p| \\ v, & \text{für } p \leq |X_p| \end{cases} \tag{4.1}$$

Die Formel 4.1 beschreibt die Mutationsfunktion für jene Eigenschaften dessen Information als Gleitkommazahl ausgedrückt ist. Dabei steht v für den Wert einer Eigenschaft im Genotyp, z.B. die maximale Energie. p steht für die Mutationswahrscheinlichkeit, welche im Genotyp spezifiziert ist (siehe Tabelle 4.1 zum Genotyp). Die *min* und *max* Werte geben jeweils die minimalen und maximalen Grenzwerte für die zu mutierende Eigenschaft an, dabei gilt $min, max \in R$. X beschreibt hierbei eine verteilte Zufallszahl $X \in [-1,1]$. Dabei werden zur Mutationsentscheidung X_p (also bei der Überprüfung der Mutationsbedingung) und Werteberechnung X_v (also der Anpassung der Eigenschaft) unterschiedliche Zufallszahlen verwendet.

Für genotypische Eigenschaften bei denen eine Menge von definierten Zeichenfolgen im Genotyp speziell interpretiert werden (wie z.B. an welcher Oberflächenseite eines Quaders, neue Glieder befestigt werden sollen), wird eine andere Mutationsfunktion nötig.

$$M_{string}(v,p) = \begin{cases} w \in Z, & \text{für } p > |X_p| \\ v, & \text{für } p <= |X_p| \end{cases} \tag{4.2}$$

Die Formel 4.2 beschreibt eine einfache Mutation, bei der im Falle einer entschiedenen Mutation ($p > |X_p|$), ein zufälliges Element $w \in Z$ ausgewählt wird. Dabei ist Z eine Menge an Zeichenkombinationen (z.B. bei der Oberflächenseite in der Form $Z = \{Front, Back, Top, Bottom, Right, Left\}$).

$$X_{dist}(r) = \begin{cases} 1 - \frac{1}{\frac{1}{|r|}+1}, & \text{für } r > 0 \\ -(1 - \frac{1}{\frac{1}{|r|}+1}), & \text{für } r < 0 \\ 0, & \text{für } r = 0 \end{cases} \tag{4.3}$$

Die in 4.3 abgebildete Verteilungsfunktion erzeugt zufällige Gleitkommawerte, welche auf dem Intervall]-1,1[liegen und nicht gleichverteilt sind. Die Verteilungsfunktion schränkt die Auslösung von Mutationen ein, abhängig von der genotypisch gegebenen Mutationswahrscheinlichkeit p. r ist hierbei eine gleichverteilte Zufallszahl, welche als Eingabe für die Verteilungsfunktion verwendet wird.

4.6 Ökosystem

Die Grundidee ist es die Kreaturen in einem Ökosystem der künstlichen Evolution zu überlassen. Dieses Ökosystem beinhaltet, ebenso wie in der natürlichen Welt, abiotische und biotische Komponenten. Die Kreaturen befinden sich in einer begrenzten Landschaft, d.h. es ist ein geschlossener Landschaftsraum vorgegeben. In diesem werden Nahrungsobjekte an zufälligen Positionen innerhalb der Landschaft erzeugt. Die Nahrungsobjekte steigern bei Verzehr den Energiegehalt der Swimbots. Es gibt eine obere Grenze an Energie, welche in dem Ökosystem im Umlauf ist. Wird diese

durch den gesamten Energiegehalt von Massenpopulationen erreicht, werden die Nahrungsobjekte nicht mehr erzeugt. Ziel der Evolution soll es sein, dass die Swimbots sich an die Umwelt durch ihre individuellen genotypischen Eigenheiten anpassen. Das Prinzip des Energieflusses bzw. der Nahrungsbeziehungen aus natürlichen Ökosystemen soll damit simuliert werden.

5 Umsetzung

5.1 Entwicklungsumgebung

5.1.1 Unreal Engine

Unreal Engine ist eine Spiele-Engine mit integrierter Entwicklungsumgebung und einer Reihe von Entwicklungswerkzeugen für die Echtzeitverarbeitung von Visualisierungen in virtuellen Welten. Es wird in der modernen Spieleindustrie aufgrund der hohen Performance und Skalierbarkeit verwendet. Durch die vielen Features und der nativen C++ Unterstützung bietet es viel Freiraum für Projekte jeglicher Art. Die KipEvo-Simulation [IB21] wurde in der Unreal Engine Version 5 entwickelt, welche zum Zeitpunkt der Implementierung als eine der modernsten Grafik-Engines galt.

5.1.1.1 Grundlagen der Engine

Die Entwicklung einer Software in C++ unter Verwendung von Unreal Engine, findet in zwei Entwicklungsumgebungen statt.

Der Engine-Kern (engl. engine core), wie das Unreal Team es bezeichnet, beinhaltet die Implementierung der Basisklassen der Engine, welche über eine API (Unreal Engine API) dem Entwickler bereitgestellt werden. Diese dient vor allem als Low-Level-Index der Engine-Klassen und -Funktionen. Durch die Unterstützung von nativem C++ Code innerhalb der Visual Studio Umgebung, lassen sich damit übersichtlich eigene C++ Komponenten entwickeln.

Der Unreal Engine Editor , in Abbildung 5.3 dargestellt, bietet eine grafische Benutzeroberfläche zur Entwicklung von Unreal Engine Projekten. Über diesen erhält man eine grundlegende Übersicht über das Projekt im Allgemeinen. Während man über C++ die Anwendungslogik schreibt, mittels der von der Unreal Engine API bereitgestellten Schnittstelle, können über den Unreal Editor Projekteinstellungen, Konfigurationen von Materialien oder Modellen, sowie vielen weiteren Features, verwaltet werden.

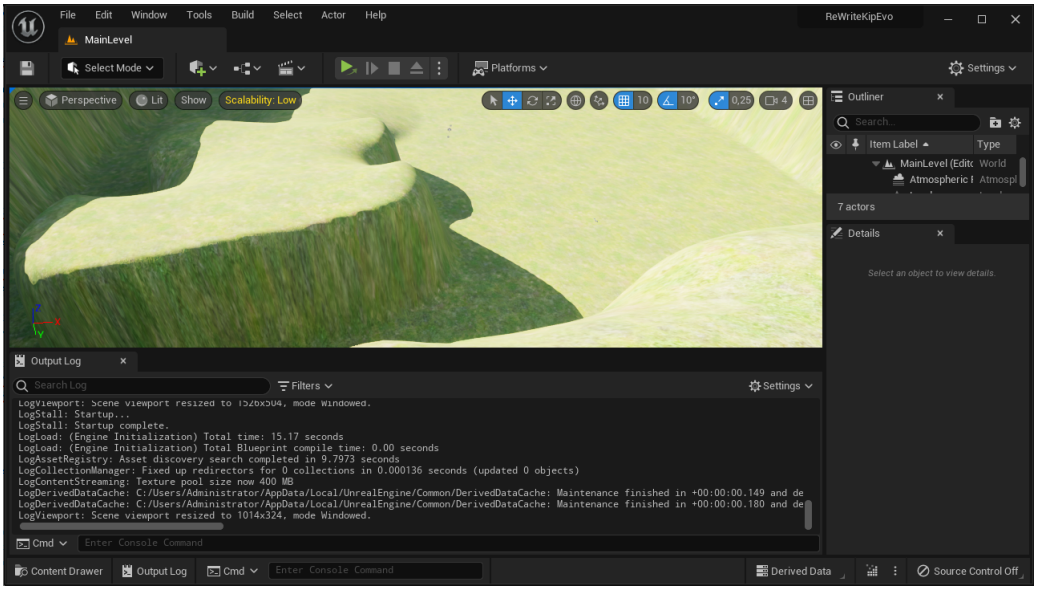


Abbildung 5.1: Der Unreal-Editor aus der Unreal Engine Version 5.

5.1.1.2 Physik

In der realen Welt unterliegen die Objekte den Gesetzen der Physik. Objekte stoßen zusammen und werden gemäß den Newtonschen Bewegungsgesetzen in Bewegung gesetzt. Damit Objekte in der Spielwelt ähnlich wie im wirklichen Leben reagieren, wird eine Physik-Engine eingesetzt. Die Unreal-Physik-Engine nutzt seit der Version 5 ihre eigen entwickelte Chaos-Engine, um Berechnungen für realitätsnahe physikalische Interaktionen wie Kollisionen und Flüssigkeitsdynamik durchzuführen. Dadurch wird die Entwicklung deutlich beschleunigt (anders als bei anderen Grafik-Engines wie OpenGL), da auch eine Schnittstelle zur Anpassung der Physiksimulation bereitgestellt wird.

5.1.1.3 Plugins

Module sind die elementaren Komponenten in der Unreal Engine. Die Engine ist als eine große Sammlung von Modulen implementiert. Mit der Engine entwickelte Anwendungen sind als Module gekapselt, welche in andere Anwendungen integriert werden können. Jedes Modul kapselt eine Reihe von Funktionen und kann eine öffentliche Schnittstelle und eine Kompilierumgebung (mit z.B. Makros und Include-Pfaden) zur Verwendung durch andere Module bereitstellen [plu].

In Unreal Engine sind Plugins Komponenten bestehend aus Modulen, welche im Unreal-Editor projektbezogen aktiviert oder deaktiviert, sowie in der Entwicklungsumgebung inkludiert werden können. Plugins können Spielfunktionen zur Laufzeit hinzufügen, eingebaute Funktionen der Engine ändern (oder neue hinzufügen). Viele bestehende Unreal Engine Subsysteme wurden so konzipiert, dass sie mit Plugins erweitert werden können. Die Implementierung einer Third-Party Bibliothek

kann auch als Plugin realisiert werden. Diese Methode der Inkludierung von Bibliotheken wird sogar als vernünftigste Methode angesehen, da es eine Modularität schafft (d.h. ein getrenntes Softwarepaket einbindet, welches jederzeit in der Projektkonfiguration ein-/ausgeschaltet werden kann).

Plugins lassen sich in Unreal Engine durch die Erzeugung einer .uplugin Datei (siehe Anhang B) im vordefinierten *Plugins* Ordner erstellen. Diese sorgt dafür, dass es im Unreal Engine Build Prozess als Plugin erkannt wird. Im Ordner des Plugins muss eine Build Konfiguration erstellt werden, welche als *.build.cs* Datei abgespeichert wird. Mit dieser spezifiziert man die Pfade der Quelldateien (dazu zählen auch die statischen oder dynamischen Bibliotheken), sowie Kompiler-Definitionen für die Bibliothek (siehe Anhang B). Der C++-Quellcode, der zu einem Plugin gehört, wird dabei direkt im Ordner der Build-Konfiguration oder in Unterverzeichnissen davon gespeichert. Die *.build.cs* Konfigurationen werden vom *UnrealBuildTool* kompiliert und aufgebaut, um die gesamte Kompilierumgebung zu bestimmen.

5.1.1.4 Unreal Engine Architektur

Project

Ein Project enthält den gesamten Inhalt eines Unreal Engine Projekts. Es enthält eine Ordnerstruktur, in der Blueprints, Quellcode, Materialien und 3D-Modelle hinterlegt sind. Die Ordnerstruktur kann beliebig erweitert werden. Eine grundlegende Struktur ist in der Tabelle 5.1 zu sehen. Jedes Projekt in Unreal Engine enthält eine Projektdatei, mit der Endung *.uproject*. Mit der *.uproject*-Datei kann das Projekt als ausführbare Anwendung gestartet, oder im Unreal-Editor bearbeitet werden. Dabei werden Plugins und Module angegeben, welche im Projekt verwendet werden.

Object

Ein Object (*UObject*) ist in Unreal Engine eine konstitutive Klasse. Sie wirkt als elementarer Baustein und enthält grundlegende Funktionen für vererbende Komponenten. Zu diesen zählen eine automatische Speicherbereinigung (engl. Garbage Collection), die Spezifikation von Metadaten (*UProperty* Eigenschaften), mit welchen Eigenschaften für den Unreal-Editor zugänglich gemacht werden, sowie eine Serialisierung. Ein Großteil der im Engine-Kern implementierten Klassen erben von *UObject*.

Actor

Ein Actor (*AActor*) ist ein beliebiges Objekt, welches in einem Level platziert werden kann. Dazu zählen beispielsweise Objekte wie die Kamera, starre Körper (engl. static meshes) oder KI-gestützte Objekte. Actors unterstützen 3D-Transformationen wie Translation, Rotation und Skalierung. Sie lassen sich durch die Engine-API erstellen und zerstören, sowie während der Laufzeit verändern. Die

Verzeichnis	Beschreibung
Binaries	Ausführbare Dateien die im Rahmen des Kompilierungsprozesses erzeugt werden.
Config	Enthält die Projekteinstellungen für Editor, Engine und Game, sowie Platformen
Content	Enthält Inhalte für die Engine oder das Projekt, einschließlich Asset-Pakete und Levels.
Intermediate	Enthält temporäre Dateien, die von UnrealBuildTool generiert werden, z. B. Visual Studio-Projektdateien.
Saved	Enthält Dateien, die von der Engine generiert werden, wie z. B. Konfigurationsdateien und Protokolle.
Source	Enthält die Quelldateien des Projekts.

Tabelle 5.1: Unreal Engine Game Projektstruktur

Klasse *AActor* wird als Basisklasse für alle Akteure verwendet.

Component

Eine Komponente ist ein nicht-phsyisches Objekt (d.h. es wird nicht gerendert und erfordert keine physikalischen Berechnungen), welches einem Actor als Unterobjekt hinzugefügt werden kann und eine erweiterte Funktionalität bietet. Komponenten sind nützlich, um gemeinsame Verhaltensweisen zu teilen, z. B. visuelle Informationen aufzunehmen oder Töne abzuspielen. Z.B. lassen sich damit Komponenten für die Audioerzeugung (AudioComponent) an den Akteur anknüpfen, wodurch dem Akteur die Möglichkeit der Audio-Wiedergabe ermöglicht wird. Komponenten müssen jedoch mit einem Akteur verknüpft sein, um zur Laufzeit erzeugt zu werden.

Controller

Controller sind nicht-physische Akteure, die eine physische Figur (darunter sind erweiterte Akteure gemeint, welche von der AActor Klasse erben) oder andere Objekte kontrollieren können und deren Aktionen zu steuern. Ein PlayerController beispielsweise, ermöglicht eine Kontrollschnittstelle für Nutzer, um Akteure zu steuern, während ein AIController die künstliche Intelligenz für die von ihnen kontrollierten Akteuren implementiert.

Controller erhalten Benachrichtigungen über viele der Ereignisse, die für den von ihnen kontrollierten Akteur eintreten. Dies gibt dem Controller die Möglichkeit, das Ereignis abzufangen und als Reaktion darauf anstelle des Standardverhaltens eventuell ein eigenes Verhalten zu implementieren.

Gamemode

Der Gamemode (dt. Spielmodus) bestimmt die Regeln einer Ausführung. Diese Regeln können die

Startsequenzen der Objekte beinhalten, Regeln zum Verhalten einer Simulation, oder Gewinnbedingungen in einem Spiel. Der Standard-Gamemode kann über die Projekteinstellungen festgelegt werden und ist für jedes Level festgelegt.

Level
Als Level wird die räumliche Struktur und Sammlung von starren Körpern, Lichtern und anderen grafisch relevanten Eigenschaften, bezeichnet. In Unreal Engine wird jedes Level mit der Endung *.umap* gespeichert.

Delegate
Delegates ermöglichen es Mitgliedsfunktionen von Objekten auf eine typsichere Weise aufzurufen. Diese können dynamisch, also auch während der Laufzeit, an eine Mitgliedsfunktion gebunden werden. Damit werden unter anderem Listener für Ereignisse erzeugt, auf welche innerhalb eines Objekts reagiert werden kann.

5.1.2 Versionsverwaltung

Als Versionsverwaltung wurde GitHub [gita] genutzt. Dieser ist ein Anbieter von Internet-Hosting für Softwareentwicklung und Versionskontrolle mit der Versionskontrollsoftware Git [gitb]. Der Hauptvorteil von GitHub ist sein Versionskontrollsystem (git), das eine nahtlose Zusammenarbeit ermöglicht, ohne die Integrität des ursprünglichen Projekts zu gefährden. Das Projekt wurde inklusive Source-Code und notwendigen Abhängigkeiten (Plugins und Bibliotheken, sowie vorkompilierte Bibliotheken) in das Git-Repository abgelegt.

5.1.3 Bibliotheken

Unreal Engine bietet viele Features, welche die Entwicklung von Computerspiel-Logiken vereinfachen. Dabei zählen z.B. vordefinierte Module für die Speicherung von Spielzuständen, sowie Sub-Frameworks zur Implementierung von KI-gesteuerten Bewegungen. Jedoch sind die Algorithmen und Datenstrukturen auf die Anwendungsfälle in diesen Bereichen limitiert. Die Nutzung von komplexeren Datenstrukturen erfordert die Nutzung einer Third-Party Bibliothek. Dafür wurden folgende Bibliotheken in das Projekt als Unreal Engine Plugins inkludiert.

5.1.3.1 Boost

Der Fokus bei der moderner Softwareentwicklung liegt auf der möglichst schnellen und einfachen Umsetzung der zugrundeliegenden Idee. Die ursprüngliche Computersprache C++ reicht nicht mehr

den diesbezüglichen Anforderungen. Um die Produktivität bei der Entwicklung zu steigern empfiehlt sich daher die Verwendung hochwertiger Bibliotheken. Eine der bekanntesten ist Boost, die mit einer Vielzahl an Tools die Entwicklung beschleunigt, unleserliche Boiler-Plate Fragmente vermeidet und dadurch auch zu weniger Fehlern führt.

5.2 Systemarchitektur und Komponenten

Die Anwendungsstruktur ist wie folgt aufgebaut:

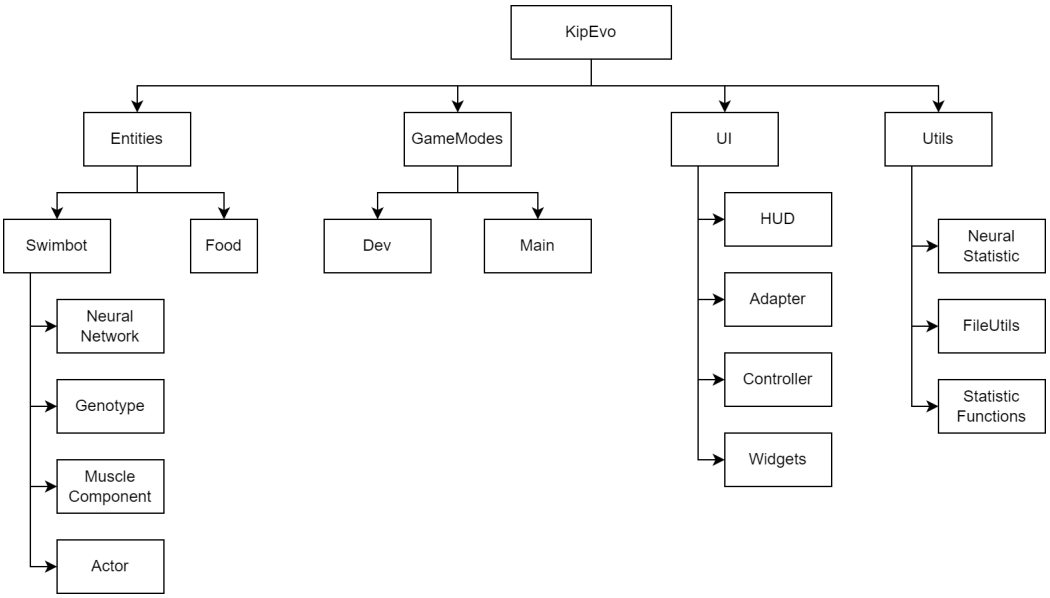


Abbildung 5.2: Systemarchitektur der Projekts, sowie grundlegende Komponenten.

Dabei enthält das Projekt 4 Unterverzeichnisse. Die Entities beschreiben die einzelnen Simulationsobjekte. Dazu zählen die Swimbots selbst und die Food Objekte für die Nahrungsaufnahme. Die Swimbots beinhalten 4 Hauptkomponenten, inklusive vereinzelter Komponenten für die Zusammensetzung eines physischen Akteurs. Die NeuralNetwork Komponente beschreibt die Funktionsweise des neuronalen Netzes. Dieses wurde grundlegend aus einer früheren Version des KipEvo Projekts übernommen und modifiziert. Die Genotyp-Komponente enthält die Implementierung der Genotypstruktur und Funktionen für die genetischen Operatoren. Die Muscle-Komponente simuliert die die Kraftwirkung in Abhängigkeit der ausgehenden Potentiale aus dem neuronalen Netz. In der Actor Implementierung werden alle Komponenten verknüpft und die entsprechenden Game-Events (dazu zählen Listener der Komponenten, sowie Delegationsfunktionen) behandelt.

5.2.1 Genotyp zu Phänotyp

Der Phänotyp wird in einem Interpretationsprozess aus dem Genotyp gebildet. Dabei wird die Beschreibung der Körper und der Basiseigenschaften, in Objekte der Unreal Engine Umgebung

abgebildet. Für jede Gliedmaße wird ein starrer Körper zur Engine-Umgebung hinzugefügt und mit den entsprechenden Elternteilen durch Constraints verbunden.

Der Genotyp ist als JSON-Datenstruktur gespeichert. Diese wird als `property_tree` Objekt in das Projekt integriert, welche aus der Property Tree Bibliothek stammt und im Boost Framework enthalten ist. Ein Property Tree ist eine Datenstruktur, welche verschachtelte Bäume speichert, in Form von Schlüssel-Wert Paaren. Die JSON-Struktur des Genotyps leitet sich aus der in Abbildung 4.1 beschriebenen Spezifikation ab und ist in Listing 5.2 abgebildet. Dabei enthält der Wert zu dem Schlüssel `dependent` die genexpressiven Werte, welche nach der Mutation des Genotyps gebildet werden. Der `body` Wert bildet den Körperbaum ab. Der Einfachheit halber bezeichnen wir jede Gliedmaße als Knoten. Die Kinderknoten eines Elternknoten sind in `children` als Array abgelegt.

Zu jedem Kindknoten, sind die Schlüssel `socket, angleToParent` und `id` relevant für die Anknüpfung zum Elternknoten. Der `socket` Wert legt dabei die Oberflächenseite des Quaders zur Anknüpfung am Elternknoten fest. Bei einem Socket-Wert "Back“wird somit der Knoten an der Rückseite des übergeordneten Quaders befestigt.

Listing 5.1: Genotyp Schemata im JSON Format.

```
{
        "dependent": {
                "color": {
                        "x": 1.0,
                        "y": 1.0,
                        "z": 1.0
                },
                "procreationThreshold": 0.5,
                "energyToChild": 100.0,
                "sensorAngle": 40
        },
        "genotype": {
                "maxEnergy": 500,
                "mutationProbability": 0.5,
                "headSize": 1.0,
                "sensorRange": 500,
                "sensorRadius": 1000
        },
        "body": {
                "id": "",
                "type": "h",
                "mesh": {
                        "width": 0.5,
                        "height": 1.0,
                        "length": 1.0,
                        "socket": "Back"
                },
                "angleToParent": {
                        "roll": 45,
                        "pitch": 0,
                        "yaw": 45
```

```
32                  },
33                  "children": [
34
35                  ]
36          }
37  }
```

Ein Swimbot erbt von der `Pawn` Klasse, welche von der `Actor` Klasse erbt und eine erweiterte Controller-Funktionalität implementiert. `Pawn`-Objekte können mit AI-Controllern des von der Engine-API bereitgestellten AI-Moduls verbunden werden. Diese sind erforderlich für die Integration der visuellen Sensorik, mit welcher der Swimbot eine periphere Sicht erlangt. Die `init()` Funktion initialisiert alle Swimbot-Komponenten und bereitet den Swimbot für die Simulation vor. Dazu zählen folgende Schritte:

- Erstellung einer Genotyp-Komponente `GenotypeComponent` mit den Erbinformationen des Elternteils. Dieser wird dann mutiert und die genexpressiven Eigenschaften werden erzeugt.
- Startender Energiegehalt wird gesetzt, entsprechend des vom Elternteil übergebenen Energiegehalts `energyToChild` (siehe Listing 5.2).
- Die Erstellung des physischen Körpers mit der Ausführung der createBody() Funktion.
- Konfiguration des AI-Controllers für die Wahrnehmungskomponente (`AIPerception`).
- Erzeugung der neuronalen Komponente `NeuralNetworkComponent` für das Bewegungsverhalten. Bei nicht vorhandenem Elternteil (initial zufällig erstellte Genotypen) wird ein neues neuronales Netz erzeugt.

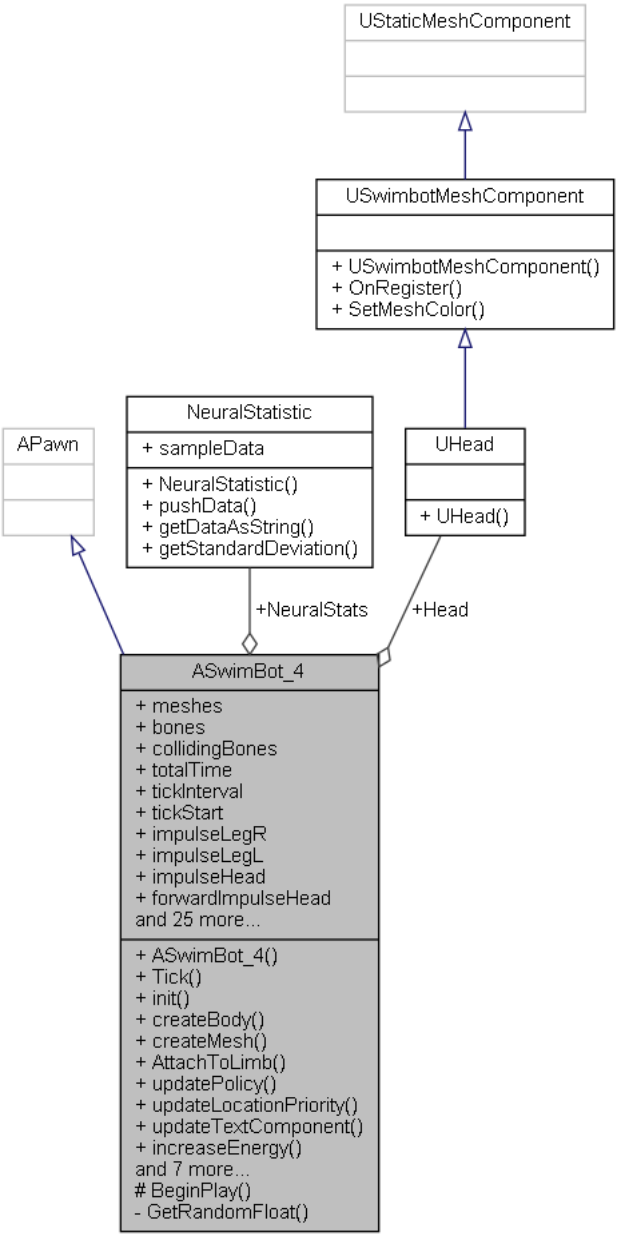


Abbildung 5.3: `ASwimBot_4` Klassendiagramm mit Abhängigkeiten.

5.2.2 Abbildung des Körperbaumes

Für die Umwandlung des im Genotyp festgeleten Körperbaums, in einen physikalisch simulierbaren `Actor`, welcher den Phänotyp darstellt, sind drei Schritte notwendig:

1. Abbildung des Kopfknoten (Wurzelknoten) als starren Körper.
2. Die Erzeugng der starren Körper inklusive aller notwendigen physikbasierten Eigenschaften.

3. Eine beschränkte physiksimulierte Verbindung zwischen zwei starren Körpern, welche das Gelenk darstellt.

Diese Prozesse sind im Swimbot-Actor integriert. Obwohl der Kopf auch als starrer Körper abgebildet wird, benötigt dieser eine separate Integration, da der Kopf auf bestimmte Ereignsse reagieren muss. Dazu zählt die Kollision mit einem Nahrungsobjekt, welches für eine Energiezunahme sorgt. In der Funktion `createBody()` wird der Körperbau-Prozess gestartet. Dabei wird der Kopf als UHead erzeugt. UHead ist eine für den Kopf spezifizierte Klasse, welche von `USwimbotMeshComponent` erbt.

`USwimbotMeshComponent` ist die Basisklasse für alle starren Körper. In dieser wird die physikalische Grundkonfiguration für alle simulierten Körperteile des Swimbots festgelegt. Dazu zählen folgende Einstellungen:

- Aktivierung der Physiksimulation.
- Mobilität setzen: Die Bewegung eines Körpers durch Kräfte oder andere Einflüsse erlauben.
- Sichtbarkeit in der 3D-Simulation aktivieren.
- Overlap-Events aktivieren. Diese erlauben das Abfangen von Kollision durch Delegates.
- Kollisionskanäle festlegen: Jeder physischer Körper in der Engine ist einem Kollisionskanal zugeordnet. Die Kanäle legen fest, wann Körper und womit kollidieren sollen.

Die Integration dafür ist in Codebeispiel B.4 abgebildet.

5.2.2.1 Verbindung der Glieder

Die Verbindung der Glieder wird mithilfe von Constraints realisiert. Unter einem Constraint versteht man in Unreal Engine eine Art Gelenk. Dieses ermöglicht es zwei simulierbare Körper (dazu zählen starre Körper der Basisklasse `UStaticMesh` oder Akteure) miteinander zu verbinden und dabei physikalische Grenzen festzulegen. Zu diesen physikalischen Grenzen zählen folgende:

- Die Winkelbegrenzungen aller einzelnen Körperachsen.
- Die Verdrehwinkelgrenze bzw. Torsion.
- Modus für den Winkelantrieb.
- Parameter für den Winkelantrieb: Positionsstärke, Geschwindigkeitsstärke, Kraftgrenze.

Ein physikalisches Constraint wird durch die Klasse `UPhysicsConstraintComponent` erzeugt. Sie wird dem Actor als Komponente hinzugefügt und verbindet zwei Körper-Komponenten

welche von der Basisklasse `UStaticMeshComponent` erben. Objekte des Typs `UBone` und `UHead` können in dieser Form daher miteinander verbunden werden.

Die Regelung, welche definiert auf welche Art und Weise zwei Körper miteinander verbunden werden, wird durch die Struktur `FAttachmentTransformRules` (als `UStruct` Datentyp) bestimmt. Dabei wird festgelegt ob neu zu verbindende Körperteile relativ zum Eltern-Körperteil transformiert werden sollen. Dies wirkt sich auf die Lage, Rotation und Skalierung aus. Bei der relativen Skalierung von verknüpften Körperteilen, werden diese abhängig vom Eltern-Körperteil skaliert. Allerdings ist dies hier nicht erwünscht. Daher wird für die Skalierung die Option `EAttachmentRule::KeepWorld` ausgewählt, mit welcher der angehängte Körper die eigene Raumlage beibehält (keine relative Anpassung). Die Rotations- und Lagetransformationen allerdings, werden zur einfachen Positionierung der neuen Körperteile als relative Transformation festgelegt. Die einzelnen starren Körper werden innerhalb der `createMesh()` Funktion auf eine Map abgebildet, welche als Schlüssel eine eindeutige ID der einzelnen Körper speichert und als Wert die Referenz auf ein `UStaticMeshComponent` Objekt. Diese wird im Verlauf der Verknüpfung zum direkten Aufruf der relevanten Körperreferenzen benutzt. Bei der Verknüpfung der beiden Körper mit der `AttachToComponent()`, ist mitunter die Angabe eines Sockets (dt. Sockel) möglich. Mit diesem lässt sich, wie bereits in Tabelle 4.1 beschrieben, die anknüpfende Oberflächenseite angeben. Die Verdrehwinkelgrenzen bzw. Winkelbegrenzungen werden mithilfe der Funktionen `SetAngularTwistLimit()`, `SetAngularSwing1Limit()`, `SetAngularSwing2Limit()` gesetzt. `AngularSwing1Limit` gibt die horizontale und
`AngularSwing2Limit` die vertikale Winkelbegrenzung an. Die Rotationstransformation unter Verwendung des Euler-Winkels `angleToParent` wird durch die Funktionen `RotateAngleAxis()` und `SetRelativeRotation()` angewandt. Die sich aus der Rotation ergebende Lageverschiebung wird durch die Aktualisierung mit `SetRelativeLocation()` gewährleistet.

Listing 5.2: `AttachToLimb` Funktion für die Verknüpfung der UPhysicsConstraintComponent Gelenke.

```
1  void ASwimBot_4::AttachToLimb(FString limbID, FString parentLimbID,
       FVector angleToParent, FString socket) {
2          if (this->meshes.Find(limbID) == NULL) {
3                  UE_LOG(LogTemp, Warning, TEXT("Error: LimbID not
                       found in the List of Meshes."));
4          }
5          else if (limbID == parentLimbID) {
6                  UE_LOG(LogTemp, Warning, TEXT("Error: Same LimbID in
                       AttachToLimb"));
7          }
8          else {
9
10                 UBodyConstraint* constraint = NewObject<
                       UBodyConstraint>(this->meshes[limbID]);
```

```
11                  FVector parentScale = this->meshes[parentLimbID]->
                        GetComponentScale();
12                  FVector scale = this->meshes[limbID]->
                        GetComponentScale();
13                  float offsetX = 0;
14
15                  FAttachmentTransformRules attachmentRule(
                        EAttachmentRule::KeepRelative, EAttachmentRule::
                        KeepRelative, EAttachmentRule::KeepWorld, false);
16
17          // Anknüpfen des Constraint an das Eltern-Körperteil am
                angegebenen Socket
18                  constraint->AttachToComponent(this->meshes[
                        parentLimbID], attachmentRule, FName(*socket));
19                  constraint->RegisterComponent();
20                  constraint->SetRelativeLocation(FVector(offsetX, 0,
                        0));
21          // Die Verbindung der starren Körper festlegen.
22                  constraint->SetConstrainedComponents(this->meshes[
                        limbID], "", this->meshes[parentLimbID], "");
23          // Die Dreh- bzw. Verdrehwinkelgrenzen  setzen.
24                  constraint->SetAngularTwistLimit(ACM_Limited, 25.f);
25                  constraint->SetAngularSwing1Limit(ACM_Limited, 25.f)
                        ;
26                  constraint->SetAngularSwing2Limit(ACM_Limited, 25.f)
                        ;
27          // Lineare Limits deaktivieren.
28                  constraint->SetLinearXLimit(LCM_Locked, 0.f);
29                  constraint->SetLinearYLimit(LCM_Locked, 0.f);
30                  constraint->SetLinearZLimit(LCM_Locked, 0.f);
31          // Modus für den Winkelantrieb setzen.
32                  constraint->SetAngularDriveMode(EAngularDriveMode::
                        SLERP);
33                  constraint->SetOrientationDriveSLERP(true);
34          // Positionsstärke festlegen, Geschwindigkeitsgrenze und
                Kraftgrenze bleiben bei 0.
35                  constraint->SetAngularDriveParams(100.f, 0, 0);
36
37                  this->meshes[limbID]->AttachToComponent(constraint,
                        attachmentRule, "");
38                  this->meshes[limbID]->RegisterComponent();
39
40                  this->meshes[limbID]->SetMassOverrideInKg(NAME_None,
                         scale.X * scale.Y * scale.Z * 10, true);
41
42                  scale.X *= 50;
43
44                  scale = scale.RotateAngleAxis(-angleToParent.X,
                        FVector(0, 1, 0));
45                  scale = scale.RotateAngleAxis(angleToParent.Z,
                        FVector(0, 0, 1));
46
47
48                  this->meshes[limbID]->SetRelativeLocation(scale);
49                  this->meshes[limbID]->SetRelativeRotation(FRotator(
                        angleToParent.X, angleToParent.Z, 0));
```

```
50
51              }
52  }
```

5.2.2.2 Zufallsbasierte Genotyperzeugung

Initial wird beim Start der Simulation eine Anzahl randomisierter Genotypen für die Swimbots erzeugt. Die Erzeugung dieser ist in der Genotyp-Komponente implementiert. Die Genotyp-Komponente beinhaltet unter anderem auch die Implementierung der genetischen Operatoren (diverse Mutationsfunktionen) und der genexpressiven Abbildungsfunktion. Bei der zufallsbasierten Erzeugung wird das JSON-Schema aus 5.2 verwendet, in welcher die Grundstruktur des Genotyps festgelegt ist. Die Mutationswahrscheinlichkeit wird am Anfang des Prozesses erzeugt.

Körperbaum

Die Basis für die Erzeugung des Körperbaums bietet eine zufällig erzeugte Zeichenfolge, welche wir als Körperstring bezeichnen [IB21]. Durch die Interpretation des Körperstrings wird die Struktur des Körperbaums Schritt für Schritt aufgebaut und in ein `property_tree` Objekt gespeichert. Die einzelnen Zeichen können aus der Menge {g,B,b,t} stammen. Jedes von Ihnen bildet eine bestimmte Funktion ab, wie in Tabelle 5.2 abgebildet. Der Körperstring spiegelt zunächst keine Struktur wieder und durchläuft daher im Anschluss einen Algorithmus welcher die Folge korrekt in einen Körperbaum überführt. In Abbildung 5.4 ist dies mit einem beispielhaften Körperstring illustriert.

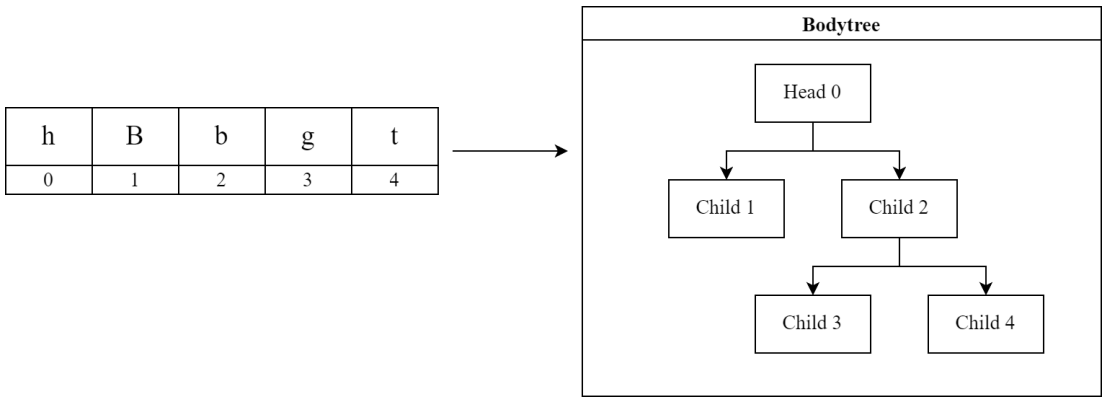


Abbildung 5.4: Beispielahfter Körperstring, welcher in eine Körperbaumstruktur umgewandelt wird.

Die Eigenschaften der einzelnen Körperteile werden zufällig erzeugt. Das sorgt dafür, dass einige Körperteile miteinander in der Simulation kollidieren können: z.B. durch Anwinkelungen die dazu führen, dass Körperteile ineinander positioniert sind. Dies ist in Abbildung 5.5 genauer zu sehen.

Zeichen	Funktion
g	Dieses Zeichen wird als Gelenk interpretiert. Dabei wird ein neuer Körper-Knoten an den übergeordneten Knoten angehängt.
B	Damit wird eine Root-Verzweigung errichtet. Bei dieser Verzweigung wird im Root-Knoten (d.h. dem Kopfknoten) ein neuer Kindknoten hinzugefügt.
b	Damit wird eine Verzweigung am zuletzt erstellten Körperknoten erstellt. D.h. es wird diesem Knoten ein Kindknoten angefügt.
t	An dem zuletzt referenzierten Verzweigungsknoten wird ein weiterer Kindknoten hinzugefügt.

Tabelle 5.2: Bedeutung der einzelnen Zeichen im Körperstring.

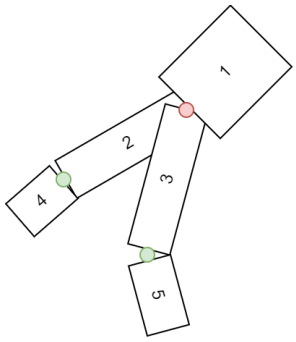


Abbildung 5.5: Beispielhafte Struktur, welches das Kollisionsproblem illustriert. Die Zahlen beziehen sich hierbei auf die Körperteile (1 steht für den Kopf).

Der Kopf (Nummer 1) hat zwei Körperteile an sich hängen, welche jedoch an fast der gleichen Position durch Constraints mit dem Kopf verbunden sind (siehe roter Punkt). Dadurch würde bei aktiver Kollision eine unrealistische Physik, durch die intern berechneten Kollisionskräfte, realisiert. Bedenkt man die Beschränkungen der Constraints, würden die Körperteile während der Simulation gelähmt wirken. Daher wird zu Beginn der Simulation, nachdem der physische Körper bereits räumlich erzeugt wurde, auf eine Kollision überprüft, in der Körperteile sich zu einem festgelegten Faktor überschneiden. Ist dies der Fall, so wird der Körperteil mit den darin angehängten Kinderknoten entfernt, da dieser nicht realistisch simulierbar ist. Dies verhindert nicht, dass Körperstrukturen erzeugt werden können, welche durch ihre Beschaffenheit in der Bewegung behindert sind. Der iterative Prozess, löscht die einzelnen Körperteile inklusive der Constraints. Der Körperbaum (Genotyp) des betroffenen Swimbots bleibt hingegen unverändert. Dadurch bleibt die

Möglichkeit offen, dass bei Swimbots späterer Generationen durch die Evolution eine Anpassung der Positionierung der Körperteile anfällt.

Genotyp

Nach dem erfolgreichen Aufbau des Körperbaums, folgen nun die Basiseigenschaften des Genotyps. Diese werden abhängig von der Mutationswahrscheinlichkeit, welche initial zufällig erzeugt wird, durch Anwendung der Mutation erzeugt. Dabei wird der Genotyp einer Anzahl an Mutationen unterworfen. Diese Mutationen sorgen dafür, dass für die Basiseigenschaften zufällige Werte erzeugt werden und der Baum entsprechend den bestimmten Wahrscheinlichkeiten auch verändert wird.

Mutation

Die Mutationsfunktionen für die verschiedenen Wertetypen, wurde bereits in Abschnitt 4.5 näher erläutert. Programmiertechnisch betrachtet muss man bei der Mutation des Genotyps zwischen zwei Datentypen unterscheiden. Zum einen der Gleitkommazahl, welche als `float` dargestellt ist, und zum anderen dem `string` für die Mutation der Sockets.

Die `getRandomDistribution()` Funktion implementiert die in 4.3 beschriebene Verteilungsfunktion der Zufallswerte. In dieser werden die verteilten Zufallswerte erzeugt, welche in den Mutationsfunktionen zur Entscheidung und Anpassung der Mutationen genutzt werden.

Die `mutateFloatWithBounds()` Funktion implementiert die Funktion zur beschränkten Mutation von Gleitkommazahlen. `mutateSocket()` hingegen implementiert die Mutation für die Sockets.

5.2.3 Krafterzeugung

Für die Kraftausführung ist im Swimbot-Akteur die Funktion `UpdateMovement` implementiert. In dieser wird die Kraft auf alle Gelenke ausgeführt. Dafür passieren folgende Schritte

- Aktuelle sensorischen Informationen für die Eingabe in das neuronale Netz werden im Vorfeld geladen.
- Eine Vorwärtspropagierung des neuronalen Netz für die neuen Eingaben.
- In einer Schleife wird für jedes Gelenk, mithilfe des Kraftpotentials aus dem neuronalen Netz, die physische Kraft ermittelt.
- Die Bewegung wird für jedes der Gelenke durch Drehmomente realisiert.

Die Drehmomentanpassung wird durch die Physik-Funktion `AddTorqueInDegrees()` ausgeführt. Als Eingabe erhält dieser einen Vektor mit den entsprechenden Kräften. Die Kräfte werden als *kilogram-force-centimeter* (kgf/cm) angegeben und die Berechnung entspricht den Newtonschen

Momentformeln [lei]. Die Krafterzeugung findet als ständige Potentialanpassung zu jeder Physikberechnung statt. Das bedeutet, dass es zu jedem physikalischen Zeitschritt, die aktuellen Kräfte berechnet werden.

Listing 5.3: Die Grundschleife für die Kraftausführung an den Gelenken

```
1  for (auto limb = this->bones.CreateConstIterator(); limb; ++limb)
2  {
3          UBone* bone = (UBone*) limb.Value();
4
5          neuronValX = neuronValues[counter];
6          neuronValY = neuronValues[counter + 1];
7          neuronValZ = neuronValues[counter + 2];
8
9          forceSinX = bone->getForce(neuronValX);
10         forceSinY = bone->getForce(neuronValY);;
11         forceSinZ = bone->getForce(neuronValZ);;
12         force = FVector(forceSinX, forceSinY, forceSinZ);
13
14         bone->AddTorqueInDegrees(force);
15
16         currentTotalForce += force.Length();
17
18         detractEnergy(force);
19
20         counter += 3;
21 }
```

`this->bones` referenziert die bereits in Abschnitt 5.2.1 näher erläuterte Abbildung der Körperteile mit Schlüssel-Wert Paaren. Durch diese wird iteriert und es werden der Reihenfolge nach mit den ermittelten Potentialen aus dem neuronalen Netz die jeweiligen Kräfte berechnet. `currentTotalForce` speichert zu jeder Kraftberechnung die aktuelle Kraftauswirkung des gesamten Körpers, im Bezug auf die maximal mögliche Kraftauswirkung (also durch volles Kraftpotential aller Glieder). Anschließend wird der Energiegehalt des Swimbots durch die Kraftauswirkung reduziert.

5.2.4 Ökosystem und Konfiguration

Das Ökosystem setzt sich aus der Lebensumgebung der Swimbots (also der Landschaft) und abiotischen Faktoren wie der Nahrungserzeugung. Die Landschaft ist als feste Struktur implementiert und ist strukturell nicht durch Handlungen der Swimbots veränderbar; es kann auch als eine Art Versuchsumgebung angesehen werden. Der Gamemode, wie bereits in 5.1.1.4 beschrieben, gibt hierbei die Grundregeln der Simulation fest und bereitet eine Startkonfiguration vor. Zu diesen Aspekten zählen folgende:

- Die Festlegung und Überprüfung der globalen Energie. Abhängig von dieser wird entschieden

ob weitere Nahrungsobjekte erzeugt werden.

- Zufällige Erzeugung von Nahrungsobjekten in der Versuchsumgebung, abhängig von der globalen Energie.
- Die Erzeugung von neuen Swimbots: Abhängig von der Reproduktionsentscheidung des Swimbots wird im Gamemode durch Aufruf der Engine-Funktion `SpawnActor()` ein Swimbot erzeugt.
- Einrichtung des Menüs, sowie der Controller für den Nutzer (Kamerasicht).
- Funktionen zum Entfernen von Swimbots und Nahrungsobjekten.
- Initiale Grafikeinstellungen durch Konsolen-Befehle, sowie das Schalten der Physiksimulation und des Renderings.

Die Konfiguration der Startparameter erfolgt mit einer JSON Datei, welche als Startkonfiguration in die Simulation geladen und interpretiert wird. In dieser befinden sich folgende Parameter:

Listing 5.4: Startkonfiguration der Simulation

```
{
        "startBots" : 100,
        "energyPerFood" : 200,
        "energyToDecreaseRatio" : 1e14,
        "maxWorldEnergy" : 100000,
        "foodSpawnInterval" : 1,
        "maxTimeDilation" : 10,
        "ratioMassToForce" : 20000000,
        "maxBodyStringSize" : 60,
        "SpawnAreaXMax": 9500,
        "SpawnAreaXMin": -9500,
        "SpawnAreaYMax" : 9500,
        "SpawnAreaYMin"  : -9500,
        "SpawnAreaZBot" : 300,
        "SpawnAreaZFood" : 1000
}
```

`startBots` gibt an wie viele Bots initial erzeugt werden sollen. Für diese werden entsprechend zufällige Genotypen nach dem bereits beschriebenem Verfahren erzeugt. `energyPerFood` gibt die Größe des Energiegehalts pro Nahrungsobjekt an; diese sollte nicht zu gering gewählt werden, da es sonst zu viele physische Nahrungsobjekte in der Simulationslandschaft gibt, welche für die Bewegung der Swimbots zum Hinderniss werden. `energyToDecreaseRatio` gibt den Divisor an, welcher bei der Quotientenberechnung für die Energieabnahme verwendet wird. Je größer dieser ist, desto weniger Energie verbraucht ein Swimbot. `maxWorldEnergy` gibt die Grenze der globalen Energie im Ökosystem an, d.h. wenn diese erreicht ist, werden keine Nahrungsobjekte mehr automatisch erzeugt. `foodSpawnInterval` gibt die Zeit an (in Sekunden) die zwischen der Erzeugung einzelner Nahrungsobjekte liegen soll. `maxTimeDilation` gibt den Faktor der Simulationsbeschleunigung

an. Dieser hat sich bei ca. 10 als maximum bewehrt, da sich durch Testläufe gezeigt, dass höhere Werte zu einer instabilen Physik führen. Über diesen Wert gibt es keine realistische Beschleunigung mehr. Dies kann jedoch von System zu System unterschiedlich sein. Dies wird noch in Abschnitt 5.3 näher erläutert. `ratioMassToForce` gibt den Kraftfaktor an; dieser wird bei der Kraftausführung (also der Ausführung der Drehmomente) mit den Potentialen aus dem neuronalen Netz multipliziert. `maxBodyStringSize` gibt, für die Erzeugung eines zufälligen Genotyps, die maximale Länge des Körperstring (Bodystring) an. Die restlichen Konfigurationsvariablen mit dem Suffix `Spawn` geben jeweils die Lagegrenzen für die Erzeugung neuer Objekte an.

5.2.4.1 Visueller Sensor-Kontroller

In Unreal Engine sind bestimmte Wahrnehmungen in Form von Stimulis implementiert. D.h. ein Akteur kann als Stimuli-Hörer fungieren, während ein anderer Akteur als Stimuli-Quelle dient. In der Implementierung reagiert der Swimbot auf visuelle Stimuli. Der visuelle Sensor für die sichtbare Wahrnehmung besteht aus einer `AIPerception` Komponente.

Diese wird mit einem `UAI_SenseConfig_Sight` Objekt konfiguriert. In diesem lassen sich Sichtreichweite, peripheren Neigungswinkel und weitere Konfigurationen vornehmen. Der Kontroller implementiert die `AIPerception` Komponente. Die Funktion `OnSeeActor()` ist eine Delegate Funktion, welche auf das `OnPerceptionUpdated` Event feuert (also wenn die Wahrnehmung durch den Sensor aktualisiert wird, z.B. durch das Erblicken eines neuen Objekts). Dabei wird die Anzahl der zum Zeitpunkt sichtbaren Objekte gezählt, so wie die Anzahl der gesamt gesichteten Objekte (dazu zählen aktuell sichtbare, sowie vergangene Sichtungen). Die Variable `stimuliCounter` speichert die Anzahl der Erblickten Objekte durch den Sensor, welche später für die Eingabe in das neuronale Netz verwendet wird.

Listing 5.5: OnSeeActor Funktion welche die Sichtung der Objekte verarbeitet.

```
1  void AVisualSensorController::OnSeeActor(const TArray<AActor*>&
       testActors)
2  {
3          int counter = 0;
4          int counterVisible = 0;
5          for (auto x = this->PerceptionComponent->
               GetPerceptualDataIterator(); x; ++x) {
6                  FActorPerceptionInfo z = x.Value();
7
8                  if (z.LastSensedStimuli[0].GetAge() == 0.f) {
9                          counterVisible++;
10                 }
11
12                 dist = (z.LastSensedStimuli[0].ReceiverLocation - z.
                       LastSensedStimuli[0].StimulusLocation).Size();
13                 if (dist < distMax) {
```

```
14                          this->bestFoodLocation = z.LastSensedStimuli
                                [0].StimulusLocation;
15                          distMax = dist;
16                  }
17                  counter++;
18
19          }
20          if (counter == 0) {
21                  this->bestFoodLocation = FVector::ZeroVector;
22          }
23
24          this->stimuliCounter = counter;
25          this->stimuliVisibleCounter = counterVisible;
26  }
```

5.2.5 Neuronales Netz

Das neuronale Netz steuert das Bewegungsverhalten eines Swimbots. Es nimmt als Eingabe eine Reihe relevanter dynamischer Variablen und gibt in der Ausgabeschicht die einzelnen Drehmomentpotentiale für alle einzelnen Glieder aus. Die Wahl der Variablen spielt hierbei eine essentielle Rolle, denn diese beeinflussen die Art der Information, welche an das neuronale Netz übergeben werden. Bei einer schlechten Informationsauswahl, wird im Laufe des evolutionären Prozesses kein geeignetes Bewegungssystem evolvieren. Die eingehenden Informationen müssen daher eine Bedeutung zum Zustand des Swimbots (z.B. der Sensorik, also der Aufnahme von Umgebungsinformationen, oder zu internenen Prozessen wie z.B. der Energie) widerspiegeln. Die geeignete Auswahl an Eingangswerten, lassen sich anhand von Probemessungen mithilfe statistischer Methoden ermitteln, welche auch im Bereich des Machine Learning große Anwendung findet [BSM$^+$].

Das neuronale Netz ist in der Klasse `NeuralNetworkComponent` implementiert. In ihr sind grundlegende Funktionalitäten zum Kopieren, Erstellen, Mutieren und Propagieren von neuronalen Netzen integriert. Die Grundimplementation stammt aus dem KipEvo Projekt [IB21] und wurde lediglich angepasst. Folgende Anpassungen wurden implementiert

- Die sigmoide Aktivierungsfunktion wurde durch TanH (siehe Abbildung 3.5b) ersetzt. Diese ermöglicht einen Wertebereich zwischen [-1,1]. Dieser Wertebereich wird für die Kraftberechnung benötigt, da die Kräfte des Drehmoments abhängig von der Drehrichtung positive und negative Werte annehmen können.
- Der Zufallsgenerator für die initiale Gewichtung, den Bias sowie der Mutation wurde auch auf den Wertebereich [-1,1] angepasst.
- Die Anzahl der Schichten wurde minimiert und die Auswahl der Eingagsneuronen wurde entsprechend der Bedeutsamkeit angepasst. Dazu im folgenden mehr.

5.2.5.1 Standardabweichung

Die Standardabweichung ist der durchschnittliche Wert der Volatilität in einer Datenreihe. Sie gibt an, wie weit jeder Wert im Durchschnitt vom Mittelwert entfernt ist. Aus einer hohen Standardabweichung folgt, dass die Werte im Allgemeinen weit vom Mittelwert entfernt sind, während eine niedrige Standardabweichung die Nähe zum Mittelwert deutet. Mathematisch ist sie wie folgt definiert.

$$\sigma = \sqrt{\frac{\sum_{i=0}^{m} = (X - \bar{x})^2}{n-1}} \tag{5.1}$$

X steht hierbei für die vereinzelten Werte einer Datenreihe. $\bar{x}$ ist der Mittelwert der Datenreihe. n ist die Anzahl der Werte in der Datenreihe.

5.2.5.2 Komplexität der Berechnungen

In Feed-Forward Netzen ist die Komplexität aus der Matrixmultiplikation und der Aktivierungsfunktion zu ermitteln. Diese lassen sich durch die Landau-Notation wie folgt beschreiben:

$$M_{ij} * M_{jk} = O(i * j * k) \tag{5.2}$$

$$A_{ik} = O(i * k) \tag{5.3}$$

5.2.5.3 Performance bei Feedforward-Propagierung

Der Algorithmus für die Feedforward-Propagierung lässt sich folgendermaßen beschreiben. Um von Schicht i nach j zu gelangen, werden die Übertragungen durch Matrixmultiplikationen berechnet (siehe 3.2.1). Anschließend wird die Aktivierung ermittelt. Für jedes versteckte Neuron wird eine Matrixmultiplikation durchgeführt, gefolgt von der Anwendung einer nichtlinearen Aktivierungsfunktion. Jedes versteckte Neuron j führt also die folgende Operation durch

$$O_j = \sum_{i=0}^{k} w_{ij} * x_i \tag{5.4}$$

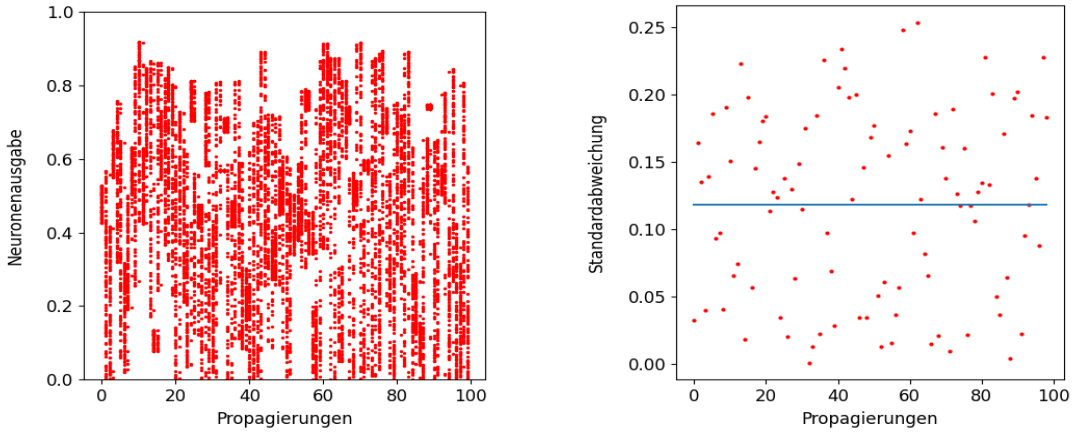
wobei i die vom Eingangsneuron i stammende Eingabe ist. w_{ij} steht für die Gewichtung der Verbindung vom Eingangsneuron i zum versteckten Neuron j.

Bei einem Netz mit 5 Eingängen, 20 versteckten Neuronen (innerhalb einer versteckten Schicht) und 20 Ausgangsneuronen sind folgende Anzahl Rechenschritte für die Ausgabe notwendig. Es

gibt 20 versteckte Neuronen, für die jeweils entsprechend der Formel 5.4 20 Multiplikationen berechnet werden müssen (ohne die Aktivierungsfunktionen). Zusätzlich werden für die Verbindung der Eingangsschicht zur versteckten Schicht $5 * 20 = 100$ Multiplikationen fällig. Analog dazu müssen von der versteckten Schicht bishin zur Ausgabe erneut $20 * 20 = 400$ Multiplikationen berechnet werden. Es wird deutlich, dass bei der Konstruktion einer geeigneten Topologie für die Simulation von vielen Bewegungssystemen die Anzahl der Eingangsneuronen und Schichten auf das Minimum beschränkt sein soll.

5.2.5.4 Problem bei invarianten Eingaben

Damit das neuronale Netz auf Veränderungen reagieren kann, muss es Werte als Eingabe bekommen, welche den aktuellen Zustand, so diversifiziert wie möglich widergeben können. Diese Zustandsinformationen können z.B. Wahrnehmungsinformationen oder interne Kraftmessungen sein. Allerdings muss dabei die Anzahl der Eingänge, aufgrund der Performanz, so gering wie möglich gehalten werden. Bei der Suche geeigneter Kandidaten für die Eingänge wurde die Streuung mithilfe der Standardabweichung ermittelt und visualisiert.



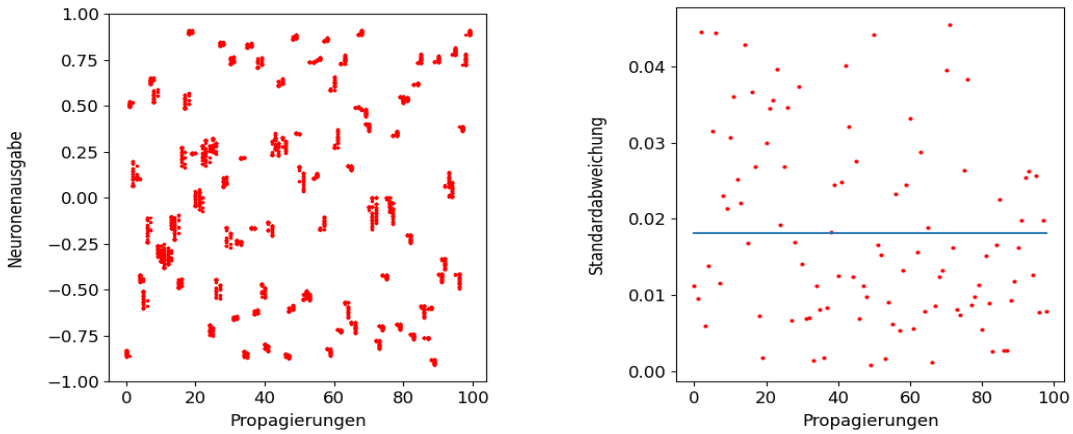
(a) Streuung der Ausgabewerte

(b) Standardabweichung

Abbildung 5.6: Streuung für die Auswahl von Eingabeneuronen

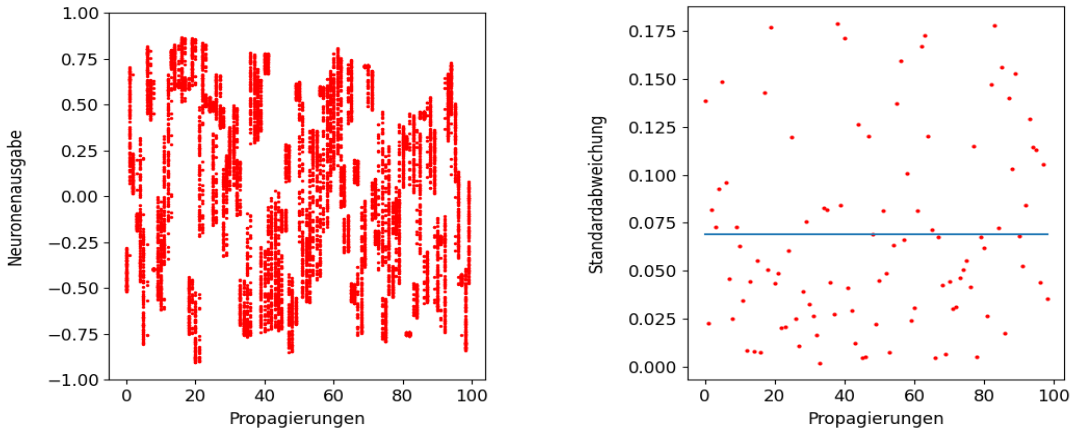
Abbildung 5.6 visualisiert eine Streuung, bei der die Eingaben des neuronalen Netzes aus nur randomisierten Werten besteht. Dabei ist keine Struktur zu erkennen, d.h. die Ausgabewerte bilden zwar einen großen Wertebereich ab (was der Streuung zu entnehmen ist), jedoch ergeben sich diese nicht aus nützlichen Information. Ohne bedeutende Informationen, welche den Zustand der Individuen abbilden, kann keine evolutionäre Anpassung des Systems statt finden.

Der zweite Ansatz ist in Abbildung 5.7 dargestellt. Hierbei wurden mehrere zustandsabhängige Eingaben in das Netz eingespeist. Dazu zählen der aktuelle Richtungsvektor zum besten Nahurngsobjekt



(a) Streuung der Ausgabewerte (b) Standardabweichung

Abbildung 5.7: Streuung für die Auswahl von Eingabeneuronen



(a) Streuung der Ausgabewerte (b) Standardabweichung

Abbildung 5.8: Streuung für die Auswahl von Eingabeneuronen

(jeweils drei Gleitkommazahlen für jede Dimension im Raum), der Quotient aus dem aktuellen Energiegehalt und der maximalen Energie, der Quotient aus der aktuell ausgeübten Kraft und der maximal möglichen Kraftausübung aller Glieder, sowie der Quotient aus der Anzahl sichtbarer und insgesamt bereits gesichteter Objekte. Allerdings wird deutlich, dass die Veränderungen zu gering sind um stark unterschiedliche Ausgaben zur jeder Propagierung zu ermöglichen. Die Standardabweichung zeigt auch deutlich, dass Ausgaben zum Teil stagnieren.

Der dritte Ansatz, welcher in Abbildung 5.8 zu sehen ist, hat zwei zusätzliche Eingaben gegenüber 5.7. Hierbei ist eine Zufallszahl (im Intervall [-1, 1]) in die Eingabe eingespeist. Zusätzlich dazu wird der Quotient aus kollidierenden Gliedern und der Gesamtzahl aller Glieder, hinzugefügt. Diese beiden Informationen sorgen für eine erweiterte Dynamik in den Ausgabewerten.

5.3 Physiksimulation

5.3.1 Finetuning der Physik

Die Performance einer numerischen Physik-Simulation hängt sehr eng mit den Teilberechnungen eines Zeitschrittes zusammen. Unter einem Zeitschritt versteht man hierbei im Kontext der Grafik, den zeitlichen Abstand zwei aufeinanderfolgend gerenderter Bilder. Je kleiner der zeitliche Abstand zwischen den Teilberechnungen ist, desto geringer sind die Ungenauigkeiten einer in Echtzeit simulierten Physik. Dadurch bietet sich unabhängig von der Integrationsmethode immer einen Bereich der Instabilität, welcher abhängig von der Performance eines Systems ist.

5.3.1.1 Physik-Substepping

Mithilfe von Substeps (dt. Unterschritt) in Unreal Engine lassen sich die Berechnungen für die Physik-Simulation, abhängig von der Bildrate, in mehrere Berechnungsschritte pro Bild unterteilen. Dies sorgt für eine stabilere und präzisere Physiksimulation bei Computern mit schwächerer Performance. Die Physikeinstellungen hierzu, finden sich in den *Framerate*-Einstellungen. Dabei sind einige Aspekte zur Festlegung der optimalen Einstellung von Bedeutung, damit eine möglichst reibungslose Physik-Simulation gewährleistet werden kann.

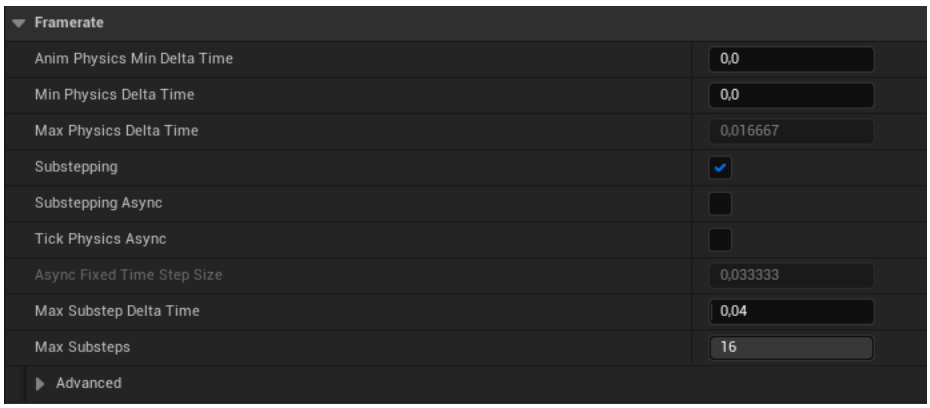


Abbildung 5.9: Substep Einstellungen im Unreal Engine Editor

5.3.1.2 Delta-Time

Die Delta-Time beschreibt den zeitlichen Abstand zwischen zwei gerenderten Bildern. Die Engine gibt die festgelegte Delta-Zeit an die Physik- und Tickfunktionen weiter und versucht gleichzeitig eine konstante Framerate zu erreichen. Hat man beispielsweise eine Bildrate von 20 Bildern pro Sekunde, so würden alle $50ms$ ein Bild gerendert werden. Mit einer Delta-Time von $12,5ms$ erreicht

man somit die Berechnung von 4 Substeps pro Bild. Bei einer Bildrate von 40 Bildern pro Sekunde, würden analog 2 Substeps pro Bild berechnet (Delta-Time wäre hierbei $6,25ms$. In der folgenden Abbildung ist die Substep-Komplexität genauer zu erkennen.

Abbildung 5.10: Eine Konfiguration zu Physik-Substepping.
Grafisch dargestellt, erkennt man die Unterteilung der Substeps bei welcher Bildrate die Anzahl der Berechnungsschritte zunimmt. Für die Darstellung wurde der interaktive Graph von Portelli zur Darstellung der Substeps benutzt [Por16].

Die sich damit ergebene stabilere Physiksimulation geht jedoch auf Kosten der CPU-Performance. Man kann bei einer niedrigen Delta-Zeit, zwar eine flüssigere Physiksimulation erreichen, jedoch gleichzeitig eine noch geringere Bildrate bewirken. Im Laufe der Suche nach der richtigen Konfiguration, hat sich dieser Nebeneffekt zur Performance und der damit eingehenden geringeren Bildrate, jedoch nicht deutlich bemerkbar gemacht. Dies kann jedoch bei schwachen Systemen anders sein.

In der aktuellen Konfiguration ist die maximale Delta-Zeit auf $40ms$ eingestellt und die Anzahl der Physik-Substeps auf 16 gesetzt. Durch Untersuchungen hat sich die Konfiguration im Entwicklungssystem als stabil herausgestellt. Bei älteren Versuchen mit einer geringereren Anzahl Substeps, wurde an einigen Stellen die Kollision zwischen Körpern nicht berücksichtigt, wodurch diese dann überlappt sind bzw. durch den Boden fielen. Die richtige Konfiguration des Physik-Substeppings ist eine sehr feinfühlige Optimierung, welche bei unterschiedlichen Computer-Systemen (besonders Systeme, welche über eine geringe Rechenkapazität verfügen) verschiedene Ergebnisse liefern kann. Eine Mindestrechenkapazität ist für eine korrekte Simulation erforderlich, da sonst nicht die Echtzeit-Physik korrekt simuliert werden kann.

6 Evaluation

6.1 Simulationsexperimente

In diesem Abschnitt werden die Simulationsexperimente vorgestellt. Im Rahmen von 4 Simulationsexperimenten soll die virtuelle Simulationsumgebung demonstriert werden. Ziel der Experimente ist es, die Evolution sowie Folgeeffekte an den Swimbots deutlich zu machen. Anders als bei gewöhnlichen genetischen Algorithmen, in denen tausende von Generationen in Sekundenschnelle entstehen, ist dieser Entwicklungsprozess bei einer physikbasierten Echtzeitsimulation von Lebewesen, deutlich langsamer. Dies liegt zum einen an den zusätzlichen räumlichen Physikberechnungen, da eine Anzahl starrer Körper gerendert und der Zustand dieser im Raum mehrmals pro Sekunde neu berechnet werden muss. Zum anderen sorgt die Festlegung einer aufgrund der festgelegten Simulationsumgebung impliziten Fitness dafür, dass die Stimulierung der Speziation nach einer Sammlung von Erfolgen und Misserfolgen bestimmt wird, statt durch eine festgelegte Bewertungsfunktion.

Jedes Experiment startet mit einer individuellen Startkonfiguration (siehe Abbildung 5.5). Im weiteren Verlauf werden die einzelnen Parametar durch die Mutation geändert, wobei sie dann die beobachteten Effekte beeinflussen.

6.2 Ergebnisse der Evolution

6.2.1 Erster Durchlauf

Für den ersten Durchlauf wurde folgende Standardkonfiguration gewählt. Die Simulationsdauer betrug ungefähr 17 Stunden. Dabei wurde eine Anzahl von 100 Swimbots in die Welt gesetzt, welche jeweils einen maximale Körperstring-Größe von 40 besitzen konnten. Die Kraft-Masse-Beziehung wurde hierbei auf $3x10^7$ gesetzt. Der Energie-Abzugsrate energyToDecreaseRatio wurde hierbei bei allen Durchläufen auf $1x10^8$ erhöht, statt der grundlegend eingestellten $1x10^{12}$, um den Evolutionsprozess zu beschleunigen. Dies hatte zur Folge, dass in vielen Durchläufen die Swimbots ausgestorben sind, jedoch bei einzelnen Durchläufen die Evolution schneller vorangetrieben wurde,

da die globale Energie durch aussterbende Swimbots reduziert wurde, und neue Nahrungsobjekte in die Welt gesetzt werden konnten.

Listing 6.1: Startkonfiguration für den ersten Simulationsdurchlauf

```
{
        "startBots" : 100,
        "energyPerFood" : 200,
        "energyToDecreaseRatio" : 1e8,
        "maxWorldEnergy" : 100000,
        "foodSpawnInterval" : 1,
        "maxTimeDilation" : 10,
        "ratioMassToForce" : 30000000,
        "maxBodyStringSize" : 40,
        "SpawnAreaXMax": 9500,
        "SpawnAreaXMin": -9500,
        "SpawnAreaYMax" : 9500,
        "SpawnAreaYMin"  : -9500,
        "SpawnAreaZBot" : 300,
        "SpawnAreaZFood" : 1000
}
```

Die folgende Grafik 6.1 zeigt die Anzahl und den zeitlichen Verlauf der Entitäten im ersten Durchlauf. Zu Beginn ist ein starkes Wachstum an Swimbots zu erkennen, was damit zu erklären ist, dass am Anfang der Simulation gleichzeitig eine bestimmte Anzahl von Nahrungsobjekten in der Versuchsumgebung generiert werden. Zwischen Minute 300 - 700, ist sowohl ein Zuwachs der Nahrungsobjekte, als auch der Swimbots zu erkennen. Daraus ist zu entnehmen, dass Swimbots mit einem geringeren Energiegehalt und einer starken Reproduktionquote bessere Überlebenschancen haben. Gegen Minute 750 gibt es ein rapides Wachstum an Swimbots, was darauf hindeutet, dass sich zu diesem Zeitpunkt mehrheitlich Swimbots mit sehr geringem Energiegehalt durchgesetzt haben.

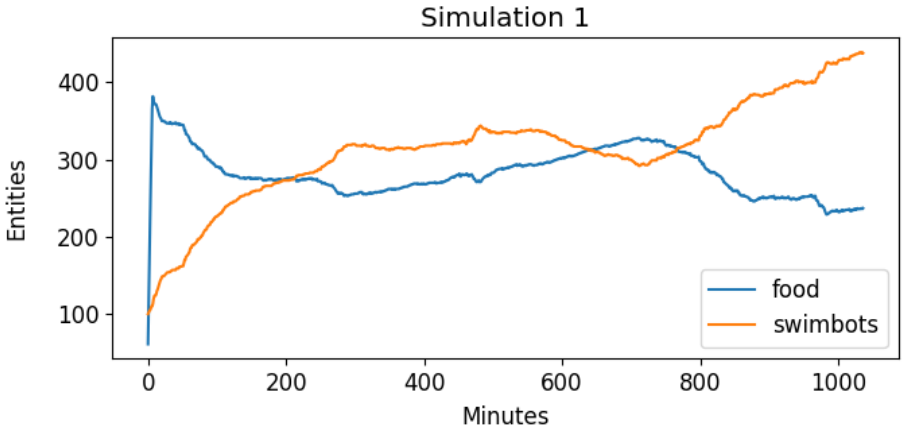
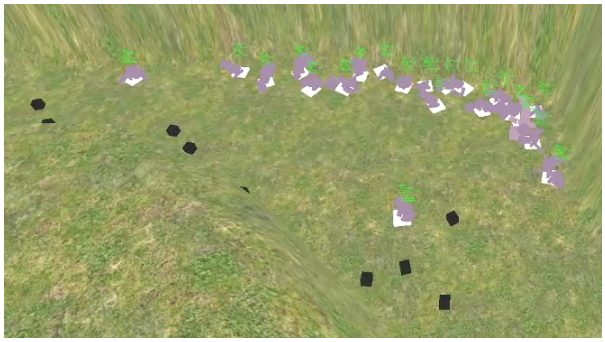


Abbildung 6.1: Zeitlicher Verlauf des Entitätenwachstums für die Simulation 1.

6.2.1.1 Verhalten und Struktur

Im Laufe der Simulation haben Swimbots eine Reihe verschiedener Verhaltensweisen entwickelt. Die meisten zeigten sich als unzulänglich, und diejenigen Swimbots sind dann ausgestorben. Es gibt allerdings einige Populationen die aufgrund ihrer Reproduktionsfähigkeiten, genug Zeit hatten erfolgreiche Bewegungsverhalten zu entwickeln. Besonders aufgefallen ist eine der dominierenden Arten, welche im Laufe des Evolution eine Art Schwingungsbewegung entwickelt hat, um den gesamten Körper voranzutreiben. Diese ist in Abbildung 6.2a abgebildet.

(a) Swimbot mit der evolvierten Eigenschaft sich mithilfe schwunghafter Ganzkörperdynamiken zu bewegen.



(b) Population von Swimbots gehäuft in einer Region.

Abbildung 6.2: Swimbots

6.2.2 Zweiter Durchlauf

Für den zweiten Durchlauf wurde die Simulation 33 Stunden laufen gelassen. Dabei wurde in der Startkonfiguration die Kraft-Masse-Beziehung verringert. Die Anzahl der Körperzeichen für die Erzeugung des Körperbaums, wurde jedoch im Vergleich zum ersten Durchlauf verdoppelt - von 40 auf 80 Zeichen Länge. Ziel war es zu testen, wie die Swimbots durch die erhöhte Anzahl der Glieder jedoch einer jeweils verringerten Kraftauswirkung sich im Vergleich zum erste Durchgang entwickeln werden. Hierbei konnte man eine interessante Beobachtung machen.

Listing 6.2: Startkonfiguration für den ersten Simulationsdurchlauf

```
{
        "startBots" : 100,
        "energyPerFood" : 200,
        "energyToDecreaseRatio" : 1e8,
        "maxWorldEnergy" : 100000,
        "foodSpawnInterval" : 1,
        "maxTimeDilation" : 10,
        "ratioMassToForce" : 23000000,
        "maxBodyStringSize" : 80,
        "SpawnAreaXMax": 9500,
        "SpawnAreaXMin": -9500,
        "SpawnAreaYMax" : 9500,
        "SpawnAreaYMin"  : -9500,
        "SpawnAreaZBot" : 300,
        "SpawnAreaZFood" : 1000
}
```

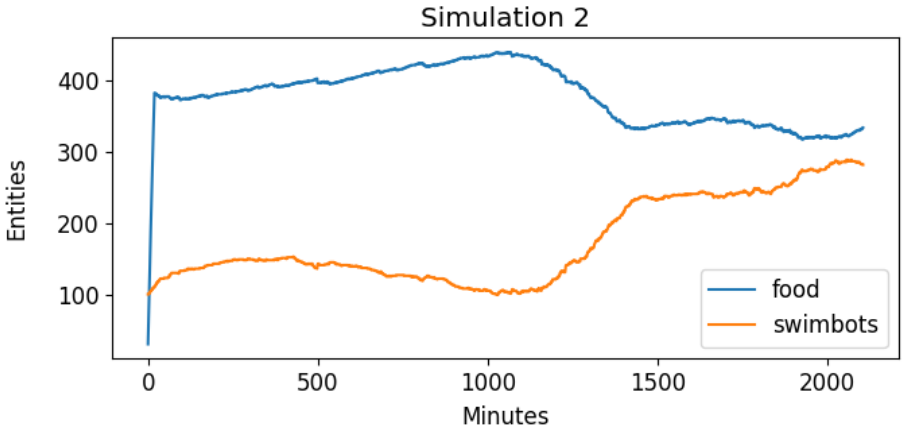


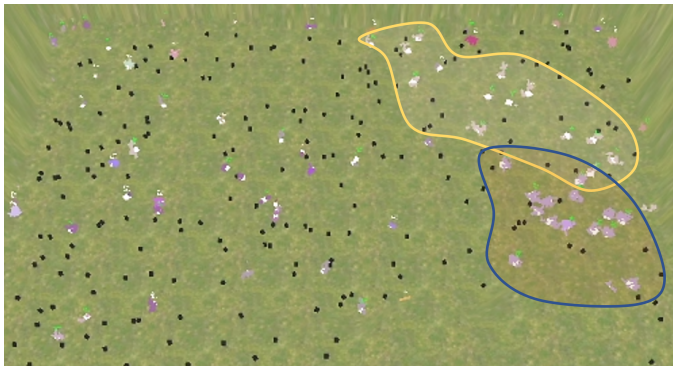
Abbildung 6.3: Zeitlicher Verlauf des Entitätenwachstums für die Simulation 2.

Hier haben sich im Bezug auf die Reproduktionsfähigkeit anfangs zwei Arten besonders dominant gezeigt, wie in der Abbildung 6.4a deutlich zu erkennen ist. Alle Individuen dieser zwei Arten konnten eine ähnliche Körperstruktur vorweisen, welche nicht aus vielen Gliedern bestand. Eine übermäßige Anzahl an Gliedern hat sich nicht vorteilhaft für das Überleben gezeigt, da der übermäßige Verbrauch an Energie nicht durch erfolgreichere Nahrungsfindung mittels einer besseren Bewegung kompensiert werden konnte. Da bei der zufälligen Körpererzeugung, die Konstruktion von effektiven Körperstrukturen nicht berücksichtigt wird, ist die Wahrscheinlichkeit gering, dass zum Start Swimbots mit optimalen Körpereigenschaften in die Welt gesetzt werden.

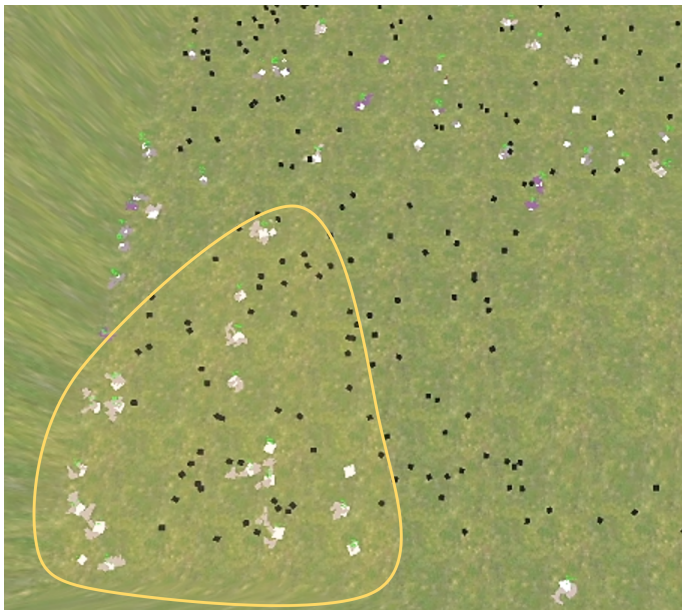
Der in 6.3 abgebildete Anstieg der Swimbots (gegen Minute 1200) ist der Art A zu verdanken, welche ein rasantes Wachstum als Eigenschaft besitzt. Zusätzlich dazu, besitzen sie die Fähigkeit sich erfolgreich fortzubewegen, wodurch Sie ungehindert auch die verschiedenen unbesuchten

Nahrungsgebiete erreichen konnten, wie in der Abbildung 6.4 verdeutlicht wird.

(a) Swimbot mit der Eigenschaft sich mithilfe schwunghafter Ganzkörperdynamik zu bewegen.



(b) Die zwei stark vermehrenden Arten. Die gelb umkreist Art *A* und die blau umkreist Art *B*



(c) Die Art *A* aus 6.4b hat sich am unteren Eck der Landschaft durch den Überschuss an Nahrung vermehren können.

Abbildung 6.4: Swimbots

6.2.3 Durchläufe mit Aussterben

In vielen Simulationsläufen sind die Swimbots nach einiger Zeit ausgestorben. Teilweise nach etlichen Stunden. Dies kann mehrere Gründe haben, von welchen einige bereits in den vorherigen Experimenten beschrieben wurden. Im folgenden wird auf zwei Simulationsdurchläufe eingegangen, welche beide das Aussterben der Kreaturen zur Folge hatten.

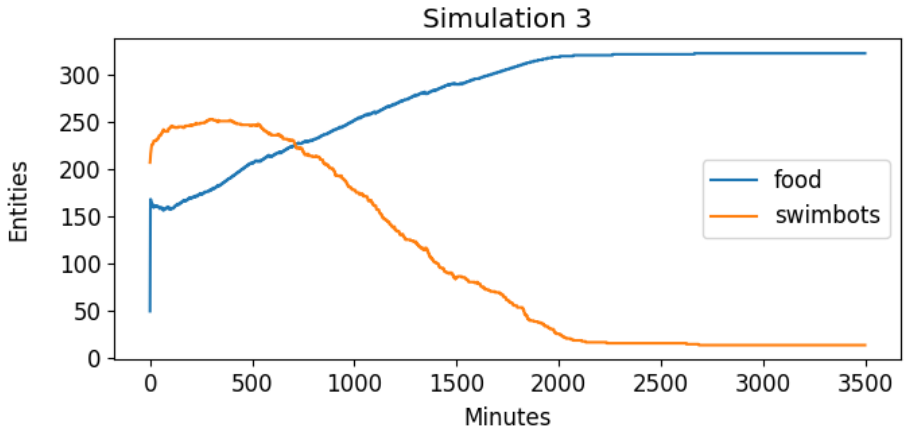


Abbildung 6.5: Zeitlicher Verlauf des Entitätenwachstums für die Simulation 3.

Im dritten Durchlauf hat sich einen Anstieg der Swimbots zu Anfang gezeigt. Die Anzahl hat sich allerdings über einen längeren Zeitraum hinweg reduziert. Die Swimbots haben sich nicht mit der spezifizierten Vorkonfiguration an die Umgebung anpassen können. Der kurze anfängliche Zuwachs an Swimbots, ist auf die Erzeugung initialer Nahrungsobjekte zurückzuführen. Da diese von Oben herab auf die Swimbots fallen, konnten einige durch den neuen Energiegehalt reproduzieren. Allerdings haben sich auch die Nachkommen als nicht überlebensfähig erwiesen, trotz des Überschusses an Nahrungsobjekten, welche in der Umgebung verteilt waren. Die entwickelten Bewegungsbeschaffenheiten waren nicht erfolgreich genug, was dazu führte, dass die Swimbots nicht existenzfähig waren.

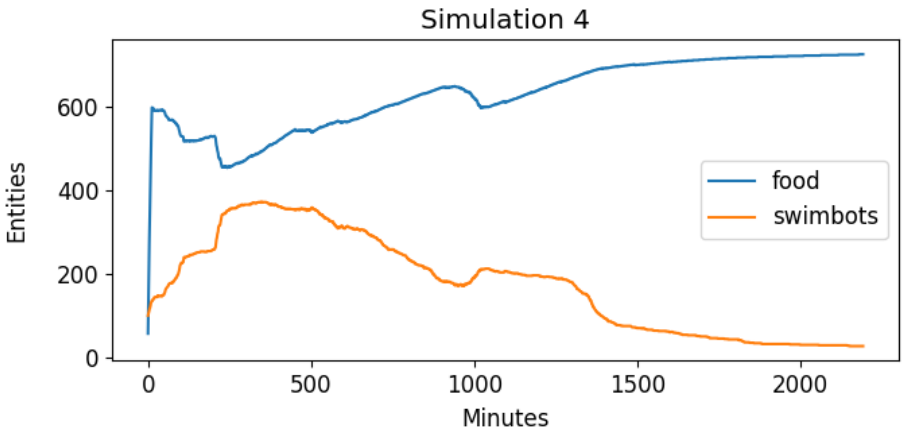


Abbildung 6.6: Zeitlicher Verlauf des Entitätenwachstums für die Simulation 4.

Im vierten Durchlauf allerdings, hat sich ein unterschiedlicher Verlauf ergeben. Ähnlich wie dritten Durchlauf, war anfangs ebenfalls ein Anstieg in der Anzahl der Swimbots zu beobachten. In der Aufnahme des Durchlaufs, konnte man sehen, dass sich eine Art um den Hügel herum angesiedelt hat und dadurch an die dort befindlichen Nahrungsobjekte gelangte. Dabei wurde eine interessante Beobachtung gemacht. Nur einige der Individuen dieser Art, waren in der Lage sich in andere Richtungen hin zu bewegen, um sich dann entsprechend um den Berg herum anzusiedeln. Nachdem der Großteil der Nahrung in diesem Bereich ausgeschöpft war, hat das Aussterben begonnen (ab Minute 300 etwa). Durch die eingeschränkte Bewegungsfähigkeit einzelner Individuen, wurde das Überleben der Art nicht gewährleistet.

6.3 Zusammenfassung

In diesem Abschnitt wurden mehrere Experimente in der Simulationsumgebung näher erläutert. Die Ergebnisse zeigen, dass es eine Vielzahl von Faktoren gibt, welche die Simulation und eine erfolgreiche Evolution beeinflussen. Durch die Simulation mit unterschiedlichen Konfigurationen wurde gezeigt, wie stark sich die Simulation beeinflussen lässt und welche Nebeneffekte diese während der Simulation erzeugen. Die Experimente umfassten Populationen von 50 - 100 Individuen, was im Vergleich mit anderen Simulationen virtueller Kreaturen, bei denen mit Populationsgrößen von 500 aufwärts gestartet wurde [PJ08b], eher gering sind. Das Hauptkriterium für ein effizientes Fortbewegungsverhalten, ist das Vorhandensein eines zyklischen Bewegungsmusters, mit welchem eine nachhaltige Fortbewegung möglich ist. Eine Fortbewegung kann stattfinden, wenn die Drehmomente der Gliedmaßen zeitlich aufeinander abgestimmt sind, sodass sich ein dynamischer Zyklus entwickelt welcher zur Fortbewegung führt. Diese Vielfalt an Bewegungsverhalten, konnte in der

Simulation, trotz der geringen Individuenzahl, an einigen Arten beobachtet werden.

7 Diskussion und Ausblick

Im Rahmen der Arbeit wurde die Evolution virtueller Kreaturen auf Basis einer grundlegenden Bewegungsmechanik gezeigt. Es wurde erfolgreich ein Bewegungsapparat entwickelt, mit welchem Evolutionseffekte veranschaulicht werden konnten, auch wenn in simpler Form und daher nicht mit vielseitigen Evolutionseffekten. Die Gründe dafür sind vielseitig und können mit einigen Verbesserungen für spätere Implementierungen gelöst werden:

- Langsames Feedback aus den Evolutionsexperimenten. Es dauerte typischerweise mehrere Stunden, um eine Population von 100 Individuen über viele Generationen zu evolvieren. Diese Simulationen wurden wegen der Rechenverfügbarkeit in der Regel nachts durchgeführt. Damit war unter gegebener Konfiguration, ein Evolutionsdurchlauf pro Tag möglich. Dies hat es erheblich erschwert , das Programm zu verbessern oder zu verfeinern. Für jede vorgenommene Änderung, war ein neues Evolutionsexperiment erforderlich gewesen, um zu überprüfen ob Änderungen eine erfolgreiche Auswirkung auf das Ergebnis haben
- Performancebeschränkte Simulationsbeschleunigung. Physik-Substepping sorgt für eine realistische Physiksimulation, unabhängig von der Bildrate. Da diese jedoch in Unreal Engine auf maximal 16 Substeps beschränkt ist, kann dadurch keine realistische Beschleunigung mehr gewährleistet werden. Dies kann besonders bei schwächeren Rechenkapazitäten der Fall sein. Durch die Beschleunigung der Simulation wird das Physik-Substepping maximal ausgeschöpft. Dies macht die Festlegung der maximalen Beschleunigung (mitsamt einer realistischen Physik) von der Rechenperformance abhängig. Ein anderes grundlegendes Performanceproblem hat die Anzahl der Akteure verursacht. Je mehr Swimbots in die Welt gesetzt werden, desto mehr Komponenten gibt es, welche die optimale Performance verringern. Komponenten wie die visuelle Sensorik oder das neuronale Netz, sorgen bei einer Vielzahl von Akteuren zu einer hohen CPU-Auslastung.
- Mutation und Aufbau des Körperbaums. Der Aufbau des Körperbaums und dessen Mutation durch die Reproduktion, sind durch dessen grundlegende Beschaffenheit (Körpereigenschaften) und den Mutationsoperator beschränkt. Die Art und Weise wie Körperteile mutiert werden können, z.B. ob gespiegelte Knoten aus vorhandenen Körperteilen erzeugt werden können, ist

simpel gestaltet. Die zufällige Konstruktion des Körperbaums ist aufgrund der fehlenden Performance ebenfalls einfach gehalten und ohne strukturelle Bedeutsamkeit (keine Spiegelungen, wiederholende Körperteile wie z.B. Arme, Beine). Dies ließe sich mit einem Lindenmayer-System [Kar97] besser lösen.

- Abstimmung der Parameter. Im Modell der Swimbots und des Ökosystems gibt es viele Parameter, wie z.B. das Ausmaß der Schwerkraft, die Festlegung der physikalischen Körpereigenschaften oder die Gelenkkräfte, welche fein abgestimmt werden müssen um eine realistische Simulationsumgebung aufzubauen. Die langsame Rückmeldung der Experimente und der Zeitmangel für diese Art von Feinabstimmung, hat diesen Prozess etwas erschwert.
- Bewegungsverhalten. Das neuronale Netz, welches die Kraftausführung der Glieder steuert ist relativ simpel konstruiert. Anpassungsprozesse bzw. Lernprozesse durch die Evolution angetrieben, finden sehr langsam statt, da nur durch Mutation die Gewichtungen angepasst werden. Man kann somit in 200 oder 300 Generationen keine großen Erfolge erwarten. Eine Verbesserung würde hierbei ein Netzwerk mit Erinnerungsfähigkeiten bieten (z.B. durch Long-Short-Term-Netze), sowie eine verbesserte Sensorik für die bessere Aufnahme von Umgebungsinformationen. Im Bezug auf die Neuroevolution wäre hierbei noch die Strukturveränderung von Netzen durch die Evolution interessant, sodass sich auch z.B. die Anzahl der Eingagsneuronen von Generation zu Generation unterscheiden kann.

Darüber hinaus muss man erwähnen, dass die Integration der Komponenten sowie der genutzten Bibliotheken, ein großes Maaß an Entwicklungsarbeit erforderlich machte. Vor allem weil die Dokumentation die für das Projekt erforderlichen Funktionalitäten zu Wünschen übrig ließ. Auch die Fehlerbearbeitung war aufgrund der häufig spärlichen Fehlerbeschreibungen und der eingeschränkten Fähigkeiten des zugrundeliegenden Unreal Engine C++ Compilers (z.B. durch beschränkte Unterstützung der C++ Standard Library) erschwert.

Zusammenfassend kann man trotzdem sagen, dass durch die Implementierung der geeigneten Bewegungsmechanik und der entsprechenden Festlegung mutationsfähiger Eigenschaften in Körper und Bewegungsverhalten, Evolution simuliert werden konnte. Der Zustandsraum, der von dem genetischen Algorithmus durchsucht wird, ist durch die implizite Fitness und der Simulationsumgebung sehr groß, und nur begrenzt durch den verfügbaren Speicher. Es ist fast unvermeidlich dass die Evolution nur begrenzt demonstriert werden kann, wenn eine kleine Populationsgröße und eine begrenzte Anzahl von Generationen verwendet wird, da die implementierten Anpassungen, sehr langsam zu Stande kommen.

A Glossar

Koevolution Beschreibt einen Prozess wechselseitiger evolutionärer Veränderungen, der zwischen Individuen oder Population im Zuge ihrer Interaktion stattfindet. Die Individuen, üben durch ihr Verhalten einen gegenseitigen Selektionsdruck aus.

Allel Eine Variante eines Gens, welche für die Merkmale eines Individuums verantwortlich ist.

Allopatrische Artbildung Beschreibt den Bildungsprozess mehrerer neuer Arten aus einer Ursprungsart. Populationen derselben Art können aufgrund geografischer Veränderungen voneinander isoliert werden, wodurch sich Populationen zu verschiedenen Arten entwickeln.

Allometrie Messung und Vergleich von Beziehungen der Körpergrößen und dem Verhältnis zu verschiedenen biologischen Größen.

Chromosom (genetische Algorithmen) Der Begriff Chromosom bezieht sich im Bereich der GAs in der Regel auf einen Lösungsvorschlag für ein Problem, welches als Bit-String kodiert ist. Die *Gene* sind entweder einzelne Bits oder kurze Blöcke von benachbarten Bits, die ein bestimmten Element des Lösungskandidaten kodieren.

Crossover Crossover (dt. Kreuzung) ist in evolutionären Algorithmen ein genetischer Operator zur Kombination von genetischen Informationen zweier Elternteile. Es bietet eine stochastische Methode zur Erzeugung neuer Lösungsvorschläge für eine Population und ist dem Crossover der sexuellen Fortpflanzung nachgeahmt.

Gendrift Eine Gendrift (auch Sewall-Wright-Effekt genannt [Jos99]) ist die Veränderung der Häufigkeit einer vorhandenen Genvariante in einer Population aufgrund des Zufalls.

Genfrequenz Die Genfrequenz ist die relative Häufigkeit eines Allels, also einer Variante eines Gens, in einer Population. Der Anteil aller Chromosomen in der Population, die dieses Allel tragen, bezogen auf die Gesamtpopulation oder die Stichprobengröße.

Generation Unter einer Generation versteht man eine Iteration des evolutionären Zyklus eines GA. Jede Generation besteht aus einer Population von Individuen, und jedes Individuum stellt einen Punkt im Suchraum und eine mögliche Lösung dar.

Mikroevolution Mikroevolution beschreibt den Prozess der Veränderung von Genvarianten, die im Laufe der Zeit innerhalb einer Population auftritt. Anders als bei der Evolution bezieht sich die Mikroevolution lediglich auf Veränderungen innerhalb einer Population.

Mutation Die Mutation beschreibt in GAs das Umdrehen von Bits (Bit-Strings) an zufälliger Position. Bei größeren Alphabeten (Zeichenketten), wird ein Symbol durch ein zufällig ausgewähltes Symbol ersetzt.

Mesh Darunter versteht man einen Körper in Grafik-Engines.

B Quellcode und Grafiken

Listing B.1: Plugin Spezifikation *Boost.uplugin* für die Integration der Boost Bibliothek.

```
{
        "FileVersion" : 3,
        "Version" : 1,
        "VersionName" : "1.0",
        "FriendlyName" : "Boost",
        "Description" : "",
        "Category" : "",
        "CreatedBy" : "",
        "CreatedByURL" : "",
        "DocsURL" : "",
        "MarketplaceURL" : "",
        "SupportURL" : "",
        "EnabledByDefault" : false,
        "CanContainContent" : false,
        "IsBetaVersion" : true,
        "Installed" : false,
        "Modules" :
        [
    {
      "Name": "Boost",
      "Type": "Runtime",
      "LoadingPhase": "Default"
    }
        ]
}
```

Listing B.2: Build Konfiguration für das Boost Plugin.

```
using UnrealBuildTool;
using System.IO;

public class Boost : ModuleRules
{
        public Boost(ReadOnlyTargetRules Target) : base(Target)
        {

                Type = ModuleType.External;

                // For boost::
                bEnableUndefinedIdentifierWarnings = false;
                bUseRTTI = true;
                bEnableUndefinedIdentifierWarnings = false;

```

```
16                  // Pfadspezifikation der Header Quelldateien
17                  PublicIncludePaths.Add(Path.Combine(ModuleDirectory,
                        "includes"));
18
19                  // Compiler-Definitionen für dieses Modul
20                  PublicDefinitions.Add("_CRT_SECURE_NO_WARNINGS=1");
21                  PublicDefinitions.Add("BOOST_DISABLE_ABI_HEADERS=1")
                        ;
22                  PublicDefinitions.Add("WITH_BOOST_BINDING=1");
23
24                  // Laden der vorkompilierten statischen Boost-
                        Bibliotheken
25                  PublicAdditionalLibraries.Add(Path.Combine(
                        ModuleDirectory, "lib/libboost_chrono-vc143-mt-
                        x64-1_78.lib"));
26                  PublicAdditionalLibraries.Add(Path.Combine(
                        ModuleDirectory, "lib/libboost_container-vc143-mt
                        -x64-1_78.lib"));
27                  PublicAdditionalLibraries.Add(Path.Combine(
                        ModuleDirectory, "lib/libboost_date_time-vc143-mt
                        -x64-1_78.lib"));
28                  PublicAdditionalLibraries.Add(Path.Combine(
                        ModuleDirectory, "lib/libboost_filesystem-vc143-
                        mt-x64-1_78.lib"));
29                  PublicAdditionalLibraries.Add(Path.Combine(
                        ModuleDirectory, "lib/libboost_graph-vc143-mt-x64
                        -1_78.lib"));
30                  PublicAdditionalLibraries.Add(Path.Combine(
                        ModuleDirectory, "lib/libboost_iostreams-vc143-mt
                        -x64-1_78.lib"));
31                  PublicAdditionalLibraries.Add(Path.Combine(
                        ModuleDirectory, "lib/libboost_json-vc143-mt-x64
                        -1_78.lib"));
32                  PublicAdditionalLibraries.Add(Path.Combine(
                        ModuleDirectory, "lib/libboost_math_c99f-vc143-mt
                        -x64-1_78.lib"));
33                  PublicAdditionalLibraries.Add(Path.Combine(
                        ModuleDirectory, "lib/libboost_math_tr1f-vc143-mt
                        -x64-1_78.lib"));
34                  PublicAdditionalLibraries.Add(Path.Combine(
                        ModuleDirectory, "lib/libboost_numpy38-vc143-mt-
                        x64-1_78.lib"));
35                  PublicAdditionalLibraries.Add(Path.Combine(
                        ModuleDirectory, "lib/libboost_system-vc143-mt-
                        x64-1_78.lib"));
36                  PublicAdditionalLibraries.Add(Path.Combine(
                        ModuleDirectory, "lib/libboost_thread-vc143-mt-
                        x64-1_78.lib"));
37
38          }
39  }
```

Listing B.3: Projektdatei für das KIP Projekt.

```
1   {
2           "FileVersion": 3,
```

```
3         "EngineAssociation": "5.0",
4         "Category": "",
5         "Description": "",
6         "Modules": [
7                 {
8                         "Name": "ReWriteKipEvo",
9                         "Type": "Runtime",
10                        "LoadingPhase": "Default",
11                        "AdditionalDependencies": [
12                                "Engine",
13                                "AIModule"
14                        ]
15                }
16        ],
17        "Plugins": [
18                {
19                        "Name": "KantanCharts",
20                        "Enabled": true,
21                        "MarketplaceURL": "com.epicgames.launcher://
                             ue/marketplace/content/89
                             bd18db011b4bd19606db2e31020f13"
22                },
23                {
24                        "Name": "Bridge",
25                        "Enabled": true,
26                        "SupportedTargetPlatforms": [
27                                "Win64",
28                                "Mac",
29                                "Linux"
30                        ]
31                }
32        ]
33 }
```

Listing B.4: USwimbotMeshComponent: Konfiguration im Konstruktor, sowie Mitgliedsfunktion zur Farbsetzung.

```
1  #include "SwimbotMeshComponent.h"
2
3  /*
4   *      Basiskonfiguration für alle starren Körper.
5   */
6  USwimbotMeshComponent::USwimbotMeshComponent() {
7          static ConstructorHelpers::FObjectFinder<UStaticMesh>
              CubeAsset(TEXT("StaticMesh'/Game/Geometry/Meshes/1M_Cube
              .1M_Cube'"));
8          this->SetStaticMesh(CubeAsset.Object);
9          this->SetSimulatePhysics(true);
10         this->SetMobility(EComponentMobility::Movable);
11         this->SetVisibility(true);
12         this->SetGenerateOverlapEvents(true);
13         this->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics
              );
14         this->SetCollisionObjectType(ECollisionChannel::
              ECC_PhysicsBody);
15         this->SetCollisionResponseToAllChannels(ECollisionResponse::
```

```
            ECR_Block);
16          this->SetNotifyRigidBodyCollision(true);
17          this->SetLinearDamping(0.f);
18  }
19
20  /*
21   *      Setzt die Farbe des Körpers durch einen RGB Vektor
22   */
23  void USwimbotMeshComponent::SetMeshColor(FVector color)
24  {
25          UMaterialInterface* Material = this->GetMaterial(0);
26          UMaterialInstanceDynamic* DynMaterial =
               UMaterialInstanceDynamic::Create(Material, this);
27          DynMaterial->SetVectorParameterValue(FName(TEXT("Color")),
               FLinearColor(color));
28          this->SetMaterial(0, DynMaterial);
29  }
```

Listing B.5: Implementierung der `getRandomDistribution()` Funktion.

```
1   float UGenotypeComponent::getRandomDistribution(float distRandVal) {
2           distRandVal = std::fmod(distRandVal * randDistributionMax,
               randDistributionMax);
3           if (distRandVal > 0) {
4                   return (1 - (1 / (1 / abs(distRandVal) + 1)));
5           }
6           else {
7                   return -(1 - (1 / (1 / abs(distRandVal) + 1)));
8           }
9   }
```

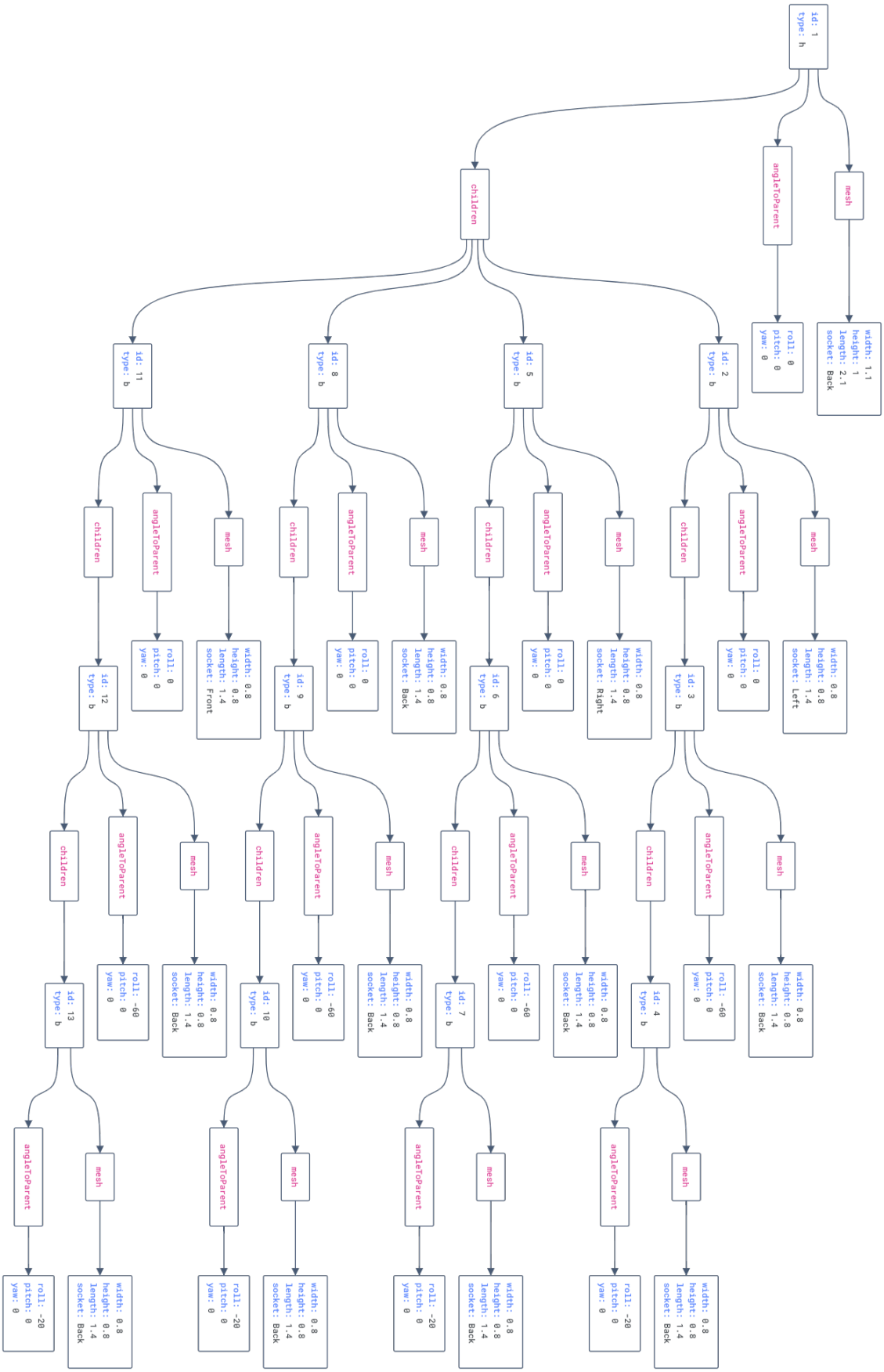


Abbildung B.1: Körperbaumbeispiel, visualisiert mit *jsonvision.com*.

Literaturverzeichnis

[AM06] ARTAMONOVA, V ; MAKHROV, Alexander: [Uncontrolled genetic processes in artificial populations: proving the leading role of selection in evolution]. In: *Genetika* 42 (2006), 04, S. 310–24

[Bar07] BARBER, Sacha: *AI: Dawkins biomorphs / and other evolving creatures*. https://www.codeproject.com/Articles/17387/AI-Dawkins-Biomorphs-And-Other-Evolving-Creatures. Version: Jan 2007

[BS93] BÄCK, Thomas ; SCHWEFEL, Hans-Paul: An overview of evolutionary optimisation for parameter optimisation. In: *Evolutionary Computation* 1 (1993), 01, S. 1–23

[BSM+] BINDER, Alexander ; SAMEK, Wojciech ; MONTAVON, Gregoire ; BACH, Sebastian ; MULLER, Klaus-Robert: *Analyzing and validating neural networks predictions*. https://iphome.hhi.de/samek/pdf/BinICML16.pdf

[Daw03] DAWKINS, Richard: The Evolution of Evolvability. In: *Artificial Life. The Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems* (2003), 12. http://dx.doi.org/10.1016/B978-012428765-5/50046-3. – DOI 10.1016/B978–012428765–5/50046–3. ISBN 9780124287655

[DJFS97] In: DE JONG, Kenneth ; FOGEL, David ; SCHWEFEL, Hans-Paul: *A history of evolutionary computation*. 1997, S. A2.3:1–12

[For] FORTUNER, Brendan: *Logistic regression*. https://ml-cheatsheet.readthedocs.io/en/latest/logistic_regression.html

[Gal20] GALVÁN, Mooney: Neuroevolution in Deep Neural Networks: Current Trends and Future Challenges, 2020

[GG01] GRASSHOFF, Manfred ; GUDO, Michael: *The evolution of animals – poster with explanations*. 2001. – 16 p. S. – ISBN 3–510–61324–4

[gita] https://github.com/

[gitb] https://git-scm.com/

[HG22] HART, Emma ; GOFF, Léni: Artificial evolution of robot bodies and control: on the interaction between evolution, learning and culture. In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 377 (2022), 01. http://dx.doi.org/10.1098/rstb.2021.0117. – DOI 10.1098/rstb.2021.0117

[HHCM97] HUSBANDS, P. ; HARVEY, I. ; CLIFF, D. ; MILLER, G.: Artificial Evolution: A New Path for Artificial Intelligence? In: *Brain and Cognition* 34 (1997), Nr. 1, 130-159. http://dx.doi.org/https://doi.org/10.1006/brcg.1997.0910. – DOI https://doi.org/10.1006/brcg.1997.0910. – ISSN 0278–2626

[Hol94] HOLLAND, J.H.: Adaptation in natural and artificial systems. In: *An Introductory Analysis with Application to Biology, Control and Artificial Intelligence* (1994), 01

[IB21] I. BOERSCH, K. Lindner F. Spitz P. Grigarzik M. A. A. Fodi F. A. Fodi: Die Expedition zu den Evolutionsinseln. (2021)

[ISPM20] ITU, Călin ; SCUTARU, Maria-Luminiţa ; PRUNCU, Cătălin I. ; MUNTEAN, Radu: Kinematic and Dynamic Response of a Novel Engine Mechanism Design Driven by an Oscillation Arm. In: *Applied Sciences* 10 (2020), Nr. 8. http://dx.doi.org/10.3390/app10082733. – DOI 10.3390/app10082733. – ISSN 2076–3417

[Joh11] JOHANNSEN, W.: The genotype conception of heredity. In: *The American Naturalist* 45 (1911), Nr. 531, S. 129–159. http://dx.doi.org/10.1086/279202. – DOI 10.1086/279202

[Jos99] JOSHI, Amitabh: The Shifting Balance Theory of Evolution. In: *Resonance* 4 (1999), 12, S. 66–75. http://dx.doi.org/10.1007/BF02838675. – DOI 10.1007/BF02838675

[Kar97] KARI, Grzegorz Salomaa A. Rozenberg: L Systems. (1997), 01

[KB06] In: KUCUK, Serdar ; BINGUL, Z.: *Robot Kinematics: Forward and Inverse Kinematics*. 2006. – ISBN 3–86611–285–8

[Koz92] KOZA, John R.: *Gentic Programming, On the programming of Computers by means of natural selection*. Cambridge, Massachussetts, London, England : MIT Press, 1992

[Kri19] KRIEGMAN, S.: Why virtual creatures matter. In: *Nature Machine Intelligence* (2019)

[lei] *Kraft und bewegungsänderung*. https://www.leifiphysik.de/mechanik/kraft-und-bewegungsaenderung

[LFLL+21] LAI, Gorm ; FOL LEYMARIE, Frederic ; LATHAM, William ; ARITA, Takaya ; SUZUKI, Reiji: Virtual Creature Morphology - A Review. In: *Computer Graphics Forum* 40

(2021), 05, S. 659–681. http://dx.doi.org/10.1111/cgf.142661. – DOI 10.1111/cgf.142661

[LP00] LIPSON, Hod ; POLLACK, Jordan: Automatic design and manufacture of robotic lifeforms. In: *Nature* 406 (2000), 09, S. 974–8. http://dx.doi.org/10.1038/35023115. – DOI 10.1038/35023115

[Mal20] MALLAWAARACHCHI, Vijini: Introduction to genetic algorithms - including example code. In: *Medium* (2020), Mar. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

[Mil88] MILLER, Gavin: The motion dynamics of snakes and worms, 1988, S. 169–173

[NRTAJ19] NICHOL, Daniel ; ROBERTSON-TESSI, Mark ; ANDERSON, Alexander R. ; JEAVONS, Peter: Model genotype–phenotype mappings and the algorithmic structure of evolution. In: *Journal of The Royal Society Interface* (2019). http://dx.doi.org/10.1098/rsif.2019.0332. – DOI 10.1098/rsif.2019.0332

[phy] *Physiksimulation Endego*. https://endego.com/de/software-sales/Physiksimulation~cse102

[PJ08a] PILAT, Marcin ; JACOB, Christian: Creature Academy: A System for Virtual Creature Evolution, 2008, S. 3289 – 3297

[PJ08b] PILAT, Marcin ; JACOB, Christian: Creature Academy: A System for Virtual Creature Evolution, 2008, S. 3289 – 3297

[PKWT] PFEIFER, Rolf ; KUNZ, Hanspeter ; WEBER, Marion M. ; THOMAS, Dale: *Artificial life - Chapter 1.2 Natural and artificial life*. https://www.ais.uni-bonn.de/SS09/skript_artif_life_pfeifer_unizh.pdf

[plu] *Plugins*. https://docs.unrealengine.com/5.0/en-US/plugins-in-unreal-engine/

[Por16] PORTELLI, Giuseppe: *Unreal-Engine-Substepping*. http://www.aclockworkberry.com/unreal-engine-substepping/. Version: 2016

[Rab16a] RABBANI, Amir H.: *Physically based kinematic editing and dynamic balance control for character animation*, Diss., 07 2016. http://dx.doi.org/10.13140/RG.2.2.17434.03527. – DOI 10.13140/RG.2.2.17434.03527

[Rab16b] RABBANI, Amir H.: *Physically based kinematic editing and dynamic balance control for character animation*, Diss., 07 2016. http://dx.doi.org/10.13140/RG.2.2.17434.03527. – DOI 10.13140/RG.2.2.17434.03527

[Sha21] SHARMA, Sagar: Activation functions in neural networks. In: *Medium* (2021), Jul. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

[Sim94] SIMS, Karl: Evolving Virtual Creatures. In: *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA : Association for Computing Machinery, 1994 (SIGGRAPH '94). – ISBN 0897916670, 15–22

[SK92] SCHWEFEL, Hans-Paul ; KURSAWE, Frank: Künstliche Evolution als Modell für natürliche Intelligenz, 1992. – ISBN 978–3–519–06195–3

[uni] *Inverse Kinematics in Unity*. https://docs.unity3d.com/Manual/InverseKinematics.html

[unr] *Inverse Kinematics in Unreal Engine*. https://docs.unrealengine.com/4.27/en-US/AnimatingObjects/SkeletalMeshAnimation/IKSetups/

[Ven05] VENTRELLA, Jeffrey: GenePool: Exploring the Interaction Between Natural Selection and Sexual Selection. In: ADAMATZKY, Andrew (Hrsg.) ; KOMOSINSKI, Maciej (Hrsg.): *Artificial Life Models in Software*. London : Springer London, 2005, S. 81–96

[VP09] V. POPOV, Prof. D. n.: Kinematik und Dynamik (Mechanik II). (2009). http://www.tu-berlin.de/uploads/media/Vorlesungsnotizen_MeII.pdf

[WH22] WINTERS, Sandra ; HIGHAM, James: Simulated evolution of mating signal diversification in a primate radiation. In: *Proceedings. Biological sciences* 289 (2022), 06, S. 20220734. http://dx.doi.org/10.1098/rspb.2022.0734. – DOI 10.1098/rspb.2022.0734

[zop] *Zusammenspiel zwischen peripherer und fovealer wahrnehmung*. http://vmrz0100.vm.ruhr-uni-bochum.de/spomedial/content/e866/e2442/e8554/e8574/e8610/e8662/index_ger.html

Abbildungsverzeichnis

Tabellenverzeichnis