Master's Thesis

Fachhochschule Gelsenkirchen

Fachbereich Informatik
Studiengang Angewandte Informatik

Vertiefungsrichtung:
Medien und Mensch-Computer-Kommunikation

Hüpfen, krabbeln und Billard spielen

Entwicklung autonomer Fortbewegung virtueller Charaktere in einer simulierten physikalischen Umgebung mit Hilfe neuronaler Netzwerke und evolutionärer Strategien

Evolving autonomous locomotion of virtual characters in a simulated physical environment via neural networks and evolutionary strategies

Stefan Marks
MatrNr.: 200210119

26. September 2005

Echtheitserklärung

Hiermit versichere ich, die Arbeit selbstständig angefertigt und keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt zu haben.

Ort, Datum	Unterschrift

Diese Masterarbeit ist ein Prüfungsdokument. Eine Verwendung zu einem anderen Zweck ist nur mit dem Einverständnis von Verfassen und Prüfern erlaubt.

Zusammenfassung

Die Animation virtueller Charaktere ist ein Prozess, der trotz vielerlei Unterstützung durch Software und Hardware aufwändig bleibt und in Kleinarbeit ausarten kann, wenn die zu animierende Figur sehr kompliziert und/oder detailliert ist.
Der hier vorgestellte Ansatz versucht, diesen Prozess zu automatisieren bzw. zu unterstützen, indem die Figur den Bewegungsvorgang autonom „lernt“.
Dazu werden zunächst zusätzlich zu den optischen die physikalischen Eigenschaften des Charakters (z.B. Masse, Trägheitsmomente, Gelenke, Freiheitsgrade) definiert, damit dieser mit einer simulierten physikalischen Umgebung interagieren kann. Im nächsten Schritt legt man fest, welche Sensoren (z.B. für Druck, Kräfte, Winkel, Geschwindigkeiten) und Aktoren (z.B. Motoren, „Muskeln“, Dämpfungselemente) der Charakter besitzt. Die Sensoren und Aktoren werden im dritten Schritt mit den Ein- und Ausgängen eines neuronalen Netzwerks verbunden, dessen Verbindungsgewichte und Schwellwerte noch uninitialisiert sind. Durch evolutionäre Strategien wird nun versucht, diese Werte so einzustellen, dass der Charakter sich möglichst natürlich in seiner physikalischen Umgebung bewegt.

Abstract

The animation of virtual characters is a process that although supported by various software and hardware can be tedious and costly especially when the character to animate is very complicated and/or detailled.
The method presented in this thesis tries to automatize or to support this process by letting the character „learn“ its movements autonomously.
This is established by first modelling physical properties (e.g. mass, inertia moments, joints, degrees of freedom) additionally to the optical ones to allow the interaction of the character with a simulated physical environment. In the second step the sensors (e.g. pressure, forces, angles, speed) and actors (e.g. motors, „muscles“, suspension elements) that the character uses are defined. Third the sensors and actors are connected with the inputs and outputs of a neural network whose bias values and link weights are still uninitialized. These values are then modified by evolutionary strategies to find naturally looking movements of the character in its physical environment.

Danksagung

Ich danke Herrn Prof. Dr. G. Lux und Herrn Prof. Dr. W. Conen für ihre Unterstützung bei dieser Thesis, für ein angenehmes Lern- und Diskussionsklima und für die zahlreichen Denkanstöße und Materialien.

Ebenfalls danken möchte ich Herrn Dipl.-Ing. T. Kollakowsky für die Unterstützung mit Hard- und Software und die konstruktiven Diskussionen über plattformübergreifende Programmierung.

Meine Hochachtung und Anerkennung gilt meiner Frau Kathrin. Ohne ihre Rückendeckung und Motivationsschübe sowie den Mut und den Elan, sich in TEX einzuarbeiten und meine Fehler direkt im Quellcode zu korrigieren, hätte die Fertigstellung dieser Thesis wesentlich mehr Zeit benötigt.

Ich möchte auch der Firma ET ELECTROTECHNOLOGY GmbH danken, die mir mit den flexiblen Arbeitszeiten überhaupt erst das Masterstudium ermöglicht hat.
Ebenfalls ein herzliches Danke geht auch an Herrn Dipl.-Ing. Heck für sein Interesse an dieser Thesis, für die Diskussionen und gemeinsam erarbeiteten Lösungsansätze.

Keinesfalls unerwähnt und ebenfalls mit Lob bedacht seien Jutta K., Kai F., Kai S., Michael L. und ein mir namentlich unbekannter Arzt. Ohne Menschen wie sie, die sich trotz des umfangreichen Werkes bereit erklärt haben, die 250 Seiten Korrektur zu lesen, hätte der Fehlerteufel viel mehr Schaden angerichtet, als es nun der Fall ist.

Nomenklatur

In der Thesis werden folgende Formelsymbole verwendet:

- **3D-Vektoren** werden in Kleinbuchstaben notiert und mit einem Pfeil gekennzeichnet.

$$\vec{v} = \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix}$$

- **Quaternionen** werden in Kleinbuchstaben notiert, mit einem Pfeil gekennzeichnet und nicht schräggestellt gesetzt.

$$\vec{\mathrm{q}} = \begin{pmatrix} 0 \\ 0.3\mathsf{i} \\ 0.4\mathsf{j} \\ 0.5\mathsf{k} \end{pmatrix}$$

- **Matrizen** werden in Großbuchstaben notiert und nicht schräggestellt gesetzt.

$$\mathrm{R} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- **imaginäre Variablen** werden in serifenloser Schrift gesetzt.

$$\mathsf{i}, \mathsf{j}, \mathsf{k} \in \mathbb{C}$$

Inhaltsverzeichnis

„Fange beim Anfang an," sagte der König ernsthaft, „und lies, bis du an's Ende kommst, dann halte an."

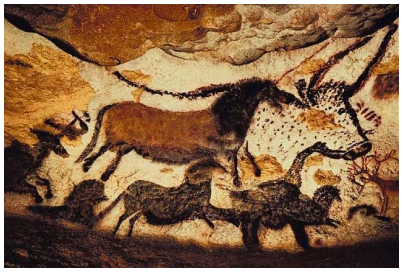
Lewis Carroll, *Alice im Wunderland*

1 Einleitung

1.1 Geschichte der Animation

Die Animation (lat. „animatio" = „Belebung") künstlicher und virtueller Charaktere ist ein umfassendes Gebiet, welches schon Generationen von Künstlern und Wissenschaftlern beschäftigt hat. Ein zeitlicher Abriss der Geschichte der Animation bis zum heutigen Zeitpunkt soll einen Überblick verschaffen [Film Education, 2005; McLaughlin, 2001]:

Am Anfang standen **Höhlenmalereien**, die ca. *18000 v.Chr.* angefertigt wurden. Nach Ansicht der Forscher wurden sie so gemalt, dass sie sich im flackernden Licht des Feuers bewegten.
(siehe Farbtafel D.1(a) auf Seite 223)


[1]

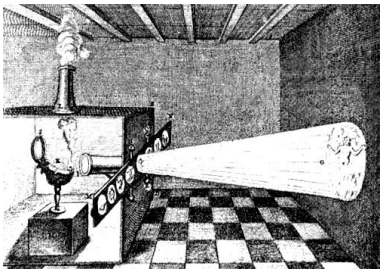

[2]

Um *7000 v.Chr.* entstand in China das **Schattentheater**, bei welchem der Schatten von beweglichen Figuren auf eine Pergamentleinwand geworfen wird. Diese Technik wurde in östlichen Ländern, speziell Indien, zu einer heute noch praktizierten Kunstform ausgeweitet.
(siehe Farbtafel D.1(b) auf Seite 223)


[3]

Auf Vasen, die aus dem *2. Jahrtausend v.Chr.* stammen, findet man Zeichnungen von Menschen, Tieren und Fahrzeugen in unterschiedlichen Stadien der Bewegung.
(siehe Farbtafel D.1(c) auf Seite 223)

[1] http://www.sanford-artedventures.com/study/images/cave_painting_1.jpg
[2] http://www.doll.at/image1/china.jpg
[3] http://www.ceramicstoday.com/images/odd/olympics.jpg

 [4]

1660 wurde die **Laterna Magica** erfunden, die es erlaubt, auf Glasplatten aufgetragene Zeichnungen an eine Wand zu projizieren. Durch Bewegung der Platten konnten so einfache Animationen erzeugt werden.
(siehe Farbtafel D.1(d) auf Seite 223)

Aus der Zeit um *1830* herum stammen Geräte wie etwa das **Zoetrop** oder das **Phenakistiskop**, in welchen eine Reihe von Einzelbildern im Innern eines Zylinders angebracht sind, die durch Schlitze in der gegenüberliegenden Wand betrachtet werden konnten. Versetzte man den Zylinder in Rotation, entstand der Eindruck eines Bewegungsablaufes.
(siehe Farbtafel D.2(a) auf Seite 225)

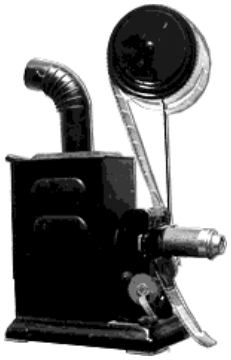
 [5]

 [6]

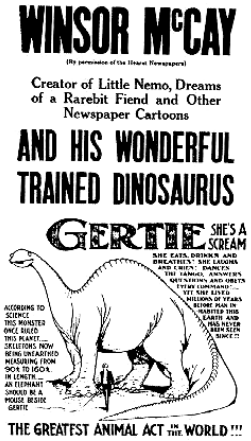
1892 wurde dieses Prinzip von Emile Raynaud durch Spiegel in seiner Wirkung verbessert. Das Resultat ist das sogenannte **Praxinoskop**.
(siehe Farbtafel D.2(b) auf Seite 225)
Aus diesem Zeitraum stammen auch eine Reihe von Maschinen, die nach dem Prinzip des Daumenkinos Einzelbilder in schneller Folge abspielen konnten.

1893 entwickelte Thomas Edison das **Kinetoskop**, welches der Urvater aller folgenden Filmprojektoren geworden ist. Die Einzelbilder waren auf einem Streifen aus **Zelluloid** (entwickelt *1887* von H.W. Goodwin) aufgebracht, der über mehrere Rollen mäanderförmig aufgespannt wurde. Sie wurden in schneller Folge nach dem Prinzip der Laterna Magica über ein Linsensystem auf eine Leinwand projiziert.

[4] `http://www.br-online.de/kultur-szene/thema/animationsfilm/foto/pop_magica.jpg`
[5] `http://web.inter.nl.net/users/anima/optical/zoetrope/zoetrope.gif`
[6] `http://web.inter.nl.net/users/anima/optical/praxino/praxino.gif`


[7]

Luois Lumière verbesserte das Gerät *1894* dadurch, dass der Zelluloidstreifen während der Projektion von einer Rolle auf eine andere Rolle umgespult wurde. Weiterhin verbesserte er die Trennung der Einzelbilder durch einen Shutter-Mechanismus und führte den heutzutage immer noch verwendeten Klauen-Transportmechanismus[8] ein. Lumière nannte dieses Gerät **Cinematograph**.
1899 gelang es erstmals, Ton magnetisch festzuhalten.
(siehe Farbtafel D.2(c) auf Seite 225)



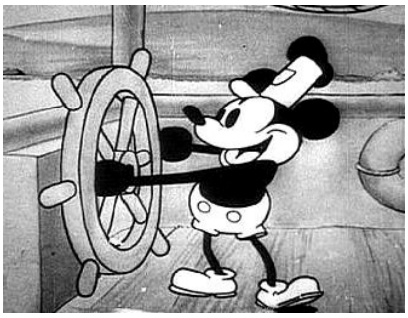
[9]

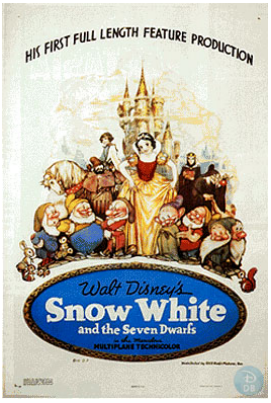
1907 wurde in dem Film „The Haunted Hotel“ erstmals die **Stop-Motion-Technik** eingesetzt. Bei dieser Technik wird die Szene nicht kontinuierlich gefilmt, sondern in Einzelbildern aufgenommen. Zwischen zwei Einzelbildern wird die Szene minimal verändert, um Bewegungen zu simulieren.
1908 entwickelte Emile Cole mit „Fantasmagorie“ nach Ansicht vieler Experten den ersten Animationsfilm überhaupt.
1909 kombinierte er in „Claire de lune“ erstmals reale Geschehnisse mit gezeichneten Animationen.
1914 fertigte Windsor McCay den Animationsfilm „Gertie the Dinosaur“ für die damaligen Lichtspielhäuser an. Dieser Schwarz-Weiß-Stummfilm wurde im Gegensatz zu seinem Erstlingswerk „Little Nemo“ (*1911*) ein echter „Kassenschlager“.
(siehe Farbtafel D.2(d) auf Seite 225)

1915 führte die Erfindung des **Zellulose-Acetats** zu neuen Möglichkeiten. Mit diesem durchsichtigen Material war man nun in der Lage, Zeichnungen in Schichten übereinander zu legen, anstatt jedes Bild inklusive Hintergrund vollständig zeichnen zu müssen.
1922 wurden die **Walt Disney Studios** gegründet.
1925 entstand mit Hilfe der Stop-Motion-Technik der abendfüllende Film „The Lost World“ mit prähistorischen Dinosauriern und anderen Kreaturen.

[7] `http://web.inter.nl.net/users/anima/optical/lantern/lantaarn.gif`
[8] Aus diesem Grund haben Filme die typischen Transportlöcher an den Rändern.
[9] `http://silentmoviemonsters.tripod.com/TheLostWorld/images/Gertiead.jpg`

1928 produzierte Walt Disney den Film „Steamboat Willie“, in welchem Mickey Mouse zum ersten Mal die Hauptrolle „spielte“. Der Film nutzte die Tatsache aus, dass kurz zuvor die Technik zur Kombination von Bild und Ton erfunden wurde, und wurde der erste wirklich erfolgreiche **Tonfilm**. (siehe Farbtafel D.2(e) auf Seite 225)

 [10]


[11]

1930 wurden die **Warner Brothers** gegründet.
1933 lief „King Kong“ in den Kinos und verblüffte mit zu dieser Zeit lebensecht wirkenden Bewegungen des Riesenaffen, die von Willis O'Brien mit Stop-Motion-Technik realisiert wurden.
1937 erschien der erste animierte **Farbfilm** „Snow White and the Seven Dwarves“ in Spielfilmlänge.
(siehe Farbtafel D.2(f) auf Seite 225)

 [12]

1963 erstaunte der Film „Jason and the Argonauts“ das Publikum durch spektakuläre Animationen aus der Hand von Ray Harryhausen [Nostalgia Central, 1998; Howe, 1999]. Für eine Kampfszene von knapp fünf Minuten Länge musste der Schauspieler Todd Armstrong in der Rolle von Jason mit sieben Skeletten kämpfen, welche er während der Dreharbeiten nicht sehen konnte. Diese wurden danach in vier Monaten Arbeit in Stop-Motion-Technik animiert und in den Film kopiert.
(siehe Farbtafel D.3(a) auf Seite 227)


[13]

1980 war das Geburtsjahr der **Computeranimation** mit dem Film „Tron“ aus dem Hause Walt Disney.
([Carlson, 2004], siehe Farbtafel D.3(b) auf Seite 227)
1990 entstand mit „Toy Story“ der erste vollständig aus dem Computer generierte Spielfilm und eröffnete den Triumphzug der **Pixar Animation Studios**.

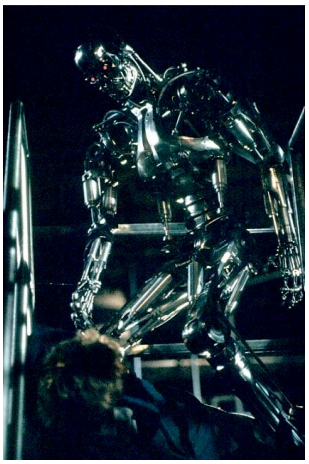
[10] `http://disneyshorts.toonzone.net/years/1928/graphics/steamboatwillie/steamboatwillie1thumb.jpg`
[11] `http://disney.go.com/vault/archives/characters/snow/snow2.jpg`
[12] `http://www.nostalgiacentral.com/images_movie/jason_03.jpg`
[13] `http://www.cybergeography.org/atlas/tron.jpg`

1.2 Computeranimation

Seit „Tron“ ist der Computer aus der Produktion visueller Medien nicht mehr wegzudenken. Die Einsatzgebiete sind zahlreich:

- Animation von Gegenständen, Tieren, Fabelwesen, Menschen und allem, was der Phantasie noch entspringen kann.
- Erschaffung von Welten und Räumen, die mit realen Kulissen nicht oder nur sehr aufwändig zu realisieren sind.
- Postproduktion (Nachbearbeitung, Retusche, Effekte etc.).

Am Beispiel von „Terminator“ (siehe Abbildung 1.1) soll der rasante Fortschritt in dieser Technik gezeigt werden:
Wurde im ersten Teil (1984) für den T-800 noch Stop-Motion-Technik eingesetzt, so entstand die Figur des T-1000 im zweiten Teil („Terminator 2 - Judgement Day“, 1995) schon vollständig im Computer.


[14]

(a) T-800 (Stop-Motion)


[15]

(b) T-1000 (Computeranimation)

Abbildung 1.1: Fortschritt der Animationstechnik am Beispiel der Filme „Terminator“ und „Terminator 2“ (siehe Farbtafel D.3(c) auf Seite 227)

Dabei nimmt die grafische Qualität der erzeugten Bilder ständig zu und erreicht immer mehr fotorealistische Ausmaße. Am Grundprinzip der Animation hat sich allerdings seither nichts geändert. Immer noch muss der zu bewegende Gegenstand für jedes Einzelbild in die entsprechende Position gebracht werden, sei dies nun mit einer leicht veränderten Zeichnung (Trickfilm), mit leicht weitergebogenen Gliedmaßen einer Figur mit einem inneren Metallskelett (Stop-Motion-Technik), mit minimal verformtem Knetgummi oder Ton (Claymation[16]) oder mit transformierten 3D-Koordinaten.

[14] `http://www.imdb.com/gallery/ss/0088247/Terminator_PUB10.jpg`
[15] `http://www.goingfaster.com/darkthoughts/t1000.jpg`
[16] Wird heute noch bei den Filmen der „Wallace and Gromit“-Reihe verwendet.

Der Computer stellt dem Animator im Gegensatz zu den „natürlichen“ Methoden einige Hilfen zur Verfügung:

- **Motion capture** (MoCap) zur Erfassung kompletter Bewegungsabläufe von realen Schauspielern mittels optischer oder mechanischer Sensoren.
- **Frame-Interpolation** zur Berechung flüssiger Übergänge zwischen zwei Schlüsselpositionen.
- Zahlreiche **Tools** zur Animation passiv bewegter Gegenstände (z.B. Fell, Haare, Kleidung, physikalische Animation, Partikeleffekte).
- Hilfestellungen durch **Constraints**, die z.B. verhindern, dass ein Gelenk unnatürlich bewegt wird oder eine Hand den Gegenstand durchdringt, den sie halten soll.

Wenn es allerdings um die Animation natürlich wirkender Bewegungen von Charakteren geht, muss jeder Animator auch heutzutage ein gutes Verständnis und eine gute Beobachtungsgabe für Bewegungsabläufe mitbringen. Denn trotz aller Hilfsmittel ist dieses Gebiet zu komplex und zu vielseitig, um es mit einfachen Mitteln zu erfassen, mathematisch oder anderweitig zu beschreiben zu können. Dadurch kann man es auch nicht in einer Software mit „einer Handvoll Reglern und Knöpfen“ verwalten. Es läuft immer wieder auf die Feinarbeit hinaus, mit der ein Animator z.B. die Fingerspitzen einer humanoiden Figur Einzelbild für Einzelbild korrigiert, die Gesamtanimation betrachtet und dann weitere Feinkorrekturen anbringt, bis er mit dem Ergebnis zufrieden ist[17].

1.3 Grundidee

Der Grundgedanke dieser Thesis ist es, dem Animator die Feinarbeit abzunehmen und durch eine Kontrolle auf einem höheren Niveau zu ersetzen. Mittels Aktivierung von Bewegungsprimitiven wie „Gehe vorwärts“, „Stehe still“, „Steige Stufen“, „Greife Gegenstand“ oder durch stufenlose Kombinationen aus diesen lässt sich der virtuelle Charakter steuern, ohne dass sich der Animator dabei um Details kümmern muss.
Im Gegensatz zu MoCap sollen diese Bewegungsprimitiven durch den virtuellen Charakter selbstständig in einer physikalisch korrekt simulierten Umgebung ausgeführt werden.
Gesteuert wird der Charakter dabei von einem neuronalen Netzwerk, welches sensorische Informationen über den Zustand des „Körpers“ und der gewünschten Bewegungsart erhält und diesen über Aktoren wie Muskeln oder Motoren steuern kann. Damit erhält das neuronale Netz in etwa den Funktionsumfang des Kleinhirns (lat. „Cerebellum“) bei Säugetieren.

[17] Bei Stop-Motion-Animationen ist für eine Sekunde Filmszene nicht selten ein ganzer Tag Arbeit notwendig. Siehe dazu auch den Text zu „Jason and the Argonauts“ auf Seite 4.

Um das neuronale Netzwerk zu trainieren, sind mindestens zwei Ansätze möglich:

1. **Direktes Training** mit Daten, die z.B. durch MoCap gewonnen wurden.
2. Man sucht mittels **evolutionärer Algorithmen** die Parameter des neuronalen Netzes, welche die Aufgaben am besten erfüllen.

Die erste dieser beiden Lösungen hat zwei wesentliche Nachteile:

- Bei Charakteren, welche erheblich andere „Proportionen“ als Menschen oder Tiere aufweisen, ist MoCap nicht durchführbar.
- Der technische Aufwand bzw. die Miete für ein professionelles Studio ist hoch.

Die zweite und in dieser Thesis näher untersuchte Lösung begegnet gerade eben diesen Nachteilen:

- Es sind prinzipiell alle Arten von Charakteren denkbar, die sich physikalisch modellieren lassen.
- Der zeitliche, technische und finanzielle Aufwand zum Betrieb des Computers/des Clusters, welcher die physikalische Simulation durchführt und die evolutionären Algorithmen anwendet, lässt sich an den gegebenen Rahmen anpassen.

Allerdings ist für die zweite Lösung auch ein höheres Wissen über Physik, Neurobiologie und die Anatomie des Charakters notwendig, um z.B. die Struktur des neuronalen Netzes und dessen Verbindungen von den Sensoren und zu den Aktoren festlegen zu können. Dieser Nachteil ließe sich allerdings dadurch kompensieren, dass man durch Untersuchungen auf Regeln, Gesetzmäßigkeiten, Empfehlungen etc. stoßen wird, die sich in vereinfachter oder automatisierter Form dem Benutzer präsentieren lassen.

1.4 Aufgabenstellung

Folgende primäre Aufgaben sind im Verlauf der Thesis zu lösen:

- **Implementierung einer virtuellen Umgebung**
  Mit Hilfe der im VUM-Labor (**V**irtuelle **UM**gebungen) der Fachhochschule Gelsenkirchen vorhandenen Gerätschaften sollen Möglichkeiten geschaffen werden, welche die Immersion in die virtuelle Welt während der Simulationsphasen ermöglicht (z.B. 3D-Projektionsleinwand[18], HMD[19], Tracker[20]).

- **Implementierung einer physikalischen Simulationsumgebung**
  Zur virtuellen Umgebung wird eine physikalische Simulation hinzugefügt, die es ermöglicht, physikalische Primitive (starre Körper, Gelenke etc.) zu erzeugen, zu testen und zu kombinieren. Auch hier sind die technischen Möglichkeiten des Labors auszuschöpfen, um die Interaktion mit der erschaffenen Szenerie zu ermöglichen (z.B. Kombination realer und virtueller Gegenstände).

- **Konstruktion virtueller Charaktere**
  Drei Arten von virtuellen Charakteren sollen modelliert und mit einem neuronalen Netz verbunden werden:

  – Monoped (lat.: „Einfüßler“)
  – Biped (lat.: „Zweifüßler“)
  – Quadruped (lat.: „Vierfüßler“)

  Sowohl die optischen und die physikalischen Eigenschaften als auch die Architektur des neuronalen Netzwerks werden dabei festgelegt.

- **Implementierung des Evolutionsalgorithmus’**
  Ein Evolutionsalgorithmus wird entworfen und auf die spezielle Problemstellung angepasst. Da die näheren Details des Evolutionsvorgangs (Parameter, Laufzeit, Strategien etc.) nicht näher bekannt sind, muss auf ein flexibles und leicht erweiterbares Design geachtet werden.

[18] Eine Leinwand, welche von zwei Projektoren angestrahlt wird, vor deren Linsen zwei um 90° zueinander verdrehte Polarisationsfilter angebracht sind. Die beiden Projektoren zeigen die gleiche Szene aufgenommen aus zwei leicht horizontal versetzen Kameras. Mit Hilfe einer speziellen Brille, welche ebenfalls mit zwei Polarisationsfiltern versehen ist, ergibt sich bei Betrachtung der Leinwand ein 3D-Effekt. Dieser Effekt ist unabhängig von der Position des Betrachters zur Leinwand.

[19] Ein HMD (engl.: „**h**ead **m**ounted **d**isplay“) ist ein Helm mit integrierten Bildschirmen, durch welche es möglich ist, den Augen eine Szene zu präsentieren, die aus leicht horizontal versetzen Perspektiven aufgenommen wurde. Dadurch ergibt sich ein 3D-Effekt. Wird die Position und Orientierung des Helms erfasst und die virtuelle Kamera entsprechend mitbewegt, so kann man sich innerhalb der virtuellen Umgebung „umsehen“.

[20] Ein Tracker erfasst optisch, magnetisch oder akustisch die Position und Ausrichtung spezieller „Marker“. Diese Daten kann man zum Abgleich der Positionen von realen und virtuellen Objekten nutzen (z.B. HMD).

- **Evolution der Bewegung der virtuellen Charaktere**
  Der letzte Schritt ist die Anwendung der evolutionären Algorithmen auf die zu animierenden Charaktere. Das neuronale Netzwerk wird so lange verändert, bis der Charakter in stabiler und natürlicher Weise Zielvorgaben wie Stehen oder Laufen erfüllt.

Sekundäre Ziele sind:

- **Plattformunabhängige Programmierung**
  Die Software soll unter Windows sowie unter Linux kompilierbar und lauffähig sein. Die grafische Ausgabe (Render-Engine) soll auch auf einer Microsoft® Xbox möglich sein.

- **Modulcharakter**
  Alle größeren Softwareblöcke sind modular zu gestalten, so dass sich einzelne Bestandteile austauschen oder verändern lassen, ohne dass ein partieller oder gar vollständiger Neuentwurf notwendig wird.

- **Netzwerkzentrierter Datenaustausch**
  Der Datenaustausch zwischen den wesentlichen Modulen soll über Netzwerkpakete möglich sein, um Aufgaben und Rechenlast im Netzwerk verteilen zu können.

- **Skalierbarkeit/Parallelisierbarkeit**
  Die Software sollte möglichst auf mehreren Rechnern (Cluster) parallel lauffähig sein, um die Rechenzeit bei komplexen Simulationen und Charakteren verkürzen zu können.

Theft from a single author is plagiarism.
Theft from two is comparative study.
Theft from three or more is research.

Unbekannt

2 Analyse

2.1 Verwandte Arbeiten

Im Folgenden werden einige Arbeiten bzw. Projekte vorgestellt, die sich mit der Thematik befassen, virtuellen Charakteren autonome Bewegung zu ermöglichen. Diese Projekte lassen sich grob in drei Kategorien einordnen:

1. Verwendung von neuronalen Netzwerken (NN)
2. Verwendung von evolutionären Algorithmen (EA)
3. Verwendung von NN in Verbindung mit EA

Eine Gesamtübersicht aller Arbeiten von Firmen, Instituten, Universitäten und auch Privatpersonen, die sich unter verschiedenen Gesichtspunkten und unter Verwendung unterschiedlichster Technologien der Thematik angenähert haben, ist aufgrund der unüberschaubaren Zahl nicht möglich. Deshalb erhebt diese Liste keineswegs den Anspruch auf Vollständigkeit. Ebensowenig soll durch die Auswahl eine Wertung der Qualität gegeben werden.

Die hier vorliegende Thesis unterscheidet sich von allen im Folgenden vorgestellten Arbeiten durch drei Aspekte:

- **Interaktion**
  Durch die Schaffung einer virtuellen Umgebung ist es möglich, mit dem virtuellen Charakter zu interagieren und ihn in seinem Evolutionsvorgang zu unterstützen[1]. Den anderen Arbeiten fehlen solche Interaktionsmöglichkeiten.
- **Charakter-Grundformen**
  Die vorgestellten Arbeiten befassen sich hauptsächlich mit jeweils einer Form eines virtuellen Charakters. Diese Thesis wendet die Mechanismen der evolutionären Algorithmen *parallel* auf ein Mono-, Bi- und ein Quadruped an.

[1] Diese Möglichkeit konnte allerdings aufgrund des eingeschränkten Zeitrahmens nicht realisiert werden.

- **Universalität**
  Anstatt ein monolithisches Programm zu entwickeln, welches speziell auf die Aufgabe zugeschnitten ist, wurde Wert darauf gelegt, sowohl die Simulationsumgebung als auch die evolutionären Algorithmen für andere Aufgaben weiterverwenden zu können.

Des Weiteren wird am Ende der Vorstellung jeder Arbeit aufgeführt, welche Nachteile und Besonderheiten diese aufweist und welche Aspekte in die vorliegende Thesis eingeflossen sind.

2.1.1 Verwendung von neuronalen Netzwerken (NN)

2.1.1.1 Intelligent Motion Control with an Artificial Cerebellum

Das Projekt ist für diese Thesis von doppelter Bedeutung. Einserseits ist Russell L. Smith der Hauptentwickler der im späteren Verlauf noch erwähnten Physik-Engine ODE (siehe Abschnitt 7.7.1 auf Seite 97), andererseits behandelt er in seiner PhD Thesis die Steuerung von Bewegungen mittels eines Controllers auf Basis von Prinzipien realer neuronaler Netzwerke [Smith, 1998]. Seine Arbeit bietet einen kompakten, aber ausführlichen Überblick über die Anatomie des Kleinhirns und wendet diese Erkenntnisse auf das Design eines robusten und lernfähigen neuronalen Controllers an, welcher neben einem realen Brückenkran und einem inversen Pendel auch ein virtuelles Monoped und ein Biped steuert.

Da diese Arbeit auf evolutionäre Anteile verzichtet, ist eine „manuelle“ Untersuchung der nötigen Voraussetzungen zur Steuerung des virtuellen Charakters notwendig. Anhand der Ergebnisse dieser Untersuchung wird ein System zur Steuerung des Charakters entworfen, in welchem der neuronale Controller eine zentrale Rolle spielt.
Diese relativ aufwändige Form der Analyse wird in dieser Thesis nicht angewendet. Stattdessen soll durch evolutionäre Algorithmen die genaue Einstellung aller Parameter erfolgen.

2.1.1.2 An Empirical Exploration of a Neural Oscillator for Biped Locomotion Control

[Endo u. a., 2004] konzentrieren sich auf das Design einer einfachen Architektur eines neuronalen Mustergenerators (CPG, engl.: „central pattern generator“) aus zwei bzw. vier Neuronen zur Steuerung eines mechanischen Biped. Die Parameter der Neuronen werden durch Experimente von Hand eingestellt.

Aus dieser Arbeit wurden wesentliche Erkenntnisse zur Parametrierung von CPGs gewonnen. Die Komplexität dieser Aufgabe bestärkt die Notwendigkeit des in dieser Thesis verwendeten Ansatzes, Paramter eines neuronalen Netzwerks mittels evolutionärer Algorithmen zu ermitteln.

2.1.2 Verwendung evolutionärer Algorithmen (EA)

2.1.2.1 Using nonlinear oscillators to control the locomotion of a simulated biped robot

In dieser Arbeit liegt der Schwerpunkt auf dem Design eines Verbundes von Oszillatoren, welche die Glieder des Biped direkt antreiben. Anstatt diese Oszillatoren aus Neuronen aufzubauen, wird eine mathematische Form mit Steuerungsgrößen gewählt und mit Hilfe von evolutionären Algorithmen optimiert [Mojon, 2004].
Ein großer Teil der Analysen der Arbeit befasst sich mit der Stabilität zweier gekoppelter Oszillatoren unter Variation von jeweils zwei Steuergrößen, um Parameter für stabile Arbeitspunkte zu finden.
Durch diese Untersuchungen ist es möglich, die Anzahl der durch die evolutionären Algorithmen zu findenden Parameter zu minimieren, indem für die Stabilität wichtige Parameter z.B. vorbelegt werden und unwichtige Parameter gar nicht erst in den Suchraum aufgenommen werden.

Für diese Thesis wurde der Aspekt der Vorbelegung von Werten für die Suche mit evolutionären Algorithmen aufgegriffen.

2.1.2.2 Virtuelle Register-Maschinen

[Wolff und Nordin, 2003b,a] verwenden eine virtuelle Register-Maschine (VRM, engl.: „**v**irtual **r**egister **m**achine“) zur Kontrolle eines realen Roboters. Diese Maschine wird mit Hilfe genetischer Programmierung (GP) entwickelt. (siehe Abschnitt 5.3.4 auf Seite 74).

Dieser Ansatz bietet eine interessante Alternative zu neuronalen Netzwerken und bietet Einblicke in die Verwendung Genetischer Programmierung. Die verwendeten Mechanismen könnten leicht abgewandelt dazu dienen, die Architektur des neuronalen Netzwerks zu entwickeln, welches den virtuellen Charakter steuert.

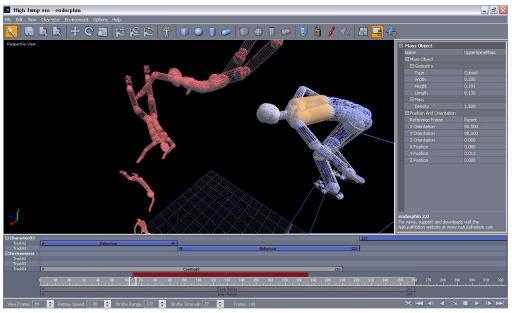
2.1.3 Verwendung von NN in Verbindung mit EA

2.1.3.1 endorphin

endorphin ist eine Software aus dem Haus NaturalMotion[2], einer Spin-Off Firma der Oxford-University, England.
Diese Software erlaubt es, virtuelle Charaktere mittels Anweisungen aus einem Katalog von Verhaltensweisen zu dirigieren. So ist es z.B. möglich, einer Figur einen Schlag zu versetzen, woraufhin diese versucht, das Gleichgewicht zu halten, stolpert, sich im Fall dreht und dabei die Hände vor das Gesicht nimmt.
Diese Verhaltensweisen stammen aus einer trainierten künstlichen Intelligenz (KI, engl.: AI für „**a**rtificial **i**ntelligence“) und nicht aus MoCap-Daten. Es gibt zusätzlich die

[2] `http://www.naturalmotion.com`

Möglichkeit, einen Charakter zunächst anhand von MoCap-Daten zu bewegen, um dann fließend in durch die KI kontrollierte Bewegungen überzugehen. Damit hat man die Möglichkeit, für reale Personen gefährliche Situationen bis kurz vor dem Gefahrenpunkt mit MoCap zu erfassen, um dann die Szene mit Hilfe des „virtuellen Stuntman“ abzuschließen.


(a) 3

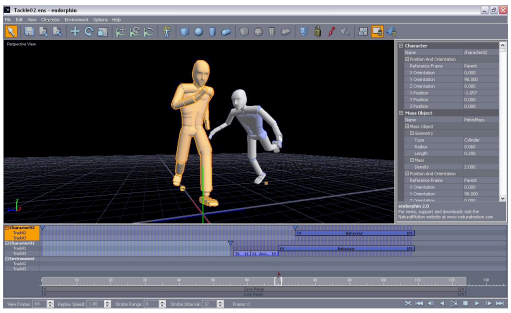

(b) 4

Abbildung 2.1: Benutzungsoberfläche von *endorphin* (siehe Farbtafel D.4 auf Seite 229)

Dieses Produkt verbirgt die Mechanismen der evolutionären Entwicklung der Bewegungen vor dem Benutzer. Dieser kann zwar die fertig entwickelten Bewegungsmuster verwenden und miteinander kombinieren, jedoch keine neuen Muster hinzufügen.
Die vorliegende Thesis verfolgt den Ansatz, aus Gründen der Flexibilität den Evolutionsvorgang selbst unter die Kontrolle des Anwenders zu stellen.

2.1.3.2 Arbeiten von Torsten Reil

In den beiden Arbeiten [Reil und Massey, 2001] und [Reil und Husbands, 2002] befasst sich Torsten Reil, ein Gründungsmitglied von NaturalMotion, mit der Aufgabe, ein einfaches Biped mit einem vollständig rekurrenten neuronalen Netzwerk zur autonomen Fortbewegung zu verhelfen.
In diesen Arbeiten sticht die sehr einfache Form des neuronalen Netzwerks hervor. Es handelt sich im Wesentlichen um zehn vollständig miteinander verbundene Neuronen, von denen sechs die Aktoren an den Gelenken steuern (siehe Abbildung 2.2 auf der nächsten Seite). Mit Hilfe eines zusätzlichen Neurons wurde später die Fähigkeit ergänzt, einer virtuellen „Schallquelle“ zu folgen.
Es ist beachtenswert, dass der virtuelle Charakter zu einem stabilen und gezielten Gang fähig ist, obwohl keinerlei Informationen über das Gleichgewicht oder sonstige Umweltinformationen in das neuronale Netzwerk eingespeist werden. Das legt allerdings auch die Vermutung nahe, dass die Stabilität des Gangs stark von den Simulationsparametern abhängt. Würde man z.B. die Schwerkraft leicht variieren, so wäre die Figur erneut instabil und ein neuer Optimierungsdurchlauf vonnöten.

[3] `http://www.naturalmotion.com/images/e20s2.jpg`
[4] `http://www.naturalmotion.com/images/e20s3.jpg`

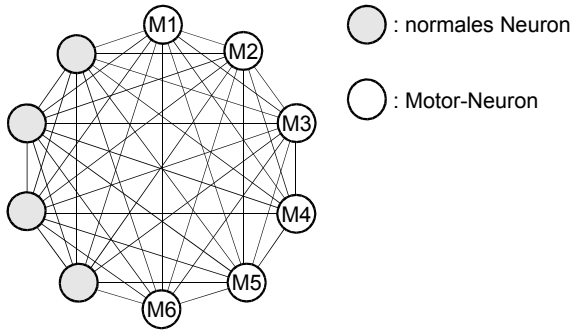


Abbildung 2.2: Vollständig verbundenes neuronales Netzwerk

In der anschließenden Betrachtung führen die Autoren an, dass sie sich durch die Eingabe von externen Informationen über die Lage des virtuellen Charakters im Raum (Gleichgewicht) und die Stellung der Gelenke eine wesentliche Stabilisierung des Biped erhoffen.

Um die Entwicklung des Laufmusters zu fördern, wurde der Charakter zunächst von außen stabilisiert. Nach Erreichen eines bestimmten Entwicklungsstadiums wurde dieser Stabilisator abgeschaltet, was sich zunächst in einer Verschlechterung der Ergebnisse auswirkte. Durch die bis dahin entwickelten Grundbewegungen konnte sich der Gang des Charakters jedoch im weiteren Verlauf über das bis dahin erreichte Stadium verbessern.
Dies entspricht in abgewandelter Form der für diese Thesis ebenfalls angedachten Hilfestellung während des Evolutionsvorgangs.

2.1.3.3 Arbeiten von Chandana Paul

Chandana Paul arbeitet zur Zeit an der Universität Zürich im Bereich Künstliche Intelligenz und Robotik an ihrer Dissertation zum Thema „Biped Robotics“. [Paul und Bongard, 2001b; Paul, 2003, 2004; Paul und Bongard, 2001a] sind vorbereitende Untersuchungen dazu.
Ähnlich wie in den Arbeiten von Reil ist das in [Paul und Bongard, 2001b] verwendete neuronale Netzwerk für die Biped von sehr einfacher Struktur (siehe Abbildung 2.3). Die mittlere Schicht stellt den zentralen Mustergenerator dar. Er ist das wesentliche Element für rhythmische Bewegungen.

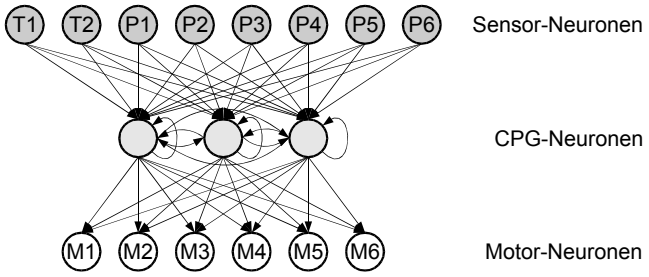


Abbildung 2.3: Einfaches neuronales Netzwerk mit zentralem Muster-Generator

[Paul, 2004] untersucht neuronale Netzwerke, welche nur mit sensorischen Informationen auskommen und keinen Mustergenerator benötigen. [Paul, 2003] erforscht Netzwerkstrukturen, welche getrennt die linke und rechte Körperhälfte behandeln.

Die in diesen Arbeiten verwendeten Netzwerkarchitekturen dienten als Vorlage für die in dieser Thesis verwendeten neuronalen Netzwerke.

2.1.3.4 Entwicklung der Morphologie und Steuerung eines zweibeinigen Laufmodells

Diese Arbeit befasst sich mit der Analyse der Übertragbarkeit von Konzepten einer passiven Laufmaschine auf eine sensorisch gesteuerte aktive Laufmaschine [Wischmann, 2003]. Es wird zunächst der Unterschied zwischen statischem und dynamischem Laufen erläutert. Statische Laufmaschinen halten ihren Schwerpunkt jederzeit innerhalb der Fläche, die von den „Fußsohlen“ aufgespannt wird. Diese Laufmaschinen können zu jedem beliebigen Zeitpunkt in ihrem Bewegungsvorgang angehalten werden, ohne umzufallen. Dafür ist die Bewegung aber auch relativ langsam und wirkt sehr künstlich und „vorsichtig“.
Dynamische Laufmaschinen erfüllen im Gegensatz dazu jederzeit die Bedingung, ihren ZMP (engl.: „**z**ero **m**omentum **p**oint“) innerhalb der Fußauflagefläche zu halten. In diesem Punkt summieren sich alle internen und externen Drehmomente zu Null auf. Diese Bewegungsart führt zu einer flüssigeren und energetisch günstigeren Fortbewegung, benötigt dafür allerdings auch einen ausgeklügelteren Regelungs- und Steuerungsalgorithmus und relativ leistungsfähige Antriebselemente.
Passiv dynamische Laufmaschinen setzen alternativ zu leistungsfähigen Antriebselementen auf Schwung, Trägheit, Schwerkraft und Energiespeicher (z.B. Federn) zur Unterstützung relativ schwach dimensionierter Antriebselemente. Passives Laufen ist nur im Rahmen der ZMP-Theorien möglich. Antriebsunterstützende Elemente sind auch in der Natur zu finden, z.B. in Form von Sehnen in Hinter- und Vorderbeinen von Tieren. Diese Sehnen speichern Bewegungsenergie beim Aufsetzen und können diese für den nächsten Sprung wieder abgeben, so dass kein hoher Kraftaufwand aus den Muskeln heraus notwendig ist. Sowohl die Morphologie des Biped als auch die Gewichte des Neurocontrollers werden durch evolutionäre Algorithmen eingestellt. Auch hier ist das neuronale Netz von relativ einfacher Struktur.
Der Autor kommt zu dem Schluss, dass die Morphologie der Laufmaschine einen wesentlichen Anteil an der Energieeffizienz trägt. Nur wenn die Proportionen stimmen, kann Schwung optimal ausgenutzt werden, und ist es demnach möglich, die Antriebselemente kleiner und energiesparender zu dimensionieren.

Im Unterschied zu dieser Arbeit ist bei der vorliegenden Thesis für die Entwicklung des neuronalen Netzwerks keine vorherige Anpassung der Morphologie des virtuellen Charakters notwendig.

2.1.3.5 Tank Wars

[Bourquin, 2004] steuert die Motoren und die Geschütztürme zweier virtueller Panzer mit Hilfe eines zeitkontinuierlichen neuronalen Netzwerks (CTRNN, engl.: „**c**ontinuous **t**ime **r**ecurrent **n**eural **n**etwork“, siehe Abschnitt 4.5 auf Seite 54). Die Parameter dieses Netzwerks werden durch evolutionäre Algorithmen eingestellt.

Die Einstellung der Parameter erfolgt in kleinen Schritten, bei welchen der Schwerpunkt jeweils auf Teilaspekten der Gesamtaufgabe liegt (steuern, ausweichen, zielen). Die Notwendigkeit dieser Vorgehensweise ist auch das Ergebnis dieser Thesis. Zu komplizierte oder umfangreiche Aufgabenstellungen überfordern evolutionäre Algorithmen und führen zu unbrauchbaren Ergebnissen.

2.1.3.6 Quadruped Robot

[Wiley, 2003] experimentiert mit einer Vielzahl von Konstellationen und Architekturen neuronaler Netzwerke inklusive von Mustergeneratoren. Gesteuert wird damit im Gegensatz zu den anderen Arbeiten ein Quadruped.

Die erfolgreiche Entwicklung von Fortbewegung kommt in dieser Arbeit erst zustande, als das entwickelte neuronale Netzwerk symmetrisch auf beiden Seiten des Quadruped eingesetzt wird.
Anstelle von CPGs verwenden die Autoren aus Zeitgründen parametrierbare Oszillatoren, schlagen in ihrem Nachwort jedoch vor, diese durch reale neuronale Oszillatoren zu ersetzen.
Die vorliegende Thesis verwendet durchgängig neuronale Netzwerke ohne externe Oszillatoren oder sonstige Hilfsmittel.

2.1.3.7 Evolving intelligent embodied agents within a physically accurate environment

[Ruebsamen, 2002] entwickelt mittels evolutionärer Algorithmen neuronale Netzwerke zur Steuerung von einfachen virtuellen Charakteren, welche sich in einer physikalischen Umgebung fortbewegen sollen. Diese Charaktere sind relativ einfach aufgebaut. Sie bestehen zumeist aus nicht mehr als einem rechteckigen Körper sowie Sprungfedern, Flossen oder Paddeln als Fortbewegungsmittel.

Das Besondere dieser Arbeit sind die verschiedenen Formen von Charakteren. Die Tatsache, dass alle diese Charaktere letztlich eine Art der Fortbewegung beherrschen, zeigt die Flexibilität der Kombination aus evolutionären Algorithmen und neuronalen Netzwerken und bestärkt die Verwendung dieser Kombination in der vorliegenden Thesis.

2.2 Zusammenfassung

Die Vielzahl der Ansätze zeigt, dass ein „Königsweg“ zur Bewältigung der Aufgabe noch nicht gefunden ist. Zu zahlreich sind die einzelnen Aspekte der Aufgabe, virtuellen Charakteren „Leben einzuhauchen“.
Jede Betrachtungsweise erlaubt Verbesserungen in Details, welche anderen Ansätzen verwehrt bleiben. Einige Ansätze mussten vorher gesteckte Ziele zurücknehmen oder dem virtuellen Charakter durch kleine „Tricks“ nachhelfen.
endorphin sticht aus dieser Reihe hervor, da es ein marktreifes Produkt darstellt. Es sollte allerdings nicht vergessen werden, dass in dieses Produkt auch eine Menge Entwicklungsarbeit, Mannmonate und wahrscheinlich eine Reihe von Abschlussarbeiten von Studenten der Oxford University geflossen sind:

> [...] unlike simple animation, characters developed with ACT (Active Character Technology) [...] have neural networks for brains and their movements have been choreographed by people with PhDs in biomechanical motion.[5]

Diese Thesis versucht, die erfolgsversprechenden Anteile aus allen vorgestellten Arbeiten zu kombinieren. So scheint die Verbindung von der Steuerung mittels neuronaler Netzwerke mit der Einstellung dieser Netzwerke über evolutionäre Algorithmen ein flexibler, universeller und praktikabler Weg zu sein, auf komplexe Aufgabenstellungen und unterschiedliche Formen von virtuellen Charakteren eingehen zu können.

2.3 Funktionsblöcke

Abbildung 2.4 auf der nächsten Seite zeigt die wesentlichen drei Blöcke, welche für die Funktionalität des Programms nötig sind, das im Rahmen dieser Master's Thesis entwickelt werden soll:

1. **Physik-Engine**
   Dieser Block ist zuständig für die physikalisch korrekte Simulation des virtuellen Charakters (z.B. Masse, Trägheitsmomente, Kräfte, Impulse, Gelenke, Schwerkraft).

2. **Neuronales Netzwerk**
   Dieser Block verarbeitet sensorische Informationen des Körpers und aus der Umgebung des virtuellen Charakters (z.B. Gelenkstellungen, Kräfte, Berührungen) und gibt Steuerinformationen für die aktiven Elemente (z.B. Muskeln, Motoren) aus.

3. **Evolutions-Engine**
   Dieser Block parametriert die Gewichte der Verbindungen, Schwellwerte der Neuronen und andere Werte des neuronalen Netzwerks anhand der Bewertungsergebnisse von Simulationsläufen, um einen optimalen Bewegungsablauf zu erreichen.

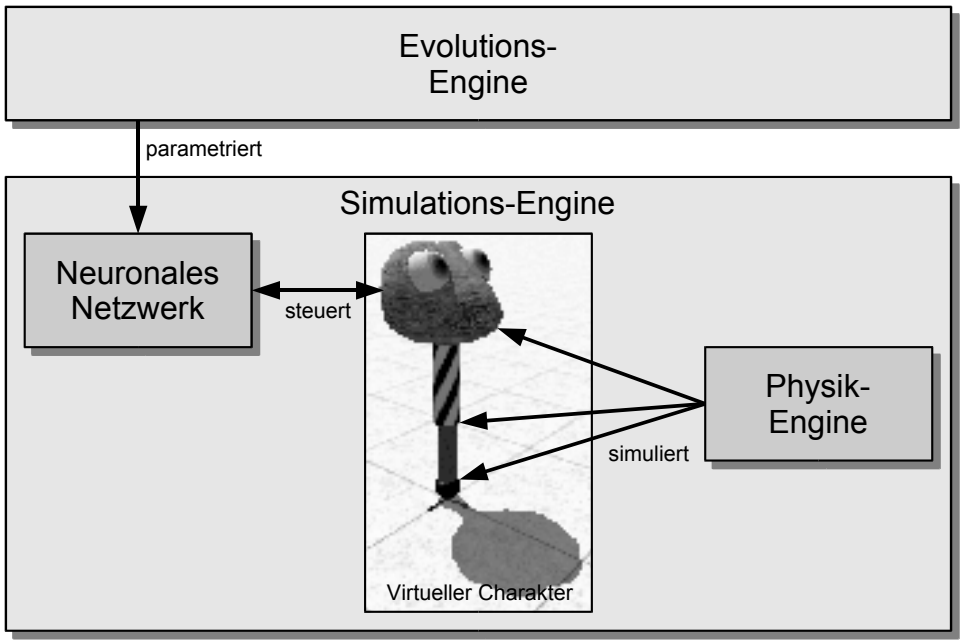


Abbildung 2.4: Funktionsblöcke des Programms

In den folgenden Kapiteln sollen die theoretischen Grundlagen für diese drei Funktionsblöcke näher erläutert werden.

[5] Aus: „Falling, flailing, virtual doll“, The Economist, 4. September 2003, `http://www.economist.com/science/tq/displayStory.cfm?story_id=2019927`

Teil I

Grundlagen

Physics is not difficult, it is just weird.

Vincent Icke, „The Force of symmetry“, 1994

3 Physik

Um den virtuellen Charakter korrekt simulieren zu können, ist eine Abbildung der realen physikalischen Gesetze in eine durch den Computer berechenbare Variante notwendig. Es müssen dabei aber bei weitem nicht alle Bereiche der Physik abgedeckt werden [Wikipedia Physik, 2005]. Für diese Thesis wird lediglich eine Simulation der Dynamik starrer Körper benötigt (RBD, engl.: „**r**igid **b**ody **d**ynamics“). Dabei handelt es sich um ein Teilgebiet der klassischen Newton'schen Mechanik. Die Bereiche der Strömungsmechanik, Optik, Akustik, Elektromagnetismus, Thermodynamik, Kern-, Molekular- und Atomphysik, Relativitätstheorie etc. werden nicht benötigt.
Eine solche sogenannte Physik-Engine wurde im Rahmen der Vorlesung Virtuelle Umgebungen A (VUM-A) im Wintersemester 2003 bereits in Ansätzen erstellt, reicht aber mit ihren Möglichkeiten für diese Thesis nicht aus.
Deshalb wurde zu der frei verfügbaren Open-Source Physik-Engine ODE (engl.: „**O**pen **D**ynamics **E**ngine“) gegriffen, welche alle benötigten Funktionen zur Verfügung stellt.
Ein Vergleich der selbst erstellten Physik-Engine mit den frei und kommerziell erhältlichen Engines wird in Abschnitt 7.7.1 auf Seite 97 noch genauer ausgeführt.
Im Folgenden soll zunächst die Funktionsweise einer solchen Engine erläutert werden. Dabei werden lediglich die Aspekte der Physik der starren Körper berücksichtigt.

3.1 Prinzipien einer Physik-Engine

3.1.1 Simulationsumfang

Eine RBD Physik-Engine sollte in der Lage sein, folgende Aspekte der natürlichen Physik zu simulieren:

- **Starre Körper**
  Ein starrer Körper besitzt Eigenschaften wie Masse, Position, Rotation, Geschwindigkeit, Impuls, Moment, Trägheit.

- **Kräfte und Impulse**
  Sie beschleunigen den Körper und sind die eigentliche Ursache von Bewegung innerhalb der Simulation. Kräfte wirken entweder stetig (z.B. Schwerkraft), kurzzeitig (z.B. Kollisionen) oder durch Ereignisse (z.B. Benutzer-Interaktion).

- **Kollisionen**
  Kollisionen sind notwendige Ereignisse in einer dynamischen Umgebung. Sie finden statt, wenn zwei oder mehrere Körper durch ihre Geschwindigkeit oder durch Kräfte aufeinander zubewegt werden und sich berühren.

- **Gelenke**
  Ein Gelenk verbindet zwei oder mehrere starre Körper und schränkt diese in ihrer Freiheit ein, sich relativ zueinander zu bewegen. Durch die Wahl der Einschränkungen kann man unterschiedliche Sorten von Gelenken simulieren, z.B. Scharniergelenk, Kreuzgelenk, Kugelgelenk.

- **Reibung**
  Reibung beeinflusst in Form von Gleitreibung die Kräfte, die zwei sich berührende, relativ zueinander bewegte Körper aufeinander ausüben. In Form von Haftreibung hingegen hindert sie zwei sich berührende, ruhende Körper daran, sich unter Einwirkung von Kräften relativ zueinander in Bewegung zu setzen.

3.1.2 Simulationsprinzip

Physik-Engines arbeiten prinzipbedingt nur zeitdiskret, das heißt, dass der Zustand der simulierten Welt nur zu bestimmten äquidistanten Zeitpunkten $t, t+\Delta t, t+2\Delta t, \ldots$ berechnet werden kann. Abbildung 3.1 auf der nächsten Seite zeigt das Simulationsprinzip.
Aus dem Zustand der zu simulierenden Welt zum Zeitpunkt t heraus werden zunächst die sogenannten **Constraints** (engl.: „Zwänge“) berücksichtigt. Dies sind Einschränkungen der Körper in ihrer Bewegung z.B. durch Gelenke. Daraus resultieren innere Kräfte, die dann zusammen mit den äußeren Kräften (z.B. Schwerkraft) und den Eigenschaften und Zuständen der Körper mit Hilfe von Differentialgleichungen (**DGL**) gelöst werden. Aus der Lösung dieser Gleichungen resultiert der Zustand der Welt für den nächsten Simulationsschritt $t + \Delta t$.

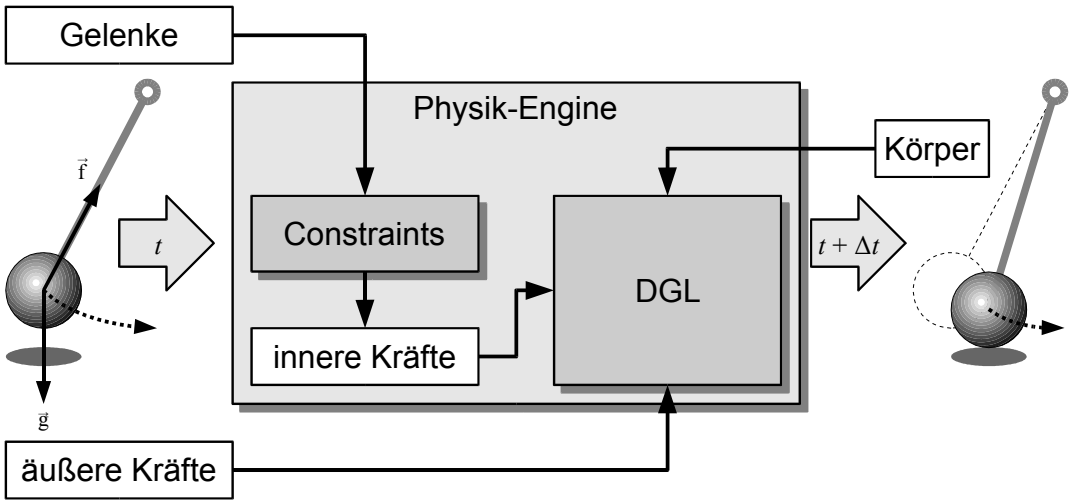


Abbildung 3.1: Simulationsprinzip einer Physik-Engine

3.1.3 Probleme

Durch die zeitdiskrete Berechnung einer Physik-Engine kann es zu diversen Problemen kommen. So ist es möglich, dass zwei Körper B_1 und B_2, die sich aufeinander zubewegen, zum Zeitpunkt t bereits sehr dicht beieinander liegen. Im nächsten Simulationsschritt $t + \Delta t$ haben sich die beiden Körper weiter aufeinander zubewegt und durchdringen sich nun. Dies ist in der Natur nicht möglich. Dort wäre eine Kollision und die darauf folgende Reaktion bereits vorher erfolgt. Die Physik-Engine hingegen muss mit solchen Situationen „rechnen“ und entsprechend darauf reagieren. Dazu gibt es drei Ansätze:

1. **Ignorieren**
   Einige Physik-Engines[1] berechnen den „Durchdringungsgrad“ und versuchen proportional dazu, die Körper wieder auseinander zu bewegen. Dies kann allerdings zu unnatürlichen, „explosionsartigen“ Reaktionen führen, wenn die Durchdringung relativ hoch ist. Diesem Phänomen kann man durch einen ausreichend kleinen Simulationsschritt Δt begegnen.

2. **Zurückverfolgen**
   Es wird versucht, den Zeitpunkt t_x mit $t < t_x < t + \Delta t$ zu ermitteln, zu dem die Kollision stattfindet und die Reaktionen exakt zu diesem Zeitpunkt erfolgen zu lassen. Dieser Vorgang ist allerdings rechenintensiv und kann im Fall von zahlreichen Kollisionen zu einer starken Belastung des Computers führen.

3. **Vorausberechnungen**
   Bei diesem Verfahren[2] wird die Position der Körper mit schnellen, aber dafür ungenauen Approximationsverfahren einen Schritt in die Zukunft voraus berechnet und so der Zeitpunkt einer Kollision vorherbestimmt.

[1] z.B. ODE, siehe Abschnitt 7.7 auf Seite 97
[2] wird z.B. bei Havok verwendet, siehe Abschnitt 7.7 auf Seite 97

3.2 Starre Körper

Ein einfacher starrer Körper im Sinne einer Physik-Engine besitzt die in Tabelle 3.1 aufgeführten Eigenschaften [Eberly, 2004; Bourg, 2002; De Loura, 2000; Smith, 2004; Ďurikovič und Numata, 2004].

Eigenschaft	**Symbol**	**Einheit**
Masse	M	[kg]
Trägheitstensor	I_0	[kg/m^2], 3×3-Matrix
Position im Raum	$\vec{x}(t)$	[m]
Orientierung im Raum	$\mathrm{R}(t)$	[m], 3×3-Matrix
Lineargeschwindigkeit	$\vec{v}(t)$	[m/s]
Linearimpuls	$\vec{P}(t)$	[kg m/s]
Rotationsgeschwindigkeit	$\vec{\omega}(t)$	[rad/s]
Rotationsimpuls	$\vec{L}(t)$	[kg m^2/s]

Tabelle 3.1: Eigenschaften eines starren Körpers

Alle Vektorgrößen in dieser Tabelle sind 3-dimensional mit einem x-, y- und z-Anteil. Die Orientierung wird in Form einer 3×3-Rotations-Matrix repräsentiert. Dabei geben die Spalten dieser Matrix die Einheitsvektoren $\vec{e}_{X/Y/Z}$ des rotierten Körperkoordinatensystems im Weltkoordinatensystem an.

$$\mathrm{R} = \begin{pmatrix} \vec{e}_X & \vec{e}_Y & \vec{e}_Z \end{pmatrix} = \begin{pmatrix} e_{X,x} & e_{Y,x} & e_{Z,x} \\ e_{X,y} & e_{Y,y} & e_{Z,y} \\ e_{X,z} & e_{Y,z} & e_{Z,z} \end{pmatrix} \tag{3.1}$$

Der Trägheitstensor wird in Abschnitt 3.3 auf Seite 29 noch genauer beschrieben. In dieser 3×3-Matrix befinden sich Informationen über die Gewichtsverteilung des Körpers.

Die Rotationsgeschwindigkeit gibt mit der Richtung des Vektors die Rotationsachse und mit seiner Länge die Geschwindigkeit an.

Um den Zustand $\boldsymbol{X}(t)$ eines starren Körpers in Abhängigkeit von der Zeit t zu beschreiben, sind im Wesentlichen nur die vier Größen Position, Orientierung, Linear- und Rotationsgeschwindigkeit notwendig:

$$\boldsymbol{X}(t) = \begin{pmatrix} \vec{x}(t) \\ \mathrm{R}(t) \\ \vec{P}(t) \\ \vec{L}(t) \end{pmatrix} \tag{3.2}$$

Alle anderen Größen aus Tabelle 3.1 stehen mit diesen Zustandsgrößen in folgenden Relationen (aus Gründen der Übersichtlichkeit wird auf die zeitliche Abhängigkeit (t) verzichtet):

- Lineargeschwindigkeit $\vec{v}$:

$$\vec{v} = \frac{d\vec{x}}{dt} \tag{3.3}$$

- Linearimpuls $\vec{P}$:

$$\vec{P} = M \cdot \vec{v} = M \cdot \frac{d\vec{x}}{dt} \tag{3.4}$$

- Rotationsgeschwindigkeit $\vec{\omega}$:
  Die Einheitsvektoren von R werden um $\vec{\omega}$ rotiert:

$$\frac{d\mathrm{R}}{dt} = \begin{pmatrix} \vec{\omega} \times \vec{e}_X & \vec{\omega} \times \vec{e}_Y & \vec{\omega} \times \vec{e}_Z \end{pmatrix} \tag{3.5}$$

$$= \begin{pmatrix} \omega_y e_{X,z} - \omega_z e_{X,y} & \omega_y e_{Y,z} - \omega_z e_{Y,y} & \omega_y e_{Z,z} - \omega_z e_{Z,y} \\ \omega_z e_{X,x} - \omega_x e_{X,z} & \omega_z e_{Y,x} - \omega_x e_{Y,z} & \omega_z e_{Y,x} - \omega_x e_{Z,z} \\ \omega_x e_{X,y} - \omega_y e_{X,x} & \omega_x e_{Y,y} - \omega_y e_{Y,x} & \omega_x e_{Z,y} - \omega_y e_{Z,x} \end{pmatrix}. \tag{3.6}$$

  Unter Einführung der antisymmetrischen Hilfsmatrix Ω

$$\text{mit} \quad \Omega = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix} \tag{3.7}$$

$$\text{gilt} \quad \frac{d\mathrm{R}}{dt} = \Omega \cdot \mathrm{R}. \tag{3.8}$$

- Drehimpuls $\vec{L}$:
  Nach Rotation des Körper-Trägheitstensors I_0 mit Hilfe der Rotationsmatrix R

$$\text{mit} \quad \mathrm{I} = \mathrm{R} \cdot \mathrm{I}_0 \cdot \mathrm{R}^T \tag{3.9}$$

$$\text{gilt} \quad \vec{L} = \mathrm{I} \cdot \vec{\omega}. \tag{3.10}$$

Durch Kollisionen, Schwerkraft oder Interaktionen wirken im Verlauf der Simulation Beschleunigungen und Kräfte auf den Körper ein (siehe Tabelle 3.2).

Größe	**Symbol**	**Einheit**
Linearbeschleunigung	$\vec{a}(t)$	$[\mathrm{m/s^2}]$
Rotationsbeschleunigung	$\vec{\alpha}(t)$	$[\mathrm{rad/s^2}]$
Kraft	$\vec{F}(t)$	$[\mathrm{N}] = [\mathrm{kg\ m/s^2}]$
Drehmoment	$\vec{\tau}(t)$	$[\mathrm{N\ m}] = [\mathrm{kg\ m^2/s^2}]$

Tabelle 3.2: Einflussgrößen eines starren Körpers

Sie stehen zu den Größen aus Tabelle 3.1 auf der vorherigen Seite in folgenden Relationen:

- Linearbeschleunigung $\vec{a}$:

$$\vec{a} = \frac{d\vec{v}}{dt} \tag{3.11}$$

- Kraft $\vec{F}$:

$$\vec{F} = \frac{d\vec{P}}{dt} = M \cdot \vec{a} \tag{3.12}$$

- Rotationsbeschleunigung $\vec{\alpha}$:

$$\vec{\alpha} = \frac{d\vec{\omega}}{dt} \tag{3.13}$$

- Drehmoment $\vec{\tau}$:

$$\vec{\tau} = \frac{d\vec{\omega}}{dt} = \mathrm{I} \cdot \vec{\alpha} \tag{3.14}$$

Aus der Gesamtheit dieser Beziehungen lässt sich nun die Differentialgleichung

$$\frac{d\boldsymbol{X}(t)}{dt} = \frac{d}{dt}\begin{pmatrix} \vec{x}(t) \\ \mathrm{R}(t) \\ \vec{P}(t) \\ \vec{L}(t) \end{pmatrix} = \begin{pmatrix} \vec{v}(t) \\ \Omega(t)\mathrm{R}(t) \\ \vec{F}(t) \\ \vec{\tau}(t) \end{pmatrix} \tag{3.15}$$

herleiten, welche die Grundlage für die physikalische Simulation bildet.

$\vec{v}(t)$ wird nach Formel 3.4 auf der vorherigen Seite mit $\vec{v}(t) = \frac{\vec{P}(t)}{M}$ gebildet. $\Omega(t)$ erhält man mit Hilfe von $\vec{\Omega}(t) = \mathrm{I}(t)^{-1} \cdot \vec{L}(t)$ und $\mathrm{I}(t)^{-1} = \mathrm{R}(t) \cdot \mathrm{I}_0^{-1} \cdot \mathrm{R}(t)^T$ aus Formel 3.10 auf der vorherigen Seite. $\vec{F}(t)$ und $\vec{\tau}(t)$ sind die äußeren Einflussgrößen des Körpers.

Mit Hilfe von Quaternionen (siehe Kapitel A auf Seite 199) ist es möglich, die Rotationsmatrix $\mathrm{R}(t)$ durch einen einfacheren Ausdruck $\vec{\mathrm{r}}(t)$ zu ersetzen [Witkin und Baraff, 2001]:

$$\frac{d\boldsymbol{X}(t)}{dt} = \frac{d}{dt}\begin{pmatrix} \vec{x}(t) \\ \vec{\mathrm{r}}(t) \\ \vec{P}(t) \\ \vec{L}(t) \end{pmatrix} = \begin{pmatrix} \vec{v}(t) \\ \frac{1}{2}\begin{pmatrix} \vec{\omega}(t) \\ 0 \end{pmatrix} \cdot \vec{\mathrm{r}}(t) \\ \vec{F}(t) \\ \vec{\tau}(t) \end{pmatrix} \tag{3.16}$$

Auf die Herleitung der Beziehung $\frac{d}{dt}\vec{\mathrm{r}}(t) = \frac{1}{2}\begin{pmatrix} \vec{\omega}(t) \\ 0 \end{pmatrix} \cdot \vec{\mathrm{r}}(t)$ wird an dieser Stelle verzichtet. Sie findet sich z.B. in [Witkin und Baraff, 2001, Anhang B].

3.3 Trägheitstensoren

Der Trägheitstensor eines starren Körpers wird allgemein wie folgt berechnet [Weisstein, 2002]:

$$\mathrm{I} = \int_V \rho(x,y,z) \begin{pmatrix} y^2+z^2 & -yx & -zx \\ -xy & x^2+z^2 & -zy \\ -xz & -yz & x^2+y^2 \end{pmatrix} dx\,dy\,dz, \tag{3.17}$$

wobei $\rho(x,y,z)$ die Dichte des Körpers an einer geometrischen Position angibt.
Für Trägheitstensoren spezieller geometrischer Grundformen gibt es vereinfachte Formeln:

- **Quader** (solide, Kantenlängen x, y, z, Masse m):

$$\mathrm{I}_{box} = \frac{m}{12} \begin{pmatrix} y^2z^2 & 0 & 0 \\ 0 & x^2z^2 & 0 \\ 0 & 0 & x^2y^2 \end{pmatrix} \tag{3.18}$$

- **Kugel** (solide, Radius r, Masse m):

$$\mathrm{I}_{sph} = \frac{2mr^2}{5} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.19}$$

- **Zylinder** (solide, Radius r, Höhe h entlang der X-Achse, Masse m):

$$\mathrm{I}_{cyl} = \frac{m}{2} \begin{pmatrix} r^2 & 0 & 0 \\ 0 & \frac{h^2}{6}+\frac{r^2}{2} & 0 \\ 0 & 0 & \frac{h^2}{6}+\frac{r^2}{2} \end{pmatrix} \tag{3.20}$$

- **Kegel** (solide, Radius r, Höhe h entlang der X-Achse, Masse m):

$$\mathrm{I}_{con} = \frac{m}{10} \begin{pmatrix} 3r^2 & 0 & 0 \\ 0 & h^2+\frac{3r^2}{2} & 0 \\ 0 & 0 & h^2+\frac{3r^2}{2} \end{pmatrix} \tag{3.21}$$

3.4 Gelenke

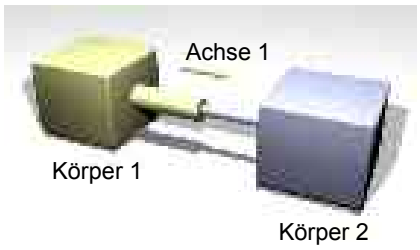
Gelenke im Sinne einer Physik-Engine sind eine Form von Constraints, welche zwei oder mehrere Körper daran hindern, sich uneingeschränkt zueinander zu bewegen [Smith, 2002; NovodeX AG, 2005].
Zwei Körper besitzen zueinander drei translatorische (t_X, t_Y und t_Z) und drei rotatorische (r_α, r_β und r_γ) Freiheitsgrade.

Werden diese Freiheitsgrade eingeschränkt, so kann man zahlreiche Formen von Gelenken modellieren. Tabelle 3.3 zeigt, welche Einschränkungen zu welcher Form von Gelenk führen und Abbildung 3.2 zeigt einige mögliche Gelenkarten.

Eingeschränkte Achsen	**Gelenktyp**	**Abbildung**
$t_{X,Y,Z}, r_{\alpha,\beta,\gamma}$	starre Verbindung	–
$t_{X,Y}, r_{\alpha,\beta,\gamma}$	Lineargelenk	3.2(a)
$t_{X}, r_{\alpha,\beta,\gamma}$	2D-Lineargelenk	–
$r_{\alpha,\beta,\gamma}$	3D-Lineargelenk	–
$t_{X,Y,Z}, r_{\alpha,\beta}$	Scharniergelenk	3.2(b)
$t_{X,Y,Z}, r_{\alpha}$	Kreuzgelenk	3.2(c)
$t_{X,Y,Z}$	Kugelgelenk	3.2(d)
$t_{X,Y}, r_{\alpha,\beta}$	z.B. Antriebsachse mit Federung	–

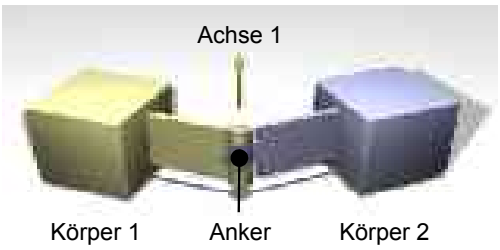
Tabelle 3.3: Einschränkungen der Freiheitsgrade und daraus resultierende Gelenk-Arten

Um kompliziertere Gelenkformen zu erzeugen, kann man zwei Gelenke miteinander verknüpfen. Dies führt aber zu einem erhöhten Rechenaufwand, da die Einhaltung von Constraints zwei mal berücksichtigt werden muss. Viele Physik-Engines definieren deshalb alle denkbaren Gelenktypen, um den Rechenaufwand in Grenzen zu halten, oder erlauben die Erweiterung um Gelenktypen (z.B. ODE).



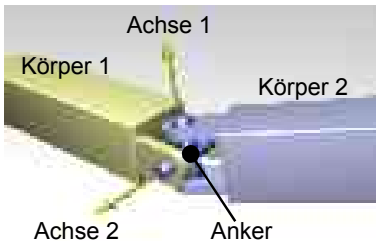
[3]

(a) Lineargelenk



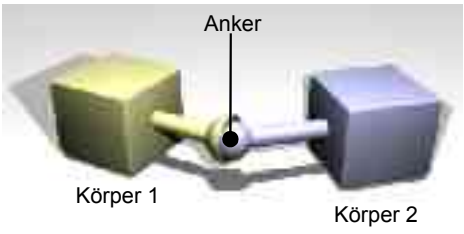
[4]

(b) Scharniergelenk



[5]

(c) Kreuzgelenk



[6]

(d) Kugelgelenk

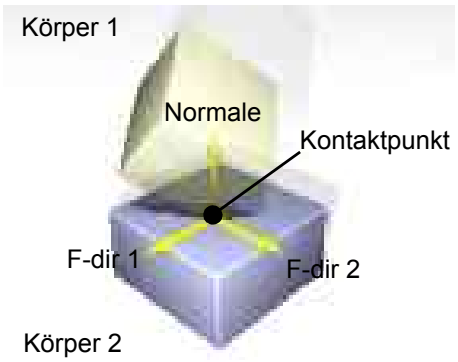
Abbildung 3.2: Gelenkarten

[3] `http://ode.org/pix/slider.jpg` (nachbearbeitet)
[4] `http://ode.org/pix/hinge.jpg` (nachbearbeitet)
[5] `http://ode.org/pix/universal.jpg` (nachbearbeitet)
[6] `http://ode.org/pix/ball-and-socket.jpg` (nachbearbeitet)

3.4.1 Kontaktgelenk

Das Kontaktgelenk ist ein Spezialfall, welcher unter anderem in der Physik-Engine ODE verwendet wird. Kontaktgelenke werden an den Kollisionspunkten zweier Körper nur für die Dauer eines Simulationsschrittes erzeugt. Sie erhalten alle Parameter, die für die Berechnung der Kollisionsantwort wichtig sind wie Reibungskoeffizienten oder Elastizität.

Nachdem die Kollisionsgelenke in die Berechnung des nächsten Simulationsschrittes eingeflossen sind, werden sie komplett entfernt.



[7]

Abbildung 3.3: Kontaktgelenk

[7] `http://ode.org/pix/contact.jpg` (nachbearbeitet)

3.5 Kollisionen

Das Thema Kollisionserkennung ist zu umfangreich, um es an dieser Stelle ausführlich zu behandeln. Deshalb wird im Folgenden nur eine grobe Zusammenfassung allgemeiner Prinzipien gegeben.

Eine Kollisions-Engine muss eng verknüpft mit der Physik-Engine zusammenarbeiten, da einerseits die Physik-Engine nach jedem Simulationsschritt die aktuellen Kollisionspunkte benötigt und andererseits die Kollisions-Engine nach jedem Simulationsschritt die Position der Kollisionsobjekte anhand der Veränderungen innerhalb der physikalischen Simulation anpassen muss.
Kollisionsobjekte sind geometrisch vereinfachte Repräsentationen der physikalischen Körper. Angaben über die Masse, den Trägheitstensor etc. sind nicht erforderlich. Die Abmessungen und ggf. detaillierte Angaben über die äußere Hülle sind ausreichend. Häufig wird ein komplexer Körper mit einer Kombination aus einfachen geometrischen Primitiven beschrieben (siehe Tabelle 3.4 und Abbildung 3.4).

geometrische Form	**Parameter**
Kugel	Mittelpunkt $\vec{x}$, Radius r
Quader	Mittelpunkt $\vec{x}$, Abmessungen x, y und z
Zylinder	Mittelpunkt $\vec{x}$, Radius r, Länge l
abgerundeter Zylinder	Mittelpunkt $\vec{x}$, Radius r, Länge l
Konus	Mittelpunkt $\vec{x}$, Radius r, Länge l

Tabelle 3.4: Arten von Kollisionsprimitiven

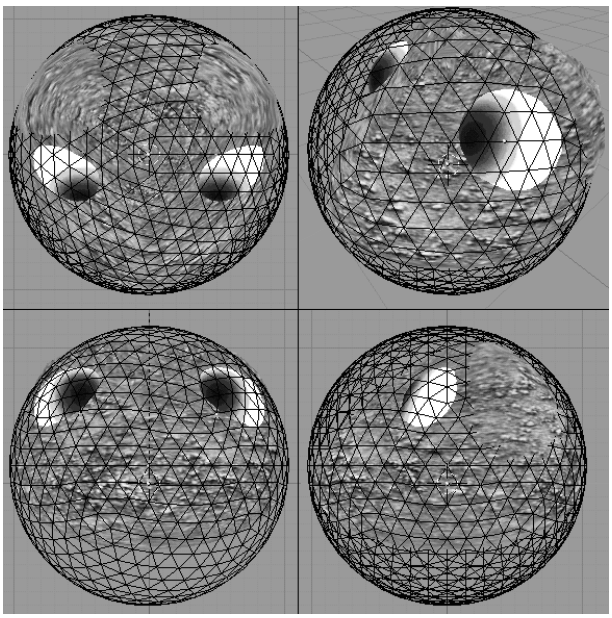


Abbildung 3.4: Komplexer grafischer Körper und sein Kollisionskörper

Bei sehr komplexen Formen kann es allerdings auch nötig sein, die äußere Form durch eine Dreieckshülle (engl.: „mesh“) anzugeben.

Im Allgemeinen läuft die Kollisionserkennung in zwei Phasen ab:

1. **Grobe Phase (broad phase)**
   In dieser Phase werden durch schnelle Verfahren Objekte vorselektiert, die als Kollisionspartner in Frage kommen. Es kommt dabei in diesem Schritt noch nicht auf Genauigkeit an. Verwendung finden z.B.:

   - Raumteilungsverfahren
     - Octrees
     - BSP (engl.: „**b**inary **s**pace **p**artitioning“)
   - Volumen-Tests
     - AABB (engl.: „**a**xis-**a**ligned **b**ounding **b**ox“)
     - OABB (engl.: „**o**bject-**a**ligned **b**ounding **b**ox“)
     - Bounding sphere

   Manche Kollisions-Engines verwalten zusätzlich Informationen darüber, welche Körper sich zuletzt berührt haben und deshalb im nächsten Schritt als erstes überprüft werden sollten (zeitlicher Zusammenhang, engl.: „temporal cohesion“).

2. **Detailphase (narrow phase)**
   Mit den Objekten dieser Vorauswahl werden die detaillierten und rechnerisch aufwändigeren Kollisionstests durchgeführt. Dabei werden vielerlei geometrische Verfahren angewendet, die alle möglichen geometrischen Primitiven auf Kollision mit anderen Primitiven überprüfen können:

   - Kugel $\leftrightarrow$ Kugel
   - Zylinder $\leftrightarrow$ Quader
   - Dreieck $\leftrightarrow$ Dreieck
   - etc.

   Die Suche nach neuen Methoden und weiteren Verbesserungen der bisherigen Verfahren bildet die Grundlage für einen breiten Forschungsbereich.

Nachdem die Kollisions-Engine die *Kollisionserkennung* durchgeführt hat, ist es die Aufgabe anderer Module, die *Kollisionsbehandlung* durchzuführen.Im Fall der Physik-Engine wären das beispielsweise Reaktionen aus den Kollisionen wie Richtungs- und Rotationsänderungen. Im Falle einer Render-Engine können dies z.B. Funken oder andere visuelle Effekte sein.

Die drei Pfund Gehirn in jedem Menschen sind, soweit uns bekannt ist, das komplexeste und geordneteste Stück Materie im ganzen Weltall.

Isaac Asimov

4 Neuronale Netzwerke

Neuronale Netzwerke (NN) oder auch KNN (künstliche neuronale Netzwerke) (engl.: ANN für „artificial neural networks“) sind informationsverarbeitende Systeme, welche aus einer großen Anzahl von einfachen, miteinander verbundenen Einheiten bestehen. Im Gegensatz dazu bestehen z.B. Computer aus einem oder wenigen, dafür allerdings komplexen Prozessoren.

4.1 Geschichte der neuronalen Netzwerke

Schon 1942 wurden die ersten mathematischen Grundlagen für die heute noch verwendeten Prinzipien neuronaler Netzwerke gelegt. 1955 entstand am Massachusetts Institute of Technology (MIT) der erste Neurocomputer. In der Zeit bis 1969 boomte die Forschung und Entwicklung, da man glaubte, kurz davor zu stehen, intelligente Systeme bauen zu können. Es folgte eine Zeit der Ernüchterung, da sich das gesamte Gebiet der künstlichen Intelligenz als komplexer und umfangreicher erwies als zunächst angenommen. Es wurde zwar weiter geforscht und es ergaben sich immer neue Ansatzpunkte, insgesamt aber wurde es still um das Thema. Erst 1985 erfolgte die Renaissance neuronaler Netze durch Arbeiten von John Hopfield und die Einführung des Backpropagation-Lernverfahrens (siehe Abschnitt 4.4.4 auf Seite 50). Durch diese neuartigen Verfahren und Methoden sowie insbesondere durch die zunehmende Rechenleistung der Computer wuchsen die Fähigkeiten neuronaler Netzwerke. Gleichzeitig erhöhte sich die Lernkapazität und -geschwindigkeit immens. Seit 1986 entwickelt sich dieses Forschungsgebiet nahezu explosionsartig. Es existiert derzeit eine beachtliche Anzahl von Teilgebieten, Publikationen, Fachzeitschriften und Fachgruppen zum Thema neuronale Netzwerke [Zell, 2000, Kap. 1.6].

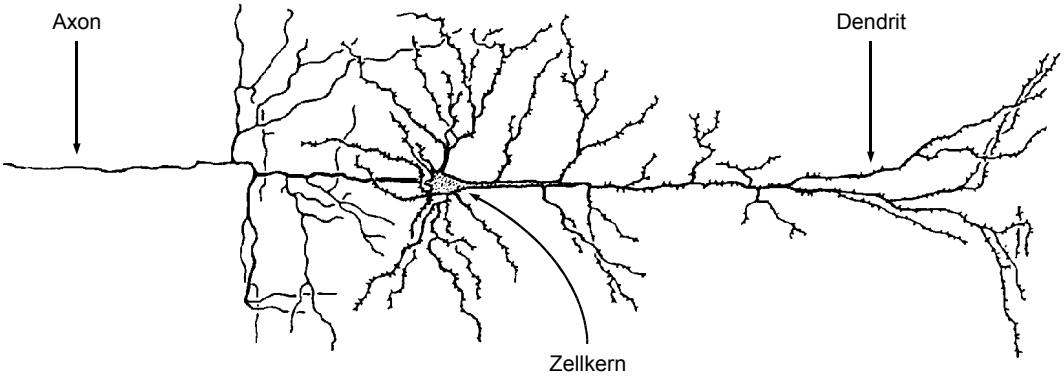
Die folgenden Kapitel können diesem Wissensumfang in keiner Weise gerecht werden. Deshalb werden lediglich Bereiche und Konzepte beschrieben, die für diese Thesis von wesentlichem Einfluss waren. Detailliertere Angaben zu den Informationen in den folgenden Abschnitten sind in [Zell, 2000, Kap. 2], [Russell und Norvig, 1995, Kap. 19], [Görz u. a., 2000, Kap.3] und [Netter, 2001] zu finden. Für Kurzinformationen zu medizinischen Fachbegriffen eignet sich der [Pschyrembel, 2001] am besten.

4.2 Neurobiologische Grundlagen

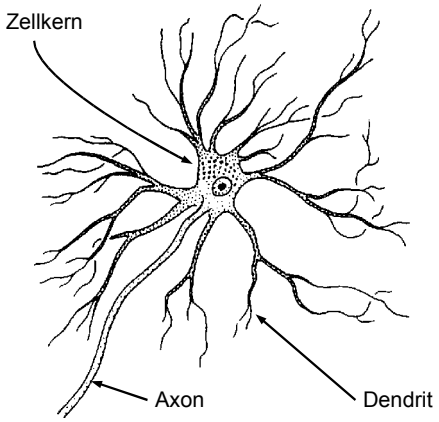
Um zu verstehen, wie die mathematischen Modelle und daraus abgeleitet die Prinzipien computergestützter neuronaler Netzwerke funktionieren, muss man sich zunächst die biologischen Grundlagen von Neuronen und deren Zusammenhänge im Menschen sowie in Tieren vor Augen führen.

4.2.1 Neuronen

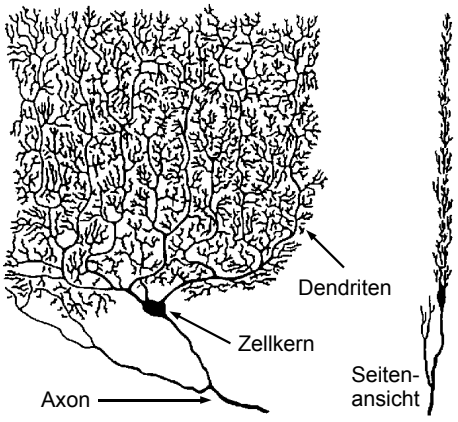
Der Mensch besitzt schätzungsweise 10^{11} Neuronen. Diese lassen sich grob nach ihrer Form und Lage im Gehirn kategorisieren (siehe Abbildung 4.1).



(a) Pyramidenzelle aus dem Cortex



(b) Motoneuron aus dem Rückenmark



(c) Purkinje-Zelle aus dem Kleinhirn

Abbildung 4.1: Formen und Größen von Neuronen (aus [Nicholls u. a., 2002, S. 13])

Allen Neuronen ist das **Neuronensoma** (Zellkörper) und der **Axon** gemeinsam. Bipolare Neuronen besitzen zusätzlich einen **Dendriten**, multipolare Neuronen mehrere. Unipolare Neuronen weisen lediglich einen Axon auf (siehe Abbildung 4.2 auf Seite 38)
Das Neuronensoma enthält den **Zellkern** und viele Organellen (z.B. Mitochondrien), welche die Zelle mit Energie versorgen. Das **endoplasmatische Retikulum** und der **Golgi-**

Apparat[1] produzieren kleine Bläschen (synaptische Vesikel) mit Neurotransmittern, welche zur Weiterleitung von Informationen an den Übergängen zwischen zwei Nervenzellen (**Synapsen**) benötigt werden.
Dendriten sind mehr oder weniger stark verästelte Fortsätze des Neurons, mit denen Eingangssignale aus anderen Neuronen aufgenommen werden. Dabei bezeichnet man das aufnehmende Neuron als **postsynaptisch** und die „sendenden“ Neuronen als **präsynaptisch**. Eine Nervenzelle ist typischerweise mit 2000 bis 10000 präsynaptischen Zellen verbunden.
Durch den Axon werden die aufgenommenen Impulse zu anderen Zellen weitergeleitet. Dieser unterscheidet sich sowohl in seiner Struktur als auch in seinen Ausmaßen von den Dendriten. Ein Axon kann von wenigen Millimetern bis zu einem Meter (z.B. Motoneuronen) lang werden. Im Gegensatz zu den Dendriten verästelt er sich erst im „Zielgebiet“ und mündet dort in die Synapsen.

4.2.2 Nervenfasern

Nervenfasern bestehen aus einer Doppelschicht von **Lipiden** (Fettmolekülen), deren hydrophile (griech.: „wasserliebend“) Enden nach außen zeigen. In diese Schicht sind zahlreiche Proteine eingebettet, die unterschiedliche Funktionen übernehmen. Wichtig für die Übertragung von Signalen sind die **Pumpenproteine**, die **Kanalproteine** und die **Rezeptorproteine**. Ihre Funktion wird in Abschnitt 4.2.3 auf Seite 39 noch genauer beschrieben.

In der Flüssigkeit, welche die Nervenzellen und -fasern umgibt, sind Salze gelöst und liegen somit in Form von Ionen vor. Wichtig für die nachfolgende Betrachtung sind vor allem Kalium- (K^+), Natrium- (Na^+) und Chlor-Ionen (Cl^-).
Die Zellmembran ist vorwiegend für die Kalium- und Chlor-Ionen durchlässig. Die Durchtrittsfähigkeit der Membran für Natrium-Ionen beträgt hingegen nur 1% derjenigen für Kalium-Ionen. Durch diese unterschiedlichen Voraussetzungen bilden die Ionen auf beiden Seiten der Membran ein Konzentrationsgefälle. Auf der Außenseite der Nervenzelle findet sich ein Überschuss an Na^+-Ionen, auf der Innenseite dagegen eine höhere Konzentration von K^+-Ionen. Zusätzlich befinden sich im Inneren der Nervenfaser organische Anionen, welche aufgrund ihrer Größe nicht durch die Membran diffundieren können. Jedes Konzentrationsgefälle einer Ionenart trägt dazu bei, dass die Summe aller dadurch hervorgerufenen Potenzialunterschiede -70mV im Inneren der Nervenfaser bezogen auf das Äußere beträgt. Dieser Wert wird als **Ruhepotenzial** bezeichnet.
Pumpenproteine sorgen zusätzlich dafür, dass das Konzentrationsverhältnis zwischen K^+- und Na^+-Ionen stetig wiederhergestellt wird. Dieses Verhältnis ändert sich sowohl durch Weiterleitung von Signalen als auch durch zurückdiffundierende Ionen. Unter Energiezu-

[1] Camillo Golgi, Pathologe, 1843-1926. Mit seinem Namen werden noch einige weitere Bestandteile des menschlichen Körpers bezeichnet. [Pschyrembel, 2001, S. 620]

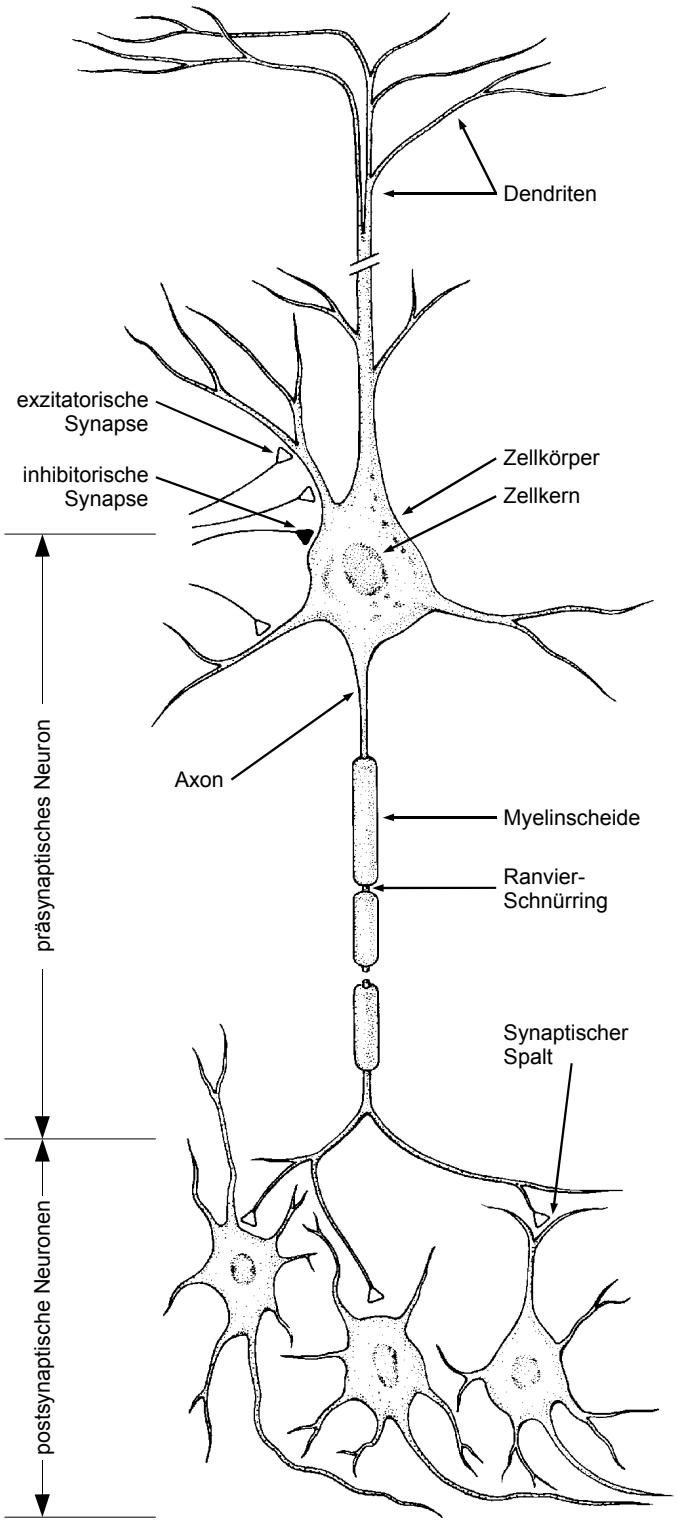


Abbildung 4.2: Aufbau eines Neurons (aus [Smith, 1998, S. 11])

fuhr von ATP (Adenosintriphosphat) „pumpen“ sie in einem Schritt jeweils drei Kalium- und zwei Natrium-Ionen auf die entgegengesetzten Seiten der Membran.

4.2.3 Signalübertragung

Zur Weiterleitung eines Signals muss das Membranpotenzial kurzfristig auf mindestens -40mV depolarisiert werden. Oberhalb dieser Schwelle stellt sich das **Aktionspotenzial** ein, welches der grundlegende Mechanismus zur Informationsübertragung ist (siehe Abbildung 4.3).

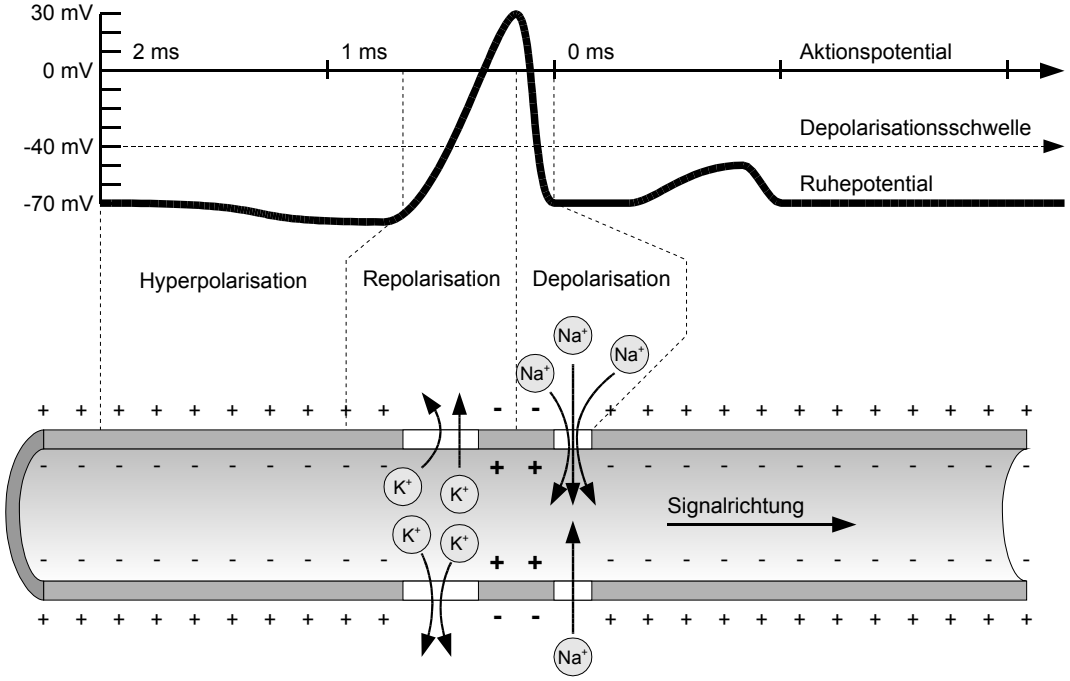


Abbildung 4.3: Signalfortpflanzung in einer Nervenfaser

Dabei steigt das Membranpotenzial während der **Depolarisationsphase** kurzfristig auf ca. +30mV an. Dies geschieht durch die Öffnung der Kanalproteine, welche für Natrium-Ionen durchlässig sind. Durch den schnellen Anstieg der Na^+-Konzentration im Inneren der Nervenfaser erhöht sich das Membranpotenzial.

0,5ms nach der Aktivierung der Natriumkanäle öffnen sich auch die Kanalproteine für Kalium-Ionen. Im Verlauf der nun folgenden **Repolarisationsphase** strömen K^+-Ionen aus dem Faserinneren nach außen und das Membranpotenzial fällt ab. Gleichzeitig schließen sich die Natriumkanäle.

Da sich die Kaliumkanäle langsamer als die Natriumkanäle schließen, folgt nun die **Hyperpolarisationsphase**, in der ein leichter Überschuß an Kaliumionen vorliegt, welcher für ein Membranpotenzial von ca. -75mV sorgt. In dieser sogenannten **Refraktärzeit** (ca. 1,5ms) ist keine weitere Aktivierung der Nervenfaser möglich. Während dieser Zeitspanne

nehmen unter anderem die Pumpenproteine ihre Arbeit auf und stellen das Konzentrationsgefälle der K^+- und Na^+-Ionen wieder her.

4.2.4 Myelinscheide

Normalerweise ist die Geschwindigkeit, mit der ein Signal entlang eines Axons weitergeleitet wird, proportional zur Quadratwurzel des Durchmessers. Für Signale, bei denen es auf schnelle Weiterleitung ankommt (z.B. Motorik), wären demnach unverhältnismäßig dicke Nervenfasern notwendig.
Die Myelinscheide dient zur schnelleren Weiterleitung des elektrischen Potenzials. Sie umgibt den Axon und ist in regelmäßigen Abständen an den sogenannten **Ranvier-Schnürringen** unterbrochen. Man kann ihre Funktion mit der eines Isolators vergleichen. Im Falle einer Aktivierung der Nervenzelle „springt“ das elektrische Potenzial quasi von Einschnürung zu Einschnürung, da es die isolierende Schicht nicht durchdringen kann. Dadurch wird erstens das Signal schneller weitergeleitet (ca. 100m/s bei 20µm Faserdicke) und zweitens ergibt sich eine erhebliche Energieersparnis, da weniger Ionenkanäle geöffnet und somit weniger Ionen von den Pumpenproteinen zurückbefördert werden müssen.
Das Aktionspotenzial einer Nervenzelle verläuft immer gleich. Somit kann unterschiedliche Information nicht im Signal*verlauf* codiert werden. Stattdessen variiert die *Frequenz* der Signale, auch **Feuerrate** genannt. Je intensiver ein Signal ist, desto höher ist die Frequenz.

4.2.5 Synapsen

Am Ende eines Axons stellen Synapsen die Verbindung zu postsynaptischen Neuronen her. Dabei kann man grundlegend zwischen **inhibitorischen** (hemmenden) und **exzitatorischen** (erregenden) Synapsen unterscheiden.[2]
Synapsen enthalten große Mengen an **synaptischen Vesikeln**, ca. 50nm kleine runde Bläschen, welche **Neurotransmitter** enthalten. Diese werden im Soma gebildet und anschließend mit ca. 400mm/d zu den Synapsen transportiert, wo sie sich dicht an der **präsynaptischen Membran** sammeln (siehe Abbildung 4.4 auf der nächsten Seite).
Angeregt durch das ankommende Signal öffnen sich die Kalzium-Kanäle und ermöglichen Ca^{2+}-Ionen das Eindringen in die Synapse. Dadurch verschmelzen die Membranen der Vesikel mit der präsynaptischen Membran und der Inhalt wird in den **synaptischen Spalt** ausgeschüttet.
Je nach Art des Neurotransmitters wird das postsynaptische Neuron angeregt oder gehemmt. Exzitatorische Neurotransmitter (z.B. Glutaminsäure) erhöhen kurzzeitig die Na^+- und K^+-Leitfähigkeit des postsynaptischen Neurons und lösen dadurch auf chemischem Weg ein Aktionspotential aus. Inhibitorische Transmitter (z.B. Gamma-aminobuttersäure) hingegen erhöhen nur die Leitfähigkeit der K^+-Kanäle und erzeugen so eine Hyperpolarisation.

[2] Eine weitere Unterscheidung betrifft chemische und elektrische Synapsen. Im Folgenden werden die besser untersuchten chemischen Synapsen beschrieben.

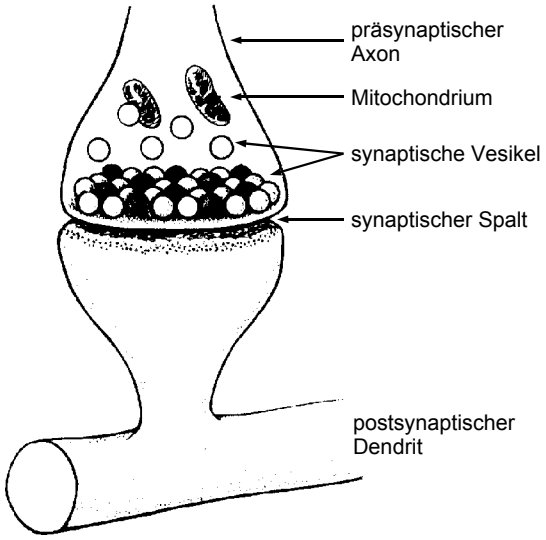


Abbildung 4.4: Aufbau einer Synapse (aus [Dayan und Abbott, 2001, S. 6])

Nach der Ausschüttung werden die Neurotransmitter durch Diffusion abtransportiert oder durch Enzyme abgebaut.

4.2.6 Summationseffekt

Die eigentliche Fähigkeit biologischer neuronaler Netze, sich zu verändern, zu lernen und zu „vergessen“, beruht auf dem Prinzip der Summation einkommender Signale.
Jede Synapse erzeugt im postsynaptischen Neuron nur eine kleine Potenzialänderung (siehe Abbildung 4.3 auf Seite 39 zum Zeitpunkt -1ms). Diese Änderung für sich genommen löst noch kein Aktionspotenzial aus. Allerdings fällt die Membranspannung relativ langsam in Richtung des Ruhepotenzials ab. Jeder weitere Impuls, der innerhalb dieser Zeit eintrifft, erhöht die Spannung um ein weiteres Stück, bis letztendlich das Aktionspotenzial erreicht ist und das Neuron ein Signal „feuert“. Bei diesem Vorgang spricht man von **zeitlicher Summation**.
Räumliche Summation beruht auf einem ähnlichen Prinzip, allerdings kommt es hier auf die Synchronität der Impulse mehrerer präsynaptischer Neuronen an.

Durch dieses wesentliche Prinzip ist es einem biologischen neuronalen Netz möglich, durch räumliche Anordnung die Anzahl und den Typ (exzitatorisch oder inhibitorisch) der Synapsen die Reaktion auf Eingangssignale zu verändern.

4.3 Funktionen in Lebewesen

Im Folgenden werden drei spezielle biologische neuronale Netze beschrieben, welche für die Thesis von besonderer Bedeutung sind, da Mechanismen oder Prinzipien dieser Netze im Verlauf dieser Thesis verwendet werden

4.3.1 Kleinhirn

> [...] Das Kleinhirn (lat.: „cerebellum“) ist ein dem Hirnstamm aufgelagerter Teil des Gehirns [...].
>
> Im Wesentlichen ist das Kleinhirn für die unbewusste Steuerung der Motorik, das motorische Lernen, die sensomotorische Integration und die richtige Zeitanpassung motorischer Reaktionen (z.B. bestimmter bedingter Reflexe) verantwortlich. [...] Das bedeutet, dass gut trainierte, automatisierte Bewegungsabläufe (für die also kein Nachdenken mehr erforderlich ist) im Kleinhirn gespeichert werden. Beispiele dafür sind die Koordination der Gesichtsmuskulatur beim Sprechen und die Bewegung der Finger beim Klavierspielen [...].
>
> [Wikipedia Cerebellum, 2005]

Die 1mm dicke Kleinhirnrinde besteht aus mehreren Schichten, welche unterschiedliche Arten von Neuronen beherbergen (siehe Abbildung 4.5).

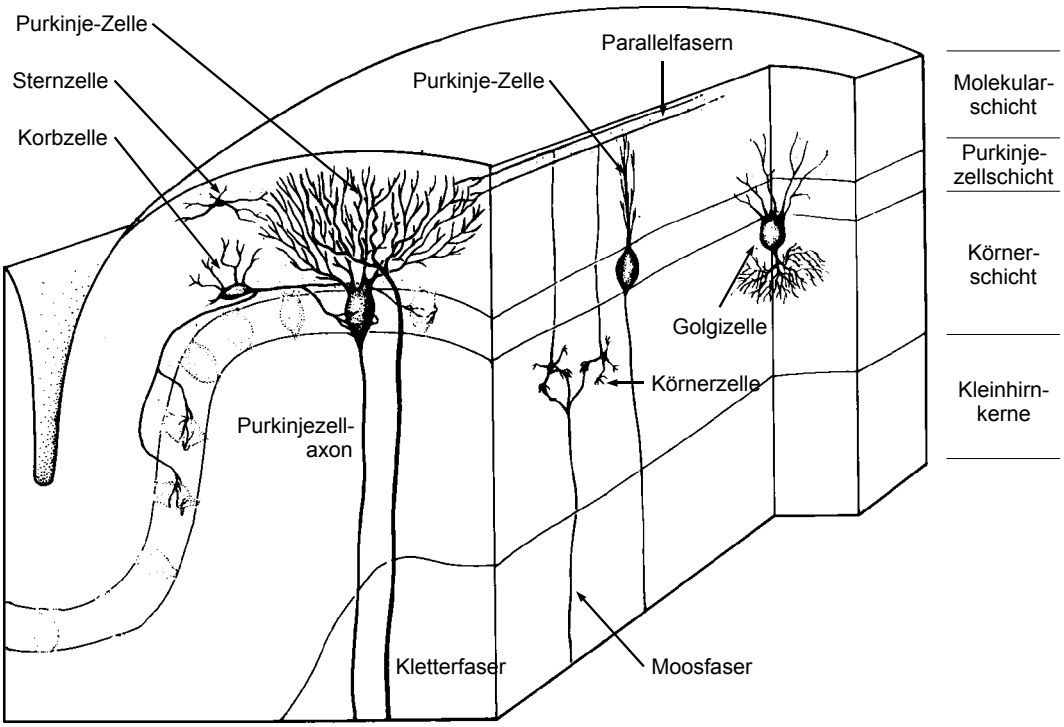


Abbildung 4.5: Aufbau der Kleinhirnrinde (aus [Smith, 1998, S. 22])

Sensorische Informationen über den Bewegungsablauf werden über die **Moosfasern** in die Kleinhirnrinde eingespeist. Diese enden in den **Körnerzellen**, welche in **Parallelfasern**

übergehen. Diese Fasern laufen parallel zueinander nahe der Kleinhirnrinde und gehen hauptsächlich Verbindungen mit den **Purkinje-Zellen** ein.
Die Purkinje-Zellen besitzen stark verzweigte Dendriten (siehe auch Abbildung 4.1(c) auf Seite 36) und können Signale von bis zu $2 \cdot 10^5$ Parallelfasern annehmen. Ihre Axone stellen die Ausgänge der Kleinhirnrinde dar. Allerdings steuern sie Bewegungsabläufe nicht aktiv, sondern modulieren nur vorgegebene Abläufe. Ihre Axone wirken lediglich inhibitorisch auf die **Basalganglien**, welche den eigentlichen motorischen Cortex steuern.
Die **Kletterfasern** steuern den Lernvorgang der Purkinje-Zellen, indem sie die Stärke der Synapsen verändern.
Sternzellen und **Korbzellen** regulieren die Feuerfrequenzen der Purkinje-Zellen, **Golgi-Zellen** die der Körnerzellen, indem sie Signale von den Parallelfasern aufnehmen und auf die regulierten Neuronen inhibitorisch einwirken.

Diese Fülle dieser Informationen vermittelt einen Eindruck darüber, dass es keine einfache Aufgabe ist, die Funktionen des Kleinhirns im Computer zu simulieren. [Smith, 1998] leitet aus der Anatomie des Kleinhirns und vor allem aus der Struktur der Purkinje-Zellen eine spezielle Form eines neuronalen Controllers ab. Dies gelingt allerdings nur unter starker Vereinfachung der Struktur.
Diese Thesis wird ebenfalls nicht versuchen, das Kleinhirn zu modellieren. Der Schwerpunkt wird zunächst auf einfacheren Funktionalitäten liegen, die sich in menschlichen und tierischen Nervensystemen außerhalb des Kleinhirns abspielen (z.B. Reflexe, Mustergeneratoren).

4.3.2 Muskeldehnreflex

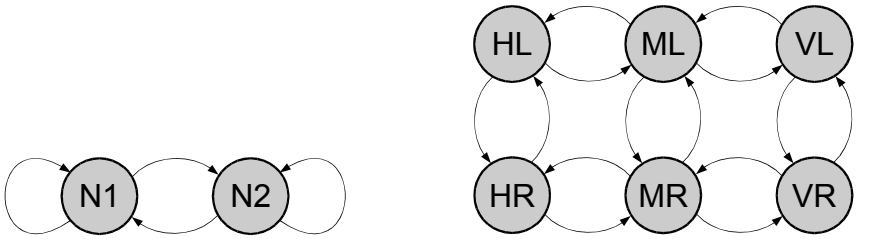
Viele Bewegungsarten des menschlichen Körpers sowie auch Bewegungen bei Tieren basieren auf dem Zusammenspiel zweier entgegengesetzt wirkender Muskeln (**Agonist** und **Antagonist**) wie z.B. dem Bizeps und dem Trizeps in den Armmuskeln. Die Koordination dieser beiden Muskeln geschieht in der Regel nicht erst im Kleinhirn, sondern bereits relativ nahe an den beteiligten Nerven im Rückenmark. Ein Grund dafür ist die Möglichkeit, schneller auf Störgrößen reagieren zu können, was durch lange Signalwege nicht möglich wäre. Das Gehirn selber muss nur noch „Sollgrößen“ wie die gewünschte Länge des Muskels und die Anspannung liefern. Die Neuronen im Rückenmark regeln die Stellgrößen entsprechend den äußeren Umständen.
Dafür benötigen sie Informationen über den Streckungszustand und die Spannung der Muskeln. Die Spannung wird vom **Golgi-Organ** gemessen. Dieser Rezeptor befindet sich an den Übergängen der Muskeln in die Sehnen. Sein Axon wird als **Ib**-Nervenfaser bezeichnet und endet im Rückenmark.
Die Streckung der Muskeln wird durch **intrafusale** Muskelspindeln gemessen. Diese Spindeln befinden sich gekapselt innerhab des Muskels und erfassen sowohl dessen aktuellen Streckungszustand als auch seine Änderung relativ zur Zeit. Ersterer wird dem Rückenmark durch **Ia**-Fasern signalisiert, letzterer durch **II**-Fasern.

Angeregt werden die Muskeln durch α- und γ-Motoneuronen. Die α-Neuronen lösen eine Kontraktion der Hauptmuskulatur aus, die γ-Neuronen eine Kontraktion der intrafusalen Muskelfasern.

Der Muskeldehnreflex ist einer der wichtigsten grundlegenden Reflexe. Er bewirkt, dass ein Muskel eine bestimmte Spannung und Stellung eines Gelenks aufrecht erhält. Ändert sich die Muskeldehnung, wird dies von den intrafusalen Muskelspindeln registriert. Diese regen daraufhin über die Ia-Fasern das α-Motoneuron des Muskels an, welcher dadurch der Änderung entgegenwirkt. Ohne eine Eingriffsmöglichkeit würde dieser Reflex dafür sorgen, dass ein Körper regungslos in Spannung verharrt. Erst durch Modulation dieses Regelkreises können kontrollierte Bewegungen erzeugt werden.
Bei Bewegungen spielt ein weiterer Aspekt des Muskeldehnungsreflexes eine wichtige Rolle: Die **reziproke Hemmung**. Soll der Muskel gezielt kontrahieren, so hemmen die Signale aus den Ia-Fasern das Motoneuron des antagonistischen Muskel, damit dieser der Bewegung *nicht* entgegenwirkt. [Baev, 1998, Kap. 9]

4.3.3 Mustergeneratoren (CPG's)

Mustergeneratoren sind kleine Verbände von untereinander verbundenen Neuronen (siehe Abbildung 4.6), welche eine bestimmte regelmäßige Signalform erzeugen. Sie finden sowohl im Menschen als auch bei Tieren Anwendung, vor allem in rhythmischen Bewegungen wie der Atmung, Peristaltik[3] und Bewegungen der Gliedmaßen zur Fortbewegung.



(a) einfacher CPG mit 2 Neuronen

(b) CPG aus dem Bewegungsapparat einer Küchenschabe

Abbildung 4.6: Arten von Mustergeneratoren

CPG's können durch Steuersignale Form, Frequenz und Phasenlage der erzeugten Signale variieren. Es wurden Versuchen mit Katzen durchgeführt, deren Hirnstamm durchtrennt wurde [Nicholls u. a., 2002, S. 366]. Sie wurden auf ein Laufband gestellt und das motorische Zentrum mit Strömen gereizt. Durch stärkere Ströme änderte sich zwar die „Kraft“ in den Bewegungen, die Schrittfrequenz blieb jedoch gleich. Bei Erhöhung der Laufbandgeschwindigkeit wechselte die Katze automatisch in die dem Tempo angemessenen Gangarten wie Schritt, Trab, Pass und Galopp. Dies zeigt, dass einfache, rhythmische Bewegungsvorgänge unabhängig von höheren Gehirnfunktionen sein können.

[3] Förderbewegung von Darm, Speiseröhre etc.

4.4 Künstliche neuronale Netzwerke

Nach einer Übersicht über die biologischen Grundlagen folgt nun die Übertragung auf künstliche, im Computer simulierbare neuronale Netzwerke.

4.4.1 Bestandteile

Künstliche neuronale Netzwerke bestehen prinzipiell aus Neuronen, auch Zellen, Elemente oder englisch „Units“ genannt, und Verbindungen zwischen diesen Neuronen, englisch auch „Links“ genannt.

4.4.1.1 Verbindungen

Eine Verbindung von einem Neuron i zu einem Neuron j wird mit einem Verbindungsgewicht $w_{i,j}$ gekennzeichnet und gibt die Stärke und die Art des Einflusses von i auf j an:

- $w_{i,j} > 0$: exzitatorisch (verstärkend, anregend)
- $w_{i,j} = 0$: kein Einfluss
- $w_{i,j} < 0$: inhibitorisch (abschwächend, hemmend)

4.4.1.2 Neuronen

Künstliche Neuronen sind durch folgende Merkmale charakterisiert:

- **Eingangssumme**
  Die Eingangssumme $\mathrm{net}_i(t)$ gibt die Summe aller eingehenden Signale an. Dies entspricht dem Summationseffekt natürlicher Neuronen (siehe Kapitel 4.2.6 auf Seite 41). Eingangssignale sind die Ausgangswerte $o_j(t)$ aller mit dem Neuron i verbundenen Neuronen multipliziert mit dem jeweiligen Verbindungsgewicht $w_{j,i}$. Zusätzlich wird ein Offset θ_i (**Bias**) addiert, um den Arbeitspunkt des Neurons verändern zu können.
  $$\mathrm{net}_i(t) = \theta_i + \sum_{j=0}^{N} o_j(t) \cdot w_{j,i} \tag{4.1}$$
- **Aktivierungszustand**
  Der Aktivierungszustand $a_i(t)$ des Neurons i entspricht dem Membranpotenzial des biologischen Neurons (siehe Abschnitt 4.2.3 auf Seite 39). Dieser Zustand ergibt sich aus der Eingangssumme $\mathrm{net}_i(t)$, über welche eine **Aktivierungsfunktion** $\mathrm{act}(x)$ gelegt wird.
  $$a_i(t) = \mathrm{act}(\mathrm{net}_i(t)) \tag{4.2}$$
  Aktivierungsfunktionen können symmetrisch ($\mathrm{act}_{\doteq}(t)$) und asymmetrisch ($\mathrm{act}_{\doteq}(t)$) zur X-Achse verlaufen. Für welche Art von Symmetrie man sich entscheidet, hängt stark von der Verwendung des künstlichen neuronalen Netzes ab.

Typische Aktivierungsfunktionen sind:

- Sprung (siehe Abbildung 4.7(a) auf der nächsten Seite)

$$\mathrm{act}_{\doteq}(x) = \begin{cases} +1 & \text{wenn } x > 0 \\ 0 & \text{wenn } x = 0 \\ -1 & \text{wenn } x < 0 \end{cases} \tag{4.3}$$

$$\mathrm{act}_{\doteq}(x) = \begin{cases} +1 & \text{wenn } x > 0 \\ 0 & \text{wenn } x \leq 0 \end{cases} \tag{4.4}$$

- Begrenzung (siehe Abbildung 4.7(b) auf der nächsten Seite)

$$\mathrm{act}_{\doteq}(x) = \begin{cases} +1 & \text{wenn } x \geq 1 \\ x & \text{wenn } 0 < x < +1 \\ 0 & \text{wenn } x \leq 0 \end{cases} \tag{4.5}$$

$$\mathrm{act}_{\doteq}(x) = \begin{cases} +1 & \text{wenn } x \geq +1 \\ x & \text{wenn } -1 < x < +1 \\ -1 & \text{wenn } x \leq -1 \end{cases} \tag{4.6}$$

- Sigmoid (Logistisch) (siehe Abbildung 4.7(c) auf der nächsten Seite)

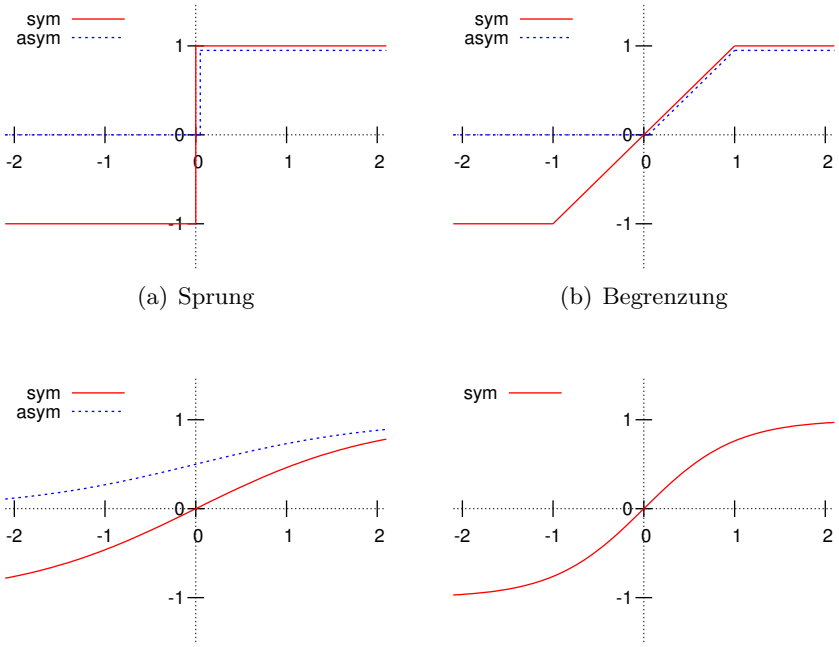
$$\mathrm{act}_{\doteq}(x) = \frac{2}{1 + e^{-x}} - 1 \tag{4.7}$$

$$\mathrm{act}_{\doteq}(x) = \frac{1}{1 + e^{-x}} \tag{4.8}$$

- Tangens Hyperbolicus (siehe Abbildung 4.7(d) auf der nächsten Seite)

$$\mathrm{act}_{\doteq}(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{4.9}$$

- **Ausgangssignal**
  Das Ausgangssignal $o_i(t)$ wird über eine **Ausgabefunktion** aus dem Aktivierungszustand berechnet. Häufig ist diese Ausgabefunktion die Identitätsfunktion $\mathrm{out}(x) = x$. Durch andere Funktionen (z.B. $\mathrm{out}(x) = \tanh x$) lässt sich das Ausgangssignal z.B. auf einen Wertebereich zwischen $(-1, +1)$ „normieren“.



(a) Sprung (b) Begrenzung

(c) Sigmoid/Logistisch (d) Tangens Hyperbolicus

Abbildung 4.7: Formen von Aktivierungsfunktionen

Zeitlich diskrete Neuronen

Somit errechnet sich das Ausgangssignal eines Neurons durch

$$o_i(t) = \text{out}\left(\text{act}\left(\text{net}_i(t)\right)\right), \tag{4.10}$$

$$= \text{out}\left(\text{act}\left(\theta_i + \sum_{j=0}^{N} o_j(t) \cdot w_{j,i}\right)\right). \tag{4.11}$$

Die Ausführung dieser Gleichung im Zuge der Simulation des neuronalen Netzes wird als **Propagation** bezeichnet. Sie findet Verwendung in zeitlich diskreten neuronalen Netzwerken (DTNN).

Zeitlich kontinuierliche Neuronen

Eine andere Variante zur Beschreibung des Aktivierungszustandes des Neurons ist die Differentialgleichung

$$\tau_i \frac{d}{dt} a_i(t) = -a_i(t) + \text{act}\left(\text{net}_i(t)\right). \tag{4.12}$$

Hierbei entfällt der Anteil $a_i(t - \delta t)$ für die Aktivierungsfunktion $\text{act}(t)$. τ_i ist eine Zeitkonstante, welche die „Trägheit“ des Neurons bei der Weiterleitung von Signalen berücksichtigt.
Diese Formel ist die Grundlage für zeitlich kontinuierliche neuronale Netzwerke (CTNN, CTRNN), welche z.B. für die Realisierung von Mustergeneratoren benötigt werden.

4.4.2 Kategorien

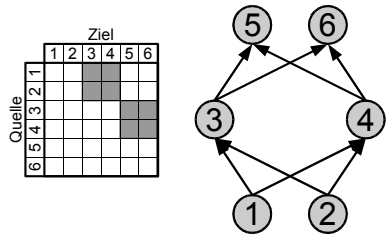
Künstliche neuronale Netze lassen sich grob anhand folgender Kriterien unterscheiden und kategorisieren:

- **Zeitliches Verhalten**
  - Zeitlich diskret, schrittweise Berechnung (siehe Formel 4.11 auf der vorherigen Seite), DTNN (engl.: „**d**iscrete **t**ime **n**eural **n**etwork“)
  - Zeitlich kontinuierlich, Berechnung über Differentialgleichung (siehe Formel 4.12 auf der vorherigen Seite), CTNN (engl.: „**c**ontinuous **t**ime **n**eural **n**etwork“)
- **Wertebereich**
  - Binär ($o_i(t) \in [1, 0]$)
  - Diskret ($o_i(t) \in \mathbb{N}$)
  - Kontinuierlich ($o_i(t) \in \mathbb{R}$)
- **Informationsfluss**
  - Von den Eingängen schichtweise zu den Ausgängen (vorwärtsgerichtetes Netzwerk, engl.: „feedforward“)
  - Von den „hinteren“ Neuronen zurück zu den „vorderen“ Neuronen (rekurrentes Netzwerk)
- **Komplexität**
  - Einlagig
  - Mehrlagig
  - Komplett untereinander verbunden

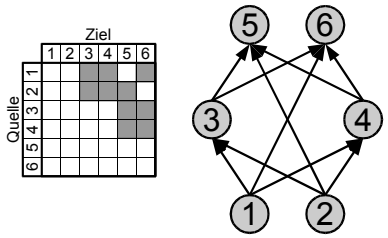
Abbildung 4.8 auf der nächsten Seite zeigt einige Strukturen neuronaler Netzwerke mit der dazugehörigen **Gewichtsmatrix**. Diese Matrix stellt die Stärke der Verbindungen zwischen den Neuronen dar. In der Abbildung wird lediglich dargestellt, *ob* eine Verbindung besteht. Andere Darstellungsformen dieser Matrix repräsentieren die Stärke der Verbindung zusätzlich durch unterschiedliche Farben oder durch unterschiedlich große Formen wie Quadrate oder Kreise.

4.4.3 Vorwärtsgerichtete Netze

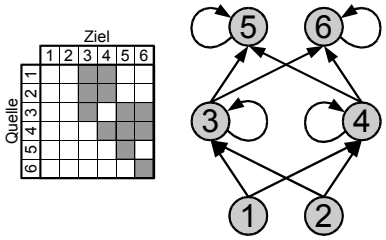
Anfang der 60er Jahre wurde von Frank Rosenblatt das **Perzeptron** entwickelt. Dieses ist charakterisiert durch seine einfache vorwärtsgerichtete Struktur (siehe Abbildung 4.9 auf der nächsten Seite). Dabei sind die Verbindungen der Eingabeschicht zur verdeckten



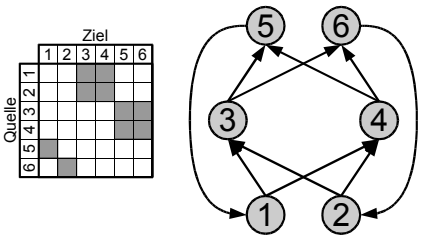
(a) Feedforward-Netzwerk



(b) Feedforward-Netzwerk mit „Abkürzungen“



(c) Netzwerk mit rückgekoppelten Neuronen



(d) Netzwerk mit Rückführung

Abbildung 4.8: Arten von neuronalen Netzen und die dazugehörige Gewichtsmatrix

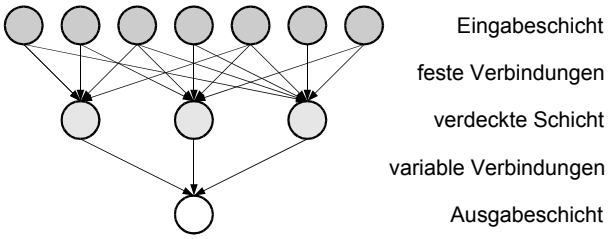


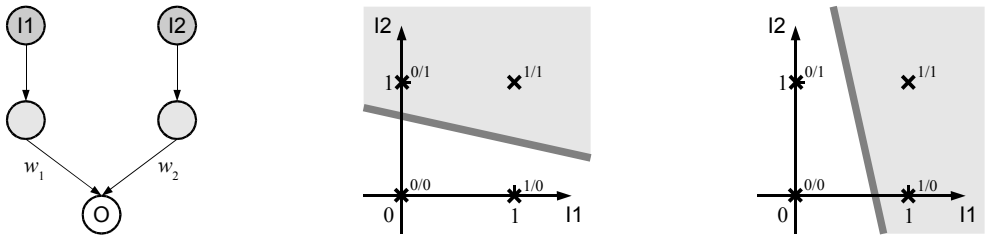
Abbildung 4.9: Aufbau eines einstufigen Perzeptrons

Schicht fest vorgegeben und können nicht verändert werden. Lediglich die Verbindungen von der verdeckten Schicht zur Ausgabeschicht lassen sich verändern und trainieren. Das Ausgabeneuron besitzt eine Schwellwertfunktion, d.h. es gibt entweder 0 oder 1 aus. Das Perzeptron eignet sich z.B. zur Erkennung einfacher Eingabemuster. Dazu ordnet man die Eingabeneuronen wie die Zellen auf der Netzhaut des Auges als Matrix an und erzeugt proportional zu der von ihnen empfangenen Helligkeit das Ausgangssignal.

Anhand des Perzeptrons lassen sich leicht Untersuchungen durchführen, die Aufschluss darüber geben, wieviele unterschiedliche Eingaben ein neuronales Netzwerk unterscheiden und an den Ausgängen repräsentieren kann.
Hinsichtlich der Lernfähigkeit besagt ein Theorem von [Rosenblatt, 1962]:

> Der Lernalgorithmus des Perzeptrons konvergiert in endlicher Zeit, d.h. das Perzeptron kann in endlicher Zeit alles lernen, was es repräsentieren kann.

Allerdings unterliegt gerade die Repräsentationsfähigkeit starken Einschränkungen. Geht man von n Eingabeneuronen aus, so unterteilt jedes Neuron den möglichen Eingaberaum durch eine $(n-1)$-dimensionale Hyperebene.



(a) Aufbau des Perzeptrons

(b) Separationsebene 1

(c) Separationsebene 2

Abbildung 4.10: XOR-Problematik eines einstufigen Perzeptrons

Diese Unterteilung macht es unmöglich, ein einstufiges Perzeptron eine XOR-Verknüpfung[4] lernen zu lassen. Die zusammengehörigen Eingabemengen $\{(0/0),(1/1)\}$ und $\{(0/1),(1/0)\}$ lassen sich nicht durch eine einfache Hyperebene separieren. Man spricht hierbei von der **XOR-Problematik** (siehe Abbildung 4.10).
Erst ein zweistufiges Perzeptron (zwei verdeckte Schichten) wäre in der Lage, dieses Problem zu lösen. Mit der Anzahl der verdeckten Schichten steigt die Komplexität der repräsentierbaren Ausgabemenge [Zell, 2000, Kap. 7].

4.4.4 Backpropagation

Backpropagation ist ein Lernverfahren für neuronale Netzwerke. Um dieses Verfahren durchzuführen, benötigt man den „Fehler“ E des Netzes. Dieser berechnet sich aus der Differenz der tatsächlichen Ausgabe o_i zur gewünschten Ausgabe t_i. Dabei kann man den Fehler als Vektor $\vec{E}$ der Differenzen an jedem Ausgabeneuron ausdrücken, aber auch als Summe E aller Differenzen.

$$\vec{E} = \begin{pmatrix} t_1 - o_1 \\ t_2 - o_2 \\ \vdots \\ t_N - o_N \end{pmatrix} \qquad \text{oder} \qquad E = \frac{1}{2}\sum_{i}^{N}(t_i - o_i)^2 \tag{4.13}$$

Jedes noch nicht korrekt eingestellte Gewicht innerhalb des Netzwerks trägt zu diesem Fehler einen Teil bei. Dabei beeinflussen die Gewichte nahe der Ausgabeschicht den Fehler mehr als die davor liegenden Schichten.
Die Verbindungsgewichte w_{i_j} werden nun durch ein **Gradientenverfahren** mit

$$\Delta w_{i,j} = \eta o_i \delta_j \tag{4.14}$$

[4] Exklusiv-Oder

und

$$\delta_j = \begin{cases} \dot{\text{act}}(\text{net}_j)(t_j - o_j) & \text{falls Neuron } j \text{ ein Ausgabeneuron ist} \\ \dot{\text{act}}(\text{net}_j) \sum_k^N \delta_k w_{j,k} & \text{falls Neuron } j \text{ ein verdecktes Neuron ist} \end{cases} \tag{4.15}$$

korrigiert, wobei η der **Lernfaktor** ist, der die „Schnelligkeit“ des Lernverfahrens angibt (Herleitung siehe [Zell, 2000, Kap. 8.4]). Diese Korrektur muss in mehreren Schritten hintereinander erfolgen, da das Gradientenverfahren die optimale Lösung nicht in einem Schritt berechnen kann, sondern sich iterativ annähern muss.
Trainiert man das neuronale Netzwerk nach jedem angelegten Eingabemuster, so spricht man von **Online-Learning**. Daneben existiert auch das **Batch-Learning** (Batch, engl.: „Stoß, Stapel“), bei dem man die Fehler für eine Reihe von Eingabemustern zunächst summiert, bevor die Gewichte mit dieser Summe korrigiert werden.

Das Backpropagation-Lernverfahren kann unter ungünstigen Umständen keine ideale Lösung finden oder zu langsam konvergieren. Aus diesem Grund wurde das Verfahren verbessert bzw. wurden andere Verfahren entworfen (siehe [Zell, 2000, Kap. 9]):

- Konjugierter Gradientenabstieg
- Lernen mit Gewichtsabnahme
- Gradientennormierung
- Backpropagation 2ter Ordnung
- Quickprop
- etc.

4.4.5 Rekurrente Netze

Rekurrente neuronale Netzwerke eignen sich hervorragend zur Erkennung oder Vorhersage von zeitlichen Folgen. Man unterscheidet prinzipiell zwei Architekturen [Zell, 2000, Kap. 11].

4.4.5.1 Jordan-Netzwerk

Ein Jordan-Netzwerk ist charakterisiert durch die Rückführung der Ausgänge auf eine spezielle Schicht sogenannter **Kontextneuronen**, welche mit den Eingabeneuronen zusammen Informationen in das Netzwerk einspeisen [Jordan, 1986] (siehe Abbildung 4.11 auf der nächsten Seite). Die Ausgangssignale werden mit der gemeinsamen Stärke γ in die Kontextneuronen eingespielt. Meistens wird dieser Wert auf 1 gesetzt. Die Kontextneuronen besitzen ihrerseits durch eine direkte Rückkopplung der Stärke λ eine Art „Langzeitgedächtnis“, das seine Informationen je nach Wahl von λ schneller oder langsamer

„vergisst“. Die Gewichtswerte γ und λ sind nicht trainierbar, sondern von vorneherein fest eingestellt.

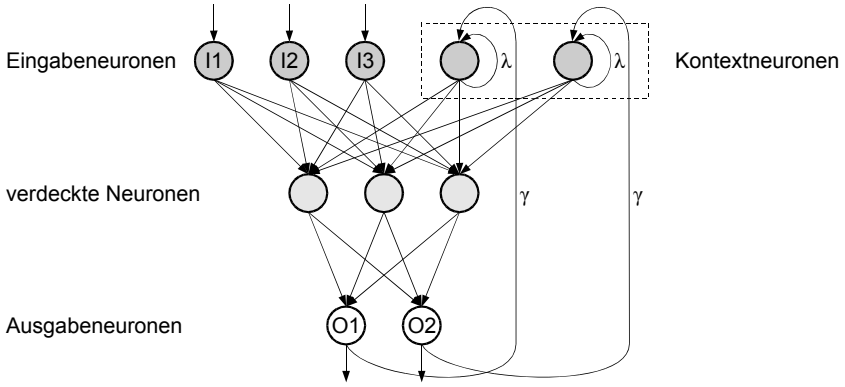


Abbildung 4.11: Aufbau eines Jordan-Netzwerks

4.4.5.2 Elman-Netzwerk

Ein Elman-Netzwerk ist prinzipiell ähnlich wie ein Jordan-Netzwerk aufgebaut [Elman, 1990]. Allerdings erhalten die Kontextneuronen ihre Eingangssignale nun von der verdeckten Schicht. Zusätzlich entfallen die rekurrenten Verbindungen der Kontextneuronen (siehe Abbildung 4.12).

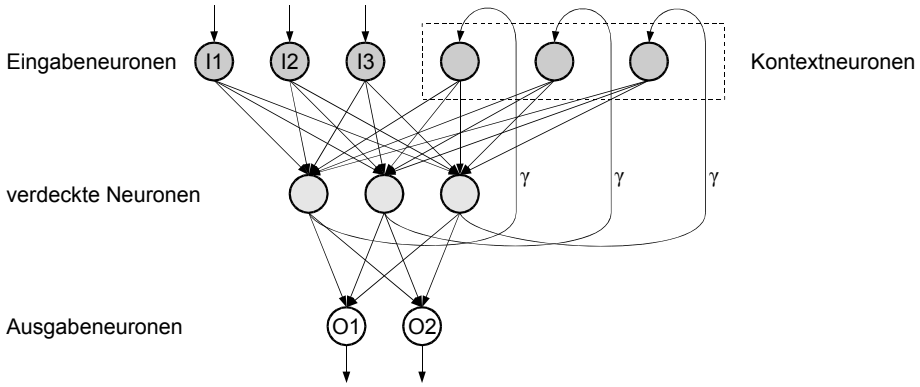


Abbildung 4.12: Aufbau eines Elman-Netzwerks

Elmann-Netzwerke sind durch die Rückführung der in den verdeckten Neuronen codierten internen Repräsentation flexibler als Jordan-Netzwerke, da sie unabhängig von der Ausgabesequenz sind.

Die Komplexität eines Elman-Netzwerks ist noch steigerbar, indem man mehrere verdeckte Schichten entwirft, welche jeweils mit einer eigenen Schicht von Kontextneuronen verbunden sind.

4.4.5.3 Lernverfahren

Um rekurrente Netzwerke lernfähig zu machen, ist eine Abwandlung des Backpropagation-Lernverfahrens notwendig. Neuronen, welche rekurrent verschaltet sind, werden zeitlich „aufgerollt“, d.h. in eine Kette von Neuronen verwandelt, welche sich mit dem Standard-Backpropagation-Lernverfahren behandeln lässt. Jedes Neuron dieser Kette besitzt dabei den Aktivierungszustand vom vorherigen Zeitschritt (siehe Abbildung 4.13).

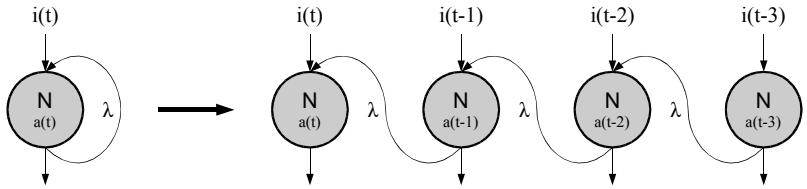


Abbildung 4.13: Umwandlung eines rekurrenten Netzwerks in ein vorwärtsgerichtetes Netzwerk

In der Praxis realisiert man dieses Verfahren durch einen Speicher in jedem Neuron, in welchem die Aktivierungswerte der letzten Schritte abgelegt sind.
Lernverfahren nach diesem Prinzip sind z.B.

- BPTT (**b**ack**p**ropagation **t**hrough **t**ime)
- RTRL (**r**eal-**t**ime **r**ecurrent **l**earning)
- Eine Kombination der beiden Verfahren

4.5 Neuronale Oszillatoren

Künstliche neuronale Oszillatoren, welche ähnliche Funktionen erfüllen wie ihre natürlichen Vorbilder, die CPG's (siehe Abschnitt 4.3.3 auf Seite 44), müssen zeitlich kontinuierlich modelliert werden. Zeitdiskrete Netzwerke wie Jordan- oder Elman-Netzwerke scheiden deshalb aus, weil sie von Schritt zu Schritt lediglich alternierende Werte erzeugen können. Neuronale Oszillatoren müssen hingegen in der Lage sein, über mehrere Simulationsschritte hinweg ein kontinuierlich fallendes bzw. steigendes Signal zu erzeugen.

4.5.1 Neuronaler Oszillator nach Wilson-Cowan

Ein neuronaler Oszillator nach Wilson-Cowan besteht aus zwei Neuronen, welche vollständig miteinander verbunden sind [Nakada u. a., 2004, Kap. 3] (siehe Abbildung 4.14).

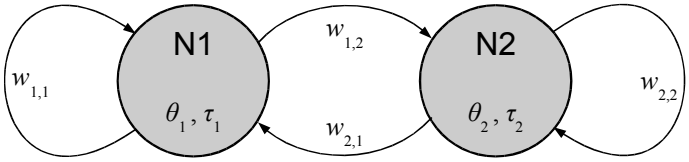


Abbildung 4.14: Mustergenerator nach Wilson-Cowan

Aufgrund seines einfachen Designs ist dieser Mustergenerator mathematisch leicht zu analysieren. Die Formel zur Berechnung der Ausgangssignale lautet

$$\tau_1 \frac{d}{dt} a_1(t) = -a_1(t) + \text{act}\left(w_{1,1} a_1(t) + w_{2,1} a_2(t) + \theta_1\right) \tag{4.16}$$

$$\tau_2 \frac{d}{dt} a_2(t) = -a_2(t) + \text{act}\left(w_{2,2} a_2(t) + w_{1,2} a_1(t) + \theta_2\right). \tag{4.17}$$

Abbildung 4.15 auf der nächsten Seite zeigt den Funktionsverlauf eines Mustergenerators mit folgenden Parametern:

$$w_{1,1} = 2{,}5 \qquad w_{1,2} = 1{,}25 \qquad \tau_1 = \tau_2 = 0{,}25$$
$$w_{2,2} = 0 \qquad w_{2,1} = -2{,}5 \qquad \theta_1 = \theta_2 = 0$$
$$\text{act}(x) = \tanh(x) \qquad \text{out}(x) = x$$

Die X-Achse zeigt das Ausgangssignal o_1 des Neurons N1, die Y-Achse das Ausgangssignal o_2 von N2. Beide Neuronen haben zu Beginn eine leichte Aktivierung von 0,01, damit sich der Schwingungsvorgang aufbauen kann.
Man sieht an der Spiralenform, dass sich die Ausgangssignale phasenverschoben zueinander aufschaukeln, bis sie bedingt durch die Begrenzung der $\tanh(x)$-Funktion auf einen Wertebereich von $(-1, 1)$ eine Maximalamplitude erreichen.
Durch Variation der Parameter $\tau_{1/2}$, $\theta_{1/2}$, $w_{1/2,1/2}$ erhält man andere Schwingungsmuster (siehe dazu die Abbildungen auf Seiten 56–57).

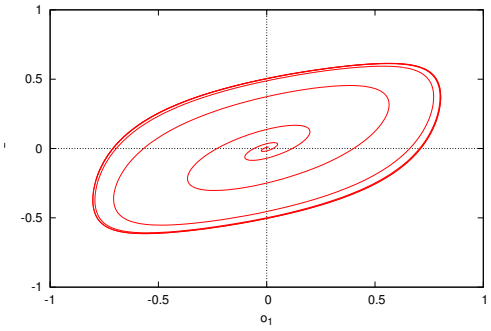


Abbildung 4.15: Signalverlauf eines Mustergenerators nach Wilson-Cowan

Es ist zu erkennen, dass sich die Amplituden und Offsets der Ausgangssignale einstellen lassen. Bei der Variation der Parameter kann es allerdings schnell passieren, dass der Oszillator aufhört zu schwingen und die Ausgänge nur noch einen stetigen Wert ausgeben (siehe Abbildung 4.16). Wann und mit welchen Parametersätzen dies geschieht, kann man mit Hilfe von **Bifurkationsdiagrammen**[5] darstellen [Wang und Blum, 1995]. Ein andere Form der Stabilitätsanalyse eines Wilson-Cowan-Oszillators ist in [Dayan und Abbott, 2001, Kap. 7.5] zu finden.

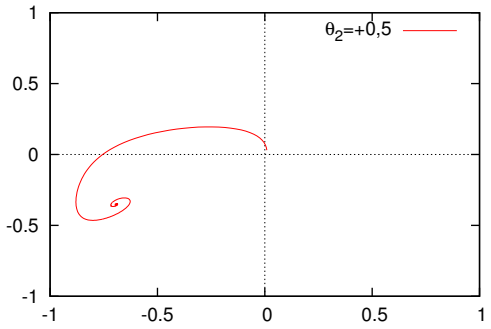
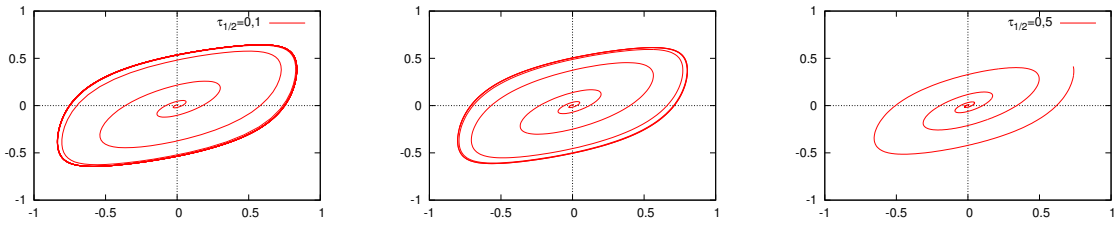


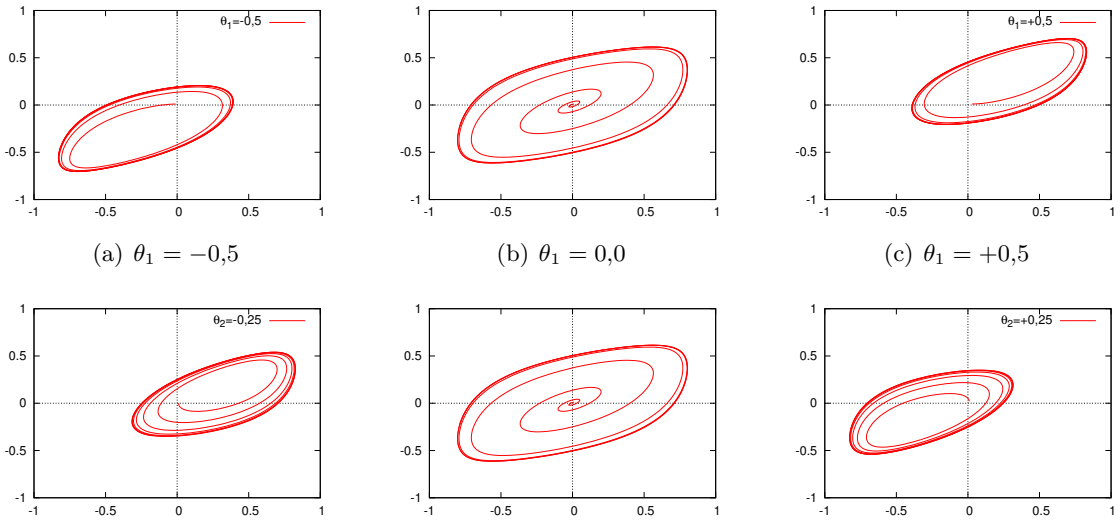
Abbildung 4.16: Abbruch des Schwingungsvorgangs

[5] Bifurkation (lat.: „bi“ zweifach, „furca“ Gabel) bezeichnet eine sprunghafte Verhaltensänderung nichtlinearer Systeme bei Veränderung eines Parameters. Strebt das System z.B. zunächst nur einem Grenzwert zu und bei Veränderung eines Parameters sprunghaft zwei oder mehreren Grenzwerten, so würde man auf einem Diagramm, welches die Grenzwerte gegen den veränderbaren Parameter aufträgt, eine gabelförmige Kurve sehen [Wikipedia Bifurkation, 2005].



(a) $\tau = 0{,}1$ (b) $\tau = 0{,}25$ (c) $\tau = 0{,}5$

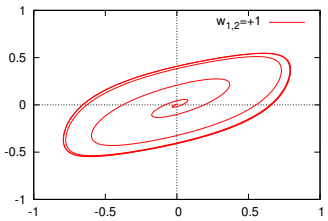
Abbildung 4.17: Variation der Zeitkonstanten $\tau_{1/2}$



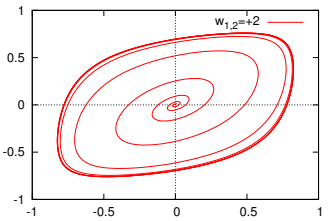
(a) $\theta_1 = -0{,}5$ (b) $\theta_1 = 0{,}0$ (c) $\theta_1 = +0{,}5$

(d) $\theta_2 = -0{,}5$ (e) $\theta_2 = 0{,}0$ (f) $\theta_2 = +0{,}5$

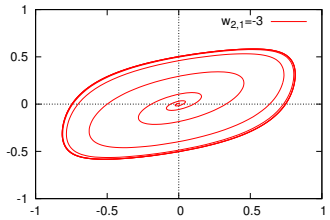
Abbildung 4.18: Variation des Bias $\theta_{1/2}$



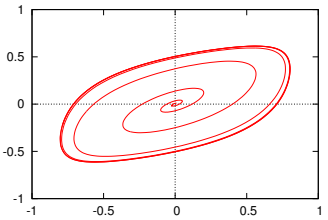
(a) $w_{1,2} = 1$



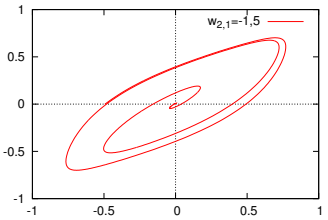
(b) $w_{1,2} = 2$



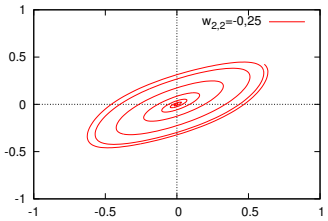
(c) $w_{2,1} = -3$



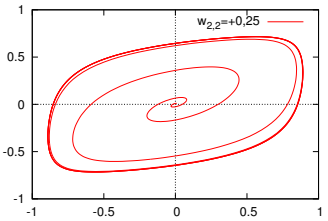
(d) Ausgangszustand



(e) $w_{2,1} = -1{,}5$



(f) $w_{2,2} = -0{,}25$



(g) $w_{2,2} = +0{,}25$

Abbildung 4.19: Variation der Verbindungsgewichte

Perfektion ist nicht dann erreicht, wenn nichts mehr hinzuzufügen ist, sondern wenn nichts mehr da ist, was weggenommen werden sollte.

Antoine de Saint Exupery

5 Evolutionäre Algorithmen

Evolutionäre Algorithmen gehören zu der Gruppe der stochastischen Suchverfahren. Sie finden vor allem bei stark nichtlinearen oder diskontinuierlichen Problemstellungen Verwendung. Eine Vielzahl möglicher Lösungen wird gleichzeitig ausgewertet und nach dem Leitsatz der Evolutionstheorie „Der Stärkere überlebt“ beibehalten oder aussortiert. Unter Zuhilfenahme zusätzlicher Faktoren finden sich so am Ende des Prozesses eine oder mehrere gute Lösungen.

5.1 Geschichte der evolutionären Algorithmen

Entgegen der weitläufigen Meinung stammt die **Evolutionstheorie** *nicht* von Charles Darwin, der 1859 mit seinem Werk „On the Origin of Species by Means of Natural Selection“ Geschichte schrieb. Schon Jean-Baptiste de Lamarck (1744-1829) vertrat die Ansicht, dass Lebewesen sich weiterentwickeln[1] und dabei neue, bessere oder andere Fähigkeiten an den Tag legen können. Wichtige Elemente bei dieser Entwicklung sind die Faktoren **Selektion** und **Mutation**.
Durch moderne Erkenntnisse aus der Biologie über Vererbungslehre, den Aufbau von Zellen und den Informationsgehalt der DNS[2] wurden die Prinzipien der Evolutionstheorie verständlicher und erklärbarer (siehe Abschnitt 5.2 auf der nächsten Seite).

Anfang der 50er Jahre wurde begonnen, die bis dahin gesammelten theoretischen Erkenntnisse für praktische Anwendungen außerhalb der Biologie zu abstrahieren und aufzubereiten. Im Bereich der Informatik entstanden aus diesen Bemühungen vier verschiedene Forschungszweige:

[1] Das Wort Evolution stammt aus dem lateinischen „evolvere“ für „abwickeln“, „entwickeln“ [Wikipedia Evolution, 2005].

[2] Die DNS (Desoxyribonukleinsäure) ist der Träger der genetischen Information. Diese Information liegt in Form von Paaren der Basen Cytosin (C) und Guanin (G) bzw. Adenin(A) und Thymin (T) vor, welche die beiden Stränge der Doppelhelix des Moleküls verbinden. Dabei entscheidet die Reihenfolge der Paare über den Informationsgehalt [Pschyrembel, 2001, S. 376].

1. Evolutionäre Strategien
2. Evolutionäre Programmierung
3. Genetische Algorithmen
4. Genetische Programmierung

Alle vier Richtungen verwenden die Grundmechanismen der Evolutionstheorie, allerdings jeweils in verschiedenen Ausprägungen.
Die folgenden Abschnitte stellen zunächst die grundlegenden Operationen vor, um danach auf die vier Verfahren und ihre Schwerpunkte bei der Verwendung der Operationen genauer einzugehen.

5.2 Grundbausteine evolutionärer Algorithmen

Abbildung 5.1 zeigt den Ablauf eines evolutionären Algorithmus' mit den dafür notwendigen Bestandteilen, welche im Folgenden als **Evolutionsoperatoren** bezeichnet werden. [Weicker, 2002, Kap. 1.2], [Polheim, 1999, Kap. 3], [Nissen, 1997, Kap. 1.3].

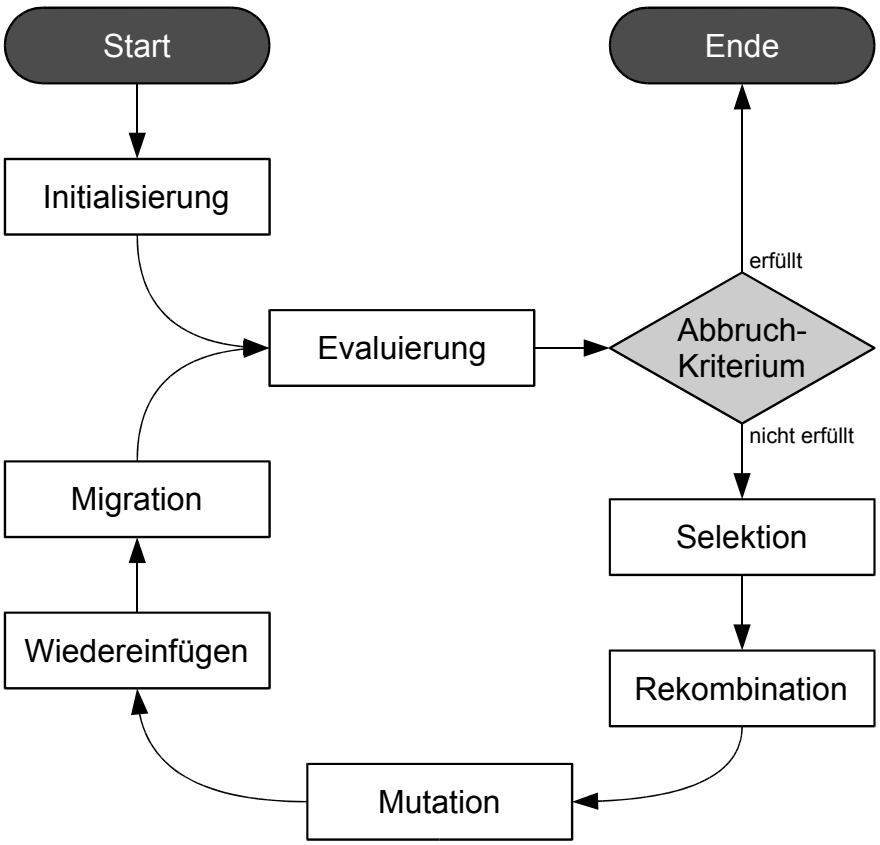


Abbildung 5.1: Ablauf eines evolutionären Algorithmus'

Nicht jeder Algorithmus verwendet dabei alle Operatoren. Die Unterschiede der vier Verfahren werden in Abschnitt 5.3 auf Seite 72 genauer erläutert.

5.2.1 Individuum

Ein Individuum steht für eine mögliche Lösung der zu optimierenden Problemstellung. Dies kann z.B. ein Punkt in einem mehrdimensionalen Raum oder ein neuronales Netzwerk sein.

Die wesentlichen Eigenschaften und die „äußerliche Erscheinung“ des Individuums bezeichnet man als **Phänotyp**.

5.2.2 Population

Eine Population ist eine einfache Ansammlung mehrerer Individuen. Sie können sich in ihrer Größe voneinander unterscheiden und sich über mehrere **Generationen** unabhängig voneinander entwickeln. Dadurch sind Effekte der Isolation und der Migration simulierbar (siehe Abschnitt 5.2.10 auf Seite 70).

5.2.3 Gen, Genom und Genotyp

In biologischen Zellen sind sämtliche Informationen zum „Aufbau und Betrieb“ des Organismus in der DNS codiert. Ein DNS-Strang wird als **Chromosom** bezeichnet. Das **Gen** bildet dabei die kleinste Informationseinheit. Die möglichen **Werte** eines Gens nennt man **Allele** [Weicker, 2002, Kap. 1.2].
Das **Genom** bezeichnet die Gesamtheit aller Gene. Dabei sind nicht alle Gene an der Ausprägung des Phänotyp beteiligt[3]. Die Gesamtheit aller Gene, welche *direkten* Einfluss auf den Phänotyp haben, wird als **Genotyp** bezeichnet.
Bei der Verwendung in evolutionären Algorithmen sind in der Regel alle Gene wesentliche Informationsträger, weshalb man in diesem Zusammenhang die Begriffe Genom und Genotyp häufig gleichgesetzt vorfindet.
Ein Genom kann z.B. eine Bit- oder Zeichenkette sein (**diskretwertiges Genom**) oder im Fall eines neuronales Netzwerks die Gewichtsmatrix als Folge von Fließkommawerten (**reellwertiges Genom**).

5.2.4 Fitness

Die Fitness gibt in numerischer oder einer sonstigen vergleichbaren Form an, wie gut ein Individuum die Zielvorgaben der Problemstellung erfüllt. Dieser Wert ist eine Möglichkeit, das Individuum mit anderen zu vergleichen, es als gut, weniger gut oder schlecht einzustufen.
Die Zweisung der Fitness erfolgt über den sogenannten **Fitnessoperator**. Ausgehend vom Genom muss dieser Operator berechnen, welche Fitness das Individuum aufweist. In einfachen Fällen geschieht dies durch eine Funktion, in komplizierteren Fällen durch

[3] Im menschlichen Erbgut scheinen 97% der DNS keine relevanten Informationen zu tragen [Wikipedia Junk DNA, 2005].

eine Simulation und Auswertung des Individuums. Dieser Vorgang wird im Folgenden als **Evaluierung** bezeichnet.
In der Literatur wird zusätzlich zwischen dem **Zielfunktionswert** $Z(I)$ und der Fitness $F(I)$ unterschieden. Der Zielfunktionswert ist das eigentliche Ergebnis der Auswertung. Die Fitnessfunktion berechnet daraus den nichtnegativen Fitnesswert [Polheim, 1999, Kap. 3.1].

5.2.4.1 Proportionale Fitnesszuweisung

Bei der proportionalen Fitnesszuweisung finden einfache Skalierungs- und Offsetoperationen statt. Je nach Zielfunktionswert kommen zusätzlich noch logarithmische und exponentielle Anpassungen und die Berücksichtigung der aktuellen Generation G dazu [Polheim, 1999, Kap.3.1.1].

$$\text{lineare Skalierung:} \quad F(I) = \alpha \cdot Z(I) + \beta \tag{5.1}$$

$$\text{linear dynamische Skalierung:} \quad F(I) = \alpha \cdot Z(I) + \beta \cdot G \tag{5.2}$$

$$\text{logarithmische Skalierung:} \quad F(I) = \log(\alpha \cdot Z(I) + \beta) \tag{5.3}$$

$$\text{exponentielle Skalierung:} \quad F(I) = (\alpha \cdot Z(I) + \beta)^{\gamma} \tag{5.4}$$

Dabei sind α, β und γ Faktoren zur Feineinstellung der Fitness.
Durch starke Unterschiede der Werte des Zielfunktionswerts kann es bei dieser Zuweisung zu Problemen kommen. Liegen viele Individuen mit ihrem $Z(I)$ dicht beieinander und in großem Abstand zu den anderen Individuen, so verlieren die Unterschiede an Bedeutung, was für den Prozess der Selektion von Nachteil ist.

5.2.4.2 Rangbasierte Fitnesszuweisung

Bei der rangbasierten Fitnesszuweisung werden die Individuen nach ihrem Zielfunktionswert sortiert und die Fitness anhand ihrer Position in der sortierten Liste bestimmt [Polheim, 1999, Kap. 3.1.2]. Dadurch verschwinden die Nachteile der proportionalen Fitnesszuweisung.
Auch bei diesem Verfahren kann man durch Anpassung der Rechenvorschrift Nichtlinearitäten einführen und dadurch die Selektion beeinflussen.

5.2.4.3 Mehrkriterielle Fitnesszuweisung

Die mehrkriterielle Fitnesszuweisung findet Anwendung, wenn ein Individuum nicht nur anhand eines Kriteriums beurteilt werden kann. So ist z.B. ein virtueller Charakter nicht nur danach zu bewerten, wie weit er innerhalb einer bestimmten Zeit kommt, sondern auch nach dem Energieaufwand für die Bewegung und der Geradlinigkeit der Fortbewegung.
Für den Vergleich zweier mehrkriterieller Fitnesswerte wird die sogenannte **Pareto-Dominanz** verwendet. Da dieses Verfahren in der Thesis nicht implementiert wurde, sei

für genauere Informationen auf [Polheim, 1999, Kap. 3.1.3] und [Weicker, 2002, Kap. 5.3] verwiesen.

5.2.5 Initialisierung

Zu Beginn des evolutionären Algorithmus' müssen die Individuen in einem initialisierten Zustand vorliegen. Dazu gibt es mehrere Möglichkeiten.

5.2.5.1 Zufällige Initialisierung

Die einfachste Methode ist das Initialisieren des Genoms mit Zufallswerten des Definitionsbereichs [Polheim, 1999, Kap. 3.6.1]. Bei einem reellwertigen Genom sind dies Zahlen zwischen dem Maximal- und Minimalwert, bei binären Genomen 0 oder 1.

5.2.5.2 Nicht-zufällige Initialisierung

Als Methoden für die nicht-zufälligen Initialisierung kommen mehrere Möglichkeiten in Betracht [Polheim, 1999, Kap. 3.6.2]:

- **Iterative Verbesserung**
  Es werden Individuen mit Werten initialisiert, welche in vorherigen Evolutionsläufen mit ggf. vereinfachten Kriterien als Ergebnis zurückgegeben wurden.

- **Expertenwissen**
  Durch Wissen über die Beschaffenheit des Lösungsraumes können Individuen mit Startwerten initialisiert werden, welche sich günstig auf eine schnelle Konvergenz zu den Lösungen auswirken.

- **Problemspezifisches Vorwissen**
  Durch Wissen über die zu lösende Problematik kann verhindert werden, dass Individuen mit „unsinnigen" Werten initialisiert werden.

5.2.6 Selektion

Wenn allen N Individuen ihre Fitness zugewiesen wurde (Evaluierung), können M Individuen anhand dieses Fitnesswerts selektiert werden [Polheim, 1999, Kap. 3.2] [Nissen, 1997, Kap. 2.2.6] (siehe Abbildung 5.2 auf der nächsten Seite). Die selektierten Individuen sind potenzielle Eltern für die nächste Generation. Gute Individuen haben höhere Chancen, selektiert zu werden als schlechte. Damit entspricht die Selektion dem Leitsatz der Evolutionstheorie „Der Stärkere überlebt".
Ein wichtiger Faktor zur Bewertung eines Selektionsverfahrens ist der sogenannte **Selektionsdruck** s. Dieser Faktor gibt die Selektionswahrscheinlichkeit des besten Individuums verglichen mit der durchschnittlichen Selektionswahrscheinlichkeit aller Individuen an. Je

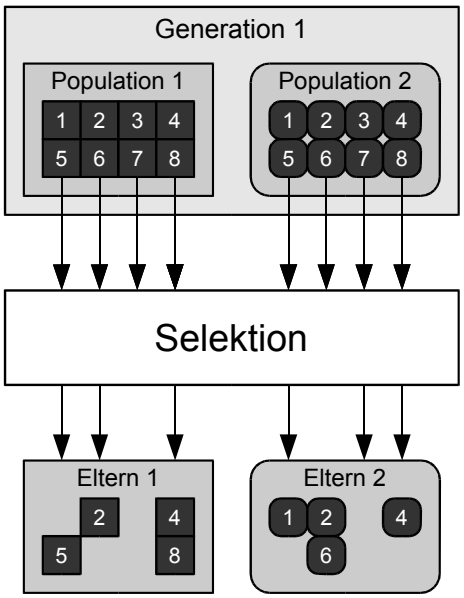


Abbildung 5.2: Selektion

höher s ist, desto wahrscheinlicher werden nur die besten Individuen in die nächste Generation gelangen. Je geringer s ist, desto eher werden es auch schlechte Individuen in die nächste Generation schaffen.

5.2.6.1 Roulettselektion

Bei der Rouletteselektion erhalten die N Individuen proportional zu ihrem Fitnesswert einen „Anteil“ an einem Bereich (z.B. ein „Glücksrad“ oder eine Linie). Nun wird M mal eine Zufallszahl ermittelt, welche eine Position auf diesem Bereich angibt. Das dieser Position zugeordnete Individuum wird dadurch selektiert [Polheim, 1999, Kap. 3.2.1] (siehe Abbildung 5.3).

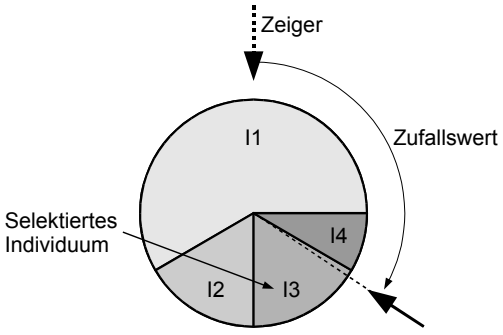


Abbildung 5.3: Rouletteselektion

Dieses Verfahren ist einfach zu realisieren, allerdings nicht optimal, was die Verteilung der Individuen betrifft. Dieser Nachteil wird durch das nachfolgende Verfahren kompensiert.

5.2.6.2 Stochastic Universal Sampling

Auch bei dem Stochastic Universal Sampling (SUS) werden den N Individuen Anteile an einem Bereich zugeordnet. Danach lässt man M Zeiger in gleichen Abständen auf diesen Bereich verweisen und verschiebt sie um einen zufälligen Betrag. Alle Individuen, auf deren Anteile die Zeiger verweisen, werden selektiert[4] [Polheim, 1999, Kap. 3.2.2] (siehe Abbildung 5.4).

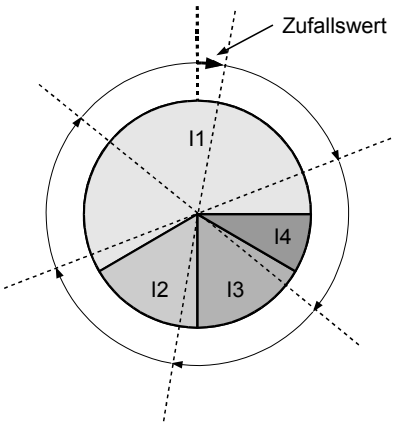


Abbildung 5.4: Stochastic Universal Sampling (SUS)

5.2.6.3 Turnierselektion

Bei der Turnierselektion werden M mal zufällig T Individuen ausgewählt und das beste Individuum selektiert [Polheim, 1999, Kap. 3.2.3].
Der Wert T gibt dabei die **Turniergröße** an. Liegt diese bei 1, werden Individuen zufällig ohne Berücksichtigung der Fitness ausgewählt und der Selektionsdruck ist somit gering. Je höher T gewählt wird, desto höher ist auch der Selektionsdruck.

5.2.6.4 Truncation-Selektion

Bei der Truncation-Selektion werden die Individuen nach ihrer Fitness sortiert und alle Individuen oberhalb einer prozentualen Schwelle selektiert [Polheim, 1999, Kap. 3.2.4]. Bei einer Schwelle von 30% und 100 Individuen wären dies also die besten 30.

5.2.7 Rekombination

Die selektierten Individuen haben die Möglichkeit, in den Prozess der Rekombination zu gelangen. Dabei geben ein, zwei oder mehrere **Eltern** $E_1 \dots E_N$ die Informationen ihrer Genome an ein neues Individuum, das **Kind** K, weiter (siehe Abbildung 5.5 auf der nächsten Seite).

[4] In dem Beispiel wird Individuum I1 4×, I2 und I3 jeweils 1× und I4 nicht selektiert.

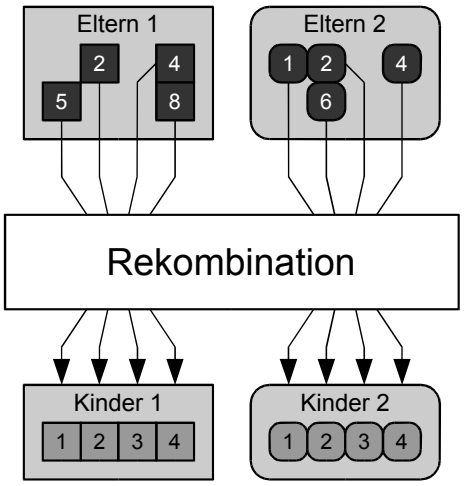


Abbildung 5.5: Rekombination

Die Rekombination entspricht dem biologischen Vorbild der Paarung. Für Letzteres sind allerdings im Regelfall zwei Individuen notwendig, die Variante bei evolutionären Algorithmen hingegen bietet mehr „kombinatorische Möglichkeiten“ (z.B. drei oder mehr Eltern).

5.2.7.1 Diskrete Rekombination

Diskrete Rekombination ist sowohl für reellwertige als auch für diskretwertige Genome möglich [Polheim, 1999, Kap. 3.3.1]. Dabei wird nach dem Zufallsprinzip für jedes Gen $G_{K,1...|G|}$ der Wert aus einem der Elterngene G_{E_e} ausgewählt.

$$\begin{aligned} G_{K,i} &= G_{E_e,i} \\ & i \in (1, 2, \ldots, |G|), \\ & e \in (1, \ldots, N) \text{ gleichverteilt ausgewählt} \end{aligned} \tag{5.5}$$

5.2.7.2 Rekombination reellwertiger Genome

Zur Rekombination **zweier** Eltern mit reellwertigen Genen stehen unter anderem folgende Methoden zur Verfügung [Polheim, 1999, Kap. 3.3.2]:

Intermediäre Rekombination

Für *jedes* Gen des Kindes wird ein Wert zwischen den beiden Allelen der Eltern berechnet.

$$\begin{aligned} G_{K,i} &= G_{E_1,i} \cdot r + G_{E_2,i} \cdot (1 - r) \\ & i \in (1, 2, \ldots, |G|), \\ & r \in [-d, 1 + d] \text{ für jedes } i \text{ neu bestimmt} \end{aligned} \tag{5.6}$$

Der Wert d erweitert den Bereich, welchen die beiden Elterngene „aufspannen“, um einen kleinen Betrag. Ohne diese Korrektur würde sich der abgedeckte Wertebereich der Gene

von Generation zu Generation immer weiter zusammenziehen. Ein statistisch optimaler Wert für d ist 0,25.

Linien-Rekombination

Dieses Prinzip arbeitet ähnlich wie die intermediäre Rekombination, mit dem Unterschied, dass der Zufallswert r für alle Gene nur ein Mal bestimmt wird. Damit liegt das Genom des Kindes auf einer Linie im Hyperraum zwischen den beiden Genomen der Eltern.

$$\begin{aligned} G_{K,i} &= G_{E_1,i} \cdot r + G_{E_2,i} \cdot (1 - r) \\ & i \in (1, 2, \ldots, |G|), \\ & r \in [-d, 1 + d] \text{ für alle } i \text{ gleich} \end{aligned} \tag{5.7}$$

5.2.7.3 Rekombination diskretwertiger Genome

Zur Rekombination **zweier** Eltern mit diskretwertigem Genom stehen unter anderem folgende Methoden zur Verfügung [Polheim, 1999, Kap. 3.3.3]:

Single-Point Crossover

Beim Single-Point Crossover wird das Genom beider Eltern an einer zufällig gewählten Stelle r unterteilt. Alle Gene des Nachkommens mit Index kleiner als r stammen von Elter 1, alle anderen von Elter 2.

$$\begin{aligned} G_{K,i} &= \begin{cases} G_{E_1,i}, \text{ wenn } i < r \\ G_{E_2,i}, \text{ wenn } i \geq r \end{cases} \\ & i \in (1, 2, \ldots, |G|), \\ & r \in [1, |G| + 1] \text{ zufällig gewählt} \end{aligned} \tag{5.8}$$

Multi-Point Crossover

Multi-Point Crossover erweitert das Konzept des Single-Point Crossover um mehrere Stellen $r_1, r_2, \ldots, r_N$, an denen die Quelle der Elterngene wechselt.

Shuffle Crossover

Das Shuffle Crossover erweitert das Prinzip des Multi-Point Crossover zusätzlich um einen Mischvorgang des Eltern-Genoms vor der Ausführung des Crossover gefolgt von einem Ent-Mischvorgang nach dem Crossover. Dadurch wird eine Bevorzugung bestimmter Genpositionen verhindert.

5.2.8 Mutation

In der freien Natur tritt Mutation durch Fehler bei der Reproduktion der DNS sowie durch ionisierende Strahlung und sonstige Umwelteinflüsse auf. Die Mutationsrate innerhalb einer Generation liegt normalerweise zwischen 10^{-4} bis 10^{-7} [Nissen, 1994, S. 8].
Durch Mutation können Veränderungen entstehen, die durch Rekombination normalerweise nicht möglich sind. Somit ist Mutation als Evolutionsfaktor der „Materiallieferant“ für wesentliche Neuerungen. Die Veränderungen können große oder auch überhaupt keine Auswirkungen auf das Individuum haben (siehe Abbildung 5.6).

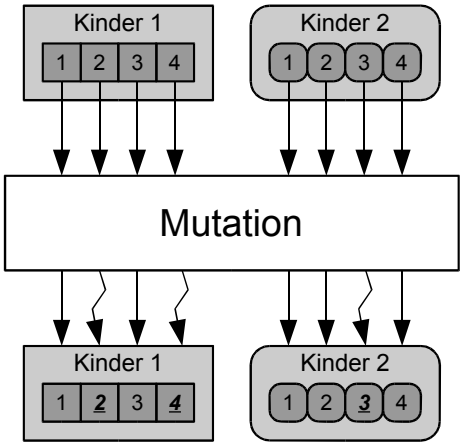


Abbildung 5.6: Mutation

Für eine genaue Beschreibung der Verfahren, ihrer Unterschiede und ihrer Vor- und Nachteile sei auf die Literatur verwiesen, z.B. [Nissen, 1997, 1994; Weicker, 2002; Polheim, 1999]. An dieser Stelle soll nur ein kurzer Überblick gegeben werden.

5.2.8.1 Mutation reellwertiger Genome

Reellwertige Genome werden hauptsächlich durch Addition eines Zufallswerts mutiert [Polheim, 1999, Kap. 3.4.1]. Dabei unterscheidet sich die Art und Weise der Applikation und Berechnung dieses Zufallswerts je nach Verfahren:

- Mutation genau eines Gens oder mehrerer Gene durch Angabe einer **Mutationswahrscheinlichkeit**.
- Feste Schrittweite oder adaptiv je nach Fortschritt des Evolutionsvorgangs.
- Zufällige Schrittweite für jedes Gen oder Skalierung der Summe der Quadrate aller zufälligen Schrittweiten auf einen Einheitswert (Hyperkugel).

Eine wichtige Regel bei der automatischen Anpassung der Schrittweite ist die **1/5-Erfolgsregel** welche besagt, dass die Schrittweite erniedrigt werden sollte, wenn weniger als $\frac{1}{5}$ der mutierten Individuen erfolgreicher sind als ihre Vorgängergeneration [Weicker,

2002, Kap. 4.3.2]. Der Grund liegt darin, dass sich der evolutionäre Algorithmus in diesem Fall dem Optimum nähert und die Änderungen vermehrt dazu führen, dass sich die mutierten Individuen vom Optimum entfernen. Umgekehrt sollte die Schrittweite erhöht werden, wenn mehr als $\frac{1}{5}$ der mutierten Individuen erfolgreicher sind.

5.2.8.2 Mutation diskretwertiger Genome

Bei diskretwertigen Genomen werden die Gene nach folgendem Verfahren verändert [Polheim, 1999, Kap. 3.4.2]:

- Mutation genau eines Gens oder mehrerer Gene gleichzeitig.
- Beibehalten der Genposition oder Verschiebung/Vertauschung mehrerer Gene (z.B. Insert Mutation, Swap Mutation, Reverse Mutation, Scramble Mutation).

5.2.9 Wiedereinfügen

Das Wiedereinfügen erzeugt aus den Eltern und deren Nachkommen eine neue Generation [Polheim, 1999, Kap. 3.5] (siehe Abbildung 5.7). Dabei unterscheiden sich die verschiedenen

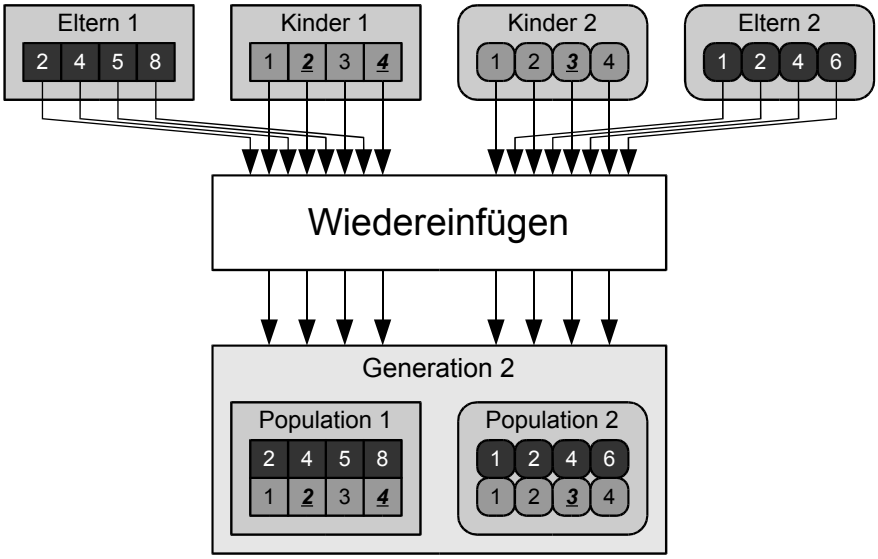


Abbildung 5.7: Wiedereinfügen

Verfahren dadurch, wieviele Nachkommen eingefügt werden und welche Individuen der alten Generation ersetzt werden. Es ist an dieser Stelle auch möglich, Individuen aufgrund von zu hohem Alter auszusortieren.
Die **Wiedereinfügerate** bestimmt den Anteil der Individuen, welcher durch Nachkommen ersetzt wird. Die **Generationslücke** gibt das Verhältnis der vorherigen Populationsgröße zu der Anzahl der Nachkommen an.

5.2.9.1 Einfaches Wiedereinfügen

Beim einfachen Wiedereinfügen werden genau so viele Nachkommen erzeugt wie die Populationsgröße beträgt (Wiedereinfügerate = 1,0). Diese ersetzen nachfolgend alle Individuen der Vorgängergeneration (Generationslücke = 1,0).

5.2.9.2 Zufälliges Wiedereinfügen

Es werden weniger Nachkommen als Individuen erzeugt (Generationslücke < 1,0). Die Nachkommen ersetzen zufällig Individuen (Wiedereinfügerate < 1,0).

5.2.9.3 Elitäres Wiedereinfügen

Es werden weniger Nachkommen als Individuen erzeugt (Generationslücke < 1,0). Die Nachkommen ersetzen die schlechtesten Individuen (Wiedereinfügerate < 1,0).

5.2.9.4 Wiedereinfügen mit Auswahl der Nachkommen

Nur der beste Teil der Nachkommen ersetzt einige Individuen (Generationslücke > Wiedereinfügerate).

5.2.10 Migration/Isolation

Eine weitere Analogie zur Evolutionstheorie ist die Bildung von Populationen, welche sich über einen längeren Zeitraum unabhängig voneinander entwickeln (**Isolation**, siehe [Nissen, 1994, S. 10]). So können sich in jeder Population unterschiedliche Arten von Individuen als diejenigen mit der besten Fitness erweisen.
In unregelmäßigen Abständen finden Individuen den Weg von einer Population zu einer anderen und tauschen so Informationen aus. Dieser Vorgang wird als **Migration** bezeichnet (siehe Abbildung 5.8 auf der nächsten Seite).
Ein ausführliche Betrachtung von Unterpopulationen und sich daraus ergebenden Effekten findet sich in [Polheim, 1999, Kap. 4].

5.2.11 Abbruchkriterium

Es existieren mehrere Ansätze, zu bestimmen, wann ein evolutionärer Algorithmus abbrechen darf [Polheim, 1999, Kap. 3.7]. Dies sollte erst geschehen, wenn mindestens eine ausreichend optimale Lösung gefunden wurde.

5.2.11.1 Direktes Abbruchkriterium

Direkte Abbruchkriterien sind z.B.

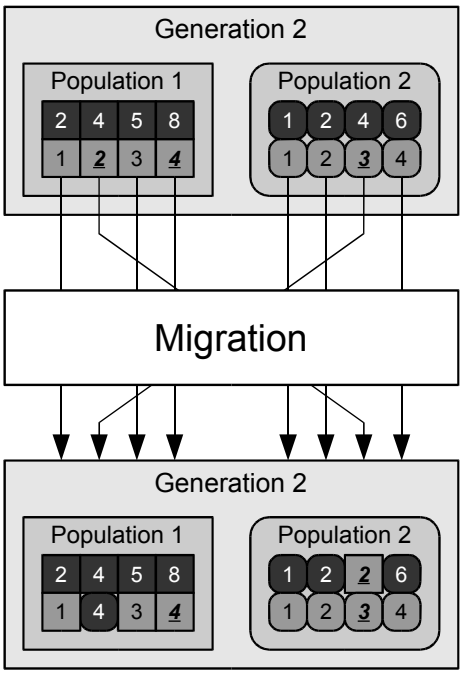


Abbildung 5.8: Migration

- Erreichen eines bestimmten Fitnesswerts
- Erreichen einer maximalen Anzahl von Generationen
- Erreichen einer maximal vorbestimmten CPU-Rechenzeit

5.2.11.2 Standardabweichung

Dieses Abbruchkriterium bestimmt die Standardabweichung der Fitnesswerte aller Individuen und bricht bei Unterschreitung eines bestimmten vorgegeben Wertes ab. Zu diesem Zeitpunkt liegen alle Lösungen dicht beieinander. Auch alle folgenden Methoden stellen diese Tatsache über rechnerische Methoden fest.
Ein Nachteil dieser Methode ist, dass der Abbruchwert in Relation zu den möglichen Fitnesswerten skaliert werden muss.

5.2.11.3 Laufendes Mittel

Liegt die Differenz des laufenden Mittels aller Fitnesswerte der aktuellen Generation zur vorherigen Generation unterhalb einer bestimmten Schwelle, ist das Abbruchkriterium erfüllt.
Meistens wird zusätzlich ein Zähler eingeführt, welcher angibt, wie oft die Schwelle hintereinander unterschritten werden muss, damit der Abbruch erfolgt. Damit vermeidet man einen zu frühen Abbruch, wenn kurzzeitig kein Fortschritt stattgefunden hat.

5.2.11.4 Bester/Schlechtester

Eine andere Methode, um festzustellen, dass alle Lösungen dicht beieinander liegen, ist die Abweichung der Fitnesswerte des besten und schlechtesten Individuums. Allerdings liegt bei diesem Verfahren derselbe Nachteil in Bezug auf die Skalierung vor wie bei der Standardabweichung.

5.2.11.5 Phi

Der Wert ϕ bezeichnet den Quotienten aus dem besten Fitnesswert eines Individuums und dem gemittelten Fitnesswert der Population. Liegt dieser Wert nahe bei 1, so kann der Algorithmus terminieren.

5.3 Verfahren

5.3.1 Evolutionäre Strategien (ES)

1964 entwarf die Forschergruppe Bienert, Rechenberg und Schwefel an der Technischen Universität Berlin ein Verfahren, welches mittels evolutionärer Methoden die Lösung experimenteller Problemstellungen erlaubte, die sich nicht analytisch beschreiben ließen [Rechenberg, 1973].
Durch dieses Verfahren optimierte die Gruppe eine Strömungsform im Windkanal und ließ den Querschnitt einer Überschalldüse und die korrekte Krümmungsform eines Abgaskanals ermitteln [Weicker, 2002].

Evolutionäre Strategien sind hauptsächlich durch Verwendung reellwertiger Genome charakterisiert. Dementsprechend müssen die Mutations- und Rekombinationsoperatoren auf solchen Mengen operieren können[5].

Eine Analogie zur Anwendung evolutionärer Strategien ist die Aufgabe, in einem Tonstudio eine Aufnahme eines Symphonieorchesters abzumischen, wobei die Dreh- und Schieberegler auf dem Mischpult keinerlei Beschriftung aufweisen. Die Fitness der aktuellen Einstellung der Regler ergibt sich aus der Qualität der daraus resultierenden abgemischten Aufnahme.

5.3.1.1 Rechenberg-Notation

Im Zusammenhang mit evolutionären Strategien hat sich die Rechenberg-Notation als wichtiges Hilfsmittel herausgestellt [Schöneburg u. a., 1994, Kap. 3]. Mit ihrer Hilfe ist es in Kurzschreibweise möglich, die Größe der Population, die Generationslücke und das Verhältnis von Eltern zu Nachkommen anzugeben. Zusätzlich existiert eine Variante dieser Schreibart für Populationskonzepte.

[5] Im Gegensatz zu den genetischen Algorithmen (siehe Abschnitt 5.3.3 auf Seite 74)

5.3.1.2 (1+1)-ES

Bei einer (1+1)-ES erzeugt genau ein Elter einen Nachkommen [Schöneburg u. a., 1994, S. 148]. Jeweils beide werden auch beim Wiedereinfügen berücksichtigt. Diese *zweigliedrige ES* ist die einfachste Form eines evolutionären Algorithmus'. Da die Population immer nur aus einem Individuum besteht, wird der Suchraum punktuell untersucht. Somit ist dieses Verfahren nur sehr eingeschränkt einzusetzen.

5.3.1.3 (μ+λ)-ES

Bei einer μ+λ-ES erzeugen μ Eltern insgesamt λ Nachkommen [Schöneburg u. a., 1994, S. 152]. Eltern und Nachkommen werden gemeinsam („+“) beim Wiedereinfügen berücksichtigt.

5.3.1.4 (μ,λ)-ES

Im Unterschied zum vorherigen Verfahren werden bei einer μ,λ-ES nur die Nachkommen („,“) wiedereingefügt [Schöneburg u. a., 1994, S. 153].

5.3.1.5 ($\mu/\rho\sharp\lambda$)-ES

Bei dieser Variante kommt mit ρ ein Faktor hinzu, welcher angibt, wieviele Eltern zur „Zeugung“ eines Nachkommen nötig sind. Das Symbol $\sharp$ steht als Platzhalter für die Zeichen „+“ (Wiedereinfügen von Eltern *und* Nachkommen) oder „,“ (Wiedereinfügen der Nachkommen alleine).

5.3.1.6 ES mit Populationen

Die Verwendung mehrerer Populationen wird durch Klammerausdrücke angegeben [Schöneburg u. a., 1994, S. 160]. So bezeichnet z.B. $[2, 3(50, 150)]$-ES ein Verfahren, bei dem aus zwei Populationen drei neue erzeugt werden, aus welchen dann zwei für die nächste Generation ausgewählt werden. Innerhalb einer Population existieren 50 Individuen, welche insgesamt 150 Nachkommen produzieren, von denen die besten 50 die vorherigen Individuen ersetzen („,“).

5.3.1.7 ES mit isolierten Populationen

Sollen sich Populationen hauptsächlich isoliert entwickeln, so wird dies durch eine Potenzzahl gekennzeichnet [Schöneburg u. a., 1994, S. 163]. Bei einer $[2, 3(50, 150)^{/100}]$-ES läuft der Vorgang innerhalb einer Population nach demselben Prinzip wie im vorherigen Beispiel. Im Unterschied dazu werden drei Populationen erzeugt, welche sich zunächst 100 Generationen lang unabhängig voneinander entwickeln. Erst danach werden aus diesen drei Populationen die zwei für die nächsten 100 Generationen ausgewählt.

5.3.2 Evolutionäre Programmierung (EP)

Zur Vorhersage von Ketten von Symbolen entwickelten Fogel, Owens und Walsh 1966 ein Verfahren, welches die dazu notwendigen Automaten mit evolutionären Methoden entwickelte [Fogel u. a., 1966], [Polheim, 1999, Kap. A.1.2].

Bei der evolutionären Programmierung liegt keine Unterscheidung von Genotyp und Phänotyp vor [Weicker, 2002, Kap. 4.3]. Die Reproduktion verliert somit ihre Bedeutung. Die Mutation hingegen muss direkt auf die Art des Individuums angepasst sein. Die Repräsentation des Individuums kann sich bei diesem Verfahren nahe an der Problemstellung orientieren.

Eine Anwendung der EP ist die Entwicklung eines Automaten, welcher eine bestimmte Eingabemenge geeignet verarbeiten soll. Die Mutation erlaubt das Hinzufügen bzw. Entfernen von Zuständen und Zustandsübergängen.

5.3.3 Genetische Algorithmen (GA)

Genetische Algorithmen wurden von John Holland an der University of Michigan entwickelt [Polheim, 1999, Kap. A.1.4]. Sein Ziel war eine allgemeine Theorie anpassungsfähiger Systeme [Holland, 1975].

Genetische Algorithmen legen starken Wert auf die Ähnlichkeit mit dem biologischen Vorbild der DNS. Der Genotyp liegt in Form von Zeichenketten oder binär codiert vor. Die Umwandlung von Genotyp zu Phänotyp ist fest vorgeschrieben und führt dazu, dass der Genotyp immer dieselbe Länge aufweist.
Die Mutation tritt gegenüber der Reproduktion in den Hintergrund [Weicker, 2002, Kap. 4.1.1]. Bei der Reproduktion finden resultierend aus der Art der Codierung hauptsächlich Verfahren Anwendung, die Symbolketten in ihrem Aufbau verändern und „vermischen“.

Eine Analogie zu genetischen Algorithmen ist das Lösen eines Kreuzworträtsels ohne Kenntnis der Sprache. Lediglich die Symbole des Alphabets dieser Sprache sind bekannt. Es müssen immer alle Felder ausgefüllt werden (feste Länge des Genotyps). Die Fitness berechnet sich aus der Anzahl der korrekt ausgefüllten Worte.

5.3.4 Genetische Programmierung (GP)

Die genetische Programmierung ist ein noch relativ junges Gebiet [Polheim, 1999, Kap. A.1.5]. Es wurde Anfang der 90er Jahre von J.R. Koza entwickelt [Koza, 1992]. Die dabei zugrunde liegende Idee war die Evolution von Computerprogrammen.

Genetische Programmierung ähnelt zunächst den genetischen Algorithmen. Im Unterschied dazu ist allerdings seine Länge nicht vorgegeben, sondern wie der Inhalt der

Veränderung unterworfen. Die Idee dahinter ist, statt einer fest vorgegebenen und lediglich parametrisierbaren Lösungsform eine flexible und sich in seiner Struktur entwickelnde Variante zu nehmen.
Anwendungen für GP sind z.B. Syntaxbäume oder einfache Programme, welche entwickelt werden, um eine Problemstellung zu lösen. Genauere Informationen finden sich in [Weicker, 2002, Kap. 4.4] und [Michalewicz, 1996, Kap. 13.2]. Eine Anwendung dieses Prinzips findet sich in [Taylor, 1999]. Hier werden „Verhaltens-Programme“ virtueller Charaktere evolutionär entwickelt, damit diese möglichst effektiv „virtuelle Nahrung“ in Form von Energiepaketen sammeln.

Eine Analogie zur genetischen Programmierung ist das Schreiben eines Textes zu einer Aufgabenstellung in einer fremden Sprache. Die Länge des Textes ist egal, das Alphabet vorgegeben. Die Fitness ergibt sich aus der „Note“ für den Text.

Teil II

Durchführung

6 Struktur

6.1 Vorhandene Hardware

Abbildung 6.1 zeigt die Infrastruktur des VUM-Laboratoriums. Alle in diesem Bild gezeigten Geräte sollten für die Ausführung dieser Thesis genutzt werden (siehe auch Farbtafeln D.5 bis D.6 auf Seiten 231–233).

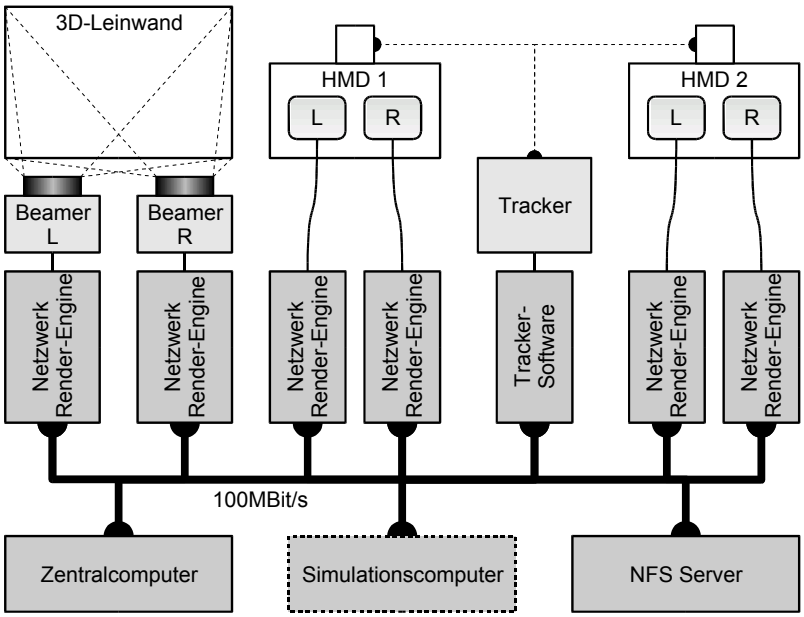


Abbildung 6.1: Infrastruktur des VUM-Labors

Es sind mehrere Computer vorhanden, deren Grafikkarten mit den Bildschirmen der HMDs bzw. zwei Beamern für die 3D-Projektionsleinwand verbunden sind. Auf diesen Maschinen soll eine Software laufen (Netzwerk-Render-Engine), welche Befehle aus dem Netzwerk entgegennimmt und in bewegte Bilder verwandelt.

Des Weiteren steht ein Computer zur Verfügung, welcher die Hardware des Trackers ansteuert und die Positions- und Orientierungsdaten der Sensoren auf den HMDs sowie zwei

weiterer Sensoren (z.B. für einen Datenhandschuh) in Form von Netzwerkpaketen sendet.

Daten, Modelle, Texturen, Konfigurationen, Szenen etc. werden auf einem zentralen NFS-Server[1] abgelegt und können von allen angeschlossenen Computern gelesen werden.
Der Zentralcomputer verwaltet alle angeschlossenen Geräte, sendet Steuerbefehle und führt die eigentliche physikalische Simulation aus. In einem späteren Schritt soll die Simulation auf einen separaten Computer ausgelagert werden können. Für die vorliegende Thesis war dies nicht erforderlich.

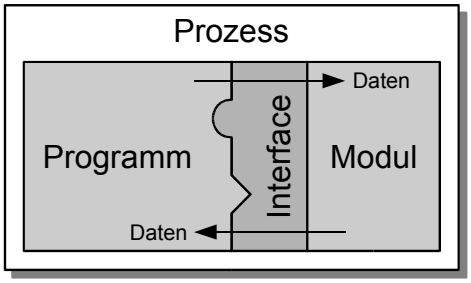
Die folgenden Abschnitte befassen sich damit, die wesentlichen Teile der nötigen Software zu identifzieren und ihre Aufgaben und die Art der Zusammenarbeit zu beschreiben.

6.2 Modularisierung

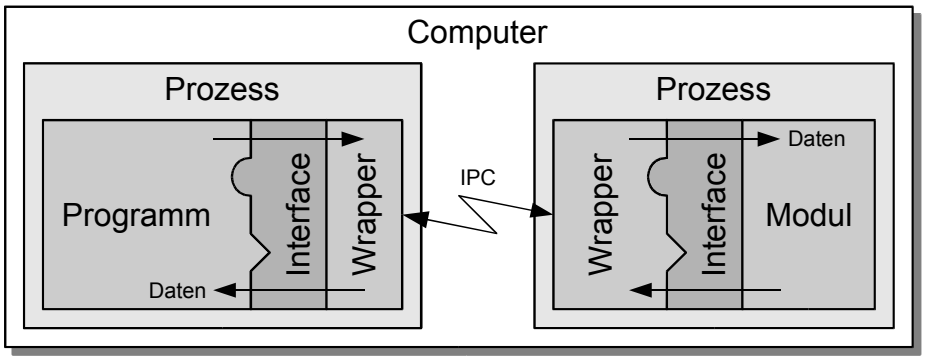
Ohne Modularisierung tendiert ein Programm schnell dazu, zu einer unüberschaubaren und erst recht unwartbaren Ansammlung von Programmzeilen zu werden. Dies kann verhindert werden, indem man das Programm in Teile mit fest umrissenen Aufgaben zerlegt. Jeder Teil, im Folgenden als **Modul** bezeichnet, stellt nach außen hin Methoden zur Verfügung, welche Zugriff auf Funktionalitäten und Eigenschaften des Moduls bieten.
In C/C++ entspricht die Deklaration dieser Methoden einer Headerdatei. So kann man z.B. die Headerdatei `gl.h` für den Grafikstandard OpenGL als Deklaration von Methoden betrachten, mit deren Hilfe man das „Modul Grafikkartentreiber“ dazu bewegen kann, 3D-Grafiken zu erzeugen.
Auf dieser Headerdatei baut z.B. die Grafikbibliothek OGRE (siehe Abschnitt 7.9 auf Seite 113) auf und bietet Klassen und Methoden an, welche von den OpenGL-Funktionen abstrahieren. Dadurch ist es OGRE möglich, auch andere hardwarenahe Grafikbibliotheken wie Direct3D zu integrieren. Der Programmierer, der die Klassen und Methoden von OGRE verwendet, muss sich dadurch über Unterschiede und Details dieser hardwarenahen Bibliotheken keine Gedanken mehr machen. Er kann z.B. auf Windows-Systemen vollkommen frei entscheiden, welche der beiden Bibliotheken er verwenden will, da OGRE von deren Unterschieden abstrahiert.
Dieses Konzept der Austauschbarkeit und Abstraktion kann noch eine Ebene weitergeführt werden. Ein Ziel der Thesis war die Austauschbarkeit von ganzen Funktionsblöcken, ohne das Programm umschreiben zu müssen. Zu diesem Zweck wurde für jeden Funktionsblock ein sogenanntes **Modulinterface** entworfen, mit dessen Hilfe die Austauschbarkeit gewährleistet sein sollte. Dieses Interface enthält alle wichtigen Funktionen, mit denen sich alle denkbaren Ausprägungen von Funktionsblöcken einheitlich steuern lassen. Programmiert man gegen dieses Modulinterface, so wird man dadurch relativ unabhängig von der konkreten Implementierung eines Moduls.

[1] Das NFS (engl.: **n**etwork **f**ile **s**ervice) ist ein Protokoll, welches den Zugriff auf ein Dateisystem im Netzwerk ermöglicht.

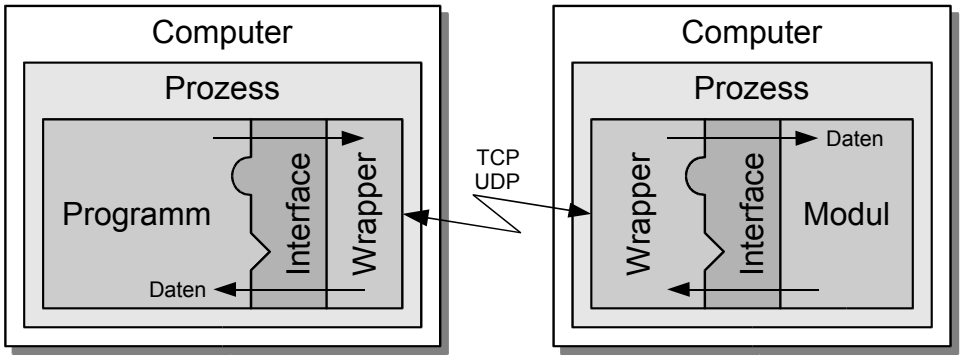
Es ist ebenfalls möglich, die Ausführung des Moduls transparent auf einen anderen Prozess oder sogar auf einen anderen Computer auszulagern. Das Interface bildet dabei ein einheitliches Bindeglied, um die Mechanismen zu verbergen, die zur konkreten Ansteuerung der Implementierungen notwendig sind (siehe Abbildung 6.2).



(a) Modularität innerhalb eines Prozesses



(b) Modularität über Prozessgrenzen hinaus



(c) Modularität über Systemgrenzen hinaus

Abbildung 6.2: Arten von Modularität

Eine Anwendung dieses Prinzips ist die Netzwerk-Render-Engine (siehe Abschnitt 7.9 auf Seite 113). Das Hauptprogramm wird gegen das Modulinterface `interface` *`render::IEngine`* erstellt und verwendet dessen Methoden, um grafische Objekte zu erstellen und zu manipulieren Die Klassen `render::Engine_Ogre` und `render::Engine_Network` implementieren jeweils dieses Interface. Der Benutzer kann nun wahlweise eine der beiden Klassen verwenden, ohne das Hauptprogramm verändern zu

müssen. Damit steht ihm flexibel die Möglichkeit zur Verfügung, die grafischen Objekte direkt auf dem eigenen Bildschirm (`class render::Engine_Ogre`) oder auf anderen Computern im Netzwerk zu sehen (`render::Engine_Network`).

Wie aus Abbildung 2.4 auf Seite 19 ersichtlich ist, lassen sich aus der Gesamtapplikation grob zwei Teile bilden:

1. Das Simulationsprogramm oder die **Simulations-Engine** `cerebellum`.
2. Die **Evolutions-Engine** `evolver`.

Es wurde entschieden, die beiden Programme nach dem Prinzip der Interprozesskommunikation (IPC, engl.: „**i**nter-**p**rocess **c**ommunication“) zu koppeln (siehe Abbildung 6.2(b) auf der vorherigen Seite). Dieses Prinzip bietet mehrereVorteile:

1. **Einfachheit**
   Kindprozesse abzuspalten und mit Hilfe von Pipes mit seinem Elternprozess kommunizieren zu lassen, ist einfach zu realisieren und erfordert wenig „Vorsichtsmaßnahmen“ (z.B. im Gegensatz zu Threads).
2. **Portabilität**
   Es ist sowohl unter Windows als auch unter Linux möglich, Kindprozesse abzuspalten.
3. **Parallelisierbarkeit**
   Unter Linux bietet sich mit der Kernel-Erweiterung Mosix[2] die Möglichkeit an, Prozesse transparent in einem Cluster von Computern zu verteilen (siehe Abschnitt 7.16 auf Seite 142). Dadurch wäre das Programm ohne Änderung in einer verteilten Umgebung lauffähig.

6.3 Das Simulationsprogramm `cerebellum`

Abbildung 6.3 auf der nächsten Seite zeigt eine Übersicht des Simulationsprogramms `cerebellum`.
Im Folgenden werden die einzelnen Funktionsblöcke und ihre Aufgaben beschrieben:

- Die **Kontroll-Logik** lädt je nach Konfiguration entsprechende Module, initialisiert und verwaltet sie. Für jede Simulation werden von ihr Daten geladen, aufbereitet und an die Module verteilt. Die Logik steuert und überwacht auch den Simulationsvorgang.
- Die **Physik-Engine** führt die physikalische Simulation durch. Sie sendet Informationen über Änderungen der Translation und Orientierung physikalischer Körper

[2] Mosix Cluster and Grid Management. ©1999-2005 Amnon Barak. `http://www.mosix.org`

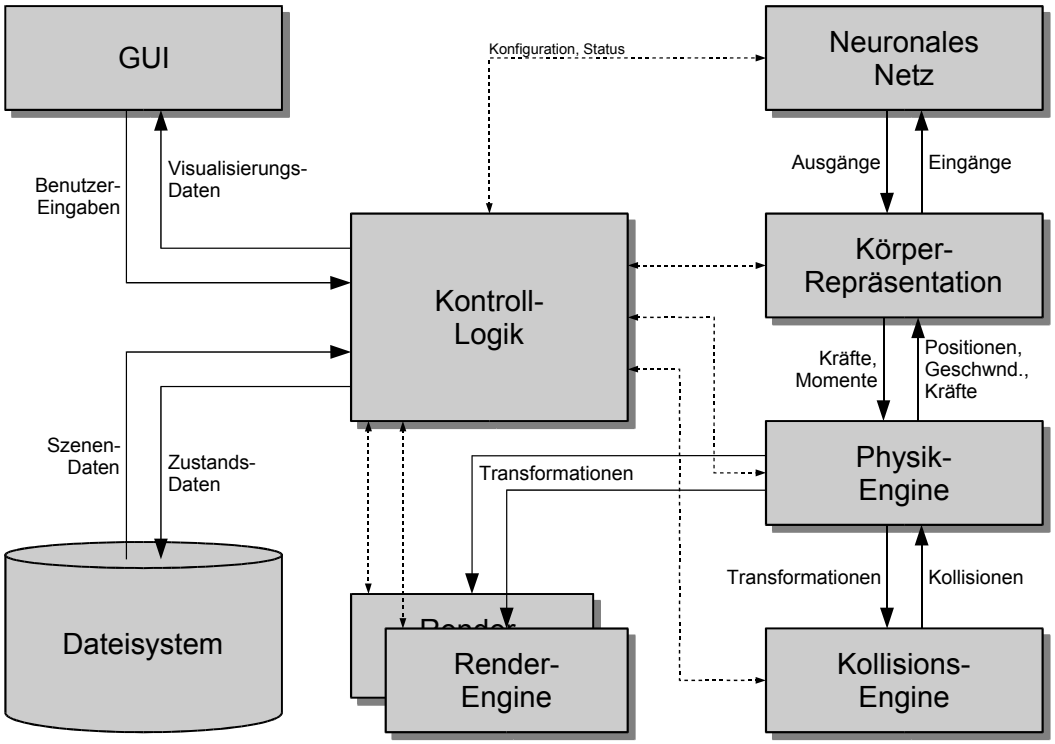


Abbildung 6.3: Blockdiagramm des Simulationsprogramms `cerebellum`

  - an die **Kollisions-Engine**, welche daraus die entstehenden Kollisionen berechnet und zurückgibt,
  - und an die **Render-Engines**, welche die Zustände der Körper grafisch darstellen.

  Informationen über Soll- und Ist-Zustände der Gelenke sowie über Kräfte werden mit der ...

- ... **Körper-Repräsentation** ausgetauscht. Diese ist dafür zuständig, Informationen vom und zum neuronalen Netzwerk aufzubereiten und anzupassen. Dazu gehört z.B. die Skalierung des Wertebereichs des neuronalen Netzwerks von $[-1, +1]$ auf $[-90°, +90°]$ eines Gelenks. Auch Informationen über Position und Orientierung des Körpers (Gleichgewicht) werden hier von der Physik-Engine erfragt, aufbereitet und weitergegeben.

- Das **neuronale Netzwerk** berechnet aus den Eingaben der Körper-Repräsentation die entsprechenden Ausgaben für Bewegungen.

- Das **GUI**[3] stellt Ein- und Ausgabewerte des neuronalen Netzwerks grafisch dar und erlaubt die Steuerung der Simulation (z.B. Szene laden, Kamera auswählen).

[3] GUI (engl.: „**g**raphical **u**ser **i**nterface“) ist der englische Begriff für eine grafische Benutzungsoberfläche.

6.4 Das Evolutionsprogramm `evolver`

Abbildung 6.4 zeigt eine Übersicht der Evolutions-Engine `evolver`.

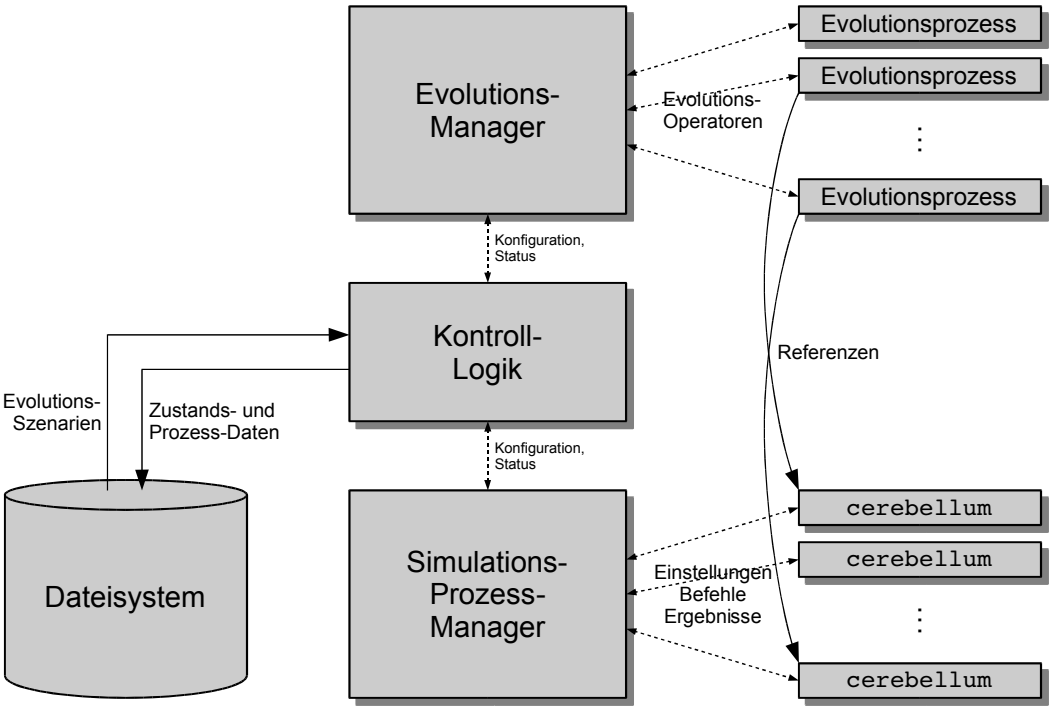


Abbildung 6.4: Blockdiagramm des Evolutionsprogramms `evolver`

Im Folgenden werden die einzelnen Funktionsblöcke und ihre Aufgaben beschrieben:

- Die **Kontroll-Logik** lädt je nach Konfiguration entsprechende Module, initialisiert und verwaltet sie. Für jeden Evolutionsvorgang werden von ihr Daten geladen, aufbereitet und an die Module verteilt. Die Logik steuert und überwacht auch den Evolutionsvorgang.
- **Simulationsprozesse** sind mehrere als Kindprozesse laufende Instanzen von `cerebellum`, die vom...
- ...**Simulationsprozess-Manager** gestartet und verwaltet werden. Der Manager lädt Simulations-Szenerien, initialisiert das neuronale Netzwerk, schreibt und liest die Genotyp-Daten, simuliert den virtuellen Charakter und liest die berechneten Fitness-Werte.
- **Evolutionsprozesse** verwalten Individuen, Populationen und Evolutionsoperatoren und führen die evolutionären Strategien aus. Sie werden vom...
- ...**Evolutionsprozess-Manager** verwaltet und gesteuert. Wenn der Prozess die Fitness von Individuen berechnen soll, so werden Simulationsprozesse durch die Kontroll-Logik angefordert und mit dem Evolutionsprozess verknüpft, bis die Berechnung abgeschlossen ist.

6.5 Zusätzliche Elemente

Zur Implementierung der beiden Hauptprogramme sind noch weitere Elemente nötig, welche wichtige integrative Aufgaben übernehmen, in den Abbildungen aber aus Gründen der Übersichtlichkeit nicht alle eingezeichnet sind:

- **Ressourcenmanagement**
  Alle von der Software benötigten Ressourcen sollten von zentraler Stelle erzeugt, geladen, verwaltet und wieder aufgeräumt werden.

- **Kommandos**
  Module müssen sowohl direkt als auch z.B. über Netzwerkpakete Kommandos ausführen. Um eine gesonderte Behandlung aller Fälle zu vermeiden, muss eine zentrale Instanz zur Behandlung von Kommandos geschaffen werden.

- **Datenfluss**
  Für die hohe Menge an Daten, welche zwischen den Komponenten ausgetauscht werden, müssen einfach zu beherrschende und universelle Mechanismen und Datenstrukturen implementiert werden.

- **Netzwerk**
  Aus Geschwindigkeitsgründen werden Befehle und Datenpakete als UDP-Pakete[4] versendet. Auf den bisher vorhandenen Bibliotheken des VUM-Labors muss eine Erweiterung aufgebaut werden, welche es erlaubt, Pakete in wichtig und unwichtig zu unterteilen und wichtige Pakete zuverlässig und fehlerfrei zu übertragen.

[4] Das UDP (engl.: „**u**ser **d**atagram **p**rotocol“) ist ein verbindungsloses Protokoll, welches Datenpakete von maximal ca. 1,5kB Größe zu einem oder mehreren Netzwerkteilnehmern transportiert. Dabei wird keine Fehlerbehandlung ausgeführt. Aus diesen Grund ist das UDP zwar unzuverlässig, aber dafür schnell.

7 Implementierung

In den folgenden Abschnitten wird die Implementierung der Interfaces und Klassen für die beiden Hauptprogramme `cerebellum` und `evolver` vorgestellt. Die ersten drei vorgestellten Klassen sind für grundlegende Funktionen zur Datenverwaltung zuständig. Die nachfolgenden Klassen bauen auf diesen Funktionen auf.

7.1 Smart-Pointer

Während der Laufzeit eines Programms werden dynamisch Objekte erzeugt, für welche vom Betriebssystem Speicher angefordert werden muss. Dieser allokierte (engl.: „allocate“, zuteilen) Speicherbereich muss spätestens bei Beendigung des Programms unter Angabe der Startadresse (Pointer) des Bereichs wieder freigegeben werden. Verliert oder „vergisst“ ein Programm diese Adresse, so entstehen Speicherlecks (engl.: „memory-leaks“), die sich unter ungünstigen Umständen während der Laufzeit des Programms aufsummieren und zu Speichermangel und damit zum Programmabbruch oder gar -absturz führen können.
Gerade die Programmiersprache C leidet stark unter diesem Phänomen, da der Programmierer selbst für die Verwaltung der allokierten Speicherbereiche zuständig ist. Im Gegensatz dazu stehen Hochsprachen wie Java, die durch einen sogenannten Garbage-Collector in regelmäßigen Abständen nicht mehr benötigte Speicherbereiche sammeln und wieder freigeben.
Ein Ausweg aus diesem Problem sind zwei Klassen, welche in Zusammenarbeit die Aufgabe übernehmen zu erkennen, wann ein Objekt nicht mehr benötigt wird und freigegeben werden kann (siehe Abbildung 7.1 auf der nächsten Seite).
Klassen, die sich unter dieses Management stellen, müssen von der Klasse `RefCountBase` abgeleitet werden. Dadurch erhalten sie einen Zähler *`m_lRefCount`*, welcher protokolliert, aus wievielen Stellen im Programm ein Objekt dieser Klasse referenziert wird. Fällt der Zähler auf 0, so zerstört sich das Objekt selbstständig.
Das Template `RefCountPtr<T>` übernimmt die Aufgabe der normalen Pointer in C++-Programmen. Durch Operator-Überladung (engl.: „operator overloading“) ist sichergestellt, dass der Programmierer von dieser Umstellung so wenig wie möglich merkt. Alle

Operationen, die mit normalen Pointern möglich sind, funktionieren auch mit den Smart-Pointern (z.B. `*`, `->`, `==`, `==`, `!=`).
Wird ein Smart-Pointer konstruiert (`RefCountPtr(...)`) oder wird ihm ein Pointerwert zugewiesen (`operator=(void* pPtr)`), so erhöht er mittels Aufruf von `m_pObject->addReference()` den Referenzzähler des verwalteten Objektes.
Wird der Smart-Pointer zerstört (`~RefCountPtr(...)`) oder der Pointerwert auf `NULL` gesetzt, so erniedrigt er vorher mittels Aufruf von `m_pObject->removeReference()` den Referenzzähler des verwalteten Objekts. Dabei kann es vorkommen, dass der Referenzzähler auf 0 fällt und sich somit das Objekt zerstört.

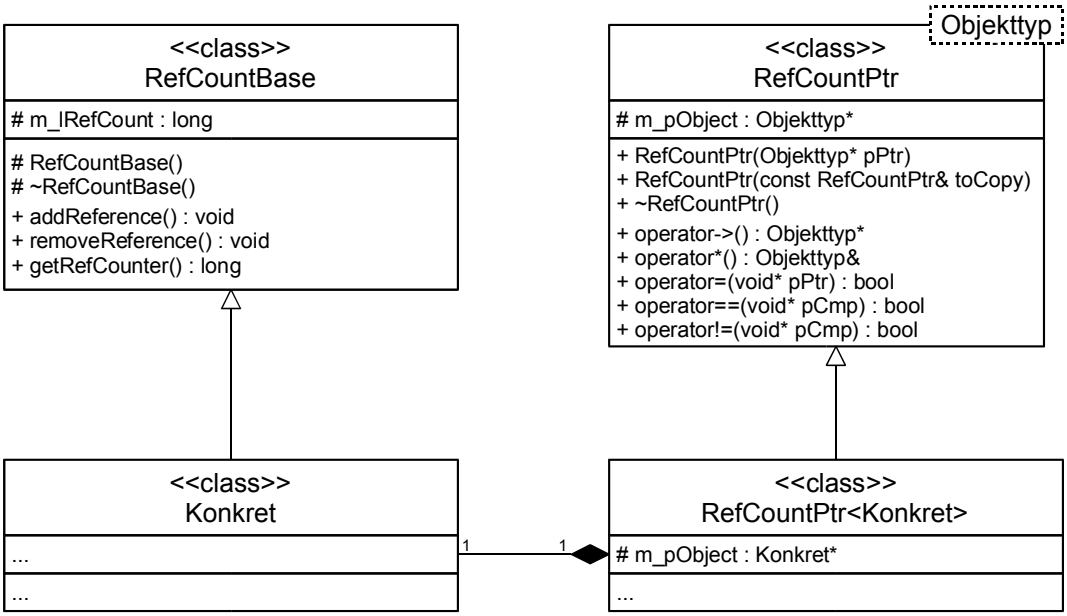


Abbildung 7.1: UML-Diagramm der Klassen für Smart-Pointer

In den folgenden Beschreibungen und UML-Digrammen wird `XyzPtr` notiert, wenn ein Smart-Pointer für die Klasse `Xyz` gemeint ist.

7.2 Datenbäume

Um hierarchisch organisierte Daten auszutauschen und XML[1]-Dateien zu lesen und zu schreiben, wurde das Konzept der Datenbäume (engl.: „data tree“) entworfen. Datenbäume bestehen aus Elementen (`class Element`), welche weitere Elemente (Kind-Elemente) und Attribute (`class Attribute`) enthalten können. Jeder Datenbaum besitzt ein Wurzelelement (`DataTree::getRoot()`). Elemente und Attribute sind abgeleitet von `class ValueHolder`, was es ihnen ermöglich, unter einem symbolischen Namen Bool-, Integer-, Fließkomma-Werte oder Strings zu enthalten (siehe Abbildung 7.2 auf der nächsten Seite).

[1] XML: e**x**tensible **m**arkup **l**anguage. „Ein Standard zur Erstellung maschinen- und menschenlesbarer Dokumente in Form einer Baumstruktur...“ [Wikipedia XML, 2005; Yergeau u. a., 2004].

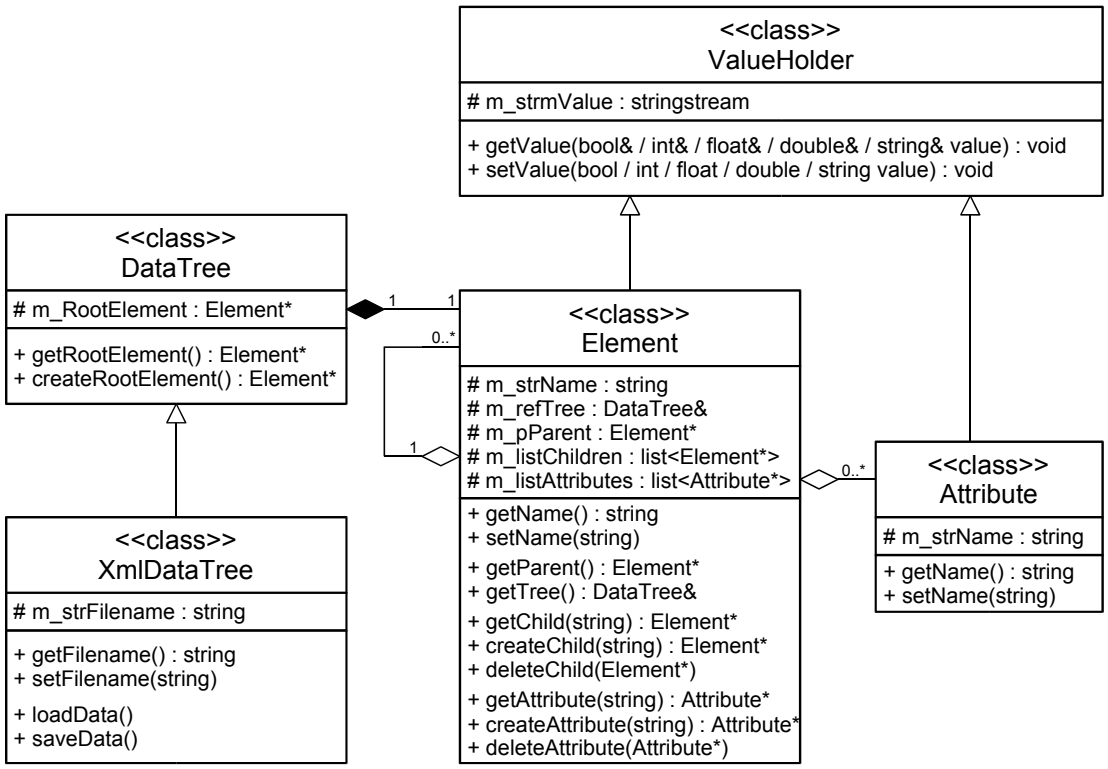


Abbildung 7.2: UML-Diagramm der Klassen für Datenbäume

Durch Ableitung von `class DataTree` kann man verschiedene Arten Datenbäume implementieren. `class XmlDataTree` liest und schreibt Datenbäume aus XML-Dateien (siehe Tabelle 7.1). `class LuaDataTree` liest Datenbäume aus LUA-Skripten (siehe Abschnitt 7.13 auf Seite 139 und Tabelle 7.6 auf Seite 139).

Datenbaum	**XML-Datei**

profile
  Name Test
  render_engine
  type ogre
  screen
  width 320
  height 200
  controller
  type kbd
	`<?xml version="1.0"?>`
	`<profile name="Test">`
	`<render_engine type="ogre">`
	`<screen>`
	`<width>320</width>`
	`<height>200</height>`
	`</screen>`
	`</render_engine>`
	`<controller type="kbd">`
	`</controller>`
	`</profile>`

Tabelle 7.1: Definition von Datenbäumen mit XML

7.3 Parameter

Bei der Implementierung von Modulinterfaces ist es nötig zu unterscheiden, welche Art von Parametern eine Methode benötigt. Es gibt Parameter, die unabdingbar zur Ausführung der Methode sind. Diese sollten fest in das Interface integriert werden. Andere Parameter können sich je nach konkreter Implementierung des Interfaces unterscheiden und verschiedene Ausprägungen annehmen. Bei der Konstruktion eines physikalischen Körpers zum Beispiel ist die ID, der Typ und der Vorgänger des Körpers unabdingbar und Bestandteil der Methode `createBody(BodyID id, BodyType type, BodyID parent, Parameters params)` (siehe Abschnitt 7.7.2.2 auf Seite 102).
Zu diesem Zweck wurde die Parameterklasse `Parameter` eingeführt (siehe Abbildung 7.3).

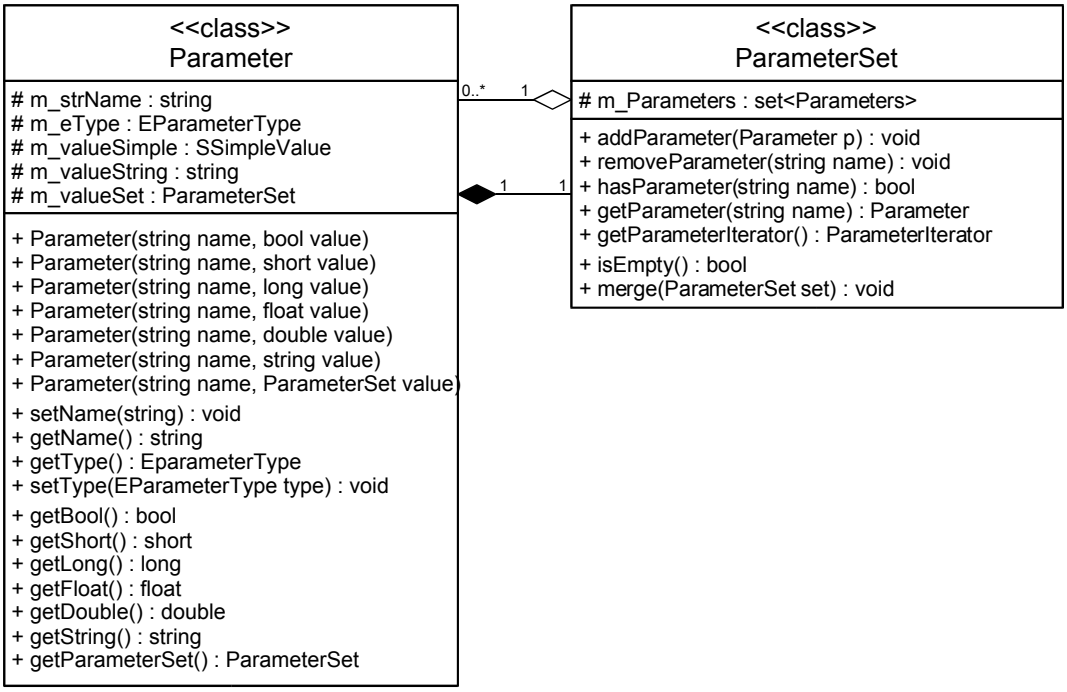


Abbildung 7.3: UML-Diagramm der Klassen für Parameter

Sie kann unter einem symbolischen Namen Boolean-, Integer-, Fließkomma-Werte, Strings und Unterparameter halten. Durch `class ParameterSet` sind auch Bäume von Parametern möglich.
Es existieren Hilfsklassen, um Parameter in Datenbäume und zurück zu konvertieren.

7.4 Kommandos

Damit Modulimplementierungen die Möglichkeit haben, abseits von den Definitionen der Modulinterfaces spezielle Befehle auszuführen, wurde die Kommandoklasse `Command` in Zusammenhang mit `interface` *`IControllable`* eingeführt (siehe Abbildung 7.4).

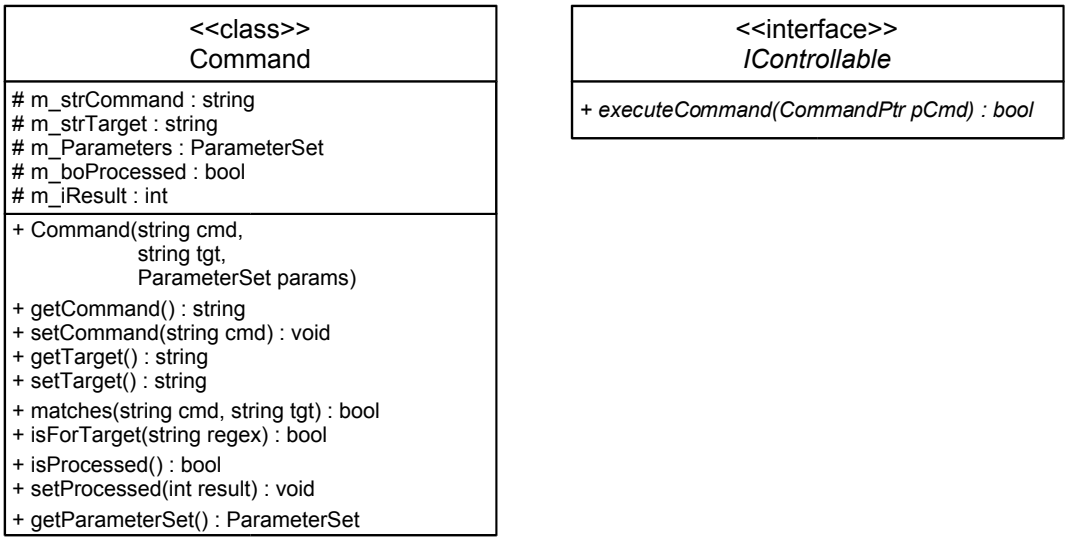


Abbildung 7.4: UML-Diagramm der Klassen für Kommandos

Ein Kommando trägt immer eine Bezeichnung (`get/setCommand(...)`) und einen Empfänger (`get/setTarget(...)`). Optional können noch zusätzliche Parameter übergeben werden. Ein Befehl gilt als abgearbeitet, wenn die Methode `isProcessed()` `true` zurückgibt. Dies wird dem Kommando durch `setProcessed(result)` mitgeteilt. `result` kann dabei als Rückgabewert signalisieren, ob ein Kommando ausgeführt werden konnte oder, wenn nicht, mit welchem Fehlercode.

Eine Klasse, welche das Interface *`IControllable`* implementiert, kann mit Kommandos gesteuert werden. Wenn sie über `executeCommand(pCmd)` ein Kommando empfängt, kann die Klasse zunächst anhand der Methoden `matches(...)` und `isForTarget(regex)` mit Hilfe von regulären Ausdrücken sehr flexibel prüfen, ob das Kommando für diese Klasse bestimmt ist. Wenn dies der Fall ist, muss das Kommando ausgeführt und mit `setProcessed(result)` als behandelt markiert werden. Tabelle 7.2 auf der nächsten Seite zeigt einige Beispiele für die Überprüfung von Kommando und Ziel.

Jedes Modul ist so implementiert, dass es die Kommandos, welche im Modulinterface festgelegt sind, auch per Kommando empfangen kann. Das erleichtert z.B. die Programmierung von Modulen, welche über das Netzwerk kommunizieren, da nicht für jede Methode ein eigenes Paket geschaffen werden muss. Statt dessen reicht die einmalige Implementierung des Sende- und Empfangsvorgangs für Kommandos.

Kommando	Ziel	Überprüfung mit	Resultat
`exit`	`*`	`matches("exit", "")`	`true`
		`matches("stop", "")`	`false`
`exit`	`render`	`matches("exit", "")`	`false`
		`matches("exit", "render")`	`true`
`exit`	`render|physics`	`matches("exit", "")`	`false`
		`matches("exit", "render")`	`true`
		`matches("exit", "physics")`	`true`
		`matches("exit", "neural")`	`false`

Tabelle 7.2: Überprüfung von Kommando und Ziel

7.5 Ressourcenmanagement

Die Klasse `ResourceManager` wurde als eine zentrale Verwaltungsstelle für Datenbäume, Skripte, Meshes usw. eingeführt (siehe Abbildung 7.5 auf der nächsten Seite).
Für alle zu verwaltenden Ressourcen muss eine typisierte Variante von `template <Objekttyp> ResourceImpl` existieren, welche über die Methode `getResource()` Zugriff auf eine geladene Ressource erlaubt. Zusätzlich muss diese Ressource über ein Plugin erzeugbar sein, welches von `interface` *`IResourceManagerPlugin`* abgeleitet wurde. Alle Plugins müssen zu Beginn des Programms beim Manager über `registerPlugin(...)` angemeldet werden.
Alle Ressourcen werden in *`m_Resources`* vom Manager verwaltet und können über `getResource(...)` abgefragt werden. Ist eine Ressource noch nicht geladen, so prüft der Manager über `canHandleResource(...)`, ob ein Plugin für diesen Typ von Ressource zuständig ist und erzeugt diese mittels `createResource(...)` wenn `true` zurückgegeben wird.

7.6 Iteratoren

Ein wichtiges Konzept der objektorientierten Programmierung ist das Verbergen von Informationen. Wenn eine Methode eine Liste von Objekten zurückgibt, so muss der Benutzer nicht unbedingt wissen, wie diese Liste organisiert ist (z.B. STL[2]-Klassen `map`, `set`, `list` oder `vector`). Er benötigt lediglich Funktionen, um sich innerhalb dieser Liste zu bewegen und Objekte zu lesen oder zu verändern.
Fast alle wichtigen Modulinterfaces, die im Verlauf dieser Thesis entworfen wurden, bieten Zugriff auf eine Sammlung von Objekten. Je nach konkreter Implementierung des Modulinterfaces werden diese Objekte unterschiedlich verwaltet. Eine Render-Engine z.B. assoziiert ein Objekt über seinen Namen und verwendet deshalb einen Assoziativspeicher (STL: „map“). Ein neuronales Netzwerk hingegen verwaltet Neuronen eher nach einer

[2] Die STL (**s**tandard **t**emplate **l**ibrary) ist eine Sammlung von C++-Templates, welche viele grundlegende Algorithmen wie Suchen, Sortieren und grundlegende Datenstrukturen wie Listen, Stacks, Maps implementiert.

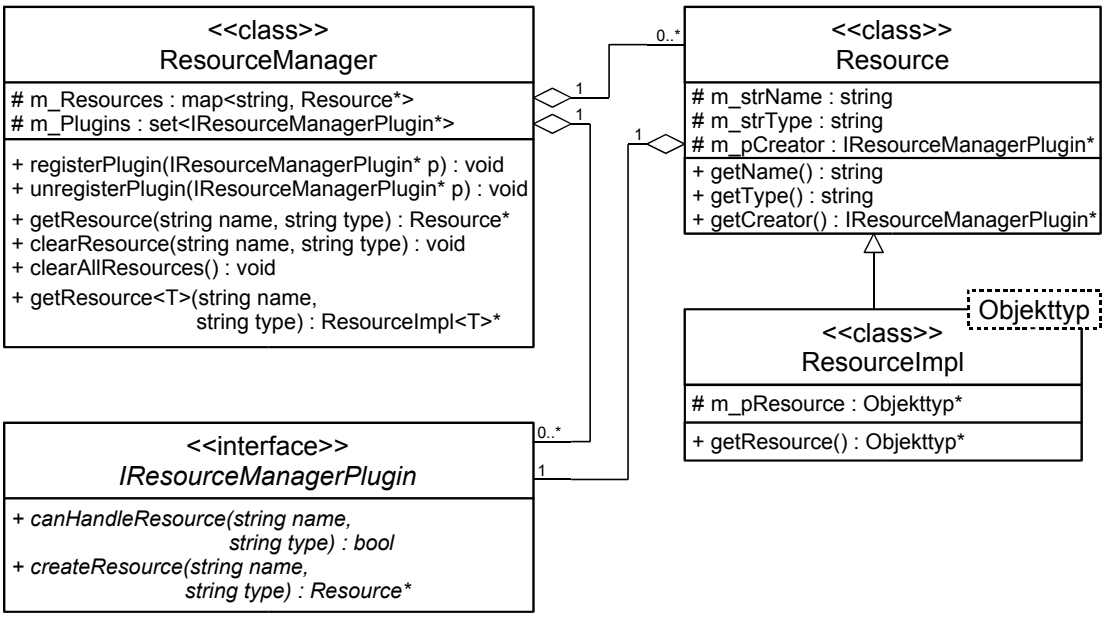


Abbildung 7.5: UML-Diagramm der Klassen für Ressourcenmanagement

numerischen ID und verwendet aus diesem Grund eine einfache Tabelle (STL: „vector“). Beide Modulinterfaces bieten Methoden, welche eine Liste der Objekte bzw. Neuronen zurückgibt (`getNodeList()`, `getNeuronList()`). Es wäre durchaus möglich, direkt eine Referenz auf die verwendeten Speicherklasse zurückzugeben. Das hätte allerdings den Nachteil, dass damit schon im Modulinterface der Typ dieser Klasse festgelegt wird.
Um diese Einschränkung zu vermeiden, wurde angelehnt an die Iteratoren der STL eine Template-Basisklasse `class Iterator<Objekttyp>` entworfen, welche grundlegende Funktionen zur Verfügung stellt (siehe Abbildung 7.6 auf der nächsten Seite und Programmcode 7.1 auf Seite 94):

- `hasNext() : bool`
  Liefert `true` zurück, wenn noch weitere Elemente vorhanden sind, d.h. der Lesezeiger noch nicht am Ende angelangt ist.

- `getNext() : Objekttyp*`
  Liefert einen Pointer auf das nächste Element in der Liste zurück und setzt den Lesezeiger ein Element weiter.

- `rewind() : void`
  Setzt den Lesezeiger zurück auf den Anfang.

Nun bietet sich die Situation, dass z.B. im Modulinterface der Render-Engine die Methode `getNodeList()` einen Iterator vom Typ `Iterator<render::INode>` zurückgibt und im Modulinterface des neuronalen Netzwerks die Methode `getNeuronList()` einen Iterator vom Typ `Iterator<neural::INeuron>`.

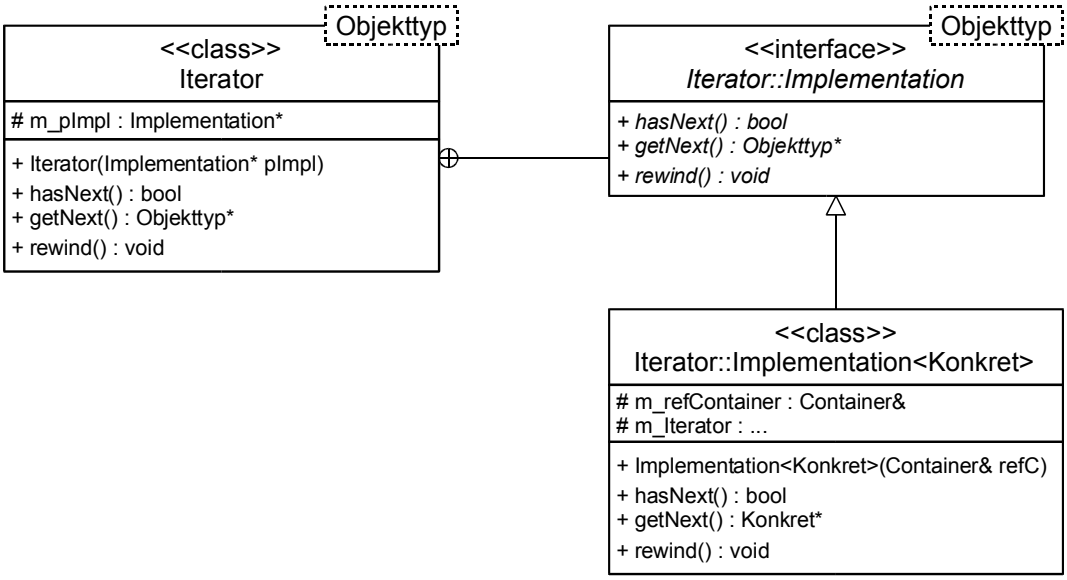


Abbildung 7.6: UML-Diagramm des Iterator-Templates

Jede konkrete Implementierung der Modulinterfaces leitet eine Hilfsklasse von `interface` *`Iterator::Implementation`* ab und implementiert die dort deklarierten Methoden, die Zugriff auf die konkrete Speicherklasse erhalten (siehe Programmcode 7.2 auf der nächsten Seite). Bei Aufruf der Methode `getXyzList()` wird ein Objekt dieser Hilfsklasse instanziiert und bei der Konstruktion des Iterators übergeben, der von der Methode zurückgegeben wird.

```
template <class T>
  class Iterator
{
  public:

    class Implementation
    {
      public:
        virtual ~Implementation() {};
        virtual bool  hasNext() const = 0;
        virtual T*    getNext() = 0;
        virtual void  rewind()  = 0;
    };

    Iterator(Implementation*  pImpl = NULL) : m_pImpl(pImpl)
    {
    };

    ~Iterator()
    {
      if ( m_pImpl != NULL )
      {
        delete m_pImpl;
        m_pImpl = NULL;
      }
    };

    bool hasNext() const
    {
      if ( m_pImpl != NULL ) return m_pImpl->hasNext();
      else                   return false;
```

```
    }

    T* getNext()
    {
      if ( m_pImpl != NULL ) return m_pImpl->getNext();
      else                   return NULL;
    }

    void rewind()
    {
      if ( m_pImpl != NULL ) m_pImpl->rewind();
    }

  protected:

    Implementation*  m_pImpl;

};
```

Programmcode 7.1: Auszug aus `Iterator.h`

```
typedef Iterator<Node>  NodeIterator;

/*****************************************************************
 * NodeMapIterator class
 *****************************************************************/

class NodeMapIterator : virtual public NodeIterator::Implementation
{
  public:

    NodeMapIterator(NodeMap& refMap) :
      m_Map(refMap),
      m_Pos(refMap.begin())
    {
    };

    virtual bool hasNext() const
    {
      return bool(m_Pos != m_Map.end());
    };

    virtual INode* getNext()
    {
      INode*  pReturn = NULL;

      if ( hasNext() )
      {
        pReturn = (*m_Pos).second.ptr();
        m_Pos++;
      }

      return pReturn;
    };

    virtual void rewind()
    {
      m_Pos = m_Map.begin();
    };

  protected:

    NodeMap&           m_Map;  ///< Reference to render node map
    NodeMap::iterator  m_Pos;  ///< Actual position in the map

};
```

```
/*************************************************************************
 * Engine class
 *************************************************************************/
.
.
.
NodeIterator Engine::getNodeIterator() const
{
  return NodeIterator(new NodeMapIterator(m_mapNodes));
}
.
.
.
```

Programmcode 7.2: Auszug aus `RenderEngine.cpp`

7.7 Physik-Engine

7.7.1 Vergleich von Physik-Engines

Tabelle 7.3 auf der nächsten Seite zeigt die Möglichkeiten von verschiedenen selbst entwickelten, frei und kommerziell erhältlichen Physik-Engines, welche in C/C++ implementiert sind:

1. **Havok**[3]
   Kommerzielle Physik-Engine
   © Havok.com Inc.
   `http://www.havok.com`

2. **NovodeX**[4]
   Kommerzielle Physik-Engine
   © AGEIA Inc.
   `http://www.ageia.com/novodex.html`

3. **ODE**[5] (**O**pen **D**ynamics **E**ngine)
   Open-Source Physik-Engine
   © 2001-2004 Russell L. Smith
   `http://ode.org`

4. **PE1**
   Selbst entwickelte Physik-Engine aus der Vorlesung Virtuelle Umgebungen A (VUM-A) im WS 2003.

In der ersten Phase des Projekts wurde PE1 benutzt, dann aber schnell durch ODE ersetzt, als die Grenzen dieser doch recht eingeschränkten Engine deutlich wurden. Allein die Implementierung von Gelenken und Constraints hätte den Rahmen der Thesis gesprengt. Hauptausschlaggebend für die Wahl von ODE war die freie Verfügbarkeit, die Eigenschaft, unter Windows und Linux kompilierbar zu sein und die gute Unterstützung durch Internet-Foren und Erfahrungsberichte zahlreicher Personen und Firmen, welche diese Engine bereits eingesetzt haben.
Gleichzeitig bietet ODE eine isoliert lauffähige Kollisions-Engine. Dadurch vereinfachte sich die Suche, Bewertung, Auswahl und Implementierung einer solchen erheblich (siehe Abschnitt 7.8 auf Seite 108).

[3] Beschreibung: `http://www.havok.com/products/physics.php`
[4] Beschreibung: `http://www.ageia.com/pdf/ds_2005_3_novodex_physics_sdk.pdf`
[5] Beschreibung: `http://ode.org/ode-latest-userguide.pdf`

Physik-Engine	Havok	NovodeX	ODE	PE1
Plattform				
Windows	+	+	+	+
Linux	-	-	+	+
Körper				
Quader	+	+	+	+
Kugel	+	+	+	+
Zylinder	+	+	+	+
Konus	+	+	+	+
beliebige Formen	+	+	+	+
Kollisionserkennung	+	+	+	+
Quader	+	+	+	-
Kugel	+	+	+	-
Zylinder	+	+	(+)[6]	-
Kapsel	+	+	+	-
Ebenen	+	+	+	-
konvexe Formen	+	+	+	-
konkave Formen	+	+	+	-
Meshes	+	+	+	+
Gelenke	+	+	+	-
Fest	+	+	+	
Scharnier	+	+	+	
Linear	+	+	+	
Kreuz	+	+	+	
Kugel	+	+	+	
beliebig	+	+	(+)[7]	
Motoren	+	+	+	-
Federn/Dämpfer	+	+	(+)[8]	-
Events	unbekannt	+	-	-
Multithreading	unbekannt	+	-	-
Hardwareunterstützung	-	(+)[9]	-	-

Tabelle 7.3: Vergleich von Physik-Engines

[6] Durch zusätzlichen Code
[7] Durch zusätzlichen Code
[8] Durch „Missbrauch“ von Simulationsparametern
[9] Geplant: Der Physik-Coprozessor PhysX (PPU: Physics Processing Unit) von AGEIA Technologies Inc. (siehe `http://www.ageia.com/technology.html`)

7.7.2 Interface

Das Modulinterface der Physik-Engine bietet Funktionen zur Erzeugung von starren Körpern und Gelenken, zur Kontrolle von Schwerkraft und Zeit sowie zur Simulation der physikalischen Welt (siehe Abbildung 7.7 und Abbildung 7.8 auf Seite 103).

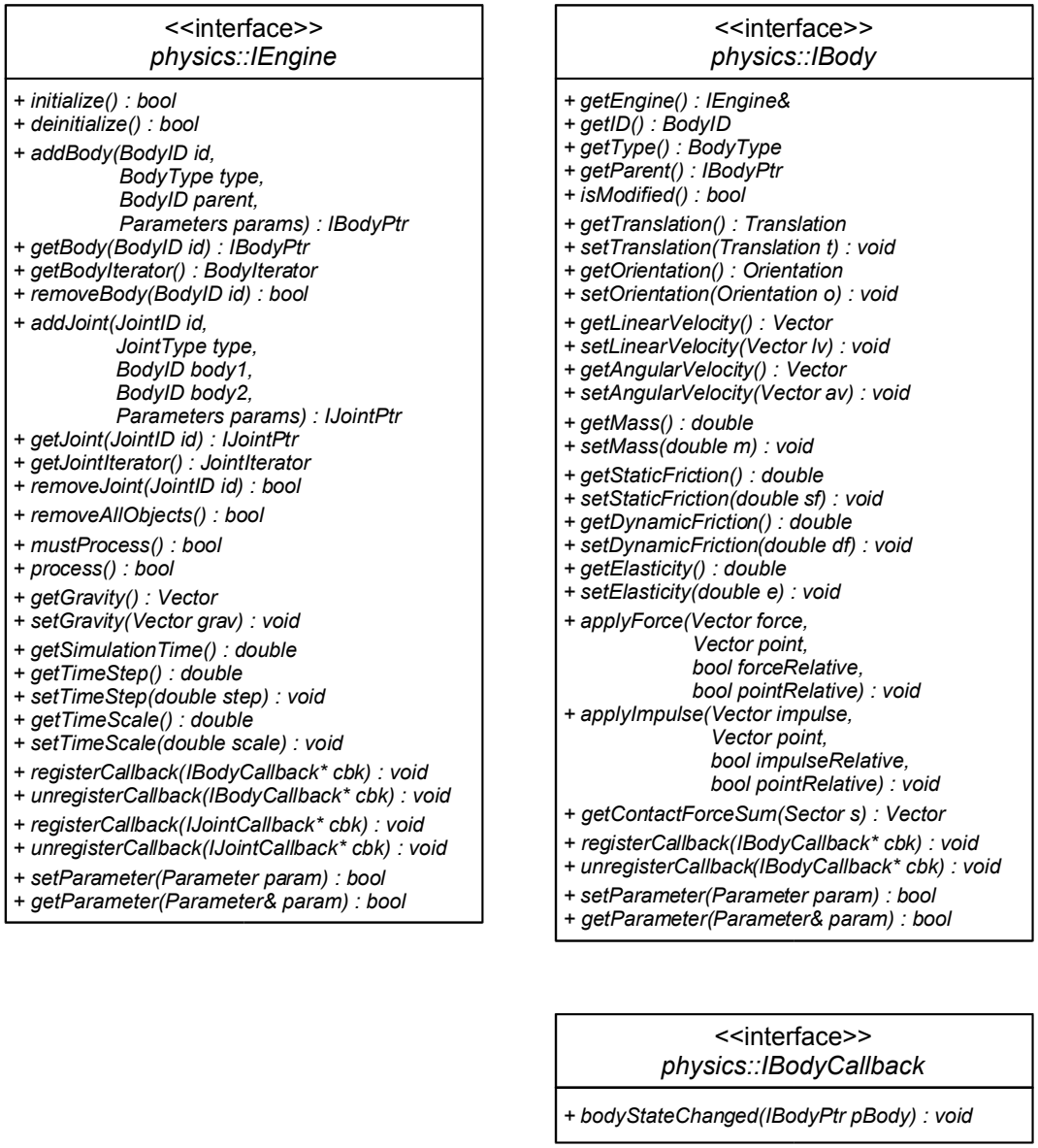


Abbildung 7.7: UML-Diagramm des Modulinterfaces der Physik-Engine (Teil 1)

7.7.2.1 `interface physics::IEngine`

Dieses Interface bietet Zugriff auf alle wesentlichen Funktionen der Physik-Engine:

- **Initialisierung / Deinitialisierung**
  Zu Beginn des Programmablaufs muss die Physik-Engine mit `initialize()` initia-

lisiert werden. Dadurch haben konkrete Implementierungen die Möglichkeit, Vorbereitungen zu treffen, die eigentliche Engine zu starten und Ressourcen anzufordern.

Nach Abschluss der Simulation wird mit `deinitialize()` die Engine wieder heruntergefahren und alle Ressourcen können freigegeben werden.

- **Körper und Gelenke**
  Mit Hilfe von `addBody(...)` bzw. `addJoint(...)` werden starre Körper bzw. Gelenke in die Simulation eingefügt. Alle zur Konstruktion unabdingbaren Daten werden direkt übergeben. Alle weiteren Parameter, die je nach konkreter Engine spezielle Ausprägungen haben können, werden über die Parameter `params` übergeben (siehe Programmcode 7.3 auf der nächsten Seite und Programmcode 7.4 auf der nächsten Seite).

  Mit den Methoden `getBody(...)` bzw. `getJoint(...)` kann man gezielt einzelne Körper bzw. Gelenke abfragen. Mit `getBodyIterator()` bzw. `getJointIterator()` erhält man Iteratoren, welche einen Gesamtüberblick über alle Objekte innerhalb der Simulation erlauben.

  `removeBody(...)` und `removeJoint(...)` dienen zum gezielten Löschen von Körpern bzw. Gelenken. `removeAllObjects()` löscht alle Objekte innerhalb der Simulation, um z.B. eine Szene neu zu laden.

- **Simulationsprozess**
  Innerhalb der Hauptschleife des umgebenden Programms kann man mittels `mustProcess()` abfragen, ob die Physik-Engine bereit für den nächsten Simulationsschritt ist. Wenn diese Methode `true` zurückgibt, kann dieser Schritt mit `process()` angestoßen werden.

- **Informationsfluss**
  Während der Simulation verändern Körper und Gelenke fortwährend ihren Zustand. Um darüber informiert zu sein, kann man mit `registerBodyCallback(...)` bzw. `registerJointCallback(...)` Callback-Funktionen anmelden, welche nach jeder Änderung mit dem aktuellen Zustand des betreffenden Objekts aufgerufen werden. `unregisterBodyCallback(...)` und `unregisterJointCallback(...)` melden diese Callback-Funktionen wieder ab.

- **Simulationsparameter**
  Einige Methoden dienen der Beeinflussung von Simulationsparametern. Mit `get/setGravity()` kann man den Schwerkraftvektor verändern. Die Zeitspanne eines Simulationsschrittes wird mit `get/setTimeStep()` festgelegt und für Zeitlupe/Zeitraffer kann man die Simulationszeit mit `get/setTimeScale(...)` variieren.

  Parameter einer konkreten Physik-Engine, welche nicht im Modulinterface enthalten sind, können mittels der Methoden `get/setParameter(...)` gelesen bzw. verändert

werden. Bei der Physik-Engine ODE kann man so z.B. globale Parameter zur Feinkontrolle der Kollisionsbehandlung einstellen.

```
'body' (Type=ParameterSet)
  'id' = 'ball_07' (Type=string)
  'type' = 'sphere' (Type=string)
  'parameters' (Type=ParameterSet)
    'dimension' (Type=ParameterSet)
      'x' = '0.1' (Type=string)
      'y' = '0.1' (Type=string)
      'z' = '0.1' (Type=string)
    'mass' = '0.25' (Type=string)
    'material' (Type=ParameterSet)
      'elasticity' = '0.9' (Type=string)
      'friction' = '0.2' (Type=string)
    'orientation' (Type=ParameterSet)
      'w' = '1' (Type=double)
      'x' = '0' (Type=double)
      'y' = '0' (Type=double)
      'z' = '0' (Type=double)
    'translation' (Type=ParameterSet)
      'x' = '-0.9' (Type=double)
      'y' = '1.05' (Type=double)
      'z' = '-0.101' (Type=double)
    'velocity' (Type=ParameterSet)
      'linear' (Type=ParameterSet)
        'x' = '0' (Type=string)
        'y' = '0' (Type=string)
        'z' = '0' (Type=string)
```

Programmcode 7.3: Parameter bei Erzeugung eines physikalischen Körpers

```
'joint' (Type=ParameterSet)
  'body1' = 'leg' (Type=string)
  'body2' = 'piston' (Type=string)
  'id' = 'piston' (Type=string)
  'type' = 'linear' (Type=string)
  'parameters' (Type=ParameterSet)
    'anchor' (Type=ParameterSet)
      'x' = '0' (Type=double)
      'y' = '0.75' (Type=double)
      'z' = '0' (Type=double)
    'axis1' (Type=ParameterSet)
      'direction' (Type=ParameterSet)
        'x' = '0' (Type=double)
        'y' = '1' (Type=double)
        'z' = '0' (Type=double)
      'limits' (Type=ParameterSet)
        'elasticity' = '0.1' (Type=string)
        'high' (Type=ParameterSet)
          'enabled' = 'true' (Type=string)
          'value' = '0' (Type=string)
        'low' (Type=ParameterSet)
          'enabled' = 'true' (Type=string)
          'value' = '-0.75' (Type=string)
      'motor' (Type=ParameterSet)
        'enabled' = 'true' (Type=string)
        'limits' (Type=ParameterSet)
          'force' = '5000' (Type=string)
          'velocity' = '5' (Type=string)
```

Programmcode 7.4: Parameter bei Erzeugung eines physikalischen Gelenks

7.7.2.2 `interface physics::IBody`

Dieses Interface repräsentiert einen einzelnen starren Körper:

- **Informationen**
  Jeder Körper besitzt eine eindeutige ID, welche über `getId()` ermittelbar ist, und einen Typ (z.B. Quader, Zylinder), den man durch `getType()` auslesen kann. Die Physik-Engine, welche diesen Körper verwaltet, kann mit `getEngine()` abgefragt werden. Ist der Körper Bestandteil eines größeren Ganzen (z.B. Auge), so gibt `getParent()` den Körper an, welcher in der Hierarchie eine Stufe höher steht (z.B. Kopf).

  Wurde der Körper im letzten Simulationsschritt verändert, so erhält man bei Aufruf von `isModified()` `true`.

- **Zustand**
  Die Translation und Orientierung des Körpers lässt sich mit `get/setTranslation(...)` bzw. `get/setOrientation(...)` abfragen bzw. festlegen.

  `get/setLinearVelocity(...)` und `get/setAngularVelocity(...)` geben Auskunft über die Linear- und Winkelgeschwindigheit bzw. erlauben es, diese zu setzen.

  Zugriff auf die Masse des Körpers hat man mit `get/setMass(...)`.

- **Oberfläche**
  Der statische und dynamische Reibungskoeffizient sowie die Elastizität des Körpers sind mit den Methoden `get/setStaticFriction(...)`, `get/setDynamicFriction(...)` und `get/setElasticity(...)` zu kontrollieren.

- **Kräfte**
  Mit `applyForce(...)` und `applyImpulse(...)` kann man mit Kräften und Impulsen auf den Körper einwirken. Dabei kann der Kraft- bzw. der Impulsvektor sowie der Angriffspunkt absolut im Weltkoordinatensystem oder relativ im Körperkoordinatensystem angegeben werden.

  `getContactForceSum(...)` ermittelt die Summe angreifender Kräfte aufgrund von Kollisionen. Mit dieser Methode erzielt man eine Art „Druckempfinden“ des Körpers. Diese Information wird in `cerebellum` verwendet, um dem neuronalen Netzwerk zu signalisieren, dass die Füße eines virtuellen Charakters Kontakt mit dem Boden haben.

- **Informationsfluss / Simulationsparameter**
  Die Methoden `registerBodyCallback(...)`, `unregisterBodyCallback(...)` und `get/setParameter(...)` haben ähnliche Bedeutungen wie ihre Pendants in `interface` *`physics::IEngine`* (auf Seiten 100–101).

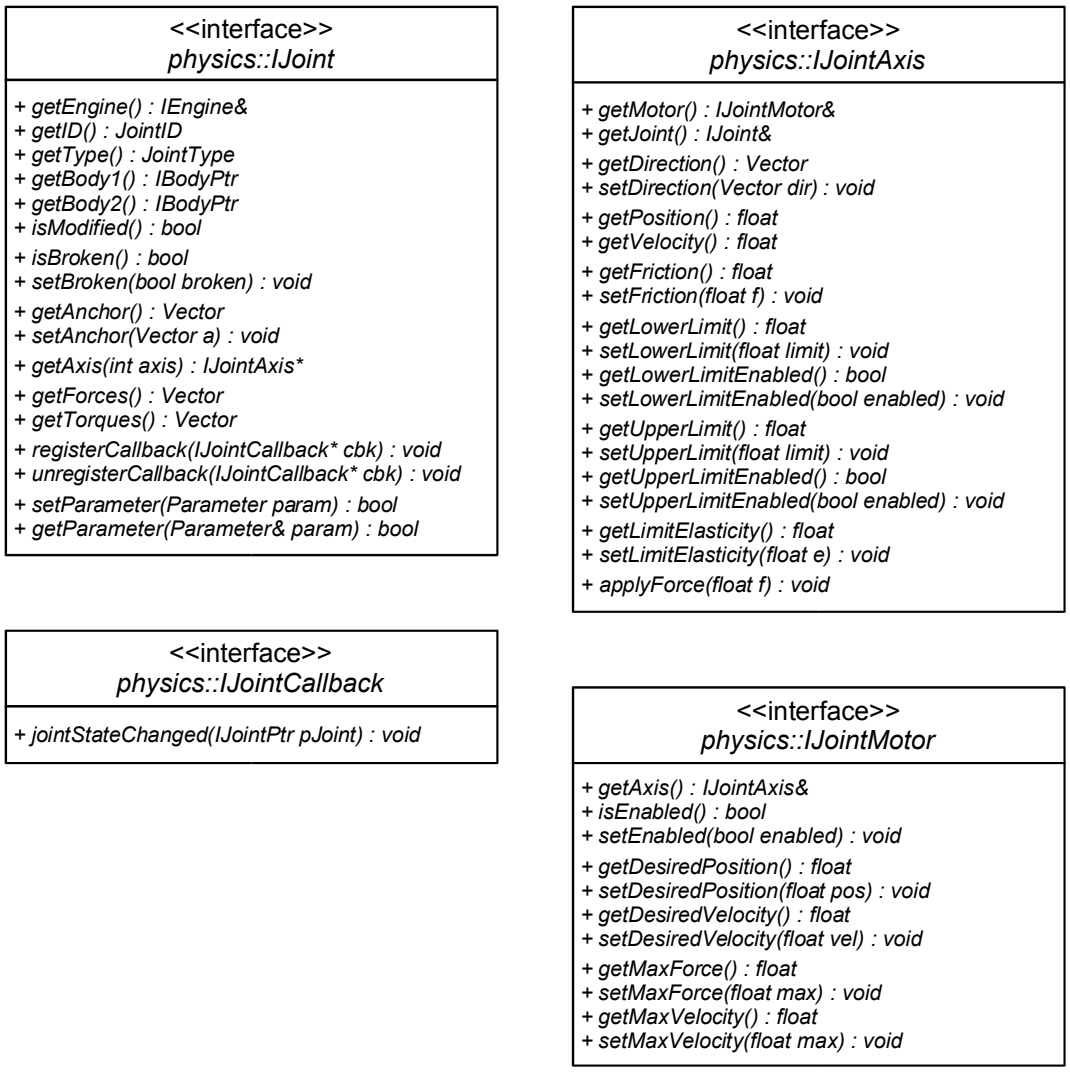


Abbildung 7.8: UML-Diagramm des Modulinterfaces der Physik-Engine (Teil 2)

7.7.2.3 `interface physics::IJoint`

Dieses Interface repräsentiert ein einzelnes Gelenk:

- **Informationen**
  Jedes Gelenk besitzt eine eindeutige ID, welche über `getId()` ermittelbar ist, sowie einen Typ (z.B. Scharnier, Kugelgelenk), den man durch `getType()` auslesen kann. Die Physik-Engine, welche dieses Gelenk verwaltet, kann mit `getEngine()` abgefragt werden.

  Es verbindet die beiden Körper `getBody1()` und `getBody2()`. Ist einer der beiden zurückgegebenen Pointer `NULL`, so bedeutet dies, dass die betreffende Seite des Gelenks unbeweglich ist – sozusagen in der simulierten Welt „festzementiert“.

  Wurde das Gelenk im letzten Simulationsschritt verändert, so erhält man bei Aufruf von `isModified() true`.

- **Zustand**
  Gelenke können bei hoher Belastung „brechen“ oder im Zuge einer Simulation aktiv aufgebrochen werden. Um abzufragen, ob das Gelenk zerbrochen ist, dient die Methode `isBroken()`. Mit `setBroken(...)` kann man ein Gelenk aktiv aufbrechen oder wieder zusammenfügen.

- **Parameter**
  Der Ankerpunkt `getAnchor()` ist der Punkt, in welchem sich die Achsen der translatorischen und rotatorischen Freiheitsgrade treffen. Diese Achsen kann man über `get/setAxis(...)` abfragen und manipulieren.

- **Kräfte**
  Mit `getForces()` und `getTorques()` erhält man Auskunft über die aktuell auf das Gelenk einwirkenden Kräfte und Drehmomente.

- **Informationsfluss / Simulationsparameter**
  Die Methoden `registerJointCallback(...)`, `unregisterJointCallback(...)` und `get/setParameter(...)` haben ähnliche Bedeutungen wie ihre Pendants in `interface` *`physics::IEngine`* (auf Seiten 100–101).

7.7.2.4 `interface physics::IJointAxis`

Dieses Interface repräsentiert eine Achse eines Gelenks:

- **Informationen**
  Eine Gelenkachse ist immer einem Gelenk `getJoint()` zugeordnet und besitzt einen Motor `getMotor()`.

  Die Richtung der Gelenkachse kann man mit `get/setDirection()` lesen oder beeinflussen.

- **Zustand**
  Die aktuelle Position und Geschwindigkeit einer Achse kann über `getPosition()` und `getVelocity()` abgefragt werden. Dabei entspricht die Position bei einem Lineargelenk der Verschiebung zum Nullpunkt, bei einem rotatorischen Gelenk dem Winkel relativ zur Nullposition.

- **Parameter**
  Bei einer Gelenkachse ist die Reibung (`get/setFriction(...)`) und die minimale und maximale Position einstellbar (`get/setUpperLimit(...)` bzw. `get/setLowerLimit(...)`). Die Minimal- und/oder Maximal-Position ist auch komplett abschaltbar (`get/setUpperLimitEnabled(...)` bzw. `get/set LowerLimitEnabled(...)`).

  `get/setLimitElasticity()` stellen die Elastizität der Limits ein. Dies ermöglicht z.B. die Simulation weicher Anschläge eines Gelenks.

- **Kräfte**
  Mit `applyForce(...)` kann man eine Kraft direkt auf ein Gelenk entlang einer Achse einwirken lassen.

7.7.2.5 `interface physics::IJointMotor`

Dieses Interface repräsentiert einen Motor, welcher eine Gelenkachse antreibt:

- **Informationen**
  Ein Achsenmotor ist immer einer Gelenkachse `getAxis()` zugeordnet.
- **Zustand**
  Der Motor kann ein oder ausgeschaltet sein (`is`/`setEnabled(...)`).
- **Parameter**
  Ein Motor kann eine Gelenkachse in eine gewünschte Position fahren oder eine gewünschte Sollgeschwindigkeit anstreben. Dazu dienen die Methoden `get`/`setDesiredPosition(...)` bzw. `get`/`setDesiredVelocity(...)`.

  Um diese Vorgaben zu erzielen, darf der Motor maximal die Kraft `get`/`setMaxForce(...)` aufwenden und maximal mit der Geschwindigkeit `get`/`setMaxVelocity(...)` fahren.

7.7.3 Konkrete Implementierung

Abbildung 7.9 und Abbildung 7.10 auf der nächsten Seite zeigen die konkrete Implementierung des Modulinterfaces durch ODE.

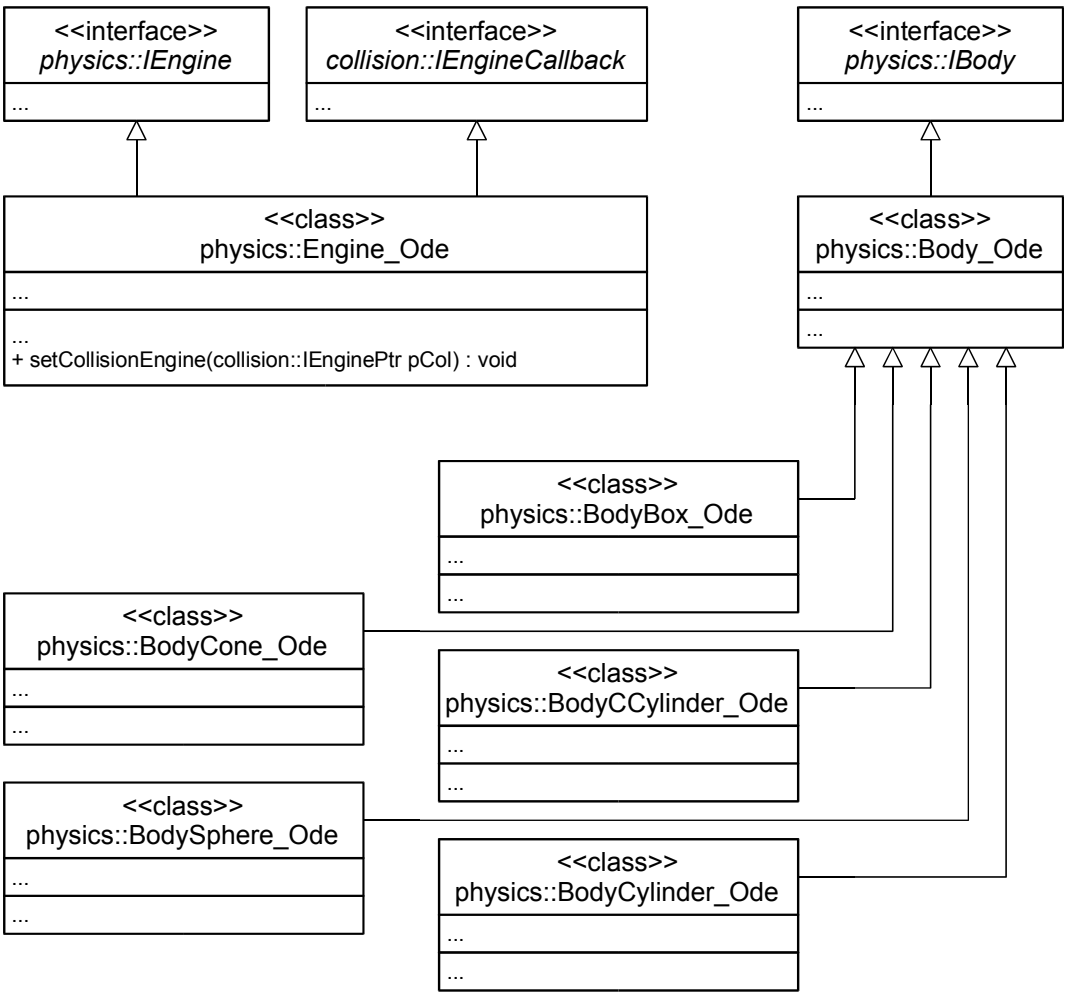


Abbildung 7.9: UML-Diagramm der Klassen der ODE Physik-Engine (Teil 1)

`class physics::Engine_Ode` implementiert das Modulinterface der Physik-Engine und erbt zusätzlich die Methode des Callback-Interfaces der Kollisions-Engine. Mit dieser zusätzlichen Ableitung ist es möglich, über die Methode `setCollisionEngine(...)` eine enge Verzahnung mit einer Kollisions-Engine zu erreichen. In diesem Fall meldet sich die Physik-Engine bei dieser als Callback an, versorgt nach jedem Simulationsschritt die Kollisions-Engine automatisch mit den aktuellen Daten und erhält über den Callback automatisch die dadurch entstehenden Kollisionen. Diese können somit direkt verarbeitet werden. Ohne diese Möglichkeit der direkten Anmeldung müsste eine exra für diesen Zweck eine Art „Verbindungsklasse“ programmiert werden.

Die verschiedenen physikalischen Körper- und Gelenkformen werden über die Zwischenklassen `class physics::Body_Ode` bzw. `class physics::Joint_Ode` abgeleitet. Diese

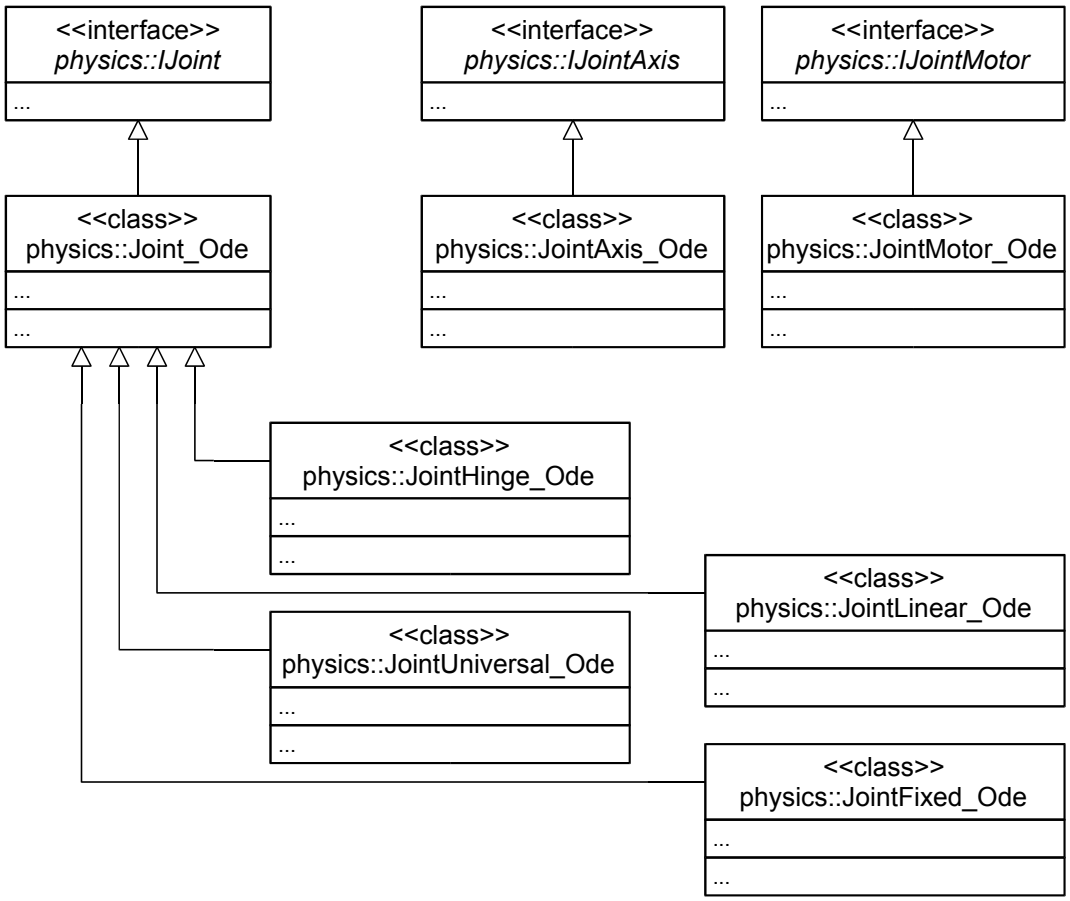


Abbildung 7.10: UML-Diagramm der Klassen der ODE Physik-Engine (Teil 2)

Klassen implementieren `interface` *`physics::IBody`* bzw. `interface` *`physics::IJoint`* und sind für Eigenschaften zuständig, welche allen Körpern bzw. Gelenken gemeinsam sind, z.B. `get/setTranslation(...)`, `get/setAnchor(...)`.
Die von diesen Klassen abgeleiteten Objekte repräsentieren letzlich die konkreten Körper wie Quader, Konus, Kugel, Zylinder und abgerundeter Zylinder bzw. die konkreten Gelenke wie Scharnier-, Kreuz-, Lineargelenk und Festverbindung.

7.8 Kollisionserkennung

7.8.1 Vergleich von Kollisions-Engines

Erste Nachforschungen zu frei verfügbaren und unter Windows und Linux lauffähigen Kollisions-Engines führten zu folgenden Bibliotheken:

1. **ColDet** (3D **col**lision **det**ection library)
   © 2000 Amir Geva
   `http://photoneffect.com/coldet`

2. **Opcode** (**op**timized **co**llision **de**tection)
   © 2001 Pierre Terdiman
   `http://www.codercorner.com/Opcode.htm`

3. **Solid** (**so**ftware **l**ibrary for **i**nterference **d**etection)
   © 1997-1998 Gino van den Bergen
   `http://www.win.tue.nl/~gino/solid/index.html`

Die Eigenschaften der drei Engines werden in Tabelle 7.4 gegenübergestellt.

Kollisions-Engine	ColDet	Opcode	Solid
Plattform			
Windows	+	+	+
Linux	+	+	+
Geometrien			
Linie	-	+	+
Kugel	-	+	+
Quader	-	+	+
Zylinder	-	-	+
Konus	-	+	+
Kapsel	-	+	-
Ebenen	-	+	-
konvexe Formen	+	+	+
konkave Formen	+	+	+
Meshes	+	+	+

Tabelle 7.4: Vergleich von Kollisions-Engines

Letztlich fiel die Wahl auf Opcode, da diese Bibliothek bereits in der Physik-Engine ODE integriert und mit einem einfachen API[10] versehen war.

[10] Ein API (engl.: „**a**pplication **p**rogrammers **i**nterface“) ist eine Festlegung und Beschreibung der Funktionen, die dem Benutzer einer Bibliothek zur Verfügung stehen.

7.8.2 Interface

Das Modulinterface der Kollisions-Engine bietet Funktionen zur Erzeugung von Kollisionsobjekten und zur Berechnung von Kollisionen dieser Objekte untereinander (siehe Abbildung 7.11).

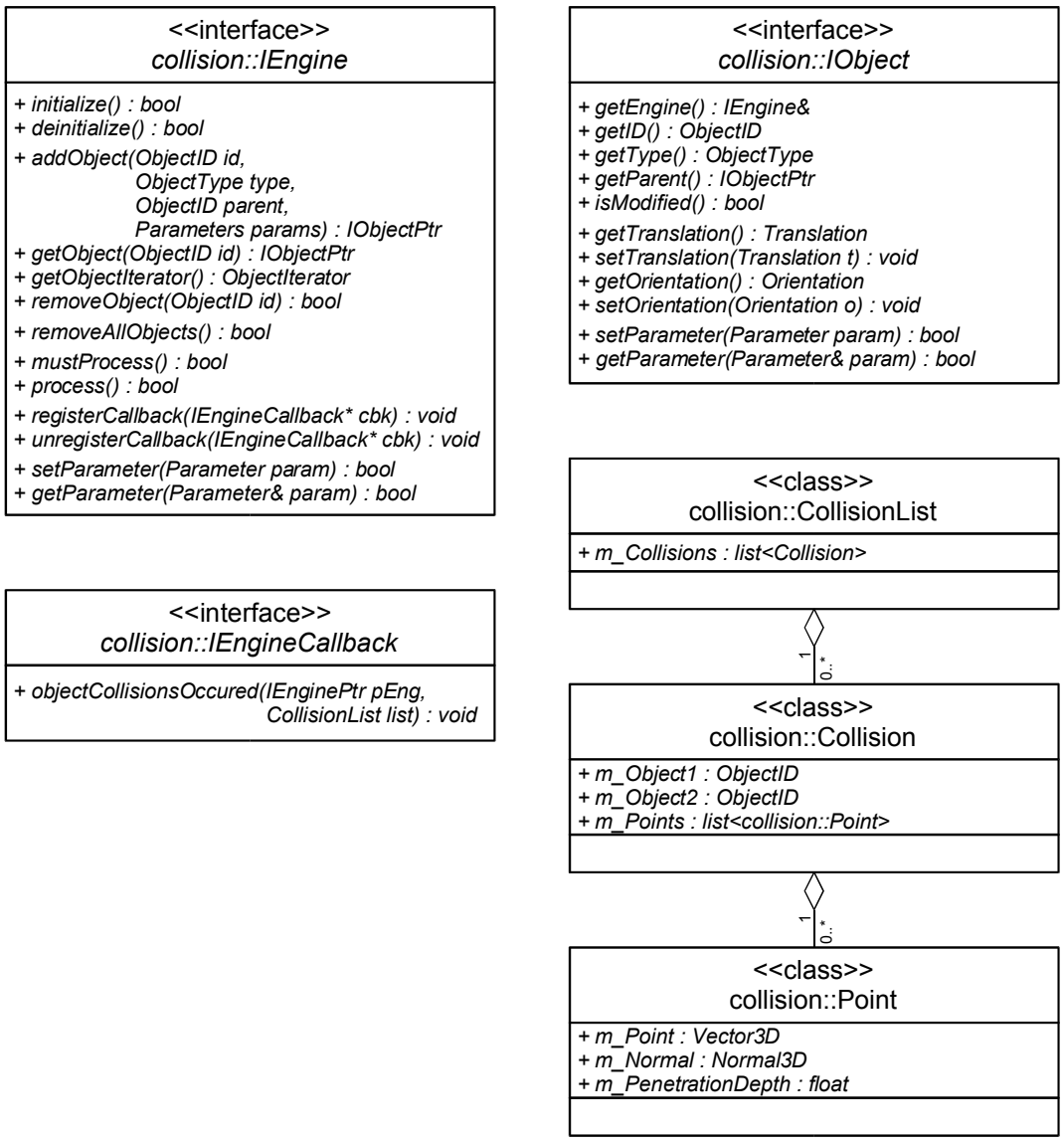


Abbildung 7.11: UML-Diagramm des Modulinterfaces der Kollisionserkennung

7.8.2.1 `interface collision::IEngine`

Dieses Interface bietet Zugriff auf alle wesentlichen Funktionen der Kollisions-Engine:

- **Initialisierung / Deinitialisierung**
  Zu Beginn des Programmablaufs muss die Kollisions-Engine mit `initialize()` initialisiert werden. Dadurch haben konkrete Implementierungen die Möglichkeit, Vor-

bereitungen zu treffen, die eigentliche Engine zu starten und Ressourcen anzufordern.

Nach Abschluss der Simulation wird mit `deinitialize()` die Engine wieder heruntergefahren und alle Ressourcen können freigegeben werden.

- **Kollisionsobjekte**
  Mit Hilfe von `addObject(...)` werden Kollisionsobjekte eingefügt. Alle zur Konstruktion unabdingbaren Daten werden direkt übergeben. Alle weiteren Parameter, die je nach konkreter Engine spezielle Ausprägungen haben können, werden über die Parameter `params` übergeben (siehe Programmcode 7.5).

  Mit der Methode `getObjecty(...)` kann man gezielt einzelne Objekte abfragen. `getObjectIterator()` liefert einen Iterator über alle Kollisionsobjekte.

  Mit `removeObject(...)` bzw. `removeAllObjects()` können einzelne bzw. alle Kollisionsobjekte gelöscht werden.

- **Simulationsprozess**
  Innerhalb der Hauptschleife des umgebenden Programms kann man mittels `mustProcess()` abfragen, ob die Kollisions-Engine Änderungen an Objekten festgestellt hat, die es erfordern, Neuberechnungen anzustellen. Wenn diese Methode `true` zurückgibt, kann dieser Schritt mit `process()` angestoßen werden.

- **Informationsfluss**
  Mit `registerCallback(...)` bzw. `unregisterCallback(...)` werden Callback-Funktionen an- bzw. abgemeldet. Diese Funktionen erhalten nach jedem Durchlauf eine Liste der berechneten Kollisionspunkte

- **Simulationsparameter**
  Parameter einer konkreten Kollisions-Engine, welche nicht im Modulinterface enthalten sind, können mittels der Methoden `get/setParameter(...)` gelesen bzw. verändert werden.

```
'object' (Type=ParameterSet)
  'id' = 'ball_07' (Type=string)
  'type' = 'sphere' (Type=string)
  'parameters' (Type=ParameterSet)
    'dimension' (Type=ParameterSet)
      'x' = '0.1' (Type=string)
      'y' = '0.1' (Type=string)
      'z' = '0.1' (Type=string)
    'orientation' (Type=ParameterSet)
      'w' = '1' (Type=double)
      'x' = '0' (Type=double)
      'y' = '0' (Type=double)
      'z' = '0' (Type=double)
    'translation' (Type=ParameterSet)
      'x' = '-0.9' (Type=double)
      'y' = '1.05' (Type=double)
      'z' = '-0.101' (Type=double)
```

Programmcode 7.5: Parameter bei Erzeugung eines Kollisionskörpers

7.8.2.2 `interface collision::IObject`

Dieses Interface repräsentiert einen einzelnen Kollisionskörper:

- **Informationen**
  Jeder Körper besitzt eine eindeutige ID, welche über `getId()` ermittelbar ist, sowie einen Typ (z.B. Quader, Zylinder), den man durch `getType()` auslesen kann. Die Kollisions-Engine, welche diesen Körper verwaltet, kann mit `getEngine()` abgefragt werden. Ist der Körper Bestandteil eines größeren Ganzen (z.B. Rad), so gibt `getParent()` den Körper an, welcher in der Hierarchie eine Stufe höher steht (z.B. Auto).

  Wurde der Körper seit der letzten Überprüfung auf Kollisionen verändert, so erhält man bei Aufruf von `isModified()` `true`.

- **Zustand**
  Die Translation und Orientierung des Körpers lässt sich mit `get/set Translation(...)` bzw. `get/setOrientation(...)` abfragen bzw. festlegen.

- **Simulationsparameter**
  Die Methoden `get/setParameter(...)` haben ähnliche Bedeutungen wie ihre Pendants in `interface` *`collision::IEngine`* (siehe Abschnitt 7.8.2.1 auf Seite 109).

7.8.2.3 `class collision::CollisionList`

Diese Datenstruktur hält alle in einem Durchlauf aufgetretenen Kollisionsdaten. `class collision::CollisionList` ist für sich genommen zunächst nur eine Liste von `class collision::Collision`-Objekten. Diese wiederum informieren über die beiden kollidierten Körper (*`m_Object1, m_Object2`*) und über alle dadurch auftretenden Kollisionspunkte (*`m_Points`*). Ein Kollisionspunkt `class collision:Point` besteht aus der Koordinate, an welcher die Kollision auftrat (*`m_Point`*), aus der Normalen der Oberfläche von Körper 1 an dieser Stelle (*`m_Normal`*) und aus der Entfernung des Kollisionspunktes von der Oberfläche (*`m_fPenetrationDepth`*), also dem „Durchdringungsgrad“.

7.8.3 Konkrete Implementierung

Abbildung 7.12 zeigt die konkrete Implementierung des Modulinterfaces durch ODE.

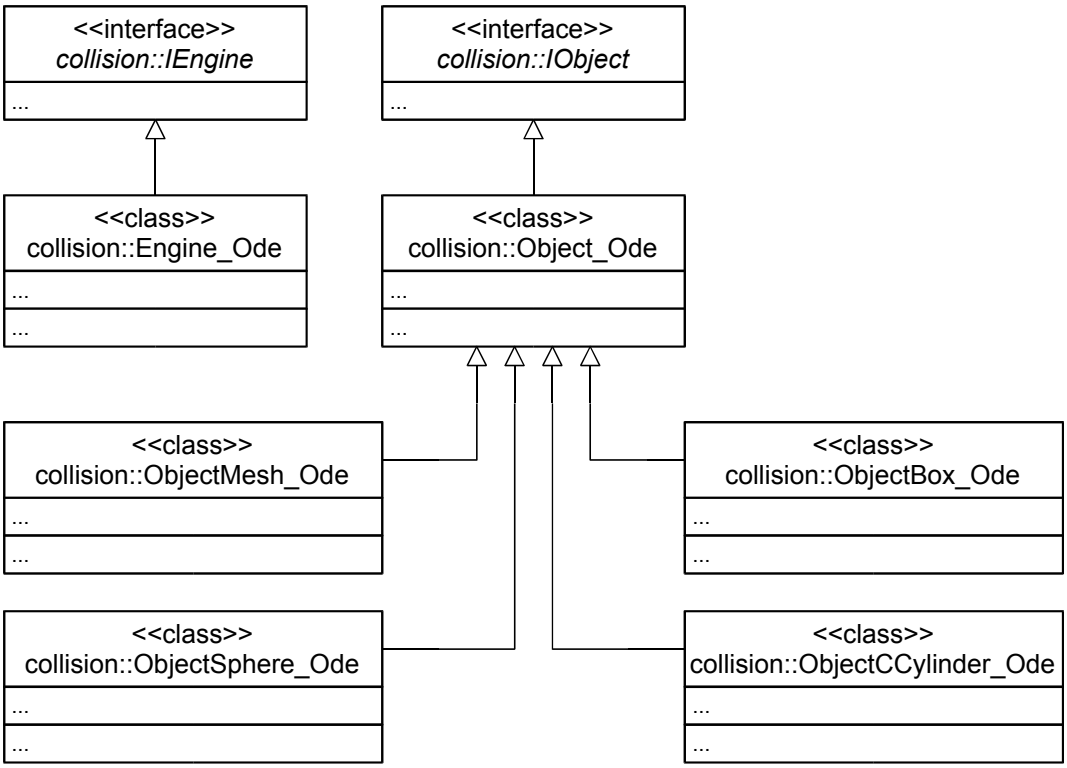


Abbildung 7.12: UML-Diagramm der Klassen der ODE Kollisions-Engine

Die unterschiedlichen Kollisionsobekte werden über die Zwischenklasse `class collision::Object_Ode` abgeleitet. Diese Klasse implementiert `interface` *`collision::IObject`* und ist für Eigenschaften zuständig, welche allen Kollisionsobjekten gemeinsam sind, z.B. `get/setTranslation(...)`, `get/setOrientation(...)`. Die von dieser Klasse abgeleiteten Objekte repräsentieren letzlich die konkreten Formen wie Kugel, Zylinder, abgerundeter Zylinder.

7.9 Render Engine

7.9.1 Vergleich von Render-Engines

Tabelle 7.5 auf der nächsten Seite zeigt die Möglichkeiten von verschiedenen frei erhältlichen Render-Engines, welche in C/C++ implementiert sind.

1. **Irrlicht**[11]
   Open-Source Render-Engine
   © 2003-2005 Nikolaus Gebhardt
   `http://irrlicht.sourceforge.net`

2. **OGRE**[12] (**O**bject-Oriented **G**raphics **R**endering **E**ngine)
   Open-Source Render-Engine
   © 2000-2005 The OGRE team
   `http://www.ogre3d.org`

3. **OSG**[13] (**O**pen**S**cene**G**raph)
   Open-Source Render-Engine
   © 2004 OSG Community
   `http://www.openscenegraph.org`

Zur Implementierung der Render-Engine wurde für diese Thesis auf OGRE zurückgegriffen. Irrlicht schied wegen der fehlenden Unterstützung von Linux aus. OGRE und OSG benötigen zunächst die Installation einiger Unterbibliotheken, bevor der eigentliche Kern kompiliert werden kann. Dies gestaltet sich für OGRE allerdings weniger aufwändig, da sich zumindest unter Debian-Systemen diese Bibliotheken einfach nachinstallieren lassen. Zudem wirkt OGRE im Gegensatz zu OSG mit seinen zahlreichen Modulen vollständiger und durchdachter. Ein Nachteil ist allerdings das spezielle Dateiformat für Modelle und Meshes. Dateien aus Programmen wie Autodesk 3ds Max können nicht direkt verwendet werden, sondern müssen jedesmal durch einen Konverter in das OGRE-Format umgewandelt werden. Dies hat jedoch den Vorteil, dass die Ladezeiten für Objekte kürzer ausfallen und die Render-Engine weniger Speicherplatz für Laderoutinen für die unterstützten Dateiformate benötigt.

[11] Beschreibung: `http://irrlicht.sourceforge.net/features.html`
[12] Beschreibung: `http://www.ogre3d.org/index.php?option=com_content&task=view&id=13&Itemid=62`
[13] Beschreibung: `http://www.openscenegraph.org/index.php?page=Introduction.Introduction`

Render-Engine	Irrlicht	OGRE	OSG
Plattform			
Windows	+	+	+
Linux	-	+	+
MacOS	-	+	+
Treiber			
DirectX	+	+	+
OpenGL	+	+	+
Shader			
Versionen	1.1-3.0	1.1-3.0	1.1-3.0
Cg	+	+	+
DX9 HLSL	+	+	-
GLSL	-	+	+
Modellformate			
3D-Studio (.3ds)	+	+	+
Maya (.obj)	+	+	+
Blender (.blend)	-	+	+
Quake (.bsp/.md2)	+	+	+
Direct-X (.x)	+	-	-
Texturformate			
BMP	+	+	+
GIF	-	-	+
DDS	-	+	+
JPG	+	+	+
PIC	-	-	+
PNG	+	+	+
RGB	-	-	+
TGA	+	+	-
Tiff	-	+	+
Level of detail (LOD)	unbekannt	+	+

Tabelle 7.5: Vergleich von Render-Engines

7.9.2 Interface

Das Modulinterface der Render-Engine bietet grundlegende Funktionen zur Erzeugung von grafischen Objekten, Lichtquellen und Kameras sowie zum Rendern einzelner Frames (siehe Abbildung 7.13 auf der nächsten Seite).

7.9.2.1 `interface render::IEngine`

Dieses Interface bietet Zugriff auf alle wesentlichen Funktionen der Render-Engine:

- **Initialisierung / Deinitialisierung**
  Zu Beginn des Programmablaufs muss die Render-Engine mit `initialize()` initia-

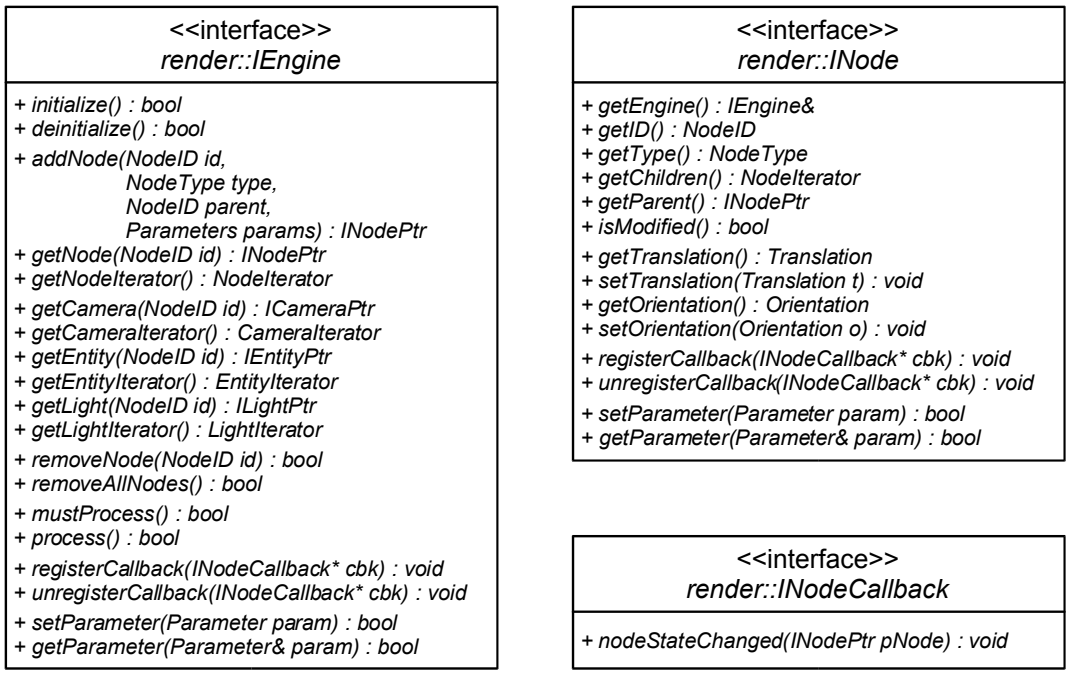


Abbildung 7.13: UML-Diagramm des Modulinterfaces der Render-Engine (Teil 1)

lisiert werden. Dadurch haben konkrete Implementierungen die Möglichkeit, Vorbereitungen zu treffen, die eigentliche Engine zu starten und Ressourcen anzufordern (z.B. Meshes, Texturen).

Nach Abschluss des Programms wird mit `deinitialize()` die Engine wieder heruntergefahren und alle Ressourcen können freigegeben werden.

- **Objekte**
  Mit Hilfe von `addNode(...)` werden grafische Objekte in die Szene eingefügt. Alle zur Konstruktion unabdingbaren Daten werden direkt übergeben. Alle weiteren Parameter, die je nach konkreter Engine spezielle Ausprägungen haben können, werden über die Parameter `params` übergeben (siehe Programmcode 7.6 auf der nächsten Seite).

  Mit der Methode `getNode(...)` kann man gezielt einzelne Objekte abfragen. Mit `getNodeIterator()` erhält man einen Iterator, welcher einen Zugriff auf alle Objekte innerhalb der Szene erlaubt. Die spezialisierten Varianten `getCamera(...)`, `getEntity(...)` und `getLight(...)` bzw. `getCameraIterator()`, `getEntityIterator()` und `getLightIterator()` beschränken sich auf den jeweiligen Objekttyp.

  `removeNode(...)` dient zum gezielten Löschen von Objekten. `removeAllObjects()` löscht alle Objekte innerhalb der Szene, um z.B. eine neue Szene zu laden.

- **Rendering**
  Innerhalb der Hauptschleife des umgebenden Programms kann man mittels `mustProcess()` abfragen, ob die Render-Engine Änderungen an Objekten festgestellt hat und diese darstellen muss. Wenn diese Methode `true` zurückgibt, kann der nächste Frame mit `process()` dargestellt werden.

- **Informationsfluss**
  Über Änderungen an den Objekten kann man mit Hilfe einer Callback-Funktion informiert bleiben. Diese wird mit `registerNodeCallback(...)` angemeldet und mit `unregisterNodeCallback(...)` wieder abgemeldet. Nach jeder Veränderung an Objekten wird die Callback-Funktion mit der betroffenen Node-ID aufgerufen.

- **Parameter**
  Parameter einer konkreten Render-Engine, welche nicht im Modulinterface enthalten sind, können mittels der Methoden `get/setParameter(...)` gelesen bzw. verändert werden. Bei der Render-Engine OGRE kann man so z.B. Methoden zum Schattenwurf, Anti-Aliasing-Einstellungen und Frameraten einstellen.

```
'entity' (Type=ParameterSet)
  'id' = 'ball_07' (Type=string)
  'type' = 'entity_mesh' (Type=string)
  'parameters' (Type=ParameterSet)
    'dimension' (Type=ParameterSet)
      'x' = '0.1' (Type=string)
      'y' = '0.1' (Type=string)
      'z' = '0.1' (Type=string)
    'mesh' (Type=ParameterSet)
      'cast_shadows' = 'true' (Type=string)
      'material' = 'billard_ball_07.material' (Type=string)
      'resource' = 'sphere_hires.mesh' (Type=string)
    'orientation' (Type=ParameterSet)
      'w' = '1' (Type=double)
      'x' = '0' (Type=double)
      'y' = '0' (Type=double)
      'z' = '0' (Type=double)
    'translation' (Type=ParameterSet)
      'x' = '-0.9' (Type=double)
      'y' = '1.05' (Type=double)
      'z' = '-0.101' (Type=double)
```

Programmcode 7.6: Parameter bei Erzeugung eines Renderobjekts

7.9.2.2 `interface render::INode`

Dieses Interface repräsentiert den Grundtypen eines Objektes innerhalb einer grafischen Szene. Von diesem Interface sind die drei Interfaces `interface` *`render::ICamera`*, `interface` *`render::IEntity`* und `interface` *`render::ILight`* abgeleitet (siehe Abbildung 7.14 auf der nächsten Seite).

- **Informationen**
  Jedes Objekt besitzt eine eindeutige ID, welche über `getId()` ermittelbar ist, sowie einen Typ (z.B. Kamera, Licht, Mesh, Partikel-Emitter), den man durch

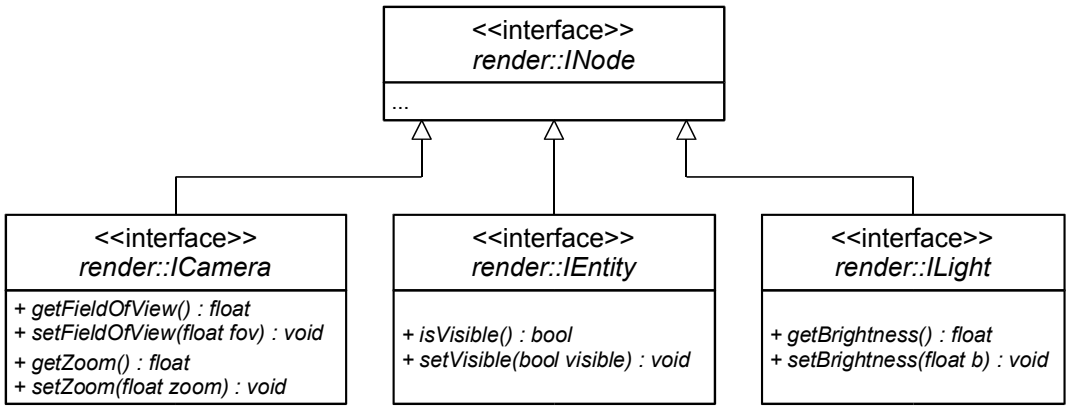


Abbildung 7.14: UML-Diagramm des Modulinterfaces der Render-Engine (Teil 2)

`getType()` auslesen kan. Die Render-Engine, welche dieses Objekt verwaltet, kann mit `getEngine()` abgefragt werden.

Zusätzlich sind die Objekte in einer Baumstruktur organisiert. Jedes Objekt kann Unterobjekte besitzen (`getChildren()`), welche ihre Translation und Orientierung relativ zu diesem Objekt ändern. Ist das Objekt selbst ein Unterobjekt, so hat es über `getParent()` Zugriff auf sein übergeordnetes Objekt.

Wurde das Objekt im letzten Simulationsschritt verändert, so erhält man bei Aufruf von `isModified()` `true`.

- **Zustand**
  Mit den Methoden `get/setTranslation(...)` und `get/setOrientation(...)` hat man Zugriff auf die Translation und Orientierung des Objektes.

- **Informationsfluss / Simulationsparameter**
  Die Methoden `registerNodeCallback(...)`, `unregisterNodeCallback(...)` und `get/setParameter(...)` haben ähnliche Bedeutungen wie ihre Pendants in `interface` *`render::IEngine`* (auf der vorherigen Seite).

7.9.2.3 `interface render::ICamera`

Dieses Interface repräsentiert alle Arten von Kameras, welche mit der konkreten Implementierung einer Render-Engine möglich sind. Das Interface ist von `interface` *`render::INode`* abgeleitet und erbt somit auch dessen Methoden.

- **Informationen**
  Die Kamera besitzt mindestens einen Sichtwinkel und einen Zoomfaktor. Diese Parameter lassen sich über `get/setFieldOfView(...)` und `get/setZoom(...)` beeinflussen.

  Alle weiteren Engine-spezifischen Parameter sind über `get/setParameter(...)` zu erreichen.

7.9.2.4 `interface render::IEntity`

Dieses Interface repräsentiert alle Arten von sichtbaren Objekten, welche mit der konkreten Implementierung einer Render-Engine möglich sind. Dazu gehören z.B. Meshes oder Partikel-Emitter. Das Interface ist von `interface` *`render::INode`* abgeleitet und erbt somit auch dessen Methoden.

- **Informationen**
  Aufgrund der Vielfalt aller möglichen Arten von sichtbaren Objekten definiert das Modulinterface nur die Eigenschaft der Sichtbarkeit. Diese ist über `is/set Visible(...)` zu steuern.

  Alle Objekt-spezifischen Parameter und alle weiteren Engine-spezifischen Parameter sind über `get/setParameter(...)` zu erreichen.

7.9.2.5 `interface render::ILight`

Dieses Interface repräsentiert alle Arten von Lichtquellen, welche mit der konkreten Implementierung einer Render-Engine möglich sind. Das Interface ist von `interface` *`render::INode`* abgeleitet und erbt somit auch dessen Methoden.

- **Informationen**
  Lichtquellen lassen sich ein- und ausschalten bzw. in ihrer Helligkeit verändern. Dazu dienen die Methoden `get/setBrightness(...)`.

  Aufgrund der Vielzahl von möglichen Lichtquellen (z.B. Punkt, Spot, Fläche, gerichtet) wird auf eine weitere Spezialisierung des Modulinterfaces verzichtet. Alle weiteren lichtquellenspezifischen Parameter und auch alle Engine-spezifischen Parameter sind über `get/setParameter(...)` zu erreichen.

7.9.3 Konkrete Implementierung

Abbildung 7.15 auf der nächsten Seite zeigt die konkrete Implementierung des Modulinterfaces durch OGRE.
Ogre ist in der Lage, mehrere Arten von Lichtquellen zu behandeln. Dies zeigt sich in den von `interface` *`render::ILight`* abgeleiteten Klassen. `class render::Light_Ogre` bietet Grundfunktionalitäten, wie z.B. das Verwalten der Helligkeit und der Farbe. Die davon abgeleiteten Klassen implementieren jeweils eine Lichtart:

- `class render::LightPoint_Ogre` repräsentiert ein Punktlicht mit diffusem[14] und spekularem[15] Lichtanteil.

- `class render::LightSpot_Ogre` repräsentiert ein Spotlicht. Dieses bietet die gleichen FUnktionen, wie ein Punktlicht, besitzt aber zusätzlich noch die Richtung, den Öffnungswinkel und die Randschärfe des Lichtkegels.

[14] Die Grundfarbe der Lichtquelle
[15] Die Farbe von Glanzpunkten, welche diese Lichtquelle hervorruft

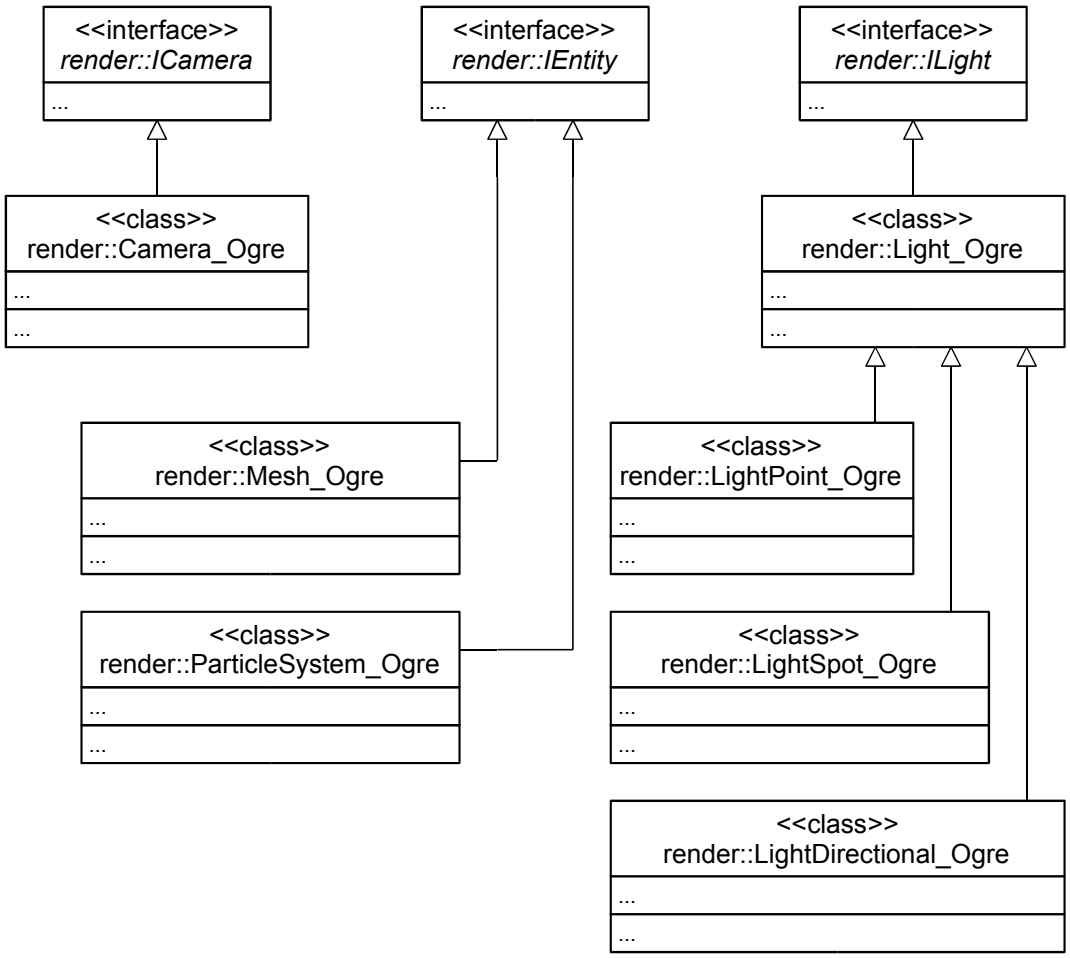


Abbildung 7.15: UML-Diagramm der Klassen der OGRE Render-Engine

- `class render::LightDirectional_Ogre` repräsentiert eine Lichtquelle, welche unendlich weit entfernt ist, aber in eine Richtung strahlt. Mit dieser Lichtquelle lässt sich z.B die Beleuchtung durch die Sonne simulieren.

Das Interface *`render::IEntity`* wird durch folgende Klassen implementiert:

- `class render::Mesh_Ogre` repäsentiert alle grafischen Objekte, welche durch ein Mesh darstellbar sind..
- `class render::ParticleSystem_Ogre` wird bei Partikelsystemen verwendet (z.B. für Bewegungsspuren an Kopf und Fuß der virtuellen Charaktere)

Das Interface *`render::ICamera`* wird durch ein Klasse `render::Camera_Ogre` implementiert. Sie steht für eine „normale“ Kamera im Sinne einer Grafik-Engine mit Position, Orientierung, Öffnungswinkel und Zoom. Zusätzlich lässt sich einstellen, ob die Szene solide, als Drahtmodell oder als Punktmodell dargestellt wird.

Eine weitere Implementierung des Modulinterfaces ist `class render::Engine_Network`. Diese Render-Engine wandelt alle Methodenaufrufe in UDP-Netzwerkpakete um, wel-

che von Netzwerk-Render-Engines entgegengenommen werden können (siehe Abschnitt 7.17 auf Seite 144). Dadurch können mehrere Computer im Netzwerk Szenerien grafisch darstellen. Mit Hilfe dieser Klasse und dem Programm `network_render_engine` wurden für diese Thesis 3D-Projektionsleinwände und HMDs realisiert.

7.10 Controller-Engine

Die Controller-Engine ist dafür zuständig, Eingaben des Benutzers über die zahlreichen Eingabelemente wie Maus, Mausrad, Tastatur, Knöpfe oder GUI-Elemente zu verwalten, zu zentralisieren und weiterzugeben.
Eine konkrete Implementierung einer solchen Engine ist die Klasse `controller::Engine_Ogre`. Diese Engine klinkt sich in die Eingabequeue von OGRE ein und fragt in regelmäßigen Abständen Tastatur und Maus ab. Diese Daten werden aufbereitet und gemäß der Deklarationen des Modulinterfaces weitergegeben.

7.10.1 Interface

Das Modulinterface der Controller-Engine bietet grundlegende Funktionen zur Abfrage von Eingabegeräten (siehe Abbildung 7.16).

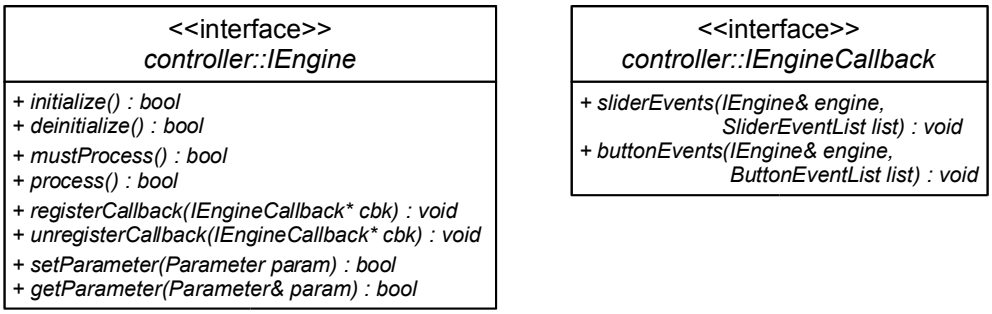


Abbildung 7.16: UML-Diagramm des Modulinterfaces der Controller-Engine

7.10.1.1 `interface controller::IEngine`

Dieses Interface bietet Zugriff auf alle wesentlichen Funktionen der Controller-Engine:

- **Initialisierung / Deinitialisierung**
  Zu Beginn des Programmablaufs muss die Controller-Engine mit `initialize()` initialisiert werden. Dadurch haben konkrete Implementierungen die Möglichkeit, Vorbereitungen zu treffen, die eigentliche Engine zu starten und Ressourcen anzufordern (z.B. Geräte).

  Nach Abschluss des Programms wird mit `deinitialize()` die Engine wieder heruntergefahren und alle Ressourcen können freigegeben werden.

- **Abfrage**
  Innerhalb der Hauptschleife des umgebenden Programms kann man mittels `mustProcess()` abfragen, ob die Controller-Engine bereit für weitere Abfragen der Eingabegeräte ist. Wenn diese Methode `true` zurückgibt, kann diese Abfrage mit `process()` gestartet werden.

- **Informationsfluss**
  Über Eingaben des Benutzers kann man mit Hilfe einer Callback-Funktion informiert bleiben. Diese wird mit `registerNodeCallback(...)` angemeldet und mit `unregisterNodeCallback(...)` wieder abgemeldet. Nach jeder Eingabe wird die Callback-Funktion mit einer Liste der betätigten Eingabeelemente (z.B. Mausrad, Taste) und eventueller Optionen (z.B. SHIFT-Taste gedrückt) aufgerufen.

- **Parameter**
  Parameter einer konkreten Controller-Engine, welche nicht im Modulinterface enthalten sind, können mittels der Methoden `get/setParameter(...)` gelesen bzw. verändert werden (z.B. Abfrageintervall).

Wurden Eingabeelemente wie Tasten oder Schalter betätigt, so erhalten die Callbacks ein `ButtonEventList`-Objekt. Bei Eingaben über kontinuierliche Eingabeelemente wie Schieberegler oder Mausrad erfolgt die Übergabe eines `SliderEventList`-Objekts (siehe Programmcode 7.7).

```
enum EButton
{
  KEY_ENTER = 13,
  KEY_ESCAPE = 27,
  KEY_SPACE = 32,

  KEY_0 = 48, KEY_0, KEY_1, KEY_2, KEY_3, KEY_4,
              KEY_5, KEY_6, KEY_7, KEY_8, KEY_9,

  KEY_A = 65, KEY_B, KEY_C, KEY_D, KEY_E, KEY_F, KEY_G, KEY_H, KEY_I,
  KEY_J, KEY_K, KEY_L, KEY_M, KEY_N, KEY_O, KEY_P, KEY_Q, KEY_R,
  KEY_S, KEY_T, KEY_U, KEY_V, KEY_W, KEY_X, KEY_Y, KEY_Z,

  KEY_UP = 256, KEY_DOWN, KEY_LEFT, KEY_RIGHT,
  KEY_INSERT, KEY_DELETE, KEY_PGUP, KEY_PGDOWN, KEY_HOME, KEY_END,

  KEY_NUMPAD0, KEY_NUMPAD1, KEY_NUMPAD2, KEY_NUMPAD3, KEY_NUMPAD4,
  KEY_NUMPAD5, KEY_NUMPAD6, KEY_NUMPAD7, KEY_NUMPAD8, KEY_NUMPAD9,

  KEY_F1, KEY_F2,  KEY_F3,  KEY_F4,  KEY_F5,  KEY_F6,  KEY_F7,  KEY_F8,
  KEY_F9, KEY_F10, KEY_F11, KEY_F12, KEY_F13, KEY_F14, KEY_F15,

  KEY_PAUSE, KEY_PRINTSCREEN,

  KEY_SHIFT_L,   KEY_SHIFT_R,
  KEY_CONTROL_L, KEY_CONTROL_R,
  KEY_ALT_L,     KEY_ALT_R,
  KEY_WIN_L,     KEY_WIN_R,

  KEY_BUTTON1, KEY_BUTTON2, KEY_BUTTON3
};

enum EModifier
{
  MODIFIER_SHIFT_L   = 0x00000001L,
  MODIFIER_SHIFT_R   = 0x00000002L,
  MODIFIER_SHIFT     = 0x00000003L,
  MODIFIER_CONTROL_L = 0x00000004L,
  MODIFIER_CONTROL_R = 0x00000008L,
  MODIFIER_CONTROL   = 0x0000000CL,
  MODIFIER_ALT_L     = 0x00000010L,
  MODIFIER_ALT_R     = 0x00000020L,
  MODIFIER_ALT       = 0x00000030L,
  MODIFIER_BUTTON1   = 0x00000100L,
  MODIFIER_BUTTON2   = 0x00000200L,
  MODIFIER_BUTTON3   = 0x00000400L
};
```

```
typedef enum EButtonState       ///< Button states
{
  PRESSED, DOWN, RELEASED, UP
};

typedef struct _SButtonEvent
{
  EButton       eButton;        ///< Button code
  char          cChar;          ///< Key char (if applicable, else 0)
  EButtonState  eState;         ///< Key state
  long          lModifier;      ///< Modifier bitmask (MODIFIER_...)
  long          lQueryInterval; ///< Keypress sampling interval in ms
} SButtonEvent;

typedef enum ESliderType        ///< Slider type
{
  MOUSE_X, MOUSE_Y,             ///< Mouse movement
  MOUSE_WHEEL                   ///< Mouse wheel change
};

typedef struct _SSliderEvent
{
  ESliderType   eType;          ///< Slider type
  float         fPosition;      ///< Absolute position
  float         fChange;        ///< Relative position
  long          lModifier;      ///< Modifier bitmask (MODIFIER_...)
  long          lQueryInterval; ///< Slider sampling interval in ms
} SSliderEvent;

typedef list<SButtonEvent>  ButtonEventList;  ///< List of button events
typedef list<SSliderEvent>  SliderEventList;  ///< List of slider events
```

Programmcode 7.7: Auszug aus `IControllerEngine.h`

Jede Liste enthält Einzelevents, in denen genauere Informationen aufgeführt sind:

- Welche Taste bzw. welcher Regler wurde betätigt? Welchen Zeichen entspricht dies auf einer Tastatur?
- Ist die Taste losgelassen oder gedrückt worden?
- Welche Änderung hat ein kontinuierlicher Regler erfahren? Wie ist seine momentane Position?
- Welche „Modifier“ sind parallel betätigt worden (z.B. SHIFT-, ALT-, CONTROL-Taste)?
- Welches Abtastintervall liegt vor?

7.11 Neuronales Netzwerk

7.11.1 Vergleich von neuronalen Netzwerken

Für die Implementierung des neuronalen Netzes wurden mehrere frei erhältliche Bibliotheken auf Funktionsumfang, Programmiersprache, Integrationsfähigkeit, Portabilität etc. untersucht. Zunächst fiel die Wahl auf den **S**tuttgarter **N**euronaler **N**etzwerk-**S**imulator (**SNNS**[16]). Der SNNS ist zwar relativ alt, dafür aber einer der Simulatoren mit dem umfangreichsten Angebot an Netzwerktopologien und Lernverfahren. Die Quellcodes liegen in C vor, sind unter Windows und Linux kompilierbar und ließen sich relativ leicht in das Hauptprogramm integrieren. Andere Bibliotheken fielen in mindestens einem der oben genannten Kriterien durch. Entweder war der Funktionsumfang zu gering (z.B. nur Feedforward-Netze) oder zu groß (komplette biochemische Simulation eines Netzwerks). Bibliotheken mit ausreichendem Umfang waren zu stark spezialisiert auf Microsoft Windows oder in einer anderen Sprache geschrieben.
Erst bei der Implementierung der CPG's stellte sich heraus, dass der SNNS keine zeitkontinuierliche Simulation erlaubt. Die Erweiterung um dieses Feature hätte zu tiefe Eingriffe in die Quellcodes erfordert, die darüber hinaus schlecht organisiert und dokumentiert waren.

Als Ausweg wurde kurzerhand eine eigene Bibliothek geschrieben, die **CTRNN**. des Simulationsprogramms abgestimmt und bietet deshalb z.B. keine Lernmethoden.
Dafür wurde zur Vereinheitlichung der Dateiformate[17] eine XML-basierte Netzwerkbeschreibung entwickelt. Um die Bibliothek leicht erweitern zu können, wurde Wert auf objektorientiertes Design gelegt. So lassen sich z.B. Aktivierungsfunktionen durch Ableitung von `class ActivationFunction_CTRNN` einfach hinzufügen und verwenden.

7.11.2 Interface

Das Modulinterface umfasst zwei Hauptklassen: Das neuronale Netzwerk `interface` *`neural::INetwork`* und die Neuronen `interface` *`neural::INeuron`* (siehe Abbildung 7.17 auf der nächsten Seite).

7.11.2.1 `interface neural::INetwork`

Dieses Interface bietet Zugriff auf alle wesentlichen Funktionen eines neuronalen Netzwerks:

- **Initialisierung / Deinitialisierung**
  Zu Beginn des Programmablaufs muss das neuronale Netzwerk mit `initialize()` initialisiert werden. Dadurch haben konkrete Implementierungen die Möglichkeit, Vorbereitungen zu treffen und Ressourcen anzufordern.

[16] ©1990-1995 University of Stuttgart, Institute for Parallel and Distributed High Performance Systems, `http://www.informatik.uni-stuttgart.de/ipvr/bv/projekte/snns/snns.html`
[17] Das Dateiformat des SNNS ist proprietär und stark fehleranfällig.

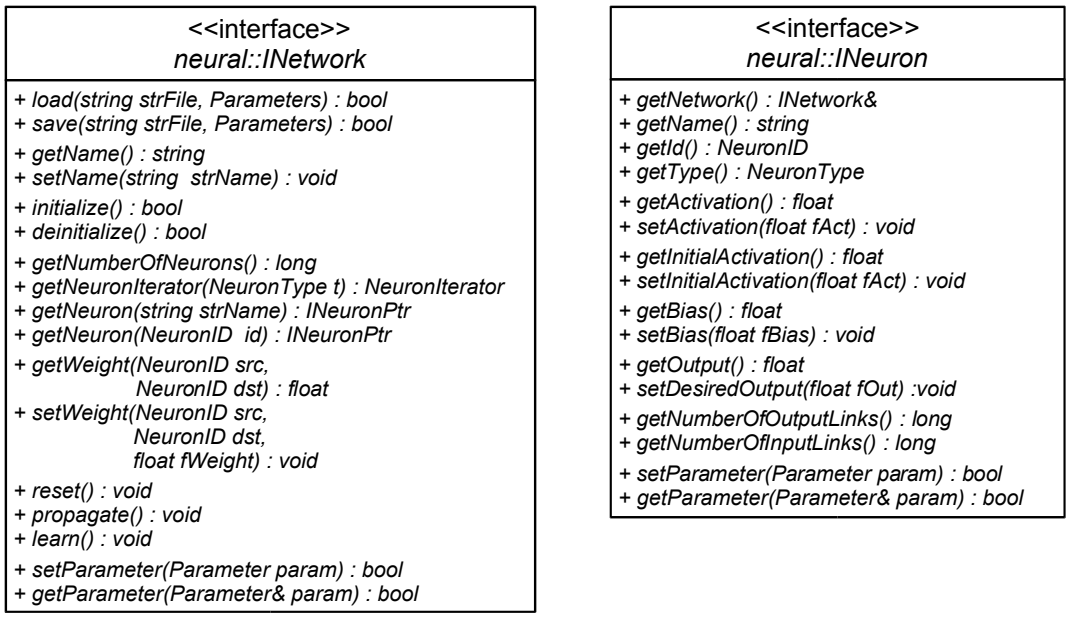


Abbildung 7.17: UML-Diagramm des Modulinterfaces des neuronalen Netzwerks

Nach Abschluss des Programms wird mit `deinitialize()` das neuronale Netzwerk angewiesen, abschließende Maßnahmen auszuführen und alle Ressourcen wieder freizugeben.

- **Laden / Speichern**
  Mit Hilfe von `load(...)` bzw. `save(...)` können neuronale Netzwerke aus Dateien geladen bzw. in Dateien geschrieben werden.

- **Neuronen**
  Die Gesamtzahl der Neuronen innerhalb des Netzwerks kann mit `getNumberOfNeurons()` abgefragt werden.

  Um Zugriff auf alle Neuronen zu haben, kann man sich mit `getNeuronIterator()` einen Iterator zurückgeben lassen. Dieser Iterator bietet auch die Möglichkeit, nur spezielle Neuronen abzufragen, wie Eingabe- oder Ausgabeneuronen. `getNeuron(...)` erlaubt in seinen beiden Ausführungen die Suche nach Neuronen anhand der ID oder des Namens.

- **Verbindungen**
  `get/setWeight(...)` fragt das Gewicht der Verbindung zwischen zwei Neuronen ab bzw. legt dieses fest. Wird ein vorhandenes Gewicht mit dem Wert `NO_LINK_WEIGHT` ($= -1^{10}$) belegt, so wird die Verbindung gelöscht. Wird das Gewicht einer bis dahin nicht vorhandenen Verbindung mit einem Wert ungleich `NO_LINK_WEIGHT` belegt, so wird diese Verbindung erzeugt.

- **Simulationsprozess**
  Mit `propagate()` wird ein Simulationsschritt des neuronalen Netzes ausgelöst.

`reset()` setzt die Neuronen auf ihre Start-Eingangswerte. `learn()` führt einen Schritt des eingestellten Lernverfahrens auf den gewünschten Ausgabemustern aus.

- **Simulationsparameter**
  Parameter eines konkreten neuronalen Netzwerks, welche nicht im Modulinterface enthalten sind, können mittels der Methoden `get/setParameter(...)` gelesen bzw. verändert werden. Bei CTRNN wird so z.B. die zeitliche Schrittweite eines Simulationsschritts eingestellt.

Sinnvollerweise sollten noch die Funktionen `addNeuron(...)` und `removeNeuron(...)` ergänzt werden. Diese waren für die Durchführung dieser Thesis nicht notwendig, da immer nur Netzwerkdateien geladen wurden. Für ein vollständiges Modulinterface wären sie jedoch unabdingbar.

7.11.2.2 `interface neural::INeuron`

Dieses Interface repräsentiert ein einzelnes Neuron:

- **Informationen**
  Jedes Neuron besitzt eine eindeutige ID (`getId()`), einen Namen (`getName()`) sowie einen Typ (z.B. Eingabe, Ausgabe, Verborgen, Dual), den man durch `getType()` auslesen kann. Das neuronale Netzwerk, welches diese Neuronen verwaltet, kann mit `getNetwork()` abgefragt werden.

  Die Anzahl der eingehenden und ausgehenden Verbindungen lässt sich mit `getNumberOfInputLinks()` bzw. `getNumberOfOutputLinks()` abfragen.

- **Zustand**
  Die aktuelle Aktivierung und die Aktivierung nach einem Reset lassen sich mit `get/setActivation(...)` bzw. `get/setInitialActivation(...)` verwalten. Ebenso hat man mit `get/setBias(...)` Zugriff auf den Schwellwert des Neurons.

  `getOutput(...)` liefert den aktuellen Ausgabezustand des Neurons.

  Für den Lernvorgang wird mit `setDesiredOutput(...)` der gewünschte Ausgangswert festgelegt.

- **Simulationsparameter**
  Die Methoden `get/setParameter(...)` haben ähnliche Bedeutungen wie ihre Pendants in `interface` *`neural::INetwork`* (auf dieser Seite).

7.11.3 Konkrete Implementierung

Abbildung 7.18 zeigt die konkrete Implementierung des Modulinterfaces durch CTRNN.

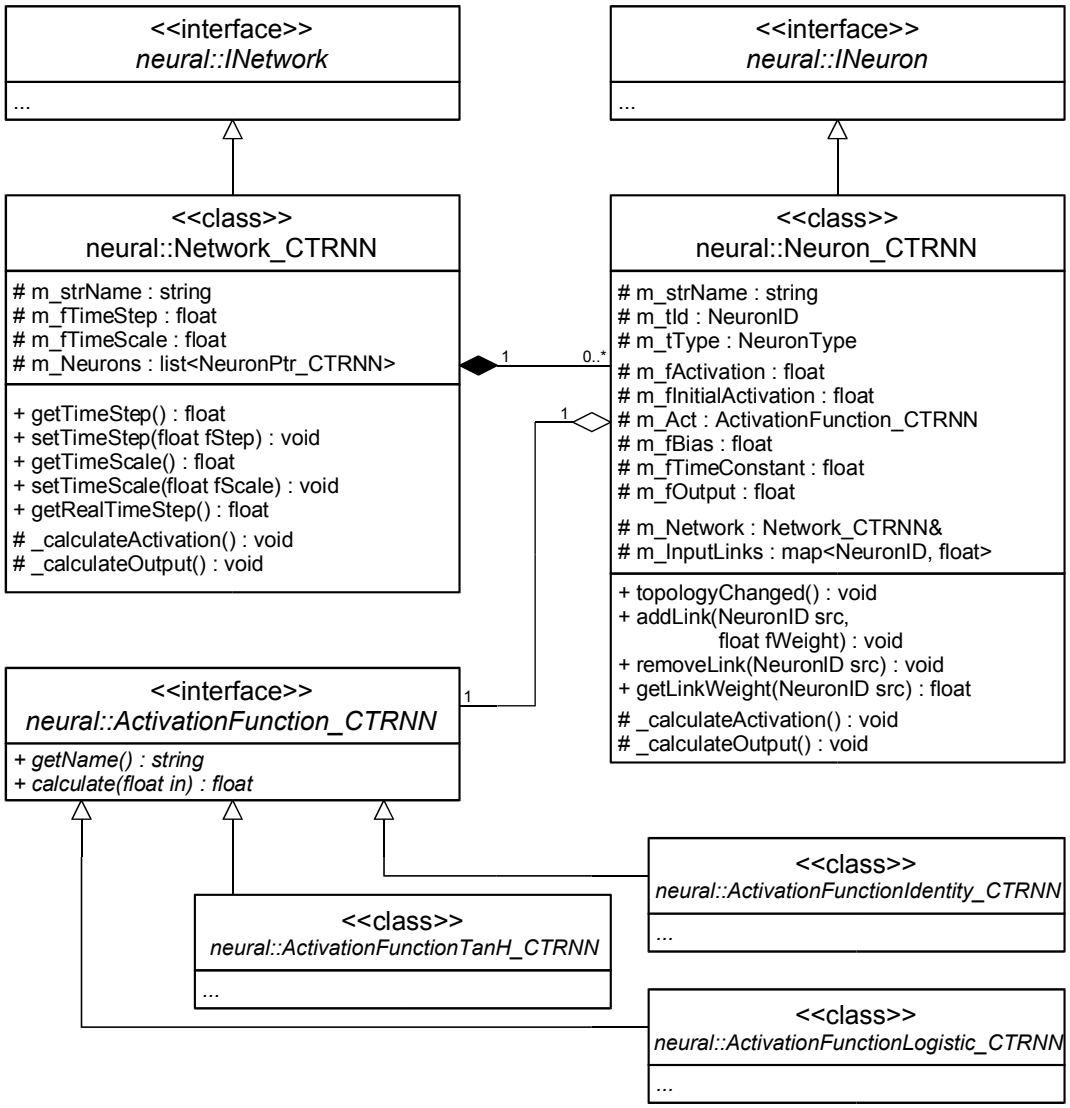


Abbildung 7.18: UML-Diagramm der Klassen des CTRNN

`class neural::Network_CTRNN` implementiert das Modulinterface und verwaltet in einer Liste *`m_Neurons`* die Neuronen vom Objekttyp `class neural::Neuron_CTRNN`.
Die Neuronen verwenden zur Berechnung des Ausgangssignals eine Instanz der von `class neural::ActivationFunction_CTRNN`. abgeleiteten Klassen für Aktivierungsfunktionen (siehe Abschnitt 4.4.1.2 auf Seite 45). Bisher sind die Identitätsfunktion ($\text{act}(x) = x$), die Logistische Funktion ($\text{act}(x) = \frac{1}{1+e^{-x}}$) und die tanh-Funktion implementiert. Durch weitere abgeleitete Klassen sind weitere Aktivierungsfunktionen leicht zu ergänzen.

7.12 Evolutions-Engine

7.12.1 Vergleich von Evolutions-Engines

Als einzige Evolutions-Engine wurde ENZO (**E**volutionärer **N**et**z**werk-**O**ptimierer) untersucht. Aus der näheren Betrachtung dieser Software wurden wichtige Erkenntnisse zum objektorientierten Design einer Evolutions-Engine gewonnen. Da ENZO allerdings erstens komplett in C geschrieben und zweitens sehr speziell auf den SNNS abgestimmt ist, welcher für die eigentliche Simulationsaufgabe nicht geeignet war (siehe Abschnitt 7.11.1 auf Seite 124), wurden keine weiteren Betrachtungen durchgeführt. Die Evolutions-Engine wurde danach komplett eigenständig entworfen und implementiert.

7.12.2 Interface

Das Modulinterface umfasst mehrere Klassen welche teilweise abstrakt sind und für eine konkrete Anwendung abgeleitet werden müssen (siehe Abbildung 7.19 auf der nächsten Seite).

7.12.2.1 `class evolution::Engine`

Diese Klasse verwaltet mehrere Evolutionsprozesse und arbeitet diese ab.

- **Prozessverwaltung**
  Evolutionsprozesse lassen sich mit `addProcess(...)` an der Evolutions-Engine anmelden. Die Gesamtzahl der Prozesse lässt sich mit `getProcessCount()` abfragen. Mit `getProcess(...)` erhält man Zugriff auf einzelne Evolutionsprozesse.

- **Evolutionsprozess**
  Wenn mindestens einer der verwalteten Evolutionsprozesse noch nicht abgearbeitet wurde, liefert `mustProcess()` `true` zurück. Ein Arbeitsschritt wird mit `process()` ausgelöst.

7.12.2.2 `class evolution::Process`

Diese abstrakte Basisklasse dient als Grundlage zur Implementierung konkreter Evolutionsprozesse.

- **Informationen**
  Evolutionsprozesse sind immer mit einen Kurznamen (`getName()`, z.B. „process01“) und einen ausführlichen Namen (`getFullName()`, z.B. „Biped01, Prozess 1 (Generation 5)“) gekennzeichnet. Der aktuelle Zustand und die aktuelle Generation lassen sich mit `getState()`, `getStateName()` und `getGeneration()` auslesen. Über den aktuellen Fortschritt des Prozesses innerhalb des aktuellen Zustands informiert die Methode `getProgress()`. Die aktuelle Gesamtfitness des Prozesses bzw. der Population(en) kann man mit *`getFitness()`* abfragen.

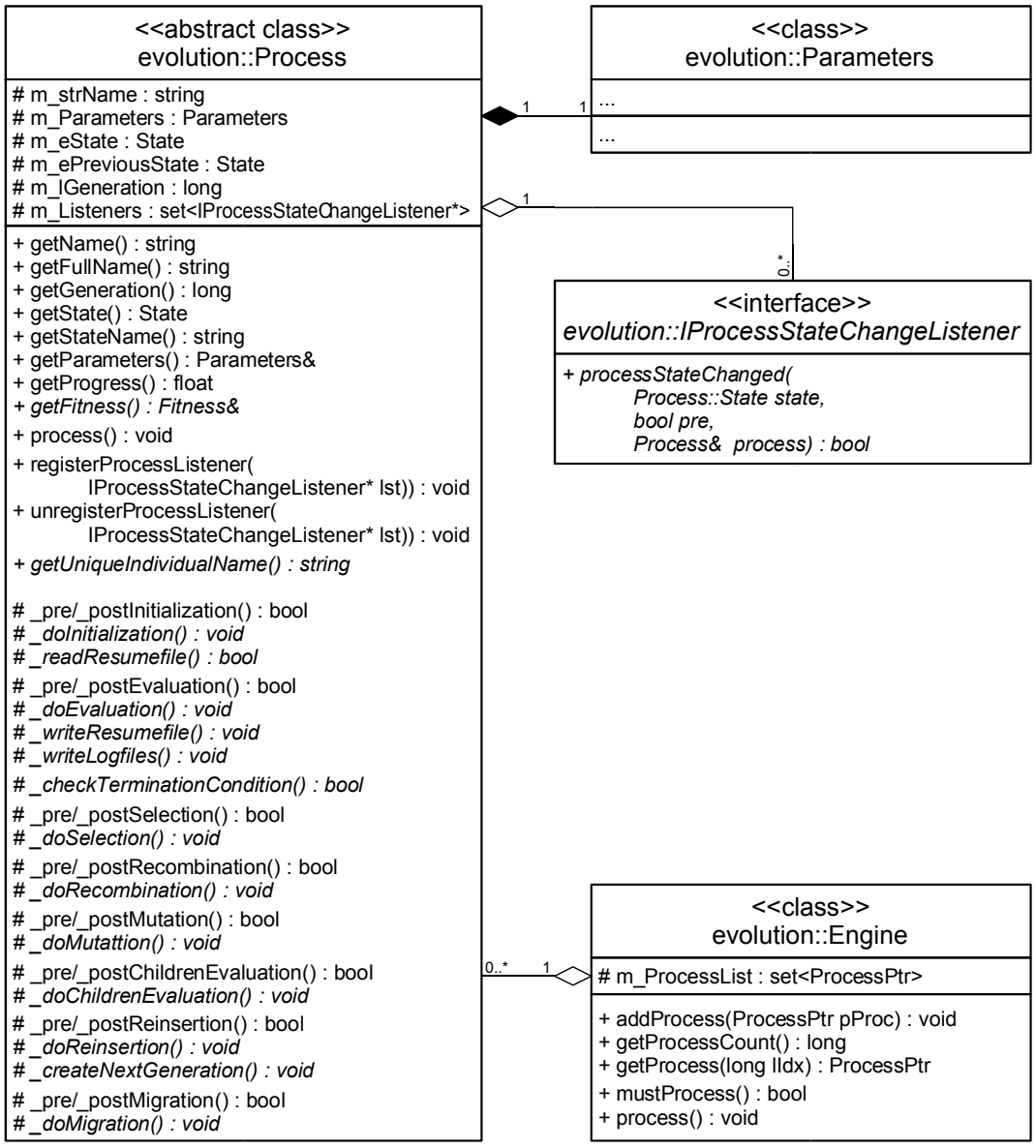


Abbildung 7.19: UML-Diagramm des Modulinterfaces der Evolutions-Engine (Teil 1)

Betriebsparameter für den Prozess sind über `getParameters()` zu lesen und zu verändern.

- **Ausführung**
  Mit der Methode `process` wird ein weiterer Schritt innerhalb des Prozesses angestoßen.

- **Informationsfluss**
  Um über den aktuellen Zustand des Evolutionsprozesses informiert zu bleiben, kann man mit `register`/`unregisterProcessStateChangeListener(...)` Objekte von Typ `interface` *`evolution::IProcessStateChangeListener`* anmelden, welche über Änderungen im Prozesszustand informiert werden.

- **Hilfsmethoden**
  Die Methode *`getUniqueIndividualName()`* liefert einen eindeutigen Namen für neu erzeugte Individuen.

- **Geschützte und abstrakte Methoden**
  Für die einzelnen Schritte des Evolutionsprozesses sind virtuelle Methoden *`_doXyz()`* vorgesehen, welche von einer konkreten Klasse implementiert werden müssen.

  Vor und nach einem solchen Schritt werden die internen Methoden `_pre`/`_postXyz()` aufgerufen. Sie rufen einerseits die registrierten Listener mit Informationen über den aktuellen Zustand und Fortschritt auf und bieten andererseits abgeleiteten Klassen Eingriffsmöglichkeiten z.B. für Plausibilitätsprüfungen. Liefert eine solche Methode `false` zurück, so bricht der Prozess umgehend ab.

7.12.2.3 `class evolution::Process_Single`

Diese von `class evolution::Process` abgeleitete Klasse implementiert einen Evolutionsprozess über eine Population[18] (siehe Abbildung 7.20 auf der nächsten Seite). Die Klasse enthält vier Member:

- *`m_pPopulation`*: Die aktuelle Population

- *`m_pParents`*: Die Eltern, welche durch den Selektionsoperator ausgewählt werden. Dabei können je nach Fitness Eltern mehrfach vorkommen.

- *`m_pChildren`*: Die Nachkommen der Eltern.

- *`m_pNextPopulation`*: Die Population, in welche Kinder und Eltern durch den Wiedereinfügeoperator eingesetzt werden. Nach diesem Vorgang erhöht sich die Anzahl der Generationen um eins und *`m_pNextPopulation`* wird zu *`m_pPopulation`*.

[18] Eine weitere Klasse `class evolution::Process_Multiple` ist zuständig für einen Evolutionsprozess über mehrere Populationen. Da eine solche Variante innerhalb dieser Thesis aber nicht verwendet wurde, ist diese Klasse nur in Ansätzen implementiert.

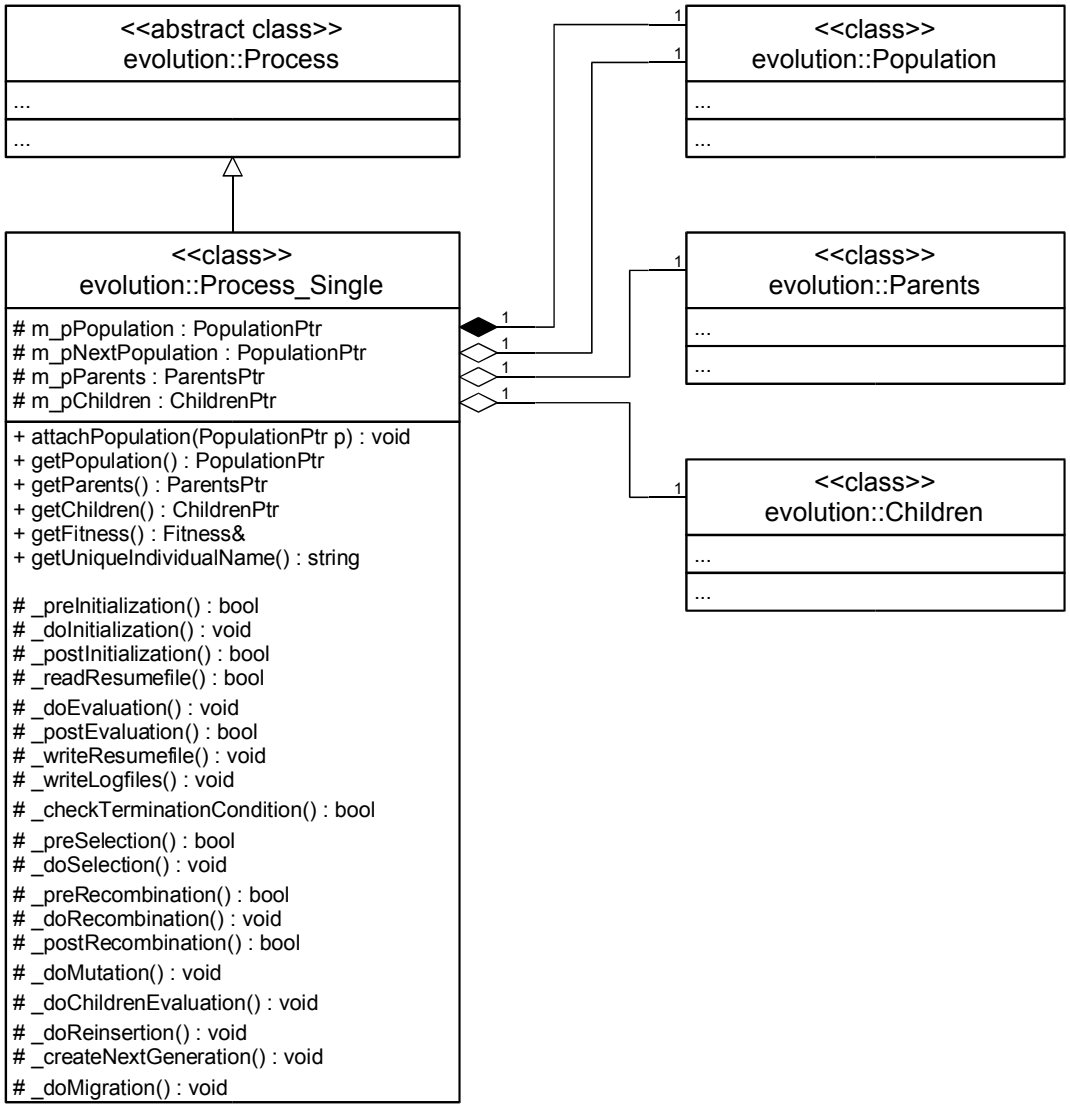


Abbildung 7.20: UML-Diagramm des Modulinterfaces der Evolutions-Engine (Teil 2)

7.12.2.4 `class evolution::Individual`

Das kleinste Element einer Population ist das Individuum, repräsentiert durch diese Klasse (siehe Abbildung 7.21).

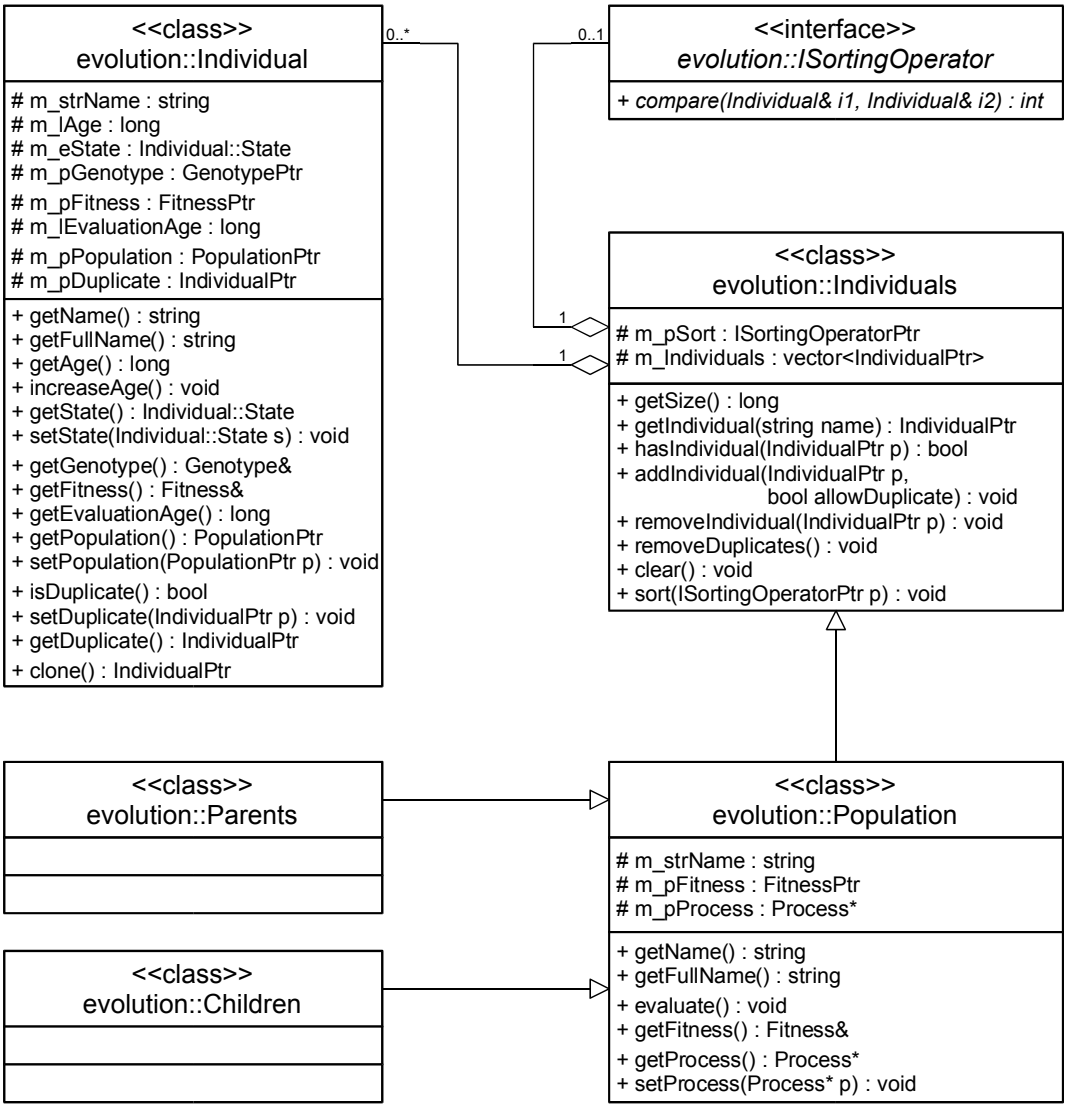


Abbildung 7.21: UML-Diagramm des Modulinterfaces der Evolutions-Engine (Teil 3)

- **Informationen**
  Ein Individuum besitzt immer einen Namen in Kurzform (`getName()`, z.B. „Idv001/024“ und in Langform (`getFullName()`, z.B. „Population 001 / Individual 001/024“). Das Alter lässt sich mit `getAge()` abfragen und mit `increaseAge()` um eins erhöhen.

  Ein Individuum besitzt eine Repräsentation seines Genotyps, auf welche man mit `getGenotype()` zugreifen kann. Die Fitness ist über `getFitness()` zugänglich.

Ob und welcher Population das Individuum zugeordnet ist, kann man mit `get/set Population(...)` abfragen bzw. festlegen. Existieren mehrere gleiche Individuen innerhalb einer Population (z.B. nach der Selektion innerhalb der Elternpopulation), so erhalten die Kopien mit `get/setDuplicate` einen Pointer auf das ursprüngliche Individuum. Die Kopien sind durch `true` als Rückgabewert von `isDuplicate()` zu erkennen.

- **Zustand**
  `get/setState(...)` fragen den Zustand ab bzw. legen diesen fest. Das Individuum kann in einem von sechs Zuständen sein:

  1. `UNINITIALIZED`: Es wurden noch keine Aktionen auf dem Individuum ausgeführt. Der Genotyp und die Fitness ist noch nicht definiert.
  2. `INITIALIZING`: Das Individuum wird gerade initialisiert.
  3. `INITIALIZED`: Das Individuum ist initialisiert, der Genotyp in einem definierten Zustand.
  4. `EVALUATING`: Das Individuum wird gerade evaluiert.
  5. `EVALUATED`: Die Evaluierung ist abgeschlossen, die Fitness enthält einen gültigen Wert.
  6. `ERROR`: Ein Fehler ist aufgetreten.

- **Aktionen**
  `clone()` fertigt eine Kopie dieses Individuums mit seinem Zustand, Genotyp, Fitness und allen anderen Informationen an.

7.12.2.5 `class evolution::Individuals`

Eine Ansammlung von Individuen wird durch diese Klasse repräsentiert (siehe Abbildung 7.21 auf der vorherigen Seite).

- **Individuen**
  Die Anzahl der Individuen ist durch `getSize()` zu ermitteln. Individuen werden mit `addIndividual(...)` hinzugefügt, wobei mit dem Flag `allowDuplicate=true` auch Duplikate erlaubt sind (z.B. bei der Selektion). Mit `hasIndividual(...)` kann man die Existenz eines bestimmten Individuums testen, mit `getIndividual(...)` auslesen und mit `removeIndividual(...)` wieder entfernen.

  `clearDuplicates()` entfernt alle Duplikate innerhalb der Gruppe und `clear()` alle Individuen insgesamt.

- **Aktionen**
  `sort(...)` sortiert die Individuen in der Gruppe nach einem Kriterium, welches durch den Operator angegeben wird. Dieses Kriterium kann z.B. die Fitness oder der Name eines Individuums sein.

7.12.2.6 `class evolution::Population`

Die Klasse `evolution::Population` (siehe Abbildung 7.21 auf Seite 132) ist eine Erweiterung von `evolution::Individuals` und bietet zusätzliche Methoden, um die Fitness der Population als Mittelwert aller Fitnesswerte der Individuen zu berechnen (`evaluate()` und `getFitness()`). Zusätzlich trägt eine Population einen Namen und einen Kurznamen (`getName()` und `getFullName()`) und kann einem Evolutionsprozess zugeordnet sein (`get/setProcess(...)`).

7.12.2.7 `class evolution::Parents` und `class evolution::Children`

Diese beiden Klassen sind direkt von `class evolution::Population` abgeleitet und erweitern diese um keinerlei Methoden oder Member (siehe Abbildung 7.21 auf Seite 132). Ihr Sinn liegt lediglich in der klaren Trennung der Bedeutungen innerhalb eines Evolutionsprozesses. Durch die verschiedenen Klassen sind Verwechslungen schon durch Warnungen des Compilers ausgeschlossen.

7.12.2.8 `class evolution::Fitness`

Die abstrakte Basisklasse `evolution::Fitness` bietet Methoden, um einfache Rechenoperationen (`add(...)`, `subtract(...)`, `multiply(...)` und `getSum()`) auszuführen (siehe Abbildung 7.22). Eine Fitness kann ungültig sein, z.B. vor der Evaluierung eines Individu-

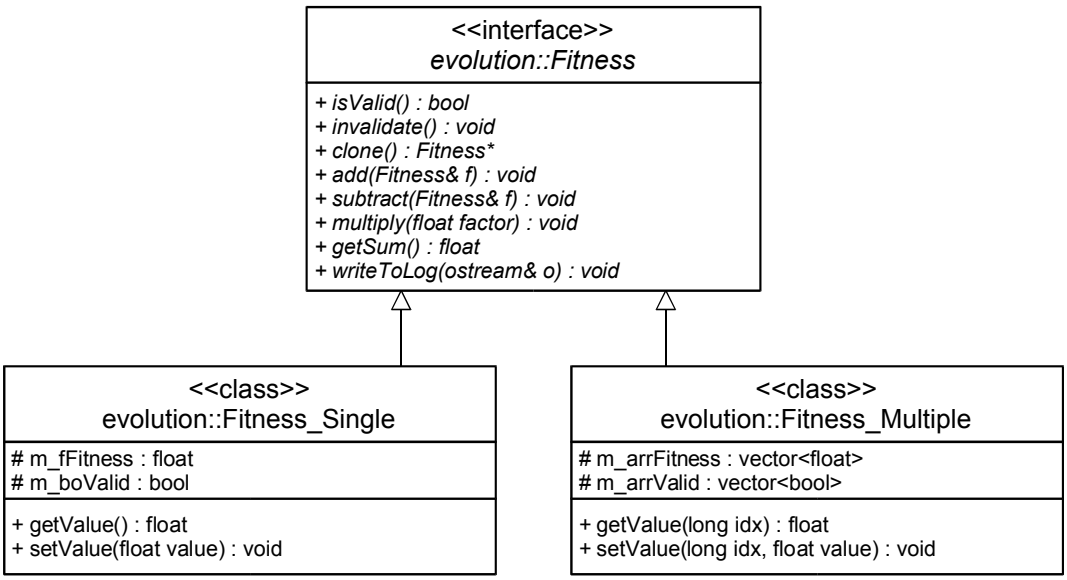


Abbildung 7.22: UML-Diagramm des Modulinterfaces der Evolutions-Engine (Teil 4)

ums. In diesem Fall gibt `isValid()` `false` zurück. Sobald der Fitness ein Wert zugewiesen wird (z.B. mit `Fitness_Single::setValue(...)`), so wechselt die Gültigkeit auf `true`. Mit `invalidate` kann man diesen Vorgang wieder rückgängig machen.

Die konkreten Klassen `evolution::Fitness_Single` und `evolution::Fitness_Multiple` implementieren die Methoden der Basisklasse auf einem einfachen Fitnesswert in Form einer Fließkommazahl bzw. auf mehreren Werten in Form eines Fließkomma-Arrays.

7.12.2.9 `class evolution::Parameters`

Diese Klasse enthält alle für einen Evolutionsprozess wichtigen Parameter. Die meisten davon sind Pointer auf Interfaces für spezielle Funktionen (siehe Abbildung 7.23 auf Seite 136).

- **Objekterzeugung** (`get/setObjectFactory(...)`)
  `interface` *`evolution::IObjectFactory`* erzeugt anwendungsspezifische Individuen und Populationen.

- **Initialisierung** (`get/setInitializationOperator(...)`)
  `interface` *`evolution::IInitializationOperator`* initialisiert ein Individuum. Darunter fällt das Initialisieren des Genotyps und das Invalidieren der Fitness.

- **Vergleich** (`get/setFitnessComparisonOperator(...)`)
  `interface` *`evolution::IFitnessComparisonOperator`* vergleicht zwei Fitnesswerte miteinander. Ist Fitness `f1` kleiner als `f2`, so gibt der Operator -1 zurück, bei Gleichheit 0 und sonst $+1$.

- **Auswertung** (`get/setEvaluationOperator(...)`)
  `interface` *`evolution::IEvaluationOperator`* evaluiert ein Individuum. Dies kann sowohl durch eine einfache Formel geschehen als auch durch einen Simulationsprozess wie `cerebellum`.

- **Terminierung** (`get/setTerminationOperator(...)`)
  `interface` *`evolution::ITerminationOperator`* überprüft, ob das Kriterium zur Beendigung der Evolution erreicht ist und gibt in diesem Fall `true` zurück.

  Eine Klasse, die von diesem Interface abgeleitet ist, ist `class Termination Operator_RunningMean`, welche die Prinzipien des „Laufendes Mittel“-Algorithmus' implementiert (siehe Abschnitt 5.2.11.3 auf Seite 71).

- **Selektion** (`get/setSelectionOperator(...)`)
  `interface` *`evolution::ISelectionOperator`* selektiert Individuen aus einer Population in die Elternpopulation.

  Ein Operator, der von diesem Interface abgeleitet ist, ist `class Selection Operator_Tournament`, bei welchem nach den Prinzipien der Selektion durch Wettkampf die Eltern der Nachkommen für die nächste Generation ausgewählt werden (siehe Abschnitt 5.2.6.3 auf Seite 65).

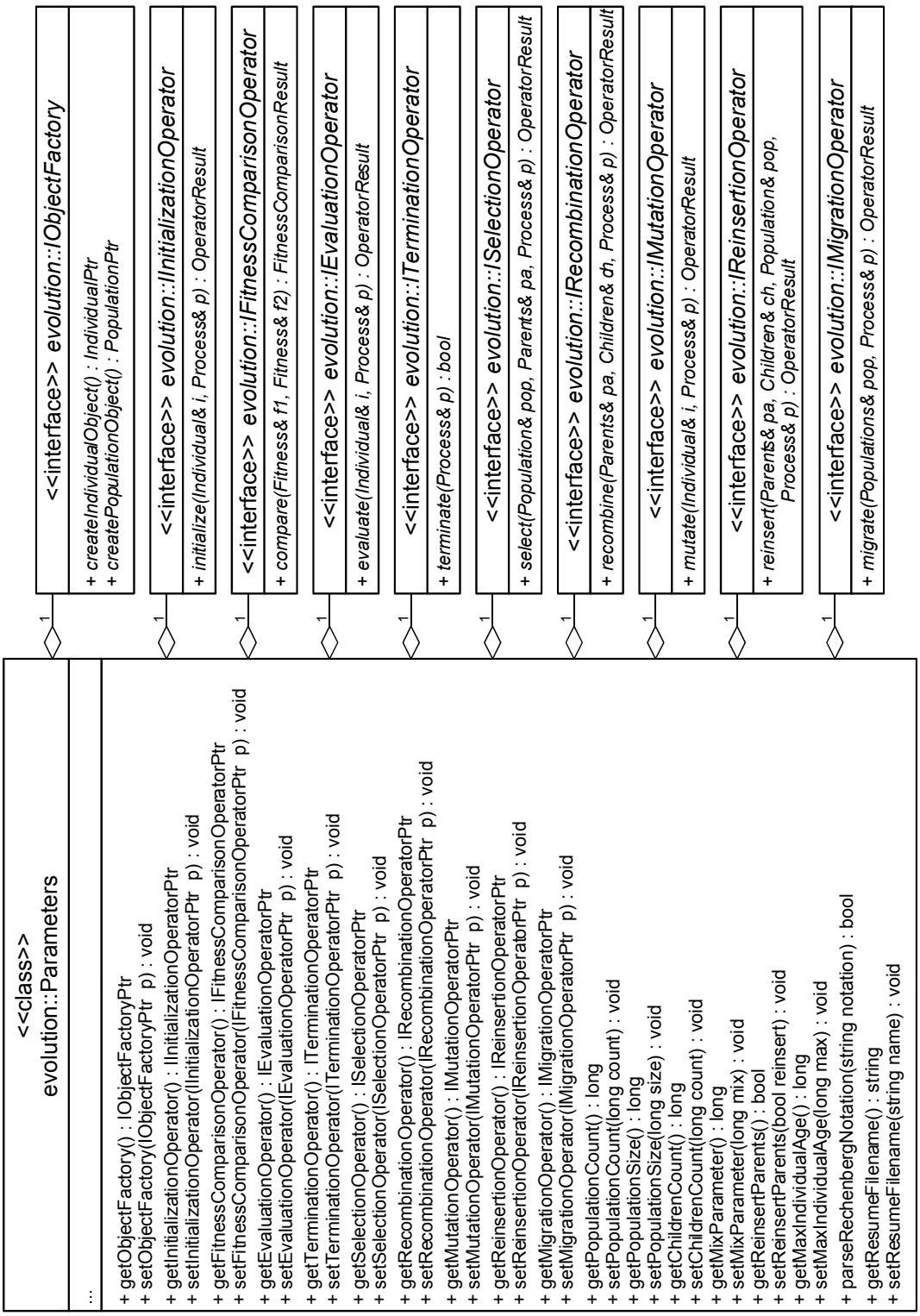


Abbildung 7.23: UML-Diagramm des Modulinterfaces der Evolutions-Engine (Teil 5)

- **Rekombination** (`get/setRecombinationOperator(...)`)
  `interface` *`evolution::IRecombinationOperator`* erzeugt aus einem oder mehreren Eltern einen Nachkommen und fügt diesen der Kindpopulation hinzu.

  Dieses Interface wird von mehreren Klassen abgeleitet. `class Recombination Operator_Discrete` selektiert zufällig Genome aus dem Genotyp der Eltern (siehe Abschnitt 5.2.7.1 auf Seite 66). `class RecombinationOperator_Intermediate` sucht einen Wert zwischen den Genom-Werten der Eltern aus (siehe Abschnitt 5.2.7.2 auf Seite 66).

- **Mutation** (`get/setMutationOperator(...)`)
  `interface` *`evolution::IMutationOperator`* mutiert Individuen aus einer Population.

  Auch hier stehen mehrere abgeleitete Klassen zur Verfügung. `class Mutation Operator_RandomValue` setzt ein Genom auf einen zufälligen Wert (siehe Abschnitt 5.2.8.1 auf Seite 68), `class MutationOperator_RandomVariation` variiert ein Genom um einen zufälligen Wert und `class MutationOperator_ RandomVariationAdaptive` erweitert letzteres Prinzip um eine adaptive Anpassung der Variation je nach Fortschritt des Evolutionsvorgangs.

- **Wiedereinfügen** (`get/setReinsertionOperator(...)`)
  `interface` *`evolution::IReinsertionOperator`* fügt Individuen aus Eltern- und Kind-Population in die Population der nächsten Generation ein.

  Hier wurde mit der Klasse `class ReinsertionOperator_Elitest` der Algorithmus aus Abschnitt 5.2.9.3 auf Seite 70 implementiert.

- **Migration** (`get/setMigrationOperator(...)`)
  `interface` *`evolution::IMigrationOperator`* sorgt bei mehreren Populationen für einen Austausch von Individuen untereinander (siehe Abschnitt 5.2.10 auf Seite 70).

  Da im Rahmen dieser Thesis noch nicht mit multiplen Populationen gearbeitet wurde, existiert von diesem Operator bisher keine Implementierung.

- **Weitere Parameter**

  - `get/setPopulationCount(...)`: Die Anzahl der Populationen.
  - `get/setPopulationSize(...)`: Die Anzahl von Individuen in einer Population.
  - `get/setChildrenCount(...)`: Die Anzahl von Nachkommen einer Population.
  - `get/setMixParameter(...)`: Die Anzahl der Eltern pro Nachkomme.
  - `get/setReinsertParents(...)`: `true`, wenn die Eltern und die Kinder in die nächste Generation eingefügt werden sollen. `false`, wenn nur die Kinder in die nächste Generation eingefügt werden.

  - `get/setMaxIndividualAge(...)`: Individuen mit diesem Alter „sterben“ automatisch.

  Mit Hilfe der Methode `parseRechenbergNotation(...)` kann man eine Zeichenkette nach der Rechenberg-Notation (siehe Abschnitt 5.3.1.1 auf Seite 72) parsen und somit die Parameter bestimmen.

- **Persistenz** (`get/setResumeFilename(...)`)
  Diese Datei enthält nach jeder neuen Generation einen Schnappschuss aller Individuen und Parameter. Damit ist es möglich, das Programm nach Beendigung oder Absturz an der Stelle des Abbruchs fortfahren zu lassen.

7.13 Skriptsprachen

Die Verwendung von Skriptsprachen in Programmen bietet den wesentlichen Vorteil, dass Parameter, Verhaltensweisen und vieles mehr, was sich mit Skripten behandeln lässt, ohne Neukompilierung des Hauptprogramms veränderbar sind.
In `cerebellum` werden Skripte für folgende Aufgaben verwendet:

- Konfiguration des Programms
- Konfiguration und dynamische Erzeugung von Szenerien
- Zufallsgesteuerte Erzeugung von Objekten oder Parametern
- Bewertung des Fitnesswerts des virtuellen Charakters
- Erzeugung von Visualisierungsparametern

Als Skriptsprache wird LUA[19] (Extensible Extension Language) verwendet. Diese Sprache zeichnet sich durch eine einfache, aber vielseitige Syntax aus, und ist in C geschrieben und somit unter Windows und Linux kompilierbar. Externe Klassen[20] schaffen einen einfachen objektorientierten Zugang zu der Sprache und ermöglichen so einerseits den Zugriff von C++ auf LUA-Methoden und andererseits den Zugriff von LUA auf C++-Methoden.
Es ist auch möglich, Datenbäume aus LUA-Skripten zu laden, als handle es sich bei diesen um XML-Dateien. Dazu dient die Klasse `LuaDataTree`. Tabelle 7.6 zeigt anhand des Beispiels aus Abschnitt 7.5 auf Seite 92, wie ein LUA-Programm eine Tabelle erzeugt, die als Datenbaum gelesen wird.

Datenbaum	**LUA-Array**

profile
├ *Name* Test
├ render_engine
│ ├ *type* ogre
│ └ screen
│ ├ width 320
│ └ height 200
└ controller
└ *type* kbd
	`profile={`
	`  _name="Test",`
	`  render_engine={`
	`  _type="ogre",`
	`  screen={`
	`  width=320,`
	`  height=200`
	`  }`
	`  },`
	`  controller={`
	`  _type="kbd"`
	`  }`
	`}`

Tabelle 7.6: Definition von Datenbäumen mit LUA

[19] ©1994-2004 Tecgraf, PUC-Rio, `http://www.lua.org`
[20] Angelehnt an die Lua-Wrapper Klassen von Yong Lin (`ly4cn@21cn.com`).

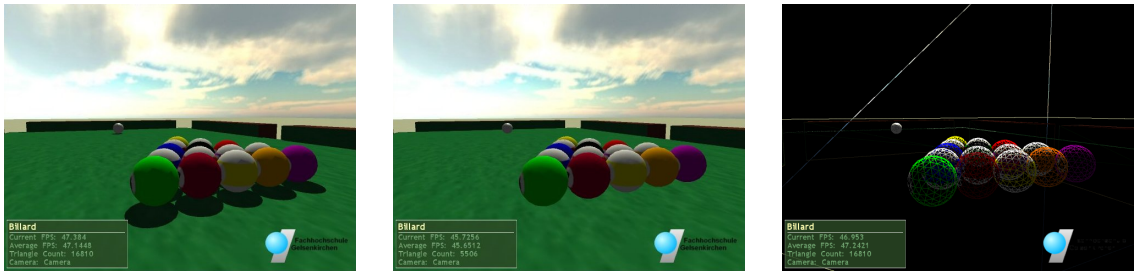
7.14 Das Simulationsprogramm `cerebellum`

`cerebellum` ist das Simulationsprogramm. Mit seiner Hilfe können Szenen geladen und simuliert werden. Konfiguration und Inhalt der zu ladenden Szenen werden mit Hilfe von LUA-Skripten generiert. Das Programm kann durch Übergabe des entsprechenden Profilnamens auch als „Server“ für Netzwerk-Render-Engines dienen.
Innerhalb einer geladenen Szene kann man sich mit Hilfe der Cursortasten und/oder der Maus bewegen, zoomen, Ansichten umschalten, Parameter der Render-Engine verändern etc. Die Simulation lässt sich beschleunigen, verlangsamen und komplett stoppen. Es können auf Tastendruck beliebige Unterszenen nachgeladen werden.
Tabelle 7.7 zeigt die Kommandozeilenparameter von `cerebellum`.

Kommandozeilenparameter	**Bedeutung**
`-D/--data_directory <Pfadname>`	Pfad für die Nutzdaten
`-C/--configuration_script <Dateiname>`	Dateiname des Konfigurationsskriptes
`-S/--scenario_script <Dateiname>`	Dateiname des Scenario-Skriptes
`-P/--port <Portadresse>`	Netzwerkport für geVE-Daten
`-L/--loopback`	Netzwerk im Loopback-Modus betreiben
`-p/--profile <Profilname>`	Simulationsprofil (z.B. slow, fast, server)
`-Y/--invert_mouse_y`	Invertiert die Y-Achse der Mausbewegungen
`-m/--mode <Modus>`	Betriebsmodus (z.B. batch, monitor, movie)
`-N/--neural_postfix <Postfix>`	Zusatz zum Dateinamen von neuronalen Netzen
`--help`	Hilfeseite anzeigen
`-v/--version`	Versionsnummer und zusätzliche Informationen anzeigen

Tabelle 7.7: Kommandozeilenparameter von `cerebellum`

Abbildung 7.24 auf der nächsten Seite zeigt Bilder vom Betrieb des Simulationsprogramms in verschiedenen Darstellungen. Die geladene Szene stellt einen physikalisch korrekt simulierten Billardtisch dar. Nähere Einzelheiten zu dieser Szene finden sich in Abschnitt 8.2.2.4 auf Seite 152.
Ein spezieller interaktiver Modus erlaubt die Steuerung des Programms mit Hilfe der Standardein- und -ausgabe. Dadurch ist es auch möglich, unter Linux das Programm als Kindprozess zu starten und mit Hilfe einer Pipe zu steuern. Dieser Mechanismus wird vom Evolutionsprogramm `evolver` verwendet (siehe Abschnitt 7.16 auf Seite 142). Abbildung 7.25 auf der nächsten Seite zeigt den Betrieb des Programms in diesem Modus. Hinter dem Render-Fenster ist ein Terminal mit den Ein- und Ausgaben des Programms zu sehen.



(a) Normaler Modus (b) Ohne Schattenwurf (c) „Wireframe“-Modus

Abbildung 7.24: Bildschirmfotos von `cerebellum` (Teil 1, siehe Farbtafel D.7 auf Seite 235)

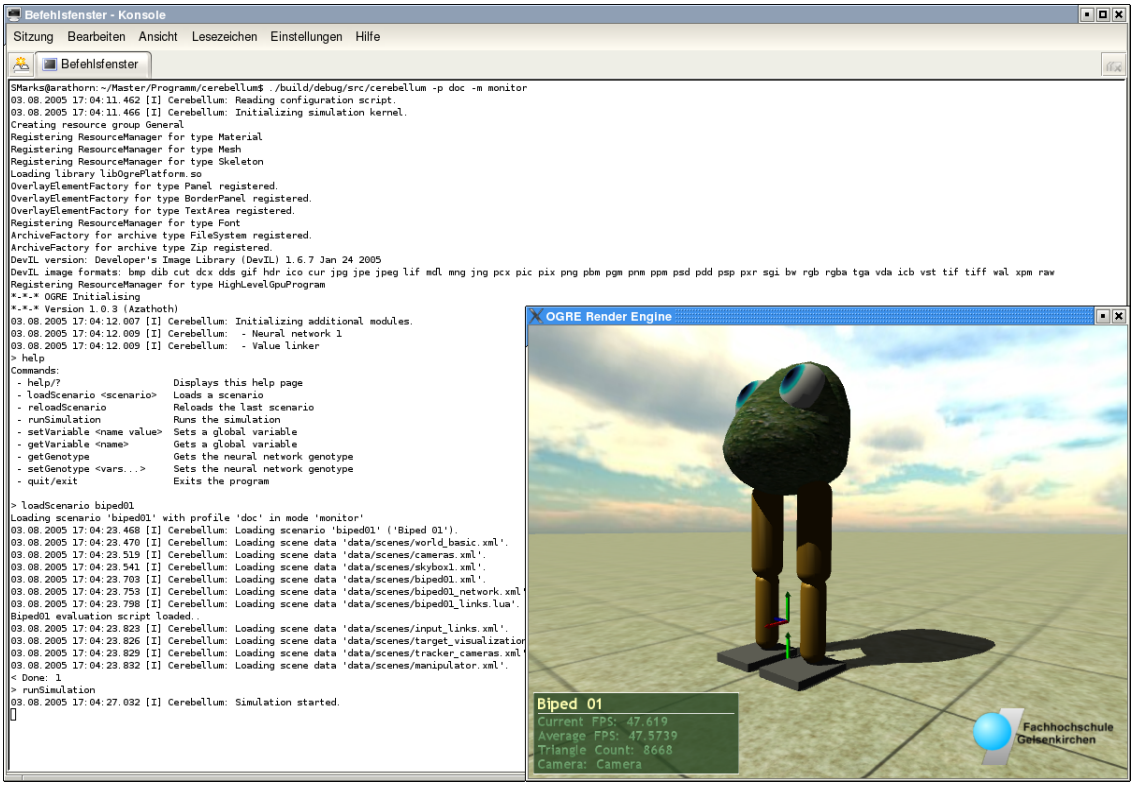


Abbildung 7.25: Bildschirmfoto von `cerebellum` (Teil 2, siehe Farbtafel D.8 auf Seite 237)

7.15 Das Visualisierungsprogramm `CerebellumObserver`

Das Programm `CerebellumObserver` ist ein Java-Programm, welches über Netzwerkpakete Informationen über Ein-, Ausgaben und Verbindungsgewichte des neuronalen Netzwerks empfängt und grafisch darstellt. Zusätzlich lassen sich Kommandos an das Simulationsprogramm senden
Abbildung 7.26 zeigt das GUI während einer Simulation.

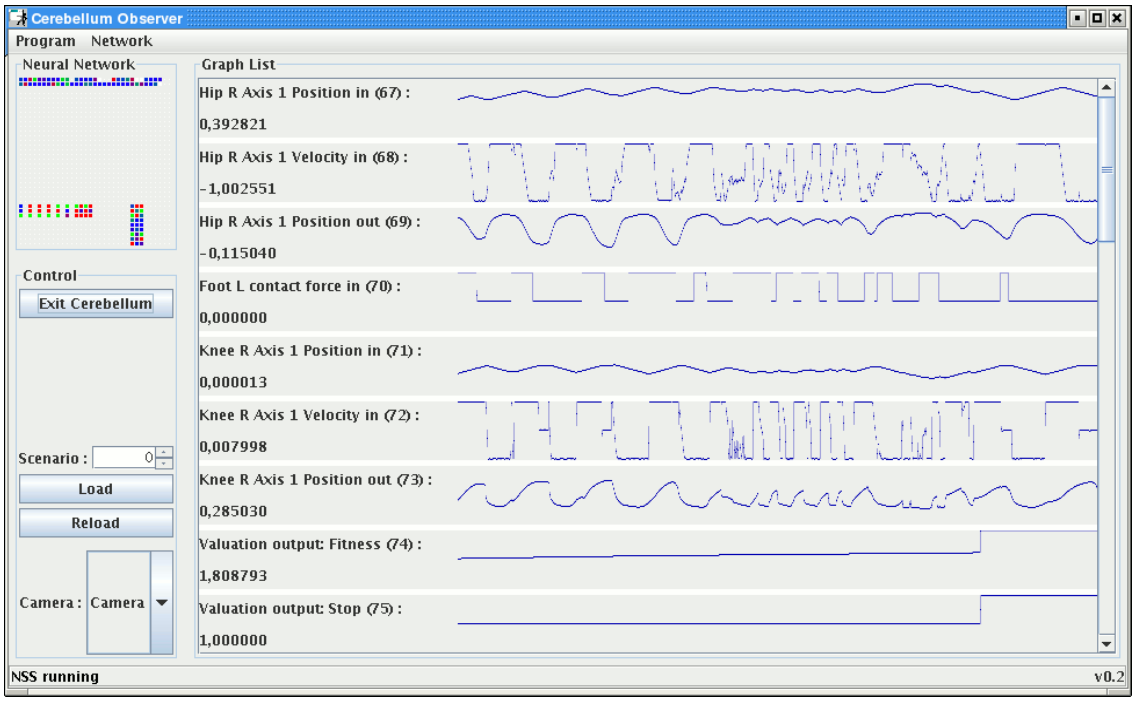


Abbildung 7.26: Bildschirmfoto von `CerebellumObserver` (siehe Farbtafel D.9 auf Seite 239)

7.16 Das Evolutionsprogramm `evolver`

Das Evolutionsprogramm `evolver` ist ein rein konsolenorientiertes Programm. Zu Beginn werden die Simulationsprozesse erzeugt und im interaktiven Modus gestartet. Im Anschluss lädt `evolver` die zu simulierende Szene, stellt alle Rahmenbedingungen ein und startet den Evolutionsprozess.
Tabelle 7.8 auf der nächsten Seite zeigt die Kommandozeilenparameter von `evolver`.
Da ein Evolutionsprozess je nach Konfiguration relativ lange dauern kann, wurden die Computer innerhalb des VUM-Labors mit Hilfe der Kernel-Erweiterung Mosix zu einem Cluster zusammengeschaltet. Mosix erfordert keine Änderung an einem Programm, um dieses im Cluster lauffähig zu machen. Sobald Prozesse oder Kindprozesse viel Rechenleistung benötigen, lagert Mosix diese automatisch und transparent auf andere Computer aus.

Kommandozeilenparameter	**Bedeutung**
`-D/--data_directory <Pfadname>`	Pfad für die Nutzdaten
`-C/--configuration_script <Dateiname>`	Dateiname des Konfigurationsskriptes
`-p/--process <Prozessname>`	Name des Evolutionsprozesses
`-c/--children <Anzahl>`	Anzahl der Simulationsprozesse
`-m/--monitor`	Einen Simulationsprozess im Monitor-Modus starten
`--help`	Hilfeseite anzeigen
`-v/--version`	Versionsnummer und zusätzliche Informationen anzeigen

Tabelle 7.8: Kommandozeilenparameter von `evolver`

Abbildung 7.27 zeigt den zentralen Computer bei der Arbeit. Neben der Konsole, auf welcher `evolver` gestartet wurde, ist auch ein Monitor-Fenster zu sehen, welches Einblick auf ein momentan simuliertes Individuum erlaubt. Rechts oben ist eine Visualisierungs-Software für den Mosix-Cluster gestartet. Diese zeigt die aktuelle Auslastung an. Unter diesem Fenster wird die Verteilung der Prozesse im Cluster visualisiert. Der zentrale Kreis repräsentiert den Computer, auf welchem das Programm gestartet wurde. Die umgebenden Kreise stehen für die anderen Computer im Cluster. Jedes Kästchen symbolisiert einen Prozess. Linien zeigen, welche Prozesse wohin ausgelagert wurden.

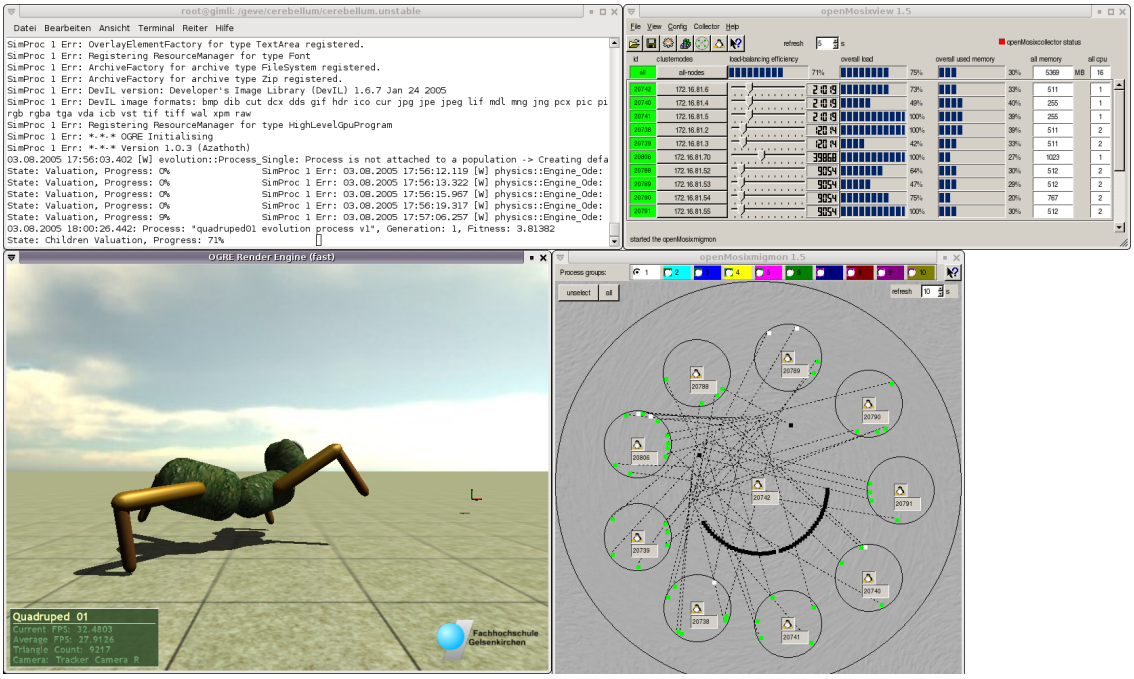


Abbildung 7.27: Bildschirmfoto von `evolver` (siehe Farbtafel D.10 auf Seite 241)

7.17 Die Netzwerk-Render-Engine `network_render_engine`

Das Programm `network_render_engine` startet eine einzelne Render-Engine und versorgt diese mit Daten aus dem Netzwerk. Diese Daten werden von der Klasse `render::Engine_Network` erzeugt, welche innerhalb des Simulationsprogramms als Implementierung des Modulinterfaces `interface` *`render::IEngine`* instanziiert wurden (siehe Abschnitt 7.9.3 auf Seite 118).
Tabelle 7.9 zeigt die Kommandozeilenparameter von `network_render_engine`.

Kommandozeilenparameter	**Bedeutung**
`-x/--x_size <Breite>`	Breite des Render-Fensters
`-y/--y_size <Hoehe>`	Höhe des Render-Fensters
`-f/--fullscreen`	Ausgabe im Vollbildmodus
`-a/--anti_aliasing <Wert>`	Anti-Aliasing der Stärke „Wert“ verwenden
`-fps/--max_frames_per_second <Wert>`	Maximal „Wert“ Bilder pro Sekunde ausgeben
`-o/--overlay`	Das Informationsfenster unten links einblenden
`-c/--camera <Kameraname>`	Immer nur die angegebene Kamera verwenden
`-mc/--mirror_camera`	Kamera-Offsets an der Y-Achse spiegeln
`-Y/--invert_camera_y`	Kamerabewegungen in Y-Richtung invertieren
`-L/--loopback`	Netzwerk im Loopback-Modus betreiben
`-P/--port <Portadresse>`	Netzwerkport für geVE-Daten
`-D/--for_id <ID>`	geVE-ForID des Programms
`--help`	Hilfeseite anzeigen
`-v/--version`	Versionsnummer und zusätzliche Informationen anzeigen

Tabelle 7.9: Kommandozeilenparameter von `network_render_engine`

Teil III

Auswertung

Problems worthy of attack prove their worth
by fighting back.

Paul Erdos (1913-1996)

8 Ergebnisse

In diesem Kapitel werden alle Ergebnisse der einzelnen Aufgabenstellungen aus Abschnitt 1.4 auf Seite 8 vorgestellt.

8.1 Implementierung einer virtuellen Umgebung

Die Implementierung der virtuellen Umgebung erfolgte in mehreren Schritten:

8.1.1 Sichtung der vorhandenen Materialien und Software

Aus bisherigen Arbeiten stand lediglich eine primär für Windows und Xbox geschriebene Netzwerkbibliothek mit Definitionen der grundlegenden Datenpakete in C++ bzw. Java zur Verfügung. Diese minimale Grundlage entspricht der Philosophie des Labors, alle Software lediglich über Netzwerk miteinander kommunizieren zu lassen.
Eine aus [Pollak, 2003] entstandene Software zur Konstruktion von virtuellen Szenerien hätte zu viel Einarbeitungszeit erfordert. Wichtige Bestandteile wie die Physik-Engine, neuronale Netzwerke oder die Evolutions-Engine standen mit dieser Software ebenfalls nicht zur Verfügung.
Des Weiteren stand der Computer mit einer Software zur Ansteuerung des Trackers und einer Software zur Ansteuerung von Datenhandschuhen bereit. Letztere wurde im Rahmen dieser Thesis nicht benötigt.

8.1.2 Linux-Portierung

Die Netzwerkbibliothek wurde auf Linux portiert. Dabei wurden wichtige Erkenntnisse zur plattformunabhängigen Programmierung gesammelt. So ist es z.B. anzuraten, die STL zu verwenden, da sie mittlerweile auch unter Windows gut portiert wurde. Allerdings sollte man zu komplizierte Template-Ausdrücke oder „Kunstgriffe“ und Tricks vermeiden, da mindestens einer der beiden Compiler Warnungen oder gar Fehler ausgibt (Windows: Microsoft Visual Studio 6 bzw. .net2003, Linux: GNU-Compiler-Collection 3, später 4).

8.1.3 Implementierung grundlegender Zusatzmodule

Im nächsten Schritt wurden die Module für Smart-Pointer, Ressourcenmanagement, XML-Datenbäume und Kommandos implementiert und getestet (siehe Kapitel 7.1 bis 7.5 auf Seiten 87–92).

8.1.4 Anbindung einer Grafikbibliothek

Aufgrund der Kompilierbarkeit unter Windows- und Linux-Plattformen fiel die Entscheidung bezüglich der Grafikbibliothek zugunsten von OGRE aus (siehe Abschnitt 2 auf Seite 113). Die Bibliothek wurde installiert und zunächst experimentell in Betrieb genommen. Dabei erwies sich das Forum[1] auf der Webseite von OGRE als große Hilfe, wenn Probleme auftraten. Auch das gute objektorientierte Design verhalf zur problemlosen Inbetriebnahme.
Da alle Ressourcen wie Meshes, Texturen oder Shaderprogramme für OGRE auf einem Dateisystem liegen müssen, wurde ein NFS-Laufwerk vorbereitet. Auf dieses haben alle Computer Zugriff, welche Render-Aufgaben ausführen.
Die Idee, mehrere Xboxen als preiswerte Render-Engines zu verwenden, musste an dieser Stelle allerdings fallen gelassen werden. Die Hardware der Xbox unterscheidet sich zu stark von der eines Standard-PCs, so dass zu viele Ausnahmen (`#ifdef _XBOX_`) in die Programmcodes hätten eingebaut werden müssen. Auch die OGRE-Bibliothek und NFS funktionieren nicht. Es ist aber nicht auszuschließen, dass eine spezielle Implementierung des Render-Engine Modulinterfaces dazu führen könnte, doch grafische Ausgaben zu erhalten. Diese Möglichkeit wurde aus Zeitmangel und wegen der den Rahmen sprengenden Komplexität nicht weiter verfolgt, bietet aber genug Material für eine eigenständige Arbeit.

8.1.5 Implementierung der Netzwerk-Render-Engine

Als zweite Implementierung des Render-Engine Modulinterfaces wurde `class render::Engine_Network` geschaffen. Sie setzt sämtliche Methodenaufrufe in Netzwerkpakete um und versendet diese als Kommandos über das Netzwerk. Ein eigenständiges Programm `network_render_engine` startet die OGRE-Render-Engine, nimmt die Befehle aus dem Netzwerk entgegen und gibt sie an die Engine weiter. Dieses Programm läuft auf allen Computern, welche nur für die grafische Ausgabe der Szene zuständig sind.
Zur Verwirklichung dieses Programms mussten die bisher vorhandenen Datenstrukturen der Netzwerkbibliothek um ein neues Paket und eine dazugehörige Klasse `ByteBuffer` erweitert werden. Mit ihrer Hilfe ist es möglich, ein Kommando in das Datenpaket hineinzuschreiben, zu senden und nach Empfang des Pakets das Kommando wieder auszulesen. Da alle wichtigen Module `interface` *`IControllable`* implementieren und alle Methoden

[1] `http://ogre3d.org/phpBB2/`

der Modulinterfaces auch über Kommandos aufrufbar sind, lassen sich somit alle Module theoretisch auch über Netzwerk steuern.
„Theoretisch“ deshalb, weil bisher noch keine Möglichkeit existiert, Rückgabewerte von Methoden zurückzusenden. Daher muss man für Methoden wie *render::IEngine::getNode(...)* oder *render::IEngine::getNumberOfNodes()* auf lokales Caching zurückgreifen. `class render::Engine_Network` speichert Informationen zu allen erzeugten Objekten lokal und sendet zusätzlich die Kommandos über das Netzwerk. Allerdings kann die Klasse so nicht prüfen, ob alle `network_render_engine`s das Kommando auch wirklich ausgeführt haben. Da die Kommandos als UDP-Broadcast[2] versendet werden, lässt sich die Bestätigung der Ausführung nicht trivial realisieren. Die Beseitigung dieser Schwäche bietet ebenfalls Material für eine eigenständige Arbeit.

8.1.6 Implementierung einer einfachen Protokollschicht

Im Betrieb zeigte sich häufig, dass die Netzwerk-Render-Engines Pakete nicht empfingen und somit Objekte in der Darstellung fehlten.
Aus diesem Grund wurde die Netzwerk-Bibliothek um die Möglichkeit erweitert, ein Paket als wichtig und unwichtig zu deklarieren. Unwichtige Pakete werden wie bis dahin üblich einfach mittels UDP-Broadcast versendet. Wichtige Pakete gelangen zusätzlich in einen Puffer, welcher die Pakete mindestens 5 Sekunden vorhält. Zusätzlich werden diese Pakete mit Sequenznummern versehen. Empfängt ein Computer ein Paket mit einer Sequenznummer, welche höher als die letzte Nummer + 1 ist, so sendet dieser Computer ein NAK-Paket[3] zurück, welches den Sender auffordert, die fehlenden Pakete erneut zu senden. Die bis dahin eintreffenden Pakete werden so lange gepuffert, bis das fehlende Paket eintrifft oder 5 Sekunden vergangen sind.
Dieses einfache Protokoll ist keine 100%ige Lösung für die Beseitigung von Datenfehlern, aber es bringt für die Netzwerk-Render-Engines eine ausreichend hohe Empfangssicherheit von Paketen.
Nach der Beseitigung der Empfangsprobleme fanden die Netzwerk-Render-Engines Einsatz in den Computern, welche die 3D-Projektionsleinwand und die beiden HMDs betreiben. Eine erste Testszene ist das Innere einer Autokarosserie (siehe Abbildung 8.1 auf der nächsten Seite). Die virtuellen Kameras der HMDs werden zusätzlich mit Translation und Orientierung aus Netzwerkpaketen des Trackers versorgt. Damit ist es möglich, um die Karosserie herumzugehen, und den Blick frei in der Szene schweifen zu lassen.

[2] UPD-Broadcast bezeichnet UDP-Pakete, welche nicht gezielt an einen, sondern an alle Computer im Netzwerk adressiert sind. Auch die Daten z.B. vom Tracker oder von den Datenhandschuhen werden so versendet.
[3] NAK: engl. „**n**egative **ac**k**k**nowledge“. Diese Abkürzung findet man oft für kleine Datenpakete, welche einen Fehler melden.

Abbildung 8.1: Virtuelle Autokarosserie (siehe Farbtafel D.11 auf Seite 243)

8.2 Implementierung einer physikalischen Simulationsumgebung

8.2.1 Anbindung einer Physik- und Kollisions-Engine

Für einen einfachen Einstieg und Test wurde zunächst auf die selbstgeschriebene Physik-Engine zurückgegriffen (siehe Abschnitt 7.7.1 auf Seite 97). Diese diente für erste Tests, wurde dann aber schnell durch die beiden getrennten Module der Physik- und Kollisions-Engine von ODE ersetzt. Auch hier stand bei der Wahl der Engine die Verfügbarkeit für Windows *und* Linux im Vordergrund.

8.2.1.1 Koordinatensysteme

Dabei galt es zu beachten, dass unterschiedliche Konventionen bezüglich der Koordinatensysteme vorlagen. Für die Thesis wurde vereinbart, dass die X- und Z-Achsen die waagerechte Ebene beschreiben, während Y nach „oben“ zeigt. ODE verwendet allerdings eine andere Konvention, was z.B. zu Schwierigkeiten bei Trägheitstensoren und den Hauptrotationsachsen von Körpern führte. Als Ausweg wurden Makros verwendet, welche Koordinaten von und nach ODE „übersetzen“.

8.2.1.2 Parameter

Eine weitere Schwierigkeit war die sehr flexible Parametrierung von ODE. Nahezu alles, was zur Berechnung der Simulation dient, lässt sich einstellen. Das kann dazu führen, dass Gelenke nicht fest in Verbindung bleiben, sondern wie an einem Gummiband auseinanderdriften. Auch das Feintuning für die Kollisionsbehandlung hatte einige Zeit benötigt. Stellt man die betroffenen Parameter zu „streng“ ein, so kann die Simulation regelrecht explodieren, wenn Körper stärker aneinanderstoßen. Grund sind die Constraints, welche von den Kollisionsgelenken stammen (siehe Abschnitt 3.4.1 auf Seite 31). Haben diese aufgrund der Parameter zu starken Einfluss, so versucht ODE, die Constraints zwischen den kollidierenden Körpern sofort im nächsten Simulationsschritt einzuhalten. Dies führt zu starken, impulsförmigen Kräften, welche die Körper explosionsartig auseinandertreiben. Stellt man im Gegenzug die Parameter zu „nachlässig“ ein, so scheinen Körper wie

aus einer Flüssigkeit zu bestehen. Sie durchdringen sich und driften dann langsam wieder auseinander.
Da sich je nach der zu ladenden Szene und der dort vorkommenden Objekte andere Voraussetzungen ergeben können, wurden die Parameter nicht fest vorgegeben, sondern durch Datensätze einstellbar gemacht. So kann man zusammen mit jeder Szene die bevorzugten Werte laden.

8.2.1.3 TriTri-Kollisionen

Ein letztes, bis zum Schluss nicht gelöstes Problem, sind Kollisionen zwischen zwei Körpern, deren Kollisionsobjekte in Form von Meshes vorliegen. Dabei bleiben die Körper entweder aneinander hängen oder durchdringen sich kurzzeitig und werden danach aus der Szene geschleudert. Eine Analyse der bei der Kollision ermittelten Kollisionspunkte und ihrer Normalen zeigte, dass prinzipiell alle Werte in Ordnung sind. Die beobachteten Effekte treten auf, wenn eine ungünstige Konstellation von Kollisionspunkten vorliegt, die zwar alle relativ dicht beieinander liegen, deren Normalen aber in unterschiedliche Richtungen zeigen. Prinzipiell könnte man dieses Problem durch eine Aufbereitung der Kollisionspunkte beseitigen.
Da sich diese Thesis vollständig mit Kollisionsprimitiven durchführen ließ, wurde auf eine genauere Betrachtung und Lösung dieses Problems verzichtet.

8.2.2 Test der Physik- und Kollisions-Engine

Es wurden mehrere Testszenarien entworfen, um die Leistungsfähigkeit und den Realismus der physikalischen Simulation zu bewerten.

(a) Fallende Objekte (b) Würfellawine (c) Kistenstapel

Abbildung 8.2: Testszenen für die Physik-Engine (Teil 1, siehe Farbtafel D.12 auf Seite 243)

8.2.2.1 Szene „Fallende Objekte“

In dieser Szene wurden alle Arten und Formen von Körpern zusammengeworfen, um die Korrektheit der Kollisionserkennung und -behandlung zu testen (siehe Abbildung 8.2(a)).

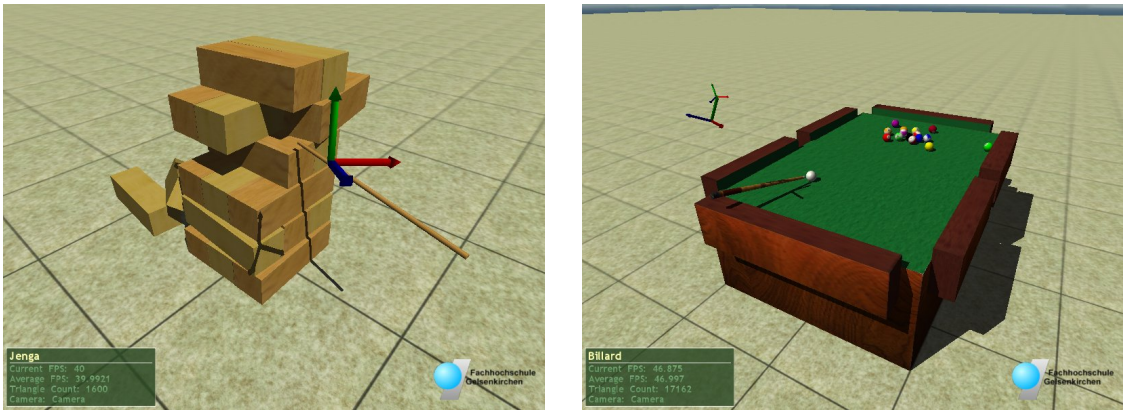
8.2.2.2 Szene „Würfellawine“

Diese Szene diente aufgrund der hohen Anzahl an Objekten (40 Würfel, 10 Treppenstufen) als Belastungstest für die Kollisions-Engine sowie für die Netzwerk-Render-Engines (siehe Abbildung 8.2(b) auf der vorherigen Seite). Wegen der hohen Anzahl von Objekten wurde die Szene auch häufig zur Überprüfung der Stabilisierung des Netzwerkprotokolls geladen (siehe Abschnitt 8.1.6 auf Seite 149).

8.2.2.3 Szene „Kistenstapel“

Der Kistenstapel diente zum Test der korrekten Reaktion der Physik-Engine auf Massenverhältnisse. Die Kugel im Bild wurde mit einer Masse von 2kg bis hin zu 2000kg versehen und in Richtung des Stapels gerollt. Die Reaktion der jeweils 10kg schweren Kisten sollte dementsprechend ausfallen (siehe Abbildung 8.2(c) auf der vorherigen Seite).

8.2.2.4 Szene „Jenga“ und „Billardtisch“



(a) Jenga (b) Billardtisch

Abbildung 8.3: Testszenen für die Physik-Engine (Teil 2, siehe Farbtafel D.13 auf Seite 245)

Diese beiden Szenen dienten zur Überprüfung der Synchronisation realer und virtueller physikalischer Gegenstände. Ein Holzstab wurde mit einem Sensor des Trackers versehen, seine realen Ausmaße und sein Gewicht in die Physik-Engine eingegeben und regelmäßig mit der gemessenen Position und Orientierung des realen Stabs abgeglichen. Das Resultat war eine verblüffende Natürlichkeit im Umgang mit dem so geschaffenen virtuellen „Manipulator“.
Bei guter Kalibrierung der Offsetwinkel der virtuellen Kamera in Bezug auf das HMD konnte man den Stab aus dem realen Sichtfeld[4] nahtlos in das virtuelle Sichtfeld hinein- und wieder hinausbewegen.

[4] Eines der HMDs deckt nur den zentralen Teil des Sichtfeldes ab und ermöglicht so „aus dem Augenwinkel heraus“ die Orientierung in der realen Umgebung. Das andere HMD ist hingegen vollständig optisch abgekapselt.

Für weitere Tests des Umgangs mit dieser Mischung aus „realer“ und virtueller Realität wurden zwei Szenen entworfen, welche die Manipulation mit dem Stab erfordern und gleichzeitig weitere Aspekte der physikalischen Simulation testen (siehe Abbildung 8.3 auf der vorherigen Seite).
Bei dem Billardtisch[5] ist auf eine gute Anpassung der Reibungskoeffizienten der Kugeln und der Oberfläche zu achten, da sonst z.B. der Eindruck einer sehr glatten Eisfläche entsteht (Reibung zu gering).
Der Jenga-Turm besteht aus länglichen „Holzklötzen“, welche mit dem Manipulator so aus dem Turm herausgeschoben werden sollen, dass der Turm dabei stehenbleibt. Zu diesem Zweck werden die Klötze mit leicht unterschiedlichen Ausmaßen erzeugt, damit einige von ihnen stärkere Reibung erfahren als andere.

8.2.3 Besonderheiten der physikalischen Simulation

Aufgrund der fehlenden Kraft-Rückkopplung (engl.: „force feedback“) kann es passieren, dass der Manipulator Bewegungen ausführt, welche in der Realität nicht möglich sind. So kann man z.B. von oben in einen am Boden liegenden virtuellen Klotz „hineinstechen“, da kein realer Klotz existiert, welcher physikalischen Widerstand bieten würde. Die Physik-Engine reagiert auf diesen „unmöglichen“ Zustand, indem sie versucht, den Klotz auf kürzestem Weg von dem Manipulator wegzubewegen. Das resultiert in heftigen, zuckenden Bewegungen des Klotzes, welche in der Realität so nicht zu beobachten wären.
Eine Möglichkeit, dieses Verhalten zu vermeiden, wäre es, die Position und Orientierung des Manipulators nicht direkt vorzugeben, sondern zu versuchen, diese durch Anwendung von Kräften und Drehmomenten zu erreichen. Durch eine solche Vorgehensweise käme die Physik-Engine nicht in die oben beschriebenen unzulässigen Situationen. Allerdings könnte man in diesem Fall wahrscheinlich Trägheitseffekte durch den notwendigen Regelvorgang beobachten.

[5] Die Billardsimulation ist der Grund für den Obertitel dieser Thesis. Diese nur für Testzwecke entworfene Szene erwies sich im Verlauf der Zeit als wahrer „Publikumsmagnet“.

8.3 Konstruktion virtueller Charaktere

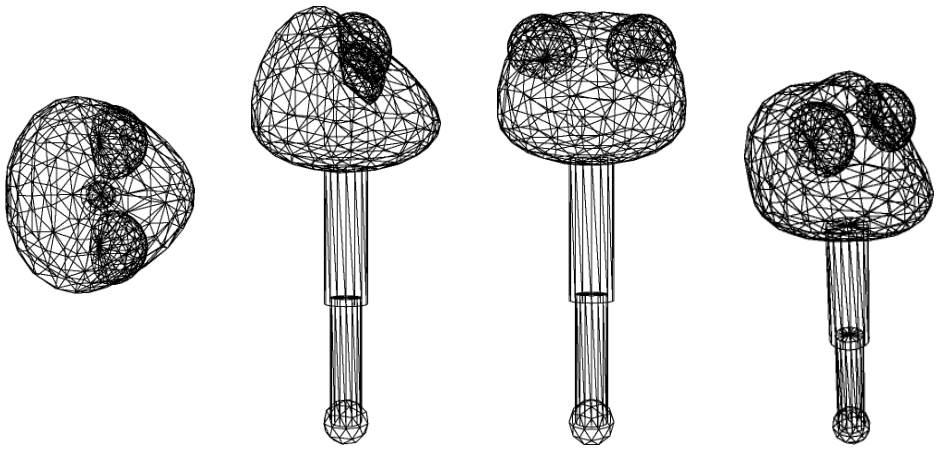
Die folgenden Abschnitte präsentieren die drei virtuellen Charaktere, ihren Aufbau, die verwendeten Gelenke und die neuronale Netzwerkarchitektur.

8.3.1 Monoped

Das Monoped besteht im Wesentlichen aus

- einem Körper,
- einem darunter angebrachten, schwenkbaren Zylinder, welcher sich ein- und ausfahren kann,
- und einem daran angebrachten Fuß.

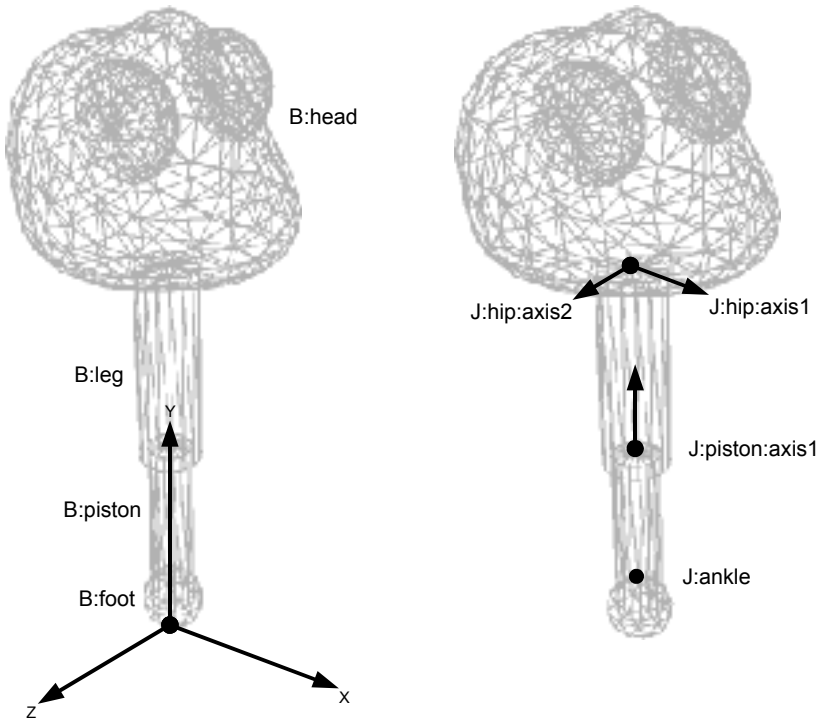
Das Monoped ist in sich instabil und muss aktiv für sein Gleichgewicht sorgen (dynamische Stabilität). Durch Schwenken des Zylinders nach vorne und hinten bzw. zur Seite können Stabilitätskorrekturen durchgeführt werden. Die Fortbewegung erfolgt durch rhythmisches Ein- und Ausfahren des Zylinders, wobei sich das Monoped vom Boden abstößt und eine kurze Zeit ohne Bodenkontakt ist (Flugphase).
Abbildung 8.4 zeigt vier Ansichten des für diese Thesis konstruierten Monoped.



(a) Draufsicht (b) Seitenansicht (c) Vorderansicht (d) Schrägansicht

Abbildung 8.4: Monoped

Tabelle 8.1 auf der nächsten Seite und Tabelle 8.2 auf Seite 156 geben eine Übersicht über die verwendeten physikalischen Körper und Gelenke und deren Namen. Abbildung 8.5 auf der nächsten Seite zeigt deren Anordnung.



(a) Körper (b) Gelenke

Abbildung 8.5: Körper und Gelenke des Monoped

Körper	**Eigenschaften**
B:head	Oberkörper, Gegengewicht für die Beinbewegung
	Form: Kugel
	Dimension: Durchmesser: 0,75m
	Masse 50kg
B:leg	Schwenkbarer Teil des Beins
	Form: Zylinder
	Dimension: Durchmesser: 0,175m
	Länge: 0,55m
	Masse: 5kg
B:piston	Ein-/ausfahrbarer Zylinder für den Sprungvorgang
	Form: Zylinder
	Dimension: Durchmesser: 0,125m
	Länge: 0,5m
	Masse: 3kg
B:foot	Fußelement, hoher Reibungskoeffizient für gute Bodenhaftung
	Form: Kugel
	Dimension: Durchmesser: 0,175m
	Masse: 1kg

Tabelle 8.1: Eigenschaften der physikalischen Körper des Monoped

Gelenk	Eigenschaften			
J:hip	Gelenk zur Bewegung des Sprungzylinders (Hüftgelenk)			
	Typ:	Kreuzgelenk		
	Achse 1:	X-Achse	Achse 2:	Z-Achse
	- Begrenzung:	$[-20° \dots +20°]$	- Begrenzung:	$[-20° \dots +20°]$
	- Drehmoment:	200Nm	- Drehmoment:	200Nm
J:piston	Gelenk für den Sprungvorgang (Sprunggelenk)			
	Typ:	Lineargelenk		
	Achse 1:	Y-Achse		
	- Begrenzung:	$[-0{,}75\text{m} \dots 0\text{m}]$		
	- Kraft:	5000N		
J:ankle	Gelenk zur Verbindung von Sprungzylinder und Fuß (Fußgelenk)			
	Typ:	starres Gelenk		

Tabelle 8.2: Eigenschaften der physikalischen Gelenke des Monoped

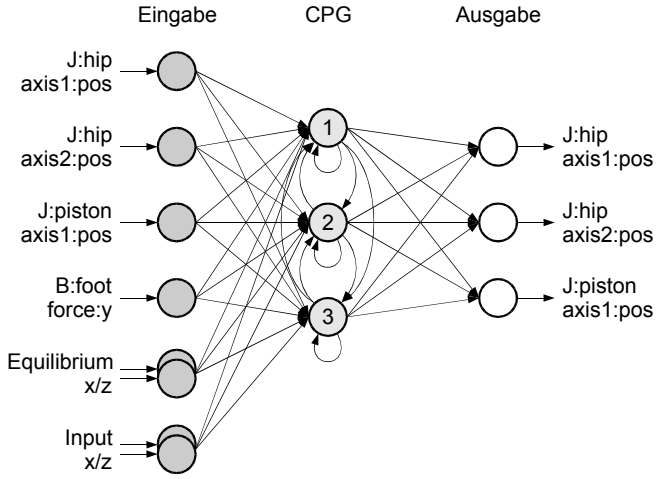


Abbildung 8.6: Neuronales Netzwerk des Monoped

Das verwendete neuronale Netzwerk wird in Abbildung 8.6 auf der vorherigen Seite dargestellt. Es handelt sich dabei um ein zeitkontinuierlich rekurrentes Netzwerk (CTRNN).

Die Eingabeschicht nimmt folgende Informationen auf:

- **J:hip:axis1/2:pos**
  Die aktuelle Position der Achsen 1 und 2 des Hüftgelenks.
- **J:piston:axis1:pos**
  Die aktuelle Position des Sprunggelenks.
- **B:foot:force:y**
  Die Kraft, welche in Y-Richtung auf den Fuß einwirkt.
- **Equilibrium:x/z**
  Die Orientierung des Kopfes in X- und Z-Richtung des Monoped (Gleichgewicht). Diese Werte sind 0, wenn sich das Monoped exakt im Gleichgewicht befindet, $0 < x < 1$, wenn das Monoped nach rechts (X-Achse) bzw. nach hinten (Z-Achse) kippt und $-1 < x < 0$, wenn das Monoped nach links (X-Achse) bzw. nach vorne (Z-Achse) kippt.
- **Input:x/z**
  Diese Eingabewerte dienen zur Steuerung des Monoped in X- und Z-Richtung. Ein positiver Wert steht für eine Bewegung in die positive Richtung der entsprechenden Achse in m/s, ein negativer Wert für eine entgegengesetzte Bewegung.

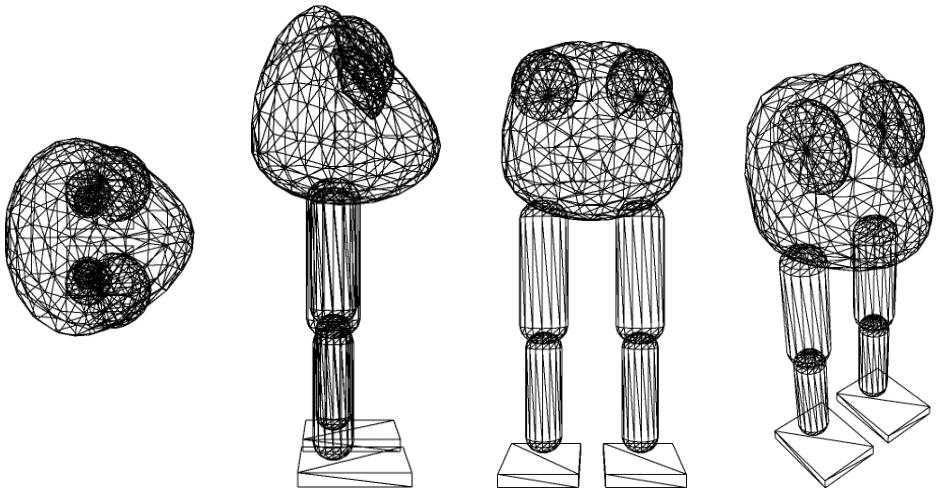
Die mittlere Schicht der Neuronen ist vollständig miteinander verbunden und stellt den Mustergenerator dar.
Die Ausgabeschicht wird nur von der mittleren Schicht angesteuert und gibt die Sollwinkel der entsprechenden Gelenke vor.
Alle Zeitkonstanten der Eingabe- und Ausgabeneuronen sind auf 0 gestellt, da hier keine zeitlichen Effekte wirken sollen. Lediglich die CPG-Neuronen besitzen Zeitkonstanten > 0s, welche ebenso wie die Verbindungsgewichte des gesamten Netzwerks mit Hilfe der evolutionären Algorithmen eingestellt werden sollen.

8.3.2 Biped

Das Biped besteht im Wesentlichen aus

- einem Körper,
- zwei Beinen, welche jeweils zweigeteilt (Ober- und Unterschenkel) und mit Hilfe eines Kniegelenks miteinander verbunden sind,
- und an den Beinen befestigten Füßen, welche ebenfalls durch Gelenke bewegt werden können.

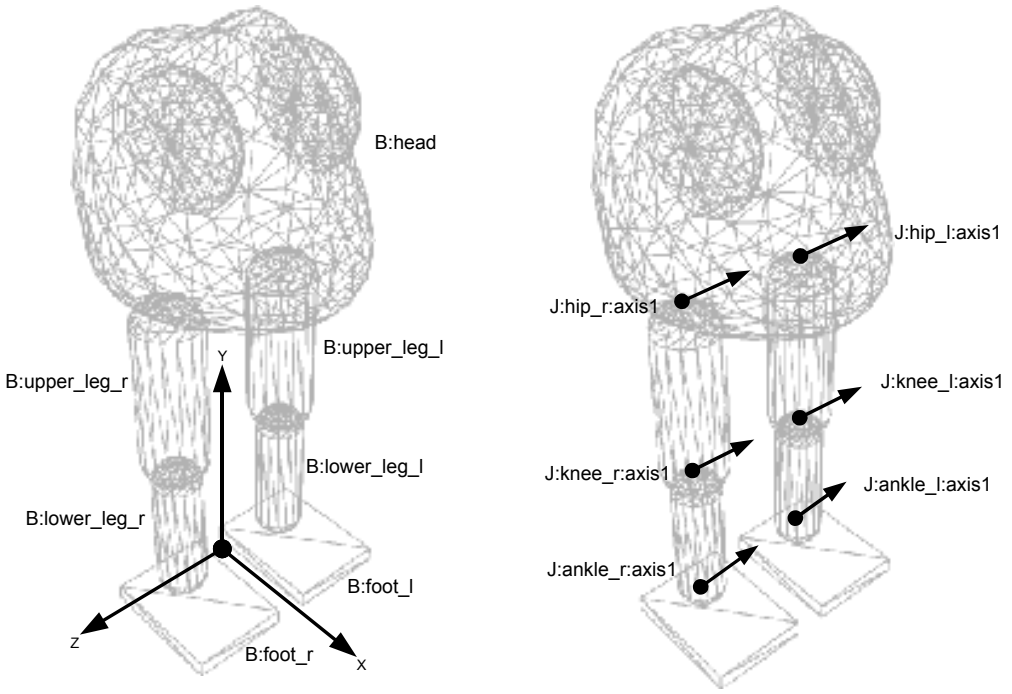
Das Biped ist lediglich im unbewegten Zustand stabil, da sich der Schwerpunkt innerhalb der Auflagefläche der Füße befindet. Im Falle der Fortbewegung hingegen muss es aktiv für sein Gleichgewicht sorgen (dynamische Stabilität). Zu diesem Zweck können die Ober- und Unterschenkel innerhalb festgelegter Grenzen vor- und zurückbewegt werden.



(a) Draufsicht (b) Seitenansicht (c) Vorderansicht (d) Schrägansicht

Abbildung 8.7: Biped

Abbildung 8.7 zeigt vier Ansichten des für diese Thesis konstruierten Biped.
Tabelle 8.3 auf der nächsten Seite und Tabelle 8.4 auf Seite 160 geben eine Übersicht über die verwendeten physikalischen Körper und Gelenke und deren Namen. Abbildung 8.8 auf der nächsten Seite zeigt deren Anordnung.



(a) Körper (b) Gelenke

Abbildung 8.8: Körper und Gelenke des Biped

Körper	**Eigenschaften**	
B:head	Oberkörper, Gegengewicht für die Beinbewegung	
	Form:	Kugel
	Dimension:	Durchmesser: 0,75m
	Masse	50kg
B:upper_leg_l/r	Oberschenkel links/rechts	
	Form:	abgerundeter Zylinder
	Dimension:	Durchmesser: 0,6m
		Länge: 0,2m
	Masse:	5kg
B:lower_leg_l/r	Unterschenkel links/rechts	
	Form:	abgerundeter Zylinder
	Dimension:	Durchmesser: 0,5m
		Länge: 0,15m
	Masse:	5kg
B:foot_l/r	Fuß links/rechts	
	Form:	Quader
	Dimension:	Breite: 0,3m
		Länge: 0,4m
		Höhe: 0,05m
	Masse:	0,5kg

Tabelle 8.3: Eigenschaften der physikalischen Körper des Biped

Gelenk	**Eigenschaften**	
J:hip_l/r	Hüftgelenk links/rechts	
	Typ:	Scharniergelenk
	Achse 1:	-Z-Achse
	- Begrenzung:	$[-45° \ldots +90°]$
	- Drehmoment:	200Nm
J:knee_l/r	Kniegelenk links/rechts	
	Typ:	Scharniergelenk
	Achse 1:	-Z-Achse
	- Begrenzung:	$[0° \ldots +90°]$
	- Drehmoment:	200Nm
J:ankle_l/r	Fußgelenk links/rechts	
	Typ:	Scharniergelenk
	Achse 1:	-Z-Achse
	- Begrenzung:	$[-45° \ldots +45°]$
	- Drehmoment:	200Nm

Tabelle 8.4: Eigenschaften der physikalischen Gelenke des Biped

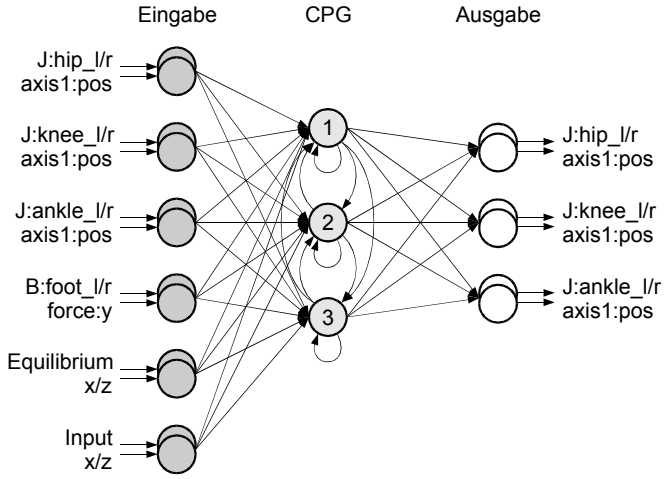


Abbildung 8.9: Neuronales Netzwerk des Biped

Das verwendete neuronale Netzwerk wird in Abbildung 8.9 auf der vorherigen Seite dargestellt. Es ist prinzipell identisch zu dem Netzwerk des Monoped aufgebaut.
Die Eingabeschicht nimmt folgende Informationen auf:

- **J:hip_l/r:axis1:pos**
  Die aktuellen Positionen des linken und rechten Hüftgelenks.
- **J:knee_l/r:axis1:pos**
  Die aktuellen Positionen des linken und rechten Kniegelenks.
- **J:knee_l/r:axis1:pos**
  Die aktuellen Positionen des linken und rechten Fußgelenks.
- **B:foot_l/r:force:y**
  Die Kraft, welche in Y-Richtung auf den linken bzw. rechten Fuß einwirkt.
- **Equilibrium:x/z**
  Die Orientierung des Kopfes in X- und Z-Richtung des Biped (Gleichgewicht). Diese Werte sind 0, wenn sich das Biped exakt im Gleichgewicht befindet, $0 < x < 1$, wenn das Biped nach rechts (X-Achse) bzw. nach hinten (Z-Achse) kippt und $-1 < x < 0$, wenn das Biped nach links (X-Achse) bzw. nach vorne (Z-Achse) kippt.
- **Input:x/z**
  Diese Eingabewerte dienen zur Steuerung des Biped in X- und Z-Richtung. Ein positiver Wert steht für eine Bewegung in die positive Richtung der entsprechenden Achse in m/s, ein negativer Wert für eine entgegengesetzte Bewegung.

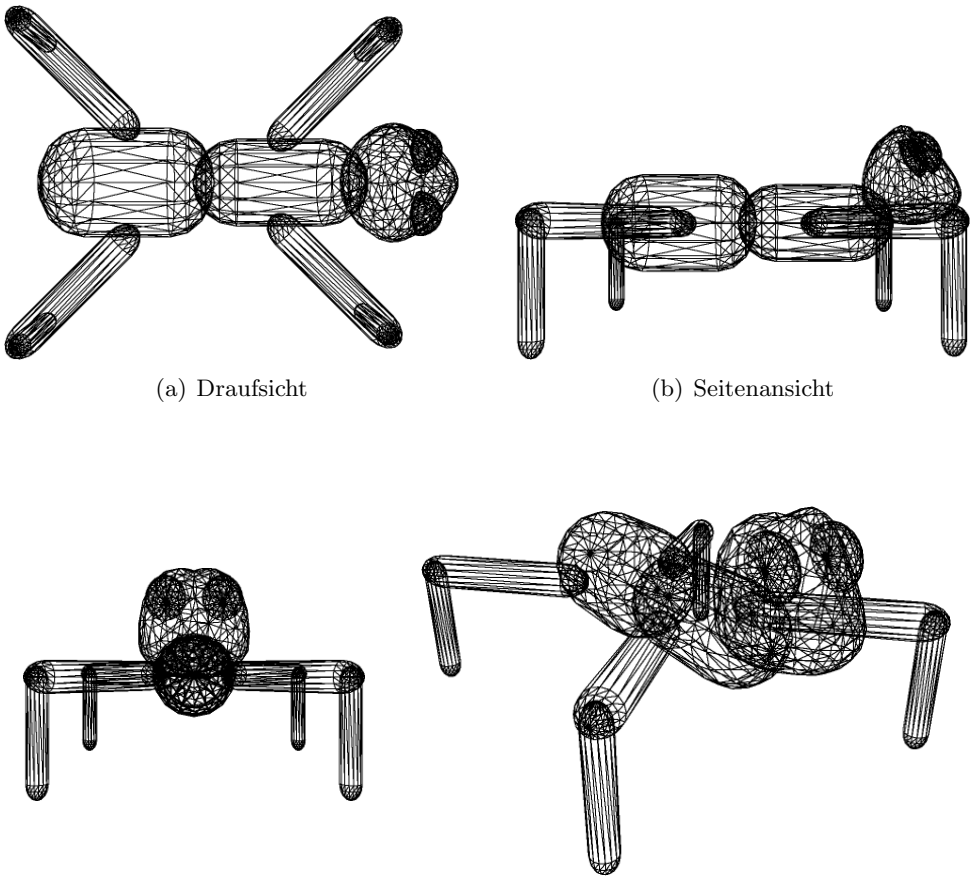
Die mittlere Schicht stellt den Mustergenerator dar und die Ausgabeschicht steuert wie beim Monoped die Sollwinkel der Gelenke.
Durch evolutionäre Algorithmen werden die Verbindungsgewichte und die Zeitkonstanten der Neuronen der mittleren Schicht eingestellt.

8.3.3 Quadruped

Das Quadruped besteht im Wesentlichen aus

- einem Körper, welcher zweigeteilt und an der Verbindungsstelle beweglich ist,
- und vier Beinen, welche jeweils zweigeteilt (Ober- und Unterschenkel) und mit Hilfe eines Kniegelenks miteinander verbunden sind.

Das Quadruped ist prinzipiell im unbewegten sowie im bewegten Zustand stabil, wenn mindestens drei der vier Beine Bodenkontakt haben und der Schwerpunkt innerhalb des durch die Auflageflächen aufgespannten Dreiecks liegt.

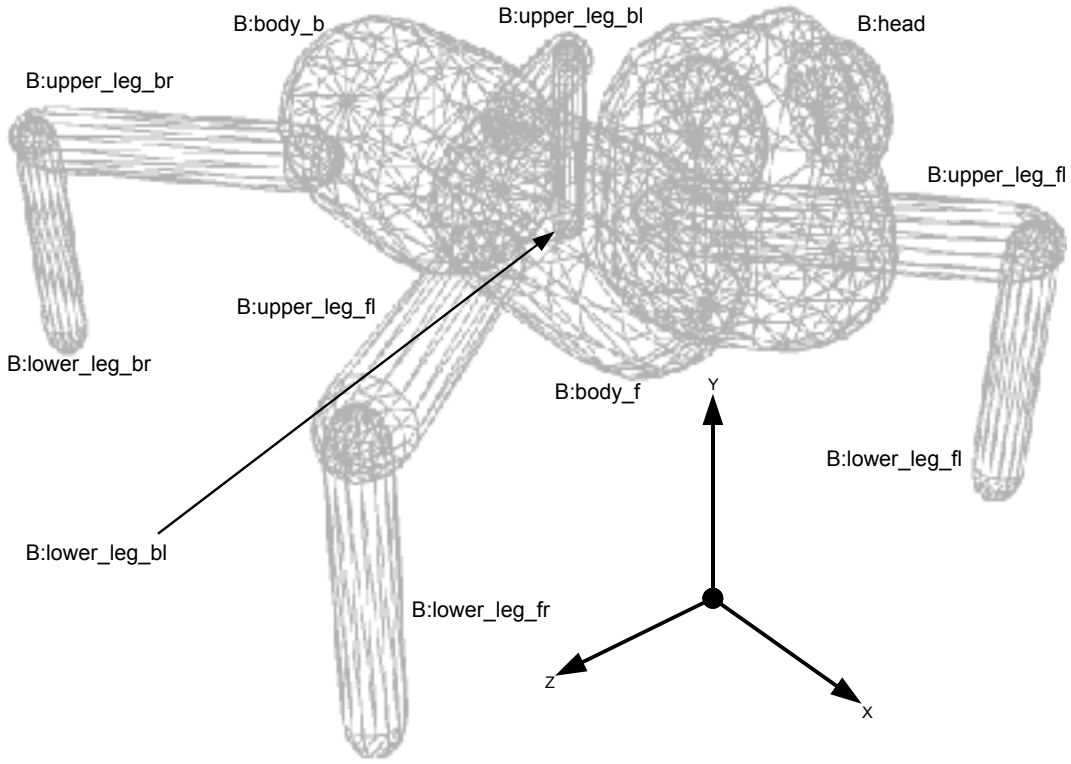


(a) Draufsicht (b) Seitenansicht

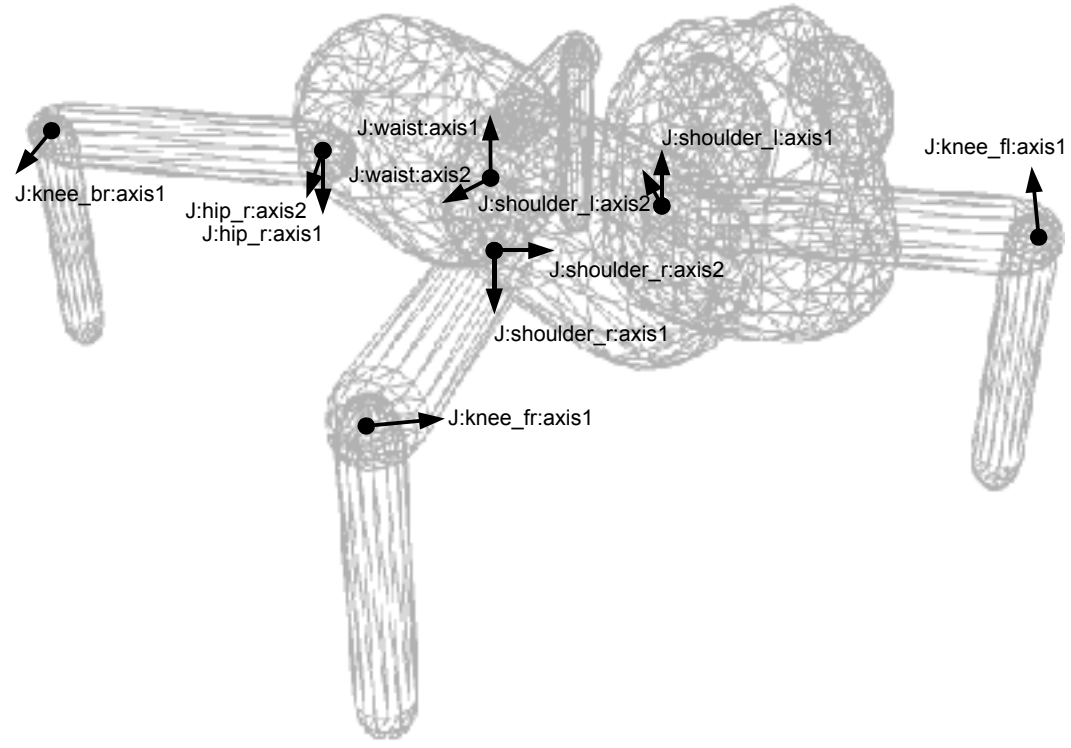
(c) Vorderansicht (d) Schrägansicht

Abbildung 8.10: Quadruped

Abbildung 8.10 zeigt vier Ansichten des für diese Thesis konstruierten Quadruped.
Tabelle 8.5 auf Seite 164 und Tabelle 8.6 auf Seite 165 geben eine Übersicht über die verwendeten physikalischen Körper und Gelenke und deren Namen. Abbildung 8.11 auf der nächsten Seite zeigt deren Anordnung.



(a) Körper



(b) Gelenke

Abbildung 8.11: Körper und Gelenke des Quadruped

Körper	**Eigenschaften**	
B:head	Kopf	
	Form:	Kugel
	Dimension:	Durchmesser: 0,5m
	Masse	1kg
B:body_f	vorderer Körperteil	
	Form:	abgerundeter Zylinder
	Dimension:	Durchmesser: 0,4m
		Länge: 0,8m
	Masse	10kg
B:body_b	hinterer Körperteil	
	Form:	abgerundeter Zylinder
	Dimension:	Durchmesser: 0,5m
		Länge: 0,8m
	Masse	10kg
B:upper_leg_fl/fr/bl/br	Oberschenkel links/rechts/vorne/hinten	
	Form:	abgerundeter Zylinder
	Dimension:	Durchmesser: 0,15m
		Länge: 0,8m
	Masse:	1kg
B:lower_leg_fl/fr/bl/br	Unterschenkel links/rechts/vorne/hinten	
	Form:	abgerundeter Zylinder
	Dimension:	Durchmesser: 0,1m
		Länge: 0,6m
	Masse:	1kg

Tabelle 8.5: Eigenschaften der physikalischen Körper des Quadruped

Gelenk	Eigenschaften			
J:waist	„Taillen-Gelenk“			
	Typ:	Kreuzgelenk		
	Achse 1:	+Y-Achse	Achse 2:	+Z-Achse
	- Begrenzung:	$[-40° \dots +40°]$	- Begrenzung:	$[-40° \dots +40°]$
	- Drehmoment:	500Nm	- Drehmoment:	500Nm
J:shoulder_l	Schultergelenk links			
	Typ:	Kreuzgelenk		
	Achse 1:	+Y-Achse	Achse 2:	-X/-Z-Achse
	- Begrenzung:	$[-65° \dots +20°]$	- Begrenzung:	$[-45° \dots +45°]$
	- Drehmoment:	500Nm	- Drehmoment:	500Nm
J:shoulder_r	Schultergelenk rechts			
	Typ:	Kreuzgelenk		
	Achse 1:	-Y-Achse	Achse 2:	+X/-Z-Achse
	- Begrenzung:	$[-65° \dots +20°]$	- Begrenzung:	$[-45° \dots +45°]$
J:hip_l	Hüftgelenk links			
	Typ:	Kreuzgelenk		
	Achse 1:	+Y-Achse	Achse 2:	-X/+Z-Achse
	- Begrenzung:	$[-20° \dots +65°]$	- Begrenzung:	$[-45° \dots +45°]$
J:hip_r	Hüftgelenk rechts			
	Typ:	Kreuzgelenk		
	Achse 1:	-Y-Achse	Achse 2:	+X/+Z-Achse
	- Begrenzung:	$[-20° \dots +65°]$	- Begrenzung:	$[-45° \dots +45°]$
J:knee_fl	Kniegelenk vorne links			
	Typ:	Scharniergelenk		
	Achse 1:	-X/-Z-Achse		
	- Begrenzung:	$[-45° \dots +45°]$		
	- Drehmoment:	200Nm		
J:knee_fr	Kniegelenk vorne rechts			
	Typ:	Scharniergelenk		
	Achse 1:	X/-Z-Achse		
J:knee_bl	Kniegelenk hinten links			
	Typ:	Scharniergelenk		
	Achse 1:	-X/Z-Achse		
J:knee_br	Kniegelenk hinten rechts			
	Typ:	Scharniergelenk		
	Achse 1:	+X/Z-Achse		

Tabelle 8.6: Eigenschaften der physikalischen Gelenke des Quadruped

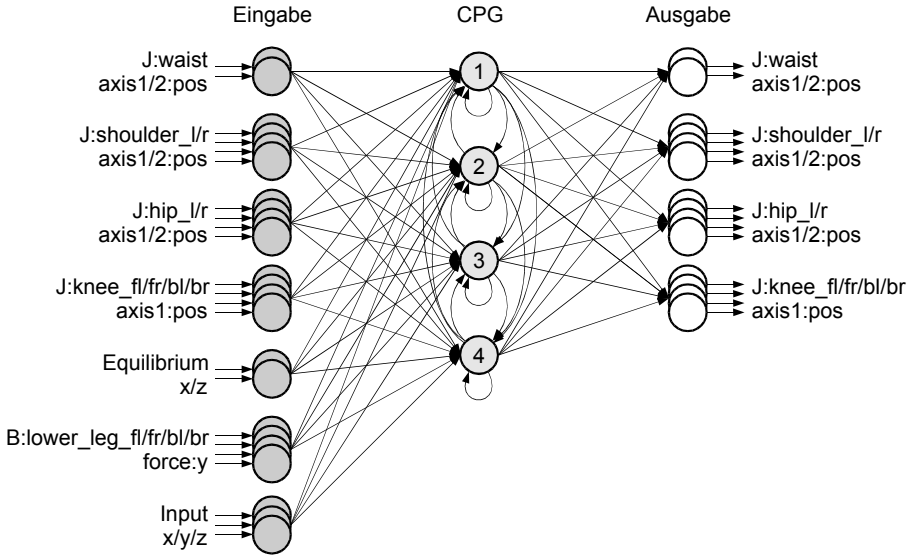


Abbildung 8.12: Neuronales Netzwerk des Quadruped

Das verwendete neuronale Netzwerk wird in Abbildung 8.12 dargestellt. Es ist prinzipell identisch zu dem Netzwerk des Mono- und Biped aufgebaut.
Die Eingabeschicht nimmt folgende Informationen auf:

- **J:waist:axis1/2:pos**
  Die aktuellen Positionen der beiden Achsen des Gelenks, welches die beiden Körperteile verbindet.

- **J:shoulder_l/r:axis1/2:pos**
  Die aktuellen Positionen der beiden Achsen des linken und rechten Schultergelenks.

- **J:hip_l/r:axis1/2:pos**
  Die aktuellen Positionen der beiden Achsen des linken und rechten Hüftgelenks.

- **J:knee_fl/fr:axis1:pos**
  Die aktuellen Positionen des linken und rechten Kniegelenks der Vorderbeine.

- **J:knee_bl/br:axis1:pos**
  Die aktuellen Positionen des linken und rechten Kniegelenks der Hinterbeine.

- **B:lower_leg_fl/fr:force:y**
  Die Kraft, welche in Y-Richtung auf den linken bzw. rechten Unterschenkel der Vorderbeine einwirkt.

- **B:lower_leg_bl/br:force:y**
  Die Kraft, welche in Y-Richtung auf den linken bzw. rechten hinteren Unterschenkel der Hinterbeine einwirkt.

- **Equilibrium:x/z**
  Die Orientierung des Kopfes in X- und Z-Richtung des Quadruped (Gleichgewicht). Diese Werte sind 0, wenn sich das Quadruped exakt im Gleichgewicht befindet, $0 < x < 1$, wenn das Quadruped nach rechts (X-Achse) bzw. nach hinten (Z-Achse) kippt und $-1 < x < 0$, wenn das Quadruped nach links (X-Achse) bzw. nach vorne (Z-Achse) kippt.

- **Input:x/y/z**
  Diese Eingabewerte dienen zur Steuerung des Quadruped in X-, Y- und Z-Richtung. Ein positiver Wert steht für eine Bewegung in die positive Richtung der entsprechenden Achse in m/s, ein negativer Wert für eine entgegengesetzte Bewegung.

Die mittlere Schicht stellt den Mustergenerator dar. Da beim Quadruped im Gegensatz zu Mono- und Biped vier Beine zu bewegen sind, wurde ein aus vier Neuronen aufgebauter CPG gewählt.
Die Ausgabeschicht steuert wie beim Monoped die Sollwinkel der Gelenke.
Durch evolutionären Algorithmen werden die Verbindungsgewichte und die Zeitkonstanten der Neuronen der mittleren Schicht eingestellt.

8.4 Implementierung eines Evolutionsalgorithmus'

Bei der Implementierung des Evolutionsalgorithmus' wurde Wert auf maximale Flexibilität und Erweiterbarkeit gelegt. Durch Erweiterung der Grundklassen (siehe Abschnitt 7.12.2 auf Seite 128) mit spezialisierten Klassen lassen sich viele Anwendungsfälle abdecken.
Zum Test der grundlegenden Funktionen der Bibliothek wurde ein einfaches Testszenario entworfen. Der evolutionäre Algorithmus sollte das Maximum der Funktionen

$$f(x, y) = 2 - (x^2 - 2\sin(2\pi x)) - (y^2 - 2\cos(2\pi y)) \tag{8.1}$$

(siehe Abbildung 8.13) finden.

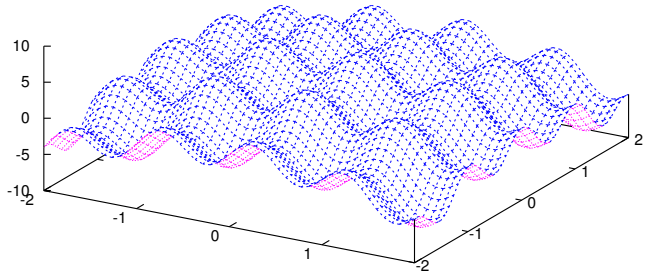
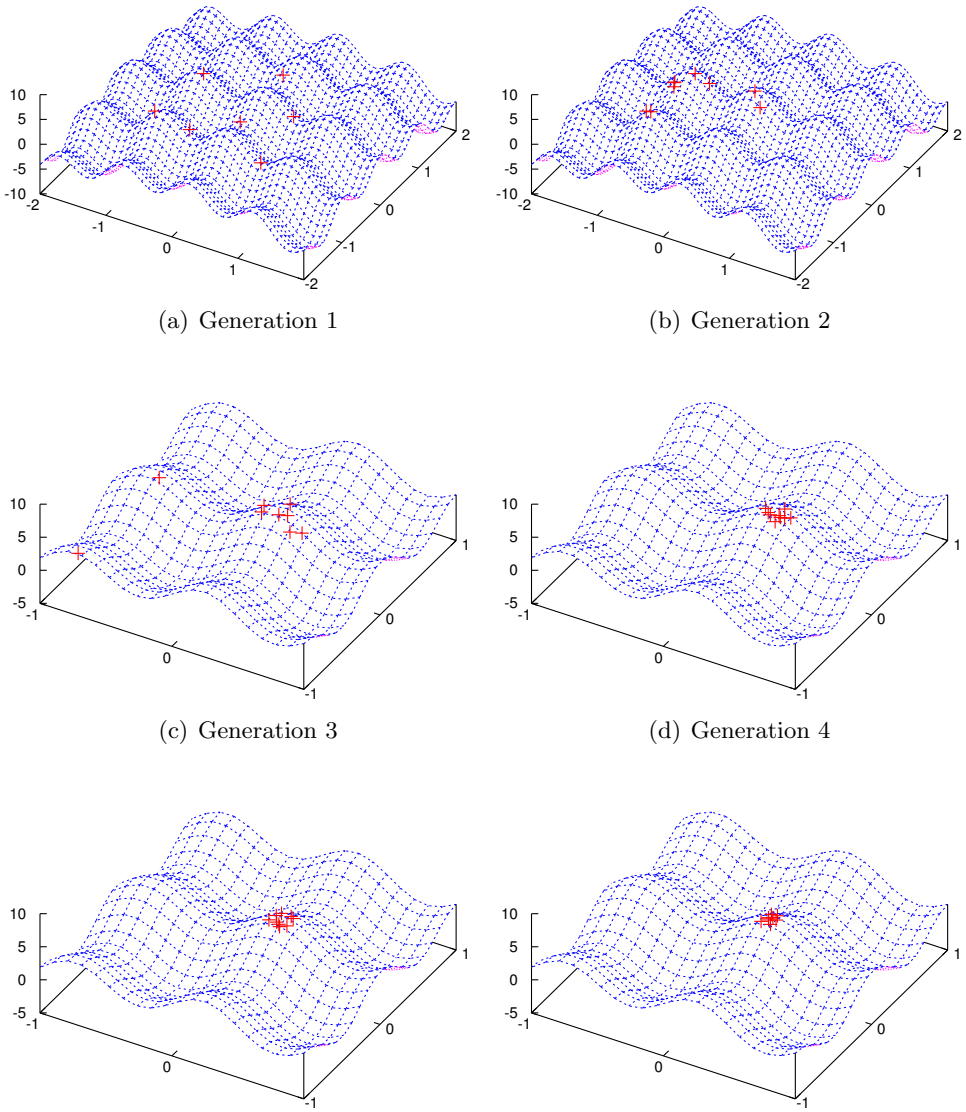


Abbildung 8.13: Testfunktion des Evolutionsalgorithmus'

Dazu wurde eine Populationsgröße von 10 Individuen verwendet. Abbildung 8.14 auf der nächsten Seite zeigt den Ablauf des Algorithmus, welcher bereits nach fünf Generationen die optimale Lösung gefunden hatte. Wie man in den Abbildungen sieht, sammeln sich die Individuen relativ schnell auf den lokalen Maxima der Funktion. In Generation 3 haben sich drei Maxima herausgebildet. Bereits in der vierten Generation sind keine Individuen mehr bei den umliegenden Maxima zu finden.
Abbildung 8.15(a) auf der nächsten Seite zeigt den Verlauf des Evolutionsvorgangs. Auf der X-Achse ist die Nummer der Generation abgetragen und auf der Y-Achse die mittlere Fitness der Population.
Abbildung 8.15(b) auf der nächsten Seite bietet einen Überblick über die Fitness jedes Individuums im Verlauf des Evolutionsvorgangs. Auf der X-Achse ist die Nummer der Generation abgetragen, auf der Y-Achse die Nummer des Individuums. Der Farbwert entspricht der Fitness des Individuums, welcher an der Skala rechts im Bild abzulesen ist.

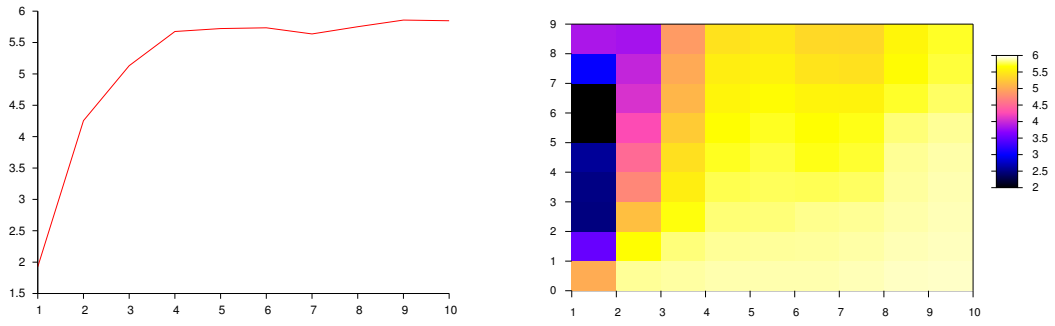


(a) Generation 1 (b) Generation 2

(c) Generation 3 (d) Generation 4

(e) Generation 5 (f) Generation 10

Abbildung 8.14: Ablauf der Testfunktion des Evolutionsalgorithmus'



(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.15: Fortschrittsdiagramm des Evolutionsalgorithmus'

8.5 Evolution der Bewegung virtueller Charaktere

8.5.1 Überlegungen

Folgende Überlegungen wurden zu Beginn und während der Evolutionsläufe angestellt:

8.5.1.1 Struktur des neuronalen Netzwerks

Zu Beginn der Arbeiten an der Thesis wurde noch davon ausgegangen, dass sich zeitdiskrete Netzwerke zur Lösung der Aufgabe eignen. Es wurde überlegt, ob es ausreichend sei, lediglich den aktuellen Zeitzustand des virtuellen Charakters als Eingabemenge des Neztwerks zu verwenden oder ob zeitlich zurückliegende Zustände ebenfalls benötigt werden. Wenn ja, wie viele Zustände müssen dies sein und in welchen zeitlichen Abständen sollten sie liegen?

[Künzell, 1996, Kap. 3.2.1] beschreibt die Architektur eines rekurrenten neuronalen Netzwerks nach Jordan zur Modellierung von Bewegungssequenzen (siehe Abschnitt 4.4.5.1 auf Seite 51). Da dieses Netzwerk nicht mit direkt eingegebenen Informationen aus der Vergangenheit arbeitet, sondern mittels Rückführung der Ausgangssignale und mit CPG-ähnlichen Strukturen, wurde der Ansatz, zeitlich zurückliegende Informationen einzugeben, zunächst zurückgestellt. Einige anfängliche Evolutionsläufe brachten lediglich entweder reflexartige Bewegungen hervor, welche keine rhythmischen Anteile erkennen ließen, oder Bewegungsvorgaben, welche so schnelle Vorzeichenwechsel aufwiesen, dass die physikalische Simulation nicht in der Lage war, diese Muster auszuführen. Dies führte zunächst zu einer Untersuchung der Fähigkeit des verwendeten neuronalen Netzwerks, Muster zu erzeugen. Ein minimales Netzwerk nach Wilson-Cowan (siehe Abschnitt 4.5.1 auf Seite 54) brachte nur von Schritt zu Schritt alternierende Muster hervor, unabhängig davon, wie die Gewichte eingestellt wurden. Diese Beobachtung deckte sich mit den alternierenden Mustern des Netzwerks, welches durch den evolutionären Algorithmus entwickelt worden war. Nach diesen Fehlschlägen wurde das zeitdiskrete neuronale Netzwerk durch das zeitkontinuierliche Netzwerk (CTRNN) ausgetauscht, was im Verlauf der weiteren Versuche zu besseren Ergebnissen führte.

Dieser Entscheidung fiel auch ein anderes Konzept zum Opfer, welches noch den Ansatz verfolgt hatte, das neuronale Netzwerk mit Lernverfahren anstelle evolutionärer Algorithmen zu trainieren. Prinzipiell sollte das Netzwerk nicht nur eine Ausgabeschicht für die Steuersignale der Gelenke besitzen, sondern auch eine nachgeschaltete Schicht für die Vorhersage der daraus resultierenden Bewegungen. Durch den Vergleich dieser Vorhersage mit der realen Bewegung ließe sich ein Fehlervektor berechnen, welcher für den Backpropagation-Algorithmus geeignet wäre. Anfängliche Versuche zeigten, dass dieser Ansatz lediglich den Vorhersageteil des Netzwerks trainiert, jedoch nicht dazu geeignet ist, Bewegungsvorgänge entstehen zu lassen. Der virtuelle Charakter blieb grundsätzlich passiv, konnte sein passives Verhalten jedoch gut im Voraus berechnen.

Eine weitere Fragestellung war, für welchen virtuellen Charakter sich welche Anzahl und Verschaltung von Neuronen als CPG am besten eignet. [Paul und Bongard, 2001b; Paul, 2003, 2004; Paul und Bongard, 2001a] verwenden hauptsächlich 3er CPGs in ihren neuronalen Netzwerken. Daher wurden in den ersten Evolutionsläufen ebenfalls 3er CPGs verwendet.
Ergänzend wurden später zusätzlich Evolutionsläufe durchgeführt, bei welchen die Generatoren durch 2er- und 4er-Varianten ersetzt wurden. Die einzige Verbesserung erfolgte beim Biped durch den Einsatz eines 2er-CPGs. Alle anderen Kombinationen führten zu schlechteren Ergebnissen.

8.5.1.2 Initialisierungsphase

[Mathayomchan und Beer, 2002] stellen in ihrer Arbeit fest, dass evolutionäre Algorithmen, welche CTRNNs auf eine bestimmte Aufgabe hin entwickeln sollen, wesentlich schneller eine optimale Lösung finden, wenn die Neuronen so eingestellt sind, dass sie nicht ständig von Sättigungszustand zu Sättigungszustand pendeln, sondern sich in den optimalen Arbeitspunkten ihrer Aktivierungsfunktionen befinden. Evolutionsläufe mit zufällig initialisierten Werten konvergierten wesentlich langsamer zur optimalen Lösung. Die Autoren entwickelten daraus eine Formel, welche aus den Verbindungsgewichten die optimalen Bias-Werte berechnet, um CTRNNs für evolutionäre Algorithmen optimal voreinzustellen. Angelehnt an diese Ergebnisse wurden die Gewichte und Zeitkonstanten der CPG-Neuronen ebenfalls voreingestellt, so dass sie von Beginn an schwingungsfähig waren. Alle anderen Gewichte und Bias-Werte des neuronalen Netzwerks wurden zu Null gesetzt.
Zum Abschluss der Arbeit wurden zum Vergleich Evolutionsläufe mit zufällig initialisierten Netzwerken durchgeführt.

8.5.1.3 Umwandlung Genotyp ↔ Phänotyp

Eine weitere Frage betraf die Umwandlung der Verbindungsgewichts- und Bias-Werte des neuronalen Netzwerks (Phänotyp) in Werte des Genotyps. Das Netzwerk arbeitet primär mit Eingabe- und Ausgabewerten im Bereich von $[-1 \ldots +1]$, kann allerdings innerhalb der verdeckten Schichten und CPGs auch mit höheren Werten arbeiten. In der Literatur findet man häufig Gewichtswerte um ± 6. Dies führte zunächst zu einer Unterscheidung von Gewichts- und Bias-Werten, welche innerhalb der rekurrenten Verbindungen der CPG-Neuronen verwendet wurden, im Gegensatz zu Werten von und zu diesen Neuronen. Anfänglich wurden diese CPG-Werte mit 5,0 skaliert, alle anderen Werte mit 2,0. Zum Ende der Thesis hin wurden beide Skalierungswerte probehalber gemeinsam auf 5,0 gesetzt, ohne jedoch einen Unterschied in den Ergebnissen zu beobachten (siehe Abschnitt 8.5.3 auf Seite 181).
Zusätzlich wurde der Bias-Wert eines Neurons mit der Anzahl der eingehenden Verbindungen skaliert. Wenn ein Neuron z.B. fünf Eingänge hat, so kann es von dort im ungünstigsten Fall (alle Gewichte $\pm 1{,}0$) eine Eingangssumme von $\pm 5{,}0$ empfangen. Nur mit einem ausrei-

chend hoch eingestellten Bias kann dieses Neuron auf diesen Extremwerten noch sinnvolle Ausgabewerte erzeugen.

8.5.1.4 Evolutionsoperatoren und -parameter

Ein weiterer Gegenstand der Untersuchungen waren die Parameter und Operatoren des evolutionären Algorithmus'. Leider reichte die Zeit nicht für eine umfassende Analyse aller Möglichkeiten. So können lediglich Vermutungen geäußert werden, was die Effektivität und die Geschwindigkeit der Konvergenz zu optimalen Werten hin betrifft.
Ein wichtiger Faktor war die Größe der Population. Es wurde dahingehend entschieden, diesen Faktor an die Dimension des zu untersuchenden Problems anzupassen. Bei dem Monoped (Genomgröße 62) und Biped (Genomgröße 93) wurde eine Populationsgröße von 100 verwendet, beim Quadruped (Genomgröße 206) hingegen 250.
Die Variation von Mutationsrate und Mutationswahrscheinlichkeit, die Anzahl der Eltern pro Nachkomme und die Anzahl der Nachkommen hatten auf die Ergebnisse weniger Einfluss als zunächst angenommen. Im Wesentlichen unterschieden sich die Verläufe der Kurven der mittleren Populationsfitness hinsichtlich ihrer Stetigkeit und Unruhe.
Verwendet wurden folgende Operatoren und Parameter:

- **Algorithmus und Genotyp** (siehe Abschnitt 5.2.3 auf Seite 61 und Abschnitt 5.3.1 auf Seite 72)
  Evolutionäre Strategien mit reellwertigem Genotyp (Fließkomma-Arrays)

- **Populationsgröße und Eltern/Nachkommen-Verhältnis** (siehe Abschnitt 5.3.1.1 auf Seite 72)
  Monoped, Biped: 100/1+200, Quadruped: 250/1+500.

- **Fitnesszuweisung** (siehe Abschnitt 5.2.4 auf Seite 61)
  Proportionale Fitnesszuweisung

- **Initialisierung** (siehe Abschnitt 5.2.5 auf Seite 63)
  Nicht-zufällige Initialisierung durch Voreinstellung der CPGs bzw. zufällige Initialisierung (siehe Abschnitt 8.5.1.2 auf der vorherigen Seite)

- **Selektion** (siehe Abschnitt 5.2.6 auf Seite 63)
  Turnierselektion mit einer Turniergröße von 5.

- **Rekombination** (siehe Abschnitt 5.2.7 auf Seite 65)
  Bei der Evolution der virtuellen Charaktere wurde keine Rekombination verwendet, da die „Vermischung“ der Gewichtswerte zweier neuronaler Netzwerke, welche identische Funktionalitäten durch unterschiedliche Verbindungsgewichte codieren, zu einem Informationsverlust führt. Dieses Phänomen nennt man „competing convention“ [Nissen, 1997, S. 287]. Es tritt bevorzugt bei der Anwendung genetischer Algorithmen auf neuronalen Netzwerken auf.

Bei der Implementierung der Testfunktion hingegen wurde sowohl diskrete als auch intermediäre Rekombination verwendet.

- **Mutation** (siehe Abschnitt 5.2.8 auf Seite 68)
  Zufällige Mutation mit adaptiver Schrittweitenanpassung über die 1/5-Erfolgsregel und 10% Mutationswahrscheinlichkeit
- **Wiedereinfügen** (siehe Abschnitt 5.2.9 auf Seite 69)
  Elitäres Wiedereinfügen
- **Migration** (siehe Abschnitt 5.2.10 auf Seite 70)
  Nicht verwendet, da keine Populationskonzepte verfolgt wurden
- **Abbruchkriterium** (siehe Abschnitt 5.2.11 auf Seite 70)
  Laufendes Mittel

8.5.1.5 Fitnessberechnung

Die Berechnung der Fitness stellte sich als das größte Problem dar. Es zeigte sich, dass ein evolutionärer Algorithmus den Suchraum so gründlich abdeckt, dass jegliche Lücke in der Interpretationsmöglichkeit eines guten Fitnesswerts genutzt wird. Diese Lücken entstanden z.B. durch

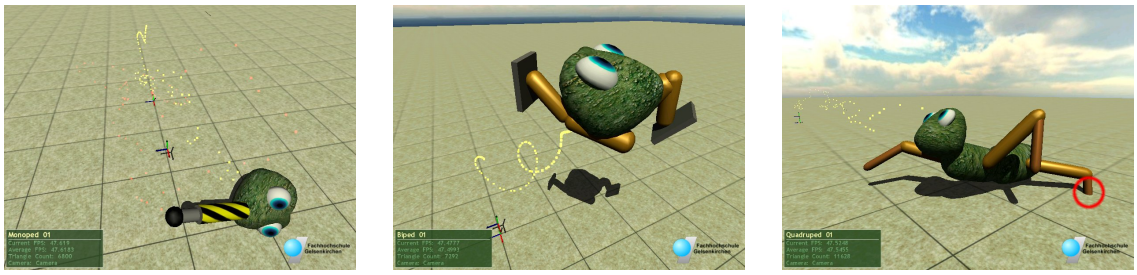
1. Fehler in der physikalischen Simulation oder
2. mathematisch „fragwürdige“ Umsetzungen eines umgangssprachlich definierten Kriteriums.

Die Stabilität einer physikalischen Simulation ist stark abhängig von den Simulationsparametern (siehe Abschnitt 8.2.1.2 auf Seite 150). Während der Evolutionsläufe fanden sich zwei Schwachstellen:

1. **Instabilität der Gelenke**
   Unter bestimmten Umständen schaffte es das Biped, die Gelenke so zu bewegen, dass die Simulation instabil wurde und das Biped sich in sich selbst „verknotete“. Die Physik-Engine versuchte mit erheblichen Kräften, die Gelenk-Constraints wieder einzuhalten, was dazu führte, dass das Biped in beliebige Richtungen fortgeschleudert wurde (siehe Abbildung 8.16(b) auf der nächsten Seite). Dies ergab unter dem Strich einen hervorragenden Fitnesswert, da dieser sich zu Beginn der Versuchsreihen primär aus der zurückgelegten Strecke errechnete. Letzlich endete der Evolutionslauf mit einer hohen Anzahl von Individuen, welche es in kürzester Zeit vollbrachten, diesen Simulationsfehler „auszunutzen“, um einen hohen Fitnesswert zu erlangen.

   Diesem Problem wurde mit zweierlei Maßnahmen begegnet. Erstens wurden Maximalgeschwindigkeiten innerhalb der physikalischen Simulation künstlich begrenzt. Zweitens wurde eine maximale Höhe des Kopfes des Biped festgelegt, bei deren Überschreitung die Simulation mit dem Fitnesswert 0 abgebrochen wurde.

2. **Instabilität der Kollisionsbehandlung**
   Ein ähnliches Problem tauchte bei der Evolution des Quadruped auf. Bei schnellen Tretbewegungen der Beine drangen diese aufgrund eines Fehlers in der Kollisionsbehandlung in den Boden ein und wurden wie von einer Art unterirdischem Fluss fortgezogen (siehe Abbildung 8.16(c)). Auch hier erlangten die Individuen den höchsten Fitnesswert, welche diesen Fehler am schnellsten ausnutzten. Die Gegenmaßnahme war auch in diesem Fall eine Begrenzung der Kopfhöhe, allerdings auf einen Minimalwert, welcher immer dann unterschritten wurde, wenn die Figur wieder „im Boden versunken“ war.



(a) Biped (b) Biped (c) Quadruped

Abbildung 8.16: Fehler in der physikalischen Simulation (siehe Farbtafel D.14 auf Seite 245)

Bestandteile der Fitnessfunktion

Die im Verlauf der Thesis entwickelte Fitnessfunktion addiert auf den Zielfunktionswert in jedem Simulationsschritt einen Wert auf, welcher sich aus mehreren Faktoren berechnet:

1. **Abstand zur Sollposition**
   Der Wert d ist ein Maß für die Distanz der Position des Kopfes $\vec{x}_{\text{Kopf}}(t)$ zur Sollposition $\vec{x}_{\text{Soll}}(t)$. Dabei entspricht d im besten Fall dem Betrag des Zeitintervalls Δt der Simulation. Je weiter der Kopf sich von einem Umkreis des Radius r von der Sollposition entfernt, desto geringer wird d.

$$d(t) = \frac{\Delta t}{\max(0, |\vec{x}_{\text{Kopf}}(t) - \vec{x}_{\text{Soll}}(t)| - r)} \tag{8.2}$$

2. **Energieterm**
   Der Energieterm e berechnet sich aus der Summe der Quadrate der Beschleunigungen a aller N_J Gelenke J_i, $i \in (1 \dots N_J)$. Grundlage für diese Berechnung ist das sogenannte „**minimum torque change model**“. Dieses besagt, dass Bewegungsabläufe am natürlichsten wirken, wenn sie mit einem Minimum an Änderungen der Gelenk-

beschleunigung $a(t)_{J_i}$ integriert über die Zeit geschehen [Künzell, 1996, Kap. 1.3.1]. Mit einem Logarithmus wird verhindert, dass der Term zu schnell wächst.

$$e(t) = \log\left(1 + \sum_{i=1}^{N_J} a(t)_{J_i}^2\right) \tag{8.3}$$

3. **Strafzustände**
   Bestimmte Bewegungen, Positionen oder Verhaltensweisen werden bestraft, indem bei deren Auftreten die Addition auf den Zielfunktionswert unterbunden wird. In diesem Fall wird der Term p auf 0 gesetzt. Im Normalfall ist $p = 1$.

$$p(t) = \begin{cases} 1 & \text{wenn kein Strafzustand} \\ 0 & \text{wenn Strafzustand} \end{cases} \tag{8.4}$$

4. **Anzahl Schritte**
   Durch die Kräfte, welche auf die Füße der virtuellen Charaktere wirken, kann festgestellt werden, ob ein Schritt erfolgt ist. Dazu wird ein Schwellwert festgelegt, ab welchem ein Fuß als am Boden befindlich gilt. Die Anzahl der Füße mit Bodenkontakt wird summiert. Bei einer Änderung dieser Summe wird ein Schritt zu der Schrittsumme s hinzugezählt. Die Schrittsumme wird auf eine maximale Zahl s_{max} begrenzt, um zu verhindern, dass ein virtueller Charakter durch „Flattern“ mit den Füßen unverhältnismäßig hohe Fitnesswerte erlangt.

Zu jedem Simulationszeitpunkt t wird ein Anteil des Zielfunktionswertes $Z(t)$ mit

$$Z(t) = \frac{d(t) \cdot p(t)}{1 + e(t) \cdot \eta} \tag{8.5}$$

berechnet. Je höher die aufgebrachte Energie, desto geringer ist der Anteil des aktuellen Simulationsschritts am Gesamtwert. Dabei ist η ein Faktor, mit welchem der Einfluss des Energieterms auf den Zielfunktionswert eingestellt werden kann. Für diesen Faktor hat sich der Wert 0,1 als geeignet erwiesen.
Der gesamte Zielfunktionswert Z und damit auch direkt die Fitness des virtuellen Charakters ergibt sich durch Aufsummierung der Anteile vom Zeitpunkt des Simulationsbeginns t_{Beginn} bis zum Simulationsende t_{Ende}. Zum Schluss wird diese Summe mit der begrenzten Schrittsumme multipliziert, um Individuen zu belohnen, welche mehr Schritte als andere geschafft haben.

$$Z = \min(s, s_{max}) \cdot \sum_{t=t_{\text{Beginn}}}^{t_{\text{Ende}}} Z(t). \tag{8.6}$$

8.5.2 Monoped

Evolutionslauf 1. . . 8

Die ersten Evolutionsläufe des Monoped dienten zunächst zur Beseitigung von Problemen und Konzeptionsfehlern. Um Rechenzeit zu sparen, wurde ein Individuum nur einmal evaluiert und die resultierende Fitness danach gespeichert. Wenn ein Simulationslauf für ein Individuum äußerst günstig verlief, so wies dieses für den Rest des Evolutionslaufes eine sehr hohe Fitness auf und hielt sich sehr lange in der Population. Wurden solche Individuen extrahiert[6] und mehrfach simuliert, so war das Ergebnis eher ernüchternd, da diese Individuen nicht besser waren als andere mit schlechterer Fitness.
Letzlich ergab dieser Evolutionslauf auch nach 20000 Generationen nur Monoped, welche sich einfach fallen ließen. Die steigende Fitnesskurve der Population (siehe Abbildung 8.17) ergab sich aus der Tatsache, dass immer mehr Individuen gelernt hatten sich *nach vorne* fallen zu lassen, um zumindest einen kleine Strecke in X-Richtung zurückzulegen (siehe Abbildung 8.23(a) auf Seite 179). Dass keine Sprungbewegungen des Monoped erfolgten, ist darauf zurückzuführen, dass dieser Evolutionslauf noch mit dem zeitdiskret rekurrenten neuronalen Netzwerk (DTRNN) erfolgte.

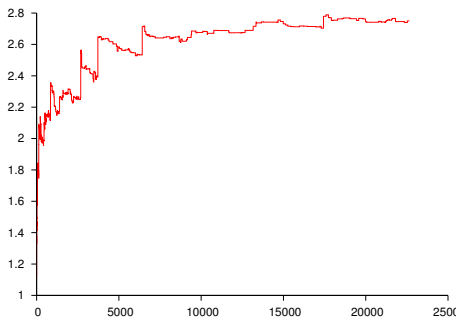


Abbildung 8.17: Monoped, Evolutionslauf 3

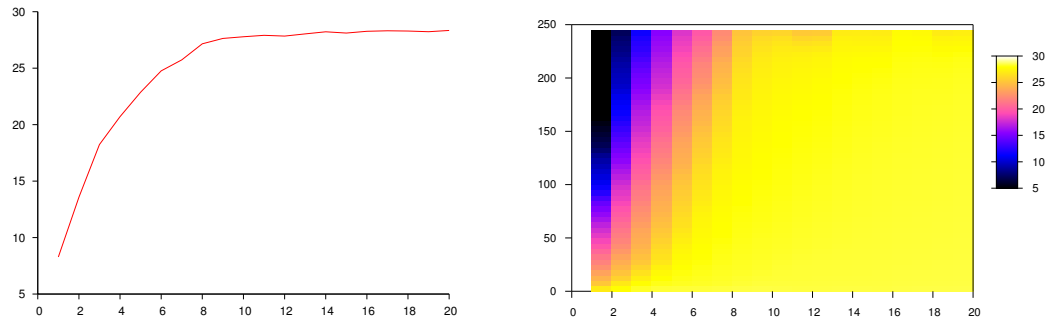
Evolutionslauf 9

Beim neunten Durchlauf (siehe Abbildung 8.18 auf der nächsten Seite) wurde das neuronale Netzwerk durch ein CTRNN ersetzt. Der Unterschied zu den vorherigen Durchläufen, welche bis zu 20000 Generationen berechnet haben, war immens. Schon nach 28 Generationen terminierte der Durchlauf automatisch. Das Ergebnis war ein Monoped, welches bis zu 6 Sprünge ausführte, bevor es instabil wurde (siehe Abbildung 8.23(b) auf Seite 179).

Evolutionslauf 10

Dieser Durchlauf baute auf den Parametern von Lauf 9 auf, begann den Evolutionsprozess jedoch mit vollständig zufällig initialisierten Individuen. Das nach 100 Generationen

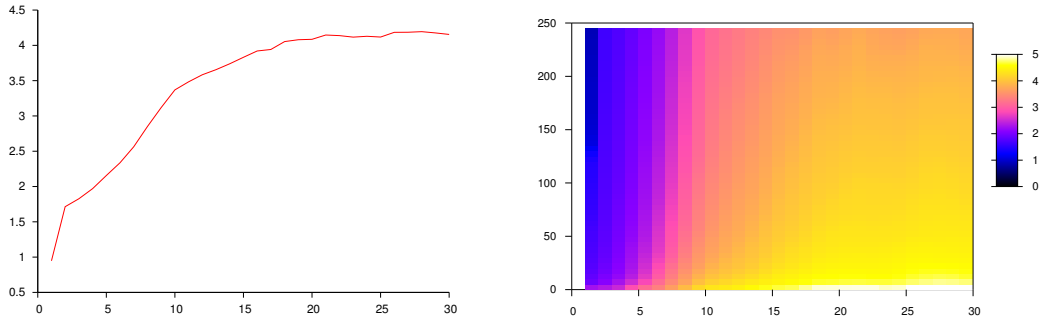
[6] Die Extraktion bezeichnet den Vorgang, aus dem Genotyp des Individuums das neuronale Netzwerk als Datei zu erzeugen, um es mit dem Simulationsprogramm „auf Herz und Nieren“ zu prüfen.



(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.18: Monoped, Evolutionslauf 9

entwickelte Individuum sprang lediglich einmal in die Luft und versuchte dabei, möglichst der Sollposition zu folgen und dabei mit dem Fuß mehrmals den Boden zu berühren (siehe Abbildung 8.19 und Abbildung 8.23(c) auf Seite 179).

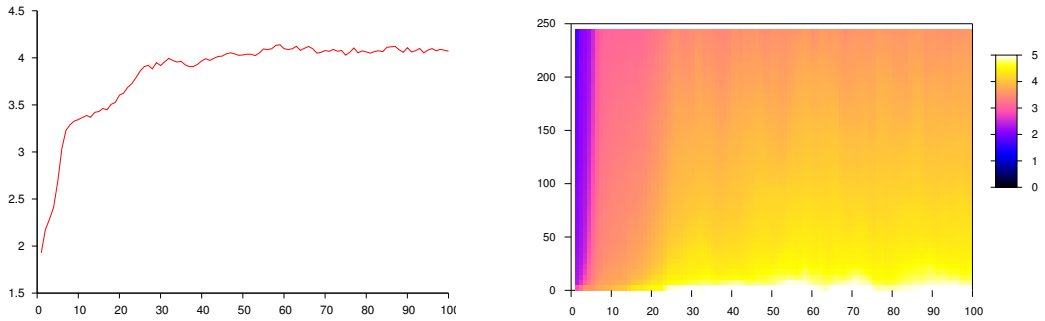


(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.19: Monoped, Evolutionslauf 10

Evolutionslauf 11

Für diesen Durchlauf wurde die mittlere Schicht des neuronalen Netzwerks durch einen 2-stufigen CPG ersetzt (siehe Abbildung 8.20 auf der nächsten Seite und Abbildung 8.23(d) auf Seite 179). Das Ergebnis war ein Monoped, welches durch schnelle Schwingvorgänge des CPG die Positionsregler des Hüftgelenks in eine Resonanzkatastrophe brachte. Die Steuersignale für die Hüftgelenke sind in Abbildung 8.21 auf der nächsten Seite). in den Zeilen „Hip Axis 1 Position out“ und „Hip Axis 2 Position out“ zu sehen. Der vollständig blaue Balken zeigt, dass die Signale schneller von einem Maximum zu einem Minimum pendelten, als dies in der Grafik dargestellt werden kann.
Durch die aus diesen Signalen resultierende Rotationsbewegung des Beins wurde das Monoped in eine beliebige Richtung geschleudert und erlangte so einen höheren Fitnesswert, als im Stillstand.



(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.20: Monoped, Evolutionslauf 11

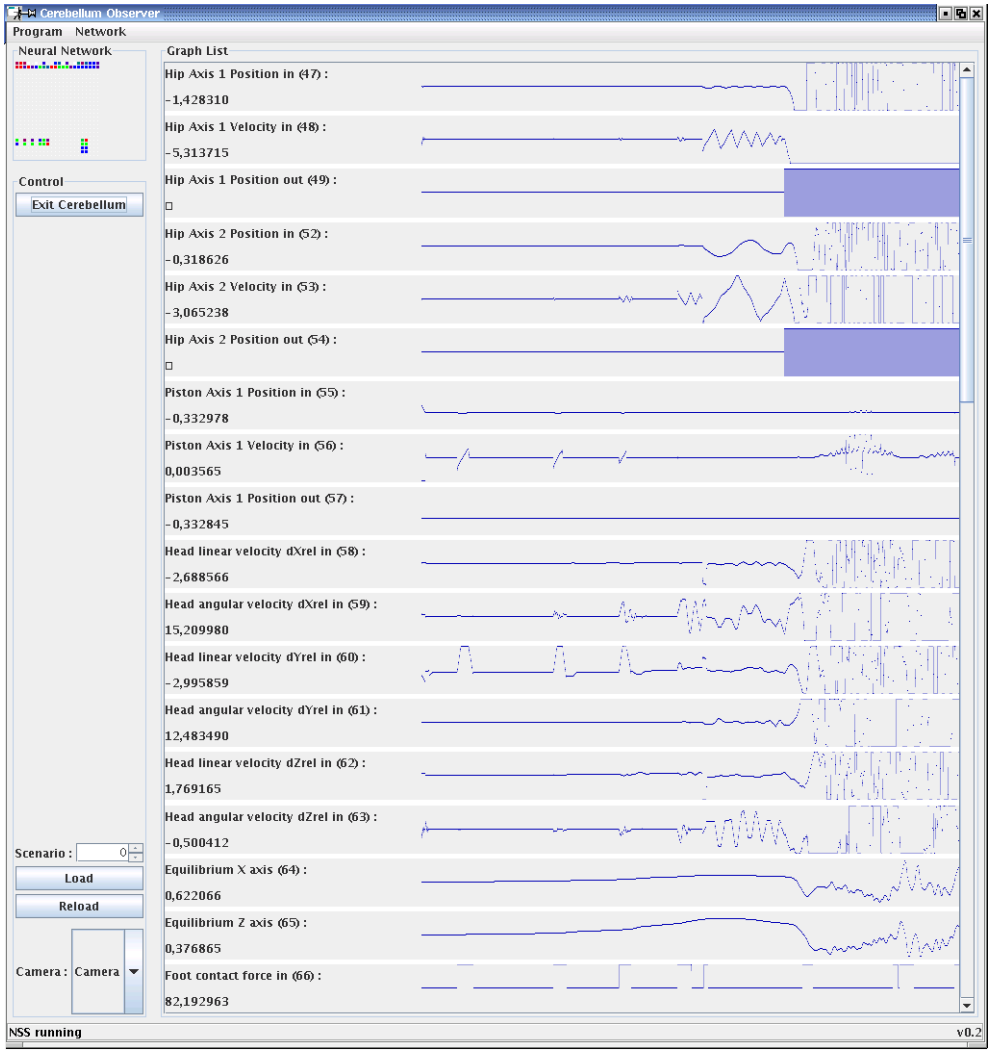
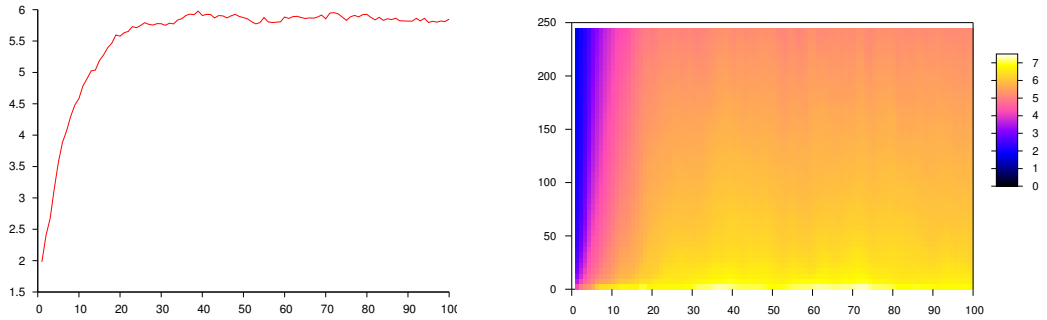


Abbildung 8.21: Resonanzkatastrophe bei Simulationslauf 11 des Monoped

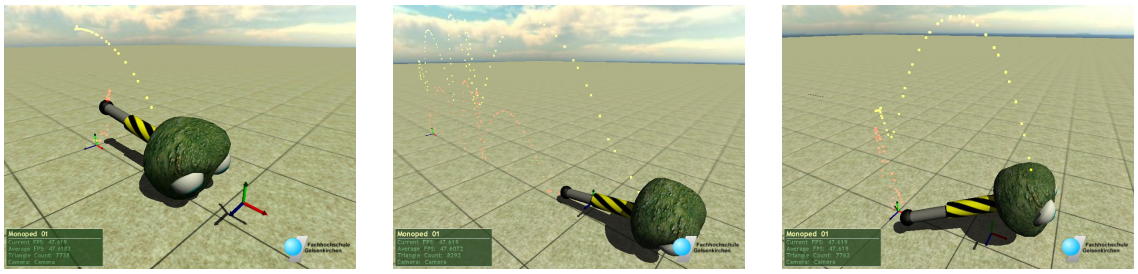
Evolutionslauf 12

Für diesen Durchlauf wurde die mittlere Schicht des neuronalen Netzwerks durch einen 4-stufigen CPG ersetzt (siehe Abbildung 8.22 und Abbildung 8.23(e)). Das resultierende Monoped vollführte wie das Exemplar aus Evolutionslauf 10 lediglich ein oder zwei Sprünge, bevor es ohne weitere Gegenmaßnahmen hinfiel.

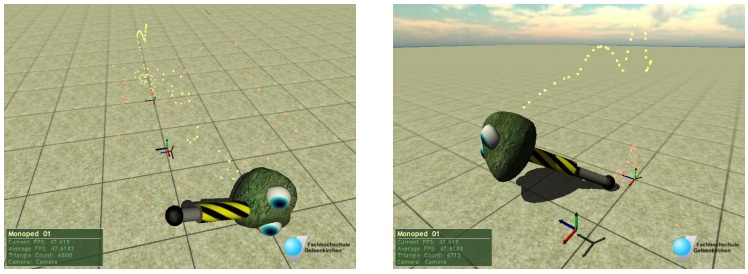


(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.22: Monoped, Evolutionslauf 12



(a) Evolutionslauf 3 (b) Evolutionslauf 9 (c) Evolutionslauf 10



(d) Evolutionslauf 11 (e) Evolutionslauf 12

Abbildung 8.23: Ergebnisse der Evolutionsläufe des Monoped (siehe Farbtafel D.15 auf Seite 247)

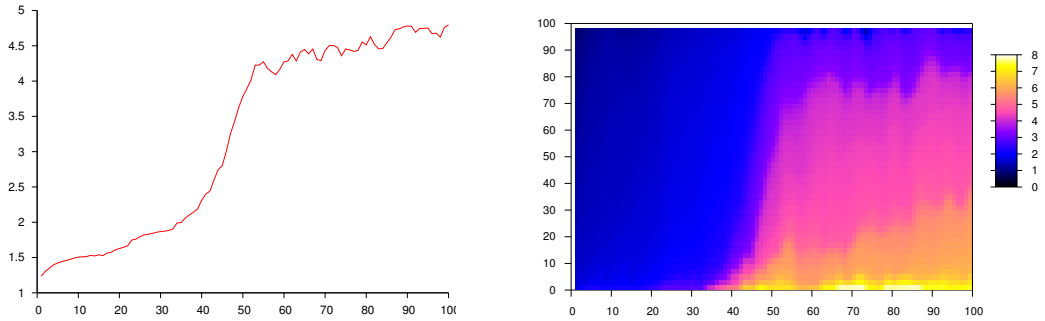
8.5.3 Biped

Evolutionslauf 1. . . 5

Auch beim Biped dienten die ersten Evolutionsläufe zur Beseitigung von Problemen und Konzeptionsfehlern.

Evolutionslauf 6

Der sechste Lauf führte zu einem unvorhergesehenen Ergebnis. Die Fitness wurde aus der Distanz zur Sollposition berechnet. Sobald der Kopf durch Sturz eine Mindesthöhe unterschritt, wurde der Simulationslauf abgebrochen.
Es entwickelte sich ein Individuum, welches es schaffte, durch einen Spagat für die Dauer der Simulationen stabil stehen zu bleiben und somit ein Maximum an Fitness zu erhalten (siehe Abbildung 8.24 und Abbildung 8.31(a) auf Seite 184).



(a) Mittlere Fitness

(b) Fitness der Individuen

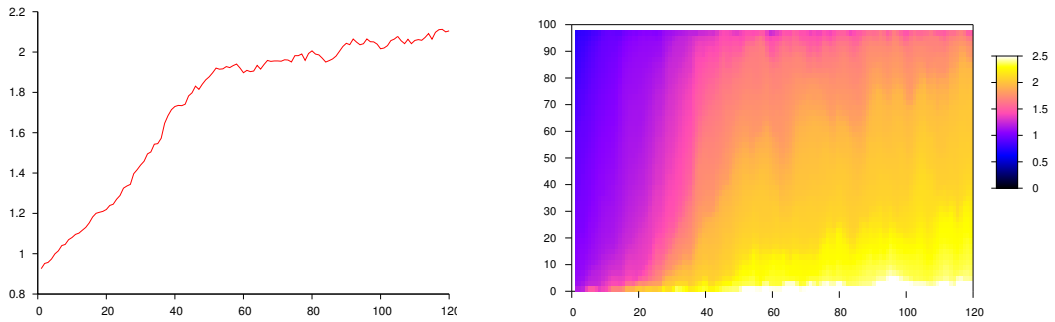
Abbildung 8.24: Biped, Evolutionslauf 6

Evolutionslauf 7

Um die Figur zu einer Bewegung zu zwingen, wurde ein Term in die Fitnessfunktion eingebaut, welcher so lange keinen Wert zur Fitness addiert, so lange beide Füße auf dem Boden sind.
Das Ergebnis dieses Evolutionslaufes (siehe Abbildung 8.25 auf der nächsten Seite) war ein Individuum, welches ein Bein rückwärts hochnahm, um dann der Sollposition „hinterherzufallen“ (siehe Abbildung 8.31(b) auf Seite 184).

Evolutionslauf 8. . . 19

Im Verlauf dieser Evolutionsläufe wurden zahlreiche Kombinationen von Evolutionsoperatoren und -Parametern ausprobiert, allerdings ohne sichtbare Unterschiede. Das Resultat blieben immer Individuen, welche sich „einfach nur fallen ließen“.

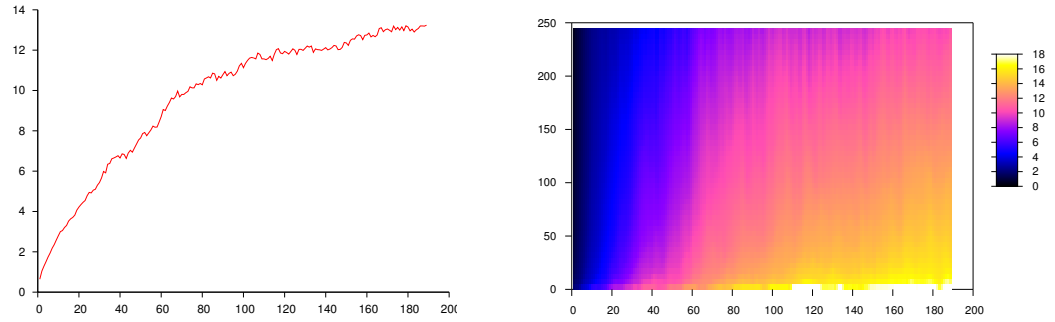


(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.25: Biped, Evolutionslauf 7

Evolutionslauf 20

Dieser Evolutionslauf brachte als Erster ein brauchbares Ergebnis hervor (siehe Abbildung 8.26). Das Individuum „humpelte“ auf den Fersen langsam aber sicher vorwärts (siehe Abbildung 8.31(c) auf Seite 184). Genauere Untersuchungen durch Abschalten der Schwerkraft zeigten, dass die Bewegungsmuster rein reflexbasiert waren. Der Fuß, welcher den Charakter vorwärts zog, führte diese Bewegung nur aus, wenn er mit dem Boden in Kontakt kam. In Schwerelosigkeit fand hingegen keine eigenständige Bewegung mehr statt.



(a) Mittlere Fitness (b) Fitness der Individuen

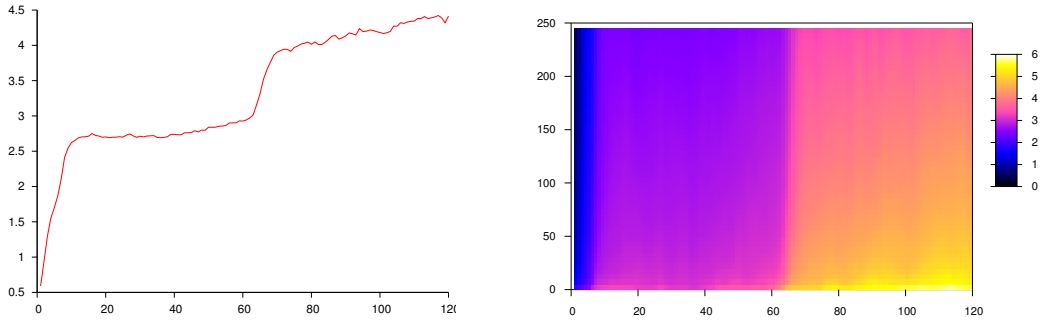
Abbildung 8.26: Biped, Evolutionslauf 20

Evolutionslauf 21 und 22

Experimente mit einer einheitlichen Skalierung aller Verbindungsgewichte und Bias-Werte mit 5 bei der Umwandlung von Genotyp in Phänotyp brachten keine wesentliche Verbesserung oder Verschlechterung.

Evolutionslauf 23

Dieser Durchlauf baute auf den Parametern von Lauf 20 auf, begann den Evolutionsprozess jedoch mit vollständig zufällig initialisierten Werten im Genotyp (siehe Abbildung 8.27). Das resultierende Individuum war instabiler und schaffte weniger Schritte als das Individuum aus Lauf 20 (siehe Abbildung 8.31(d) auf Seite 184).
Auffällig ist der Knick in der Fitnesskurve ab Generation 65. Hier schienen die ersten Individuen mehr als ein oder zwei Schritte geschafft zu haben.

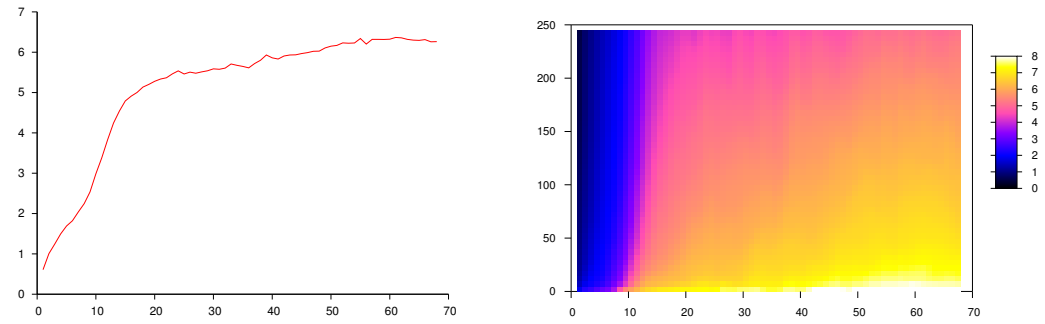


(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.27: Biped, Evolutionslauf 23

Evolutionslauf 24

Für diesen Durchlauf wurde die mittlere Schicht des neuronalen Netzwerks durch einen 2-stufigen CPG ersetzt (siehe Abbildung 8.28 und Abbildung 8.31(e) auf Seite 184). Die resultierende Fortbewegung war zwar nicht sehr stabil, sah dafür aber einer natürlichen Laufbewegung sehr viel ähnlicher als bei dem Ergebnis von Evolutionslauf 20. Ein weiterer Unterschied lag darin, dass auch nach Abschaltung der Schwerkraft die Bewegung der Beine fortgesetzt wurde.

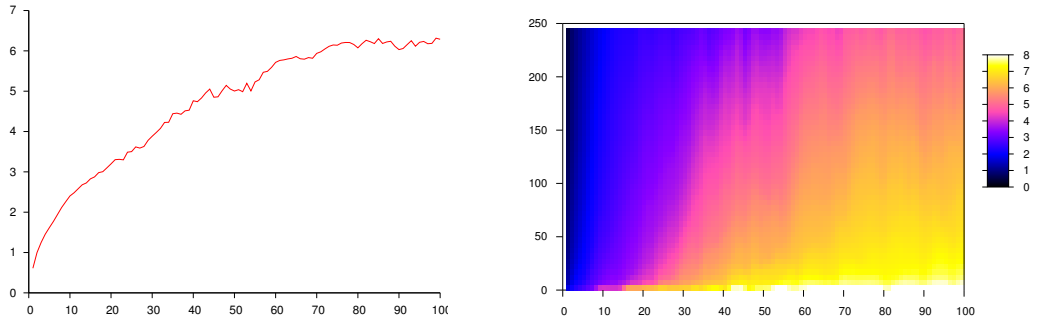


(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.28: Biped, Evolutionslauf 24

Evolutionslauf 25

Wie in Lauf 24 wurde auch für diesen Durchlauf die mittlere Schicht des neuronalen Netzwerks ersetzt, diesmal durch einen 4-stufigen CPG (siehe Abbildung 8.29 und Abbildung 8.31(f) auf der nächsten Seite). Der resultierende virtuelle Charakter ähnelte in seinem Verhalten wieder dem Charakter aus Evolutionslauf 20. Zusätzlich war er unbeweglicher, steifer und fiel leichter um.

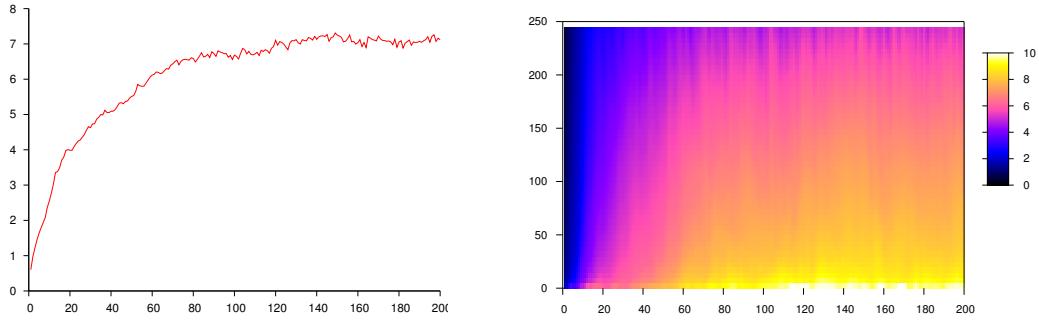


(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.29: Biped, Evolutionslauf 25

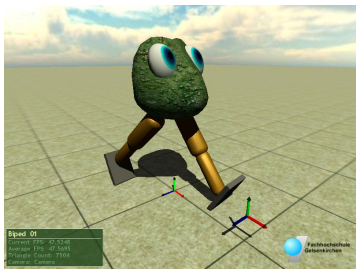
Evolutionslauf 26

Dieser Lauf diente zur Langzeitbeobachtung (siehe Abbildung 8.30). Das Abbruchkriterium wurde so eingestellt, dass die Population über 150 Generationen einen nahezu identischen Fitnesswert aufweisen musste. Das Ergebnis war ein reflexbasiertes Biped, welches eine erstaunlich stabile Fortbewegung aus den Fußgelenken heraus produzieren konnte (siehe Abbildung 8.31(g) auf der nächsten Seite).

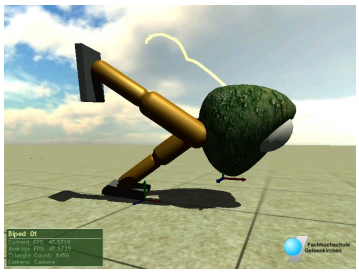


(a) Mittlere Fitness (b) Fitness der Individuen

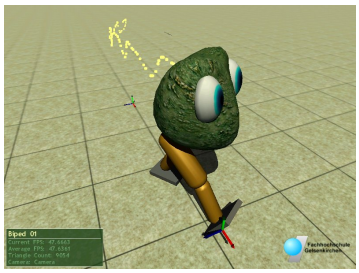
Abbildung 8.30: Biped, Evolutionslauf 26



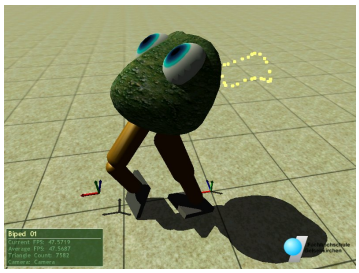
(a) Evolutionslauf 6



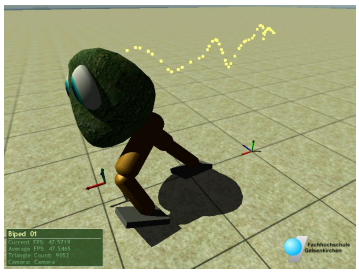
(b) Evolutionslauf 7



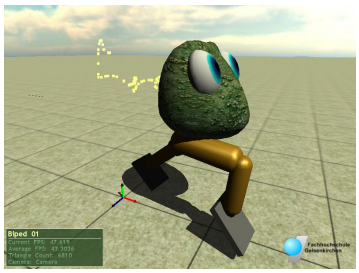
(c) Evolutionslauf 20



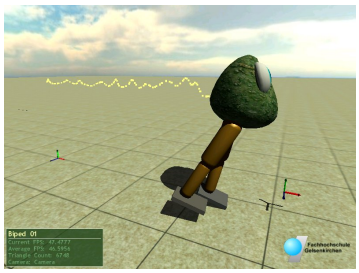
(d) Evolutionslauf 23



(e) Evolutionslauf 24



(f) Evolutionslauf 25



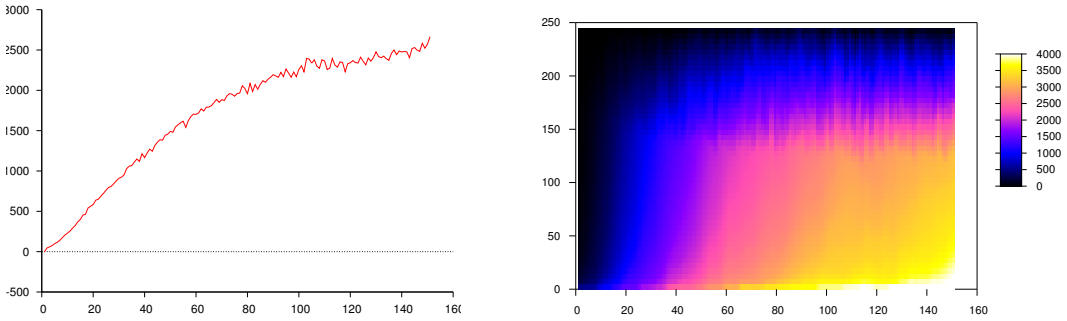
(g) Evolutionslauf 26

Abbildung 8.31: Ergebnisse der Evolutionsläufe des Biped (siehe Farbtafel D.16 auf Seite 249)

8.5.4 Quadruped

Evolutionslauf 1

Der erste Evolutionslauf (siehe Abbildung 8.32) wurde lediglich mit einer Bewertung der nach 10s erreichten Distanz in X-Richtung relativ zum Startpunkt durchgeführt.



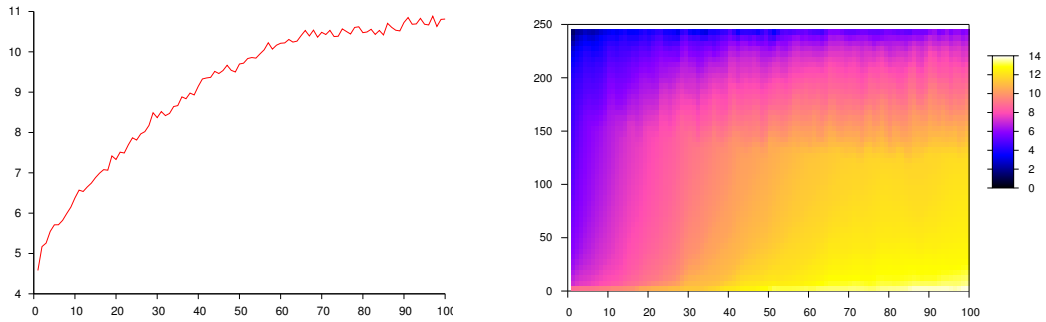
(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.32: Quadruped, Evolutionslauf 1

Bei diesem Vorgang wurde eine sehr hohe Fitness erreicht, weil die in Abschnitt 8.5.1.5 auf Seite 173 beschriebene Lücke in der Kollisionsbehandlung ausgenutzt wurde (siehe Abbildung 8.38(a) auf Seite 188). Somit ist dieser Lauf nicht zu werten.

Evolutionslauf 2

Um die starke Beinbewegung zu unterbinden, wurde der Energieterm (siehe Abschnitt 8.5.1.5 auf Seite 173) eingeführt. Dieser führt allerdings dazu, dass sämtliche Beinbewegung unterblieb und die Figur nur der Sollposition hinterherfiel. Also wurde zusätzlich ein Strafzustand eingeführt, wenn der Bauch des Charakters mit dem Boden Kontakt hatte. Dieser Evolutionslauf (siehe Abbildung 8.33) endete darin, dass die Figur alle vier Beine unter den Bauch zog und dabei wiederum geschickt der Sollposition „hinterherfiel“ (siehe Abbildung 8.38(b) auf Seite 188).

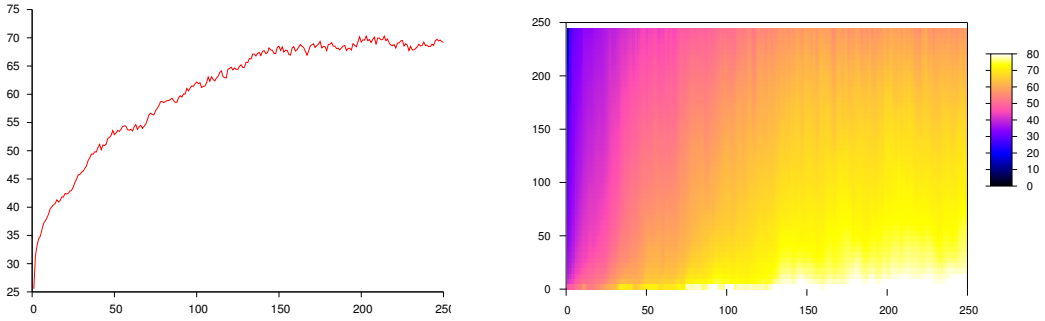


(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.33: Quadruped, Evolutionslauf 2

Evolutionslauf 3

Eine Reduktion des Energieterms um den Faktor 10 brachte einen Charakter hervor, welcher selbstständig eine rhythmische Bewegung ausführte. Durch eine Asymmetrie der Beinbewegungen verlief diese in einem großen Bogen (siehe Abbildung 8.34 und Abbildung 8.38(c) auf Seite 188).

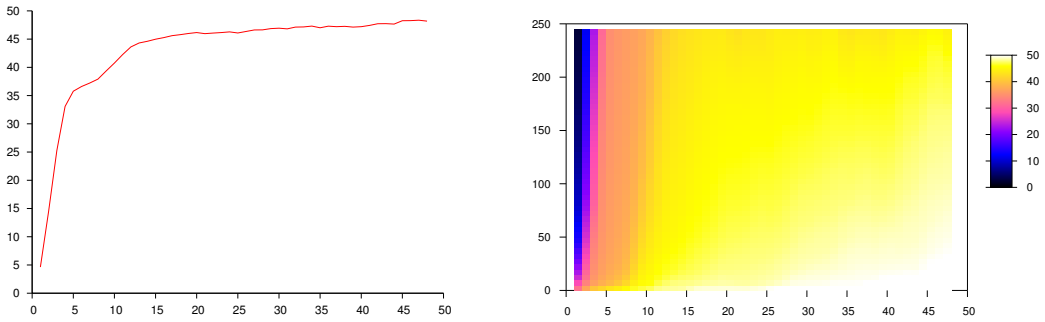


(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.34: Quadruped, Evolutionslauf 3

Evolutionslauf 4

Dieser Durchlauf baute auf den Parametern von Lauf 3 auf, begann den Evolutionsprozess jedoch mit vollständig zufällig initialisierten Werten im Genotyp. Das Ergebnis war ein nicht lauffähiger Charakter, welcher wie in Evolutionslauf 2 lediglich der Sollposition „hinterherfiel“ (siehe Abbildung 8.35 und Abbildung 8.38(d) auf Seite 188).

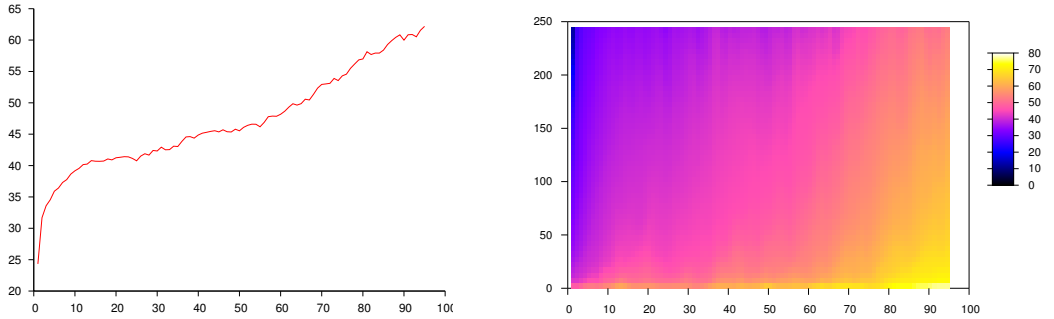


(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.35: Quadruped, Evolutionslauf 4

Evolutionslauf 5

Dieser Lauf diente zur Kontrolle des Laufes 3. Er ging von identischen Voraussetzungen aus und entwickelte eine andersartige, jedoch ebenfalls erfolgreiche Art der Fortbewegung (siehe Abbildung 8.36 auf der nächsten Seite und Abbildung 8.38(e) auf Seite 188).

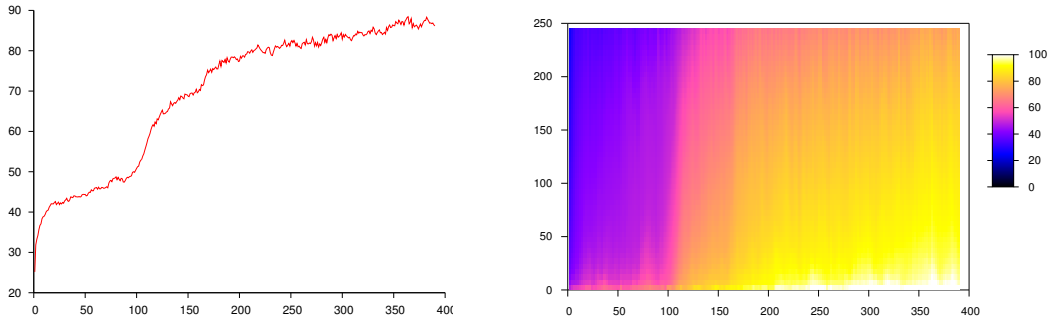


(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.36: Quadruped, Evolutionslauf 5

Evolutionslauf 6

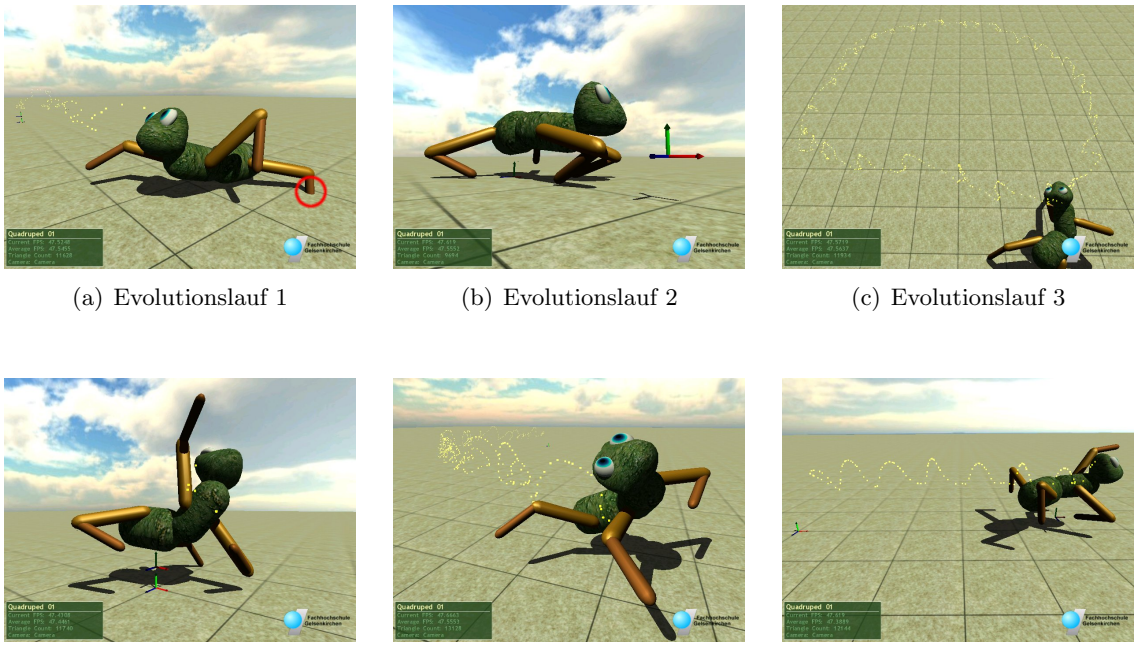
Dieser Lauf diente zur Langzeitbeobachtung (siehe Abbildung 8.37). Das Abbruchkriterium wurde so eingestellt, dass die Population über 150 Generationen einen nahezu identischen Fitnesswert aufweisen musste. Das Ergebnis[7] war ein Quadruped, welches mit drei Beinen eine Sprungbewegung vollführte und sich mit dem vierten Bein dabei stabilisierte (siehe Abbildung 8.38(f) auf der nächsten Seite).



(a) Mittlere Fitness (b) Fitness der Individuen

Abbildung 8.37: Quadruped, Evolutionslauf 6

[7] Der Lauf musste nach 390 Generationen durch einen Stromausfall abgebrochen werden.



(a) Evolutionslauf 1 (b) Evolutionslauf 2 (c) Evolutionslauf 3

(d) Evolutionslauf 4 (e) Evolutionslauf 5 (f) Evolutionslauf 6

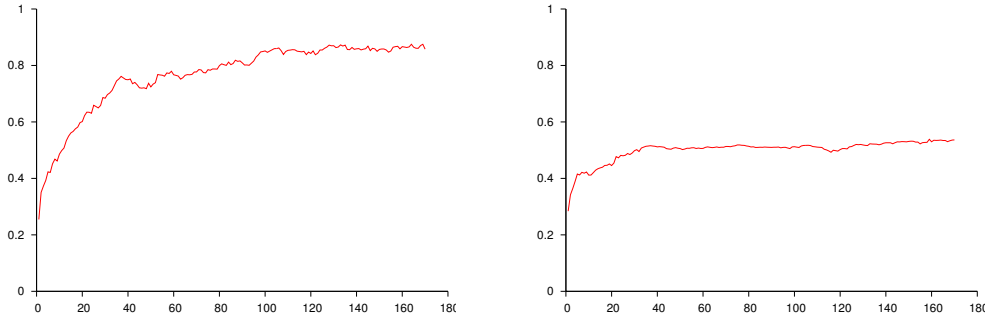
Abbildung 8.38: Ergebnisse der Evolutionsläufe des Quadruped (siehe Farbtafel D.17 auf Seite 251)

8.5.5 Zusammenfassung

Im Folgenden werden die während der Durchführung der Thesis festgestellten Einflussgrößen auf die Evolutionsläufe zusammengefasst.

8.5.5.1 Einfluss der Operatoren

Eine zu groß gewählte Mutations*schrittweite* kann dazu führen, dass der Lauf schlechter ausfällt als einer mit geringerer Schrittweite. Abbildung 8.39 zeigt zwei bis auf die Schrittweite identische Evolutionsläufe. Es ist zu erkennen, dass derjenige mit der höheren Schrittweite schlechtere Individuen erzeugt und früher stagniert. Der Grund könnten die zu starken Unterschiede der Individuen aufgrund der höheren Distanz des Genotyp zum Ursprungsindividuum sein. Der Suchraum wird dadurch zu „unruhig und hektisch“ abgedeckt.



(a) Mutationsschrittweite 0,5 (b) Mutationsschrittweite 2,0

Abbildung 8.39: Einfluss der Mutationsschrittweite

Auch eine zu hohe oder zu geringe Mutations*wahrscheinlichkeit* ist für einen guten Prozessverlauf eher abträglich. In allen Evolutionsläufen, welche in den vorherigen Abschnitten vorgestellt wurden, beträgt die Mutationswahrscheinlichkeit durchgängig 10%.

Der Einfluss der Wahl des Selektions- und Wiedereinfügeoperators sowie deren Parameter konnte im zeitlich begrenzten Rahmen der Thesis nicht untersucht werden. Eine Vermutung ist, dass vor allem die Steigung und die „Unruhe“ im Verlauf der Populationsfitness damit zu beeinflussen ist.

8.5.5.2 Einfluss der Populationsgröße

Versuche mit Variationen der Populationsgröße wurden nicht ausführlich durchgeführt. Es wurden zwischendurch lediglich einige kleine „schnelle“ Testläufe gestartet, bei denen aus Geschwindigkeitsgründen die Populationsgröße auf zehn Individuen reduziert wurde. Diese Läufe zeigten vergleichbare Ergebnisse wie solche mit hoher Populationgröße. Es ist jedoch nicht weiter untersucht worden, wie sich diese jeweiligen Ergebnisse unterscheiden.

8.5.5.3 Einfluss der Reproduktionsparameter

Da für den speziellen Anwendungsfall der Evolution eines neuronalen Netzwerks die Reproduktion immer nur ein Elter erfordert, konnte der Einfluss verschiedener Reproduktionsparameter ebenfalls nicht überprüft werden.

8.5.5.4 Einfluss der Fitnesszuweisung

Die Fitnesszuweisung ist der wichtigste Einflussfaktor aller Evolutionsläufe. Kleine Änderungen an dieser Stelle können zwischen einem erfolgreichen und einem unbrauchbaren Ergebnis entscheiden.
Es wurde festgestellt, dass eine zu strikte Fitnesszuweisung dazu führt, dass keine Entwicklung stattfinden kann. Zu oft wird in diesem Fall eine Entwicklung bestraft, welche unter Umständen in die richtige Richtung führen könnte. Alle Mono- und Bipeds, welche sich zu streng an die Sollposition halten mussten (vor Einführung des Toleranzradius r, siehe Abschnitt 8.5.1.5 auf Seite 173), ließen sich lediglich nach vorne fallen und führten keinen einzigen Sprung bzw. Schritt aus.
Eine zu lockere Fitnesszuweisung hingegen (z.B. die nach 10s zurückgelegte Strecke) führte zu Individuen, welche Lücken in der physikalischen Simulation ausnutzten, um sich von den dadurch entstehenden Phänomenen vorwärts katapultieren zu lassen.
Ein zu stark berücksichtigter Energieterm führt zu vollständiger Einstellung sämtlicher Bewegung. *Wenn* Bewegung stattfindet, dann lediglich, um Strafzuständen zu entgehen (siehe Abschnitt 8.5.4 auf Seite 185).

Das Geheimnis eines guten Evolutionslaufs liegt in der genauen Balance aller an der Fitnesszuweisung beteiligten Terme und an der exakten Definition dieser Terme. Die Schwierigkeit, diese Definitionen zu formulieren, die korrekte Balance zu finden und der dafür notwendige Zeitaufwand stellen wesentliche Nachteile bei der Entwicklung autonomer Bewegung virtueller Charaktere mittels evolutionärer Algorithmen dar.

I have not failed.
I've just found 10000 ways that won't work.
Thomas Alva Edison (1847-1931)

9 Zusammenfassung

Dieses Kapitel fasst abschließend noch einmal die Aufgabenstellungen und deren Bearbeitung zusammen. Zum Schluß folgt eine Auflistung von Verbesserungsvorschlägen und Aufgaben, welche im Anschluss an diese Thesis folgen könnten.

9.1 Aufgabenstellung

9.1.1 Simulationsumgebung

Alle Aufgaben betreffend der Simulationsumgebung wurden zur vollsten Zufriedenheit gelöst. Das Ergebnis ist eine virtuellen Umgebung, welche dem Benutzer umfangreiche Möglichkeiten bietet, sich in einer virtuellen Welt zu bewegen und diese zu beeinflussen. Auch wenn eine gewisse Gewöhnungsphase nötig ist, bevor man sich intuitiv in der virtuellen Umgebung bewegen kann, so ist diese Phase doch relativ kurz. Viele Personen, welche die Testszenen „Billard“ und „Jenga“ ausprobieren wollten, konnten sich ohne umfangreiche Einweisungen intuitiv darin bewegen und agieren.

Die 3D-Projektionsleinwand ist gerade bei Vorführungen vor einer größeren Anzahl von Personen von Nutzen. Sie bietet sich an, um entweder einen Überblick über die Szene zu erlangen, oder um die Kameraperspektive einer Person zu verfolgen, welche sich mit dem HMD in der Szene bewegt. Zahlreichen Studenten, Gruppen und anderen Besuchern des VUM-Labors konnte auf diese Weise eine hautnahe Erfahrung mit virtuelle Umgebungen angeboten werden.

Die physikalische Simulation bietet Manipulationsmöglichkeiten der virtuellen Realität, welche weit über die Möglichkeiten der realen Welt hinausgehen. Durch Verlangsamung der Zeit konnten Bewegungsabläufe der virtuellen Charaktere gut analysiert werden. Tests unter verschiedenen Bedingungen wie reduzierter Schwerkraft oder mit anderen Reibungskoeffizienten waren auf diese Weise leichter möglich als z.B. durch Versuche mit realen Robotern.

9.1.2 Evolution von Bewegungen

Das Prinzip der Entwicklung von autonomer Bewegung virtueller Charaktere mittels evolutionärer Algorithmen ist prinzipiell ein guter Ansatz. Seine Vorteile liegen darin, von der Kleinarbeit an jedem Glied des Charakters zu einer Betrachtung des Ganzen zu gelangen und Figuren zu modellieren, für die es nicht leicht oder unmöglich ist, MoCap-Daten zu erlangen. Die physikalische Simulation führt zudem zu einer besseren Akzeptanz der fertigen Szene beim Betrachter, da alle Bewegungsabläufe natürlich aussehen und physikalisch korrekt mit der Umgebung interagieren.

Der wesentliche Nachteil liegt bisher darin, dass die Kleinarbeit lediglich verlagert wird in die Einstellung der Fitnesszuweisung. An dieser Stelle sind Verbesserungen erforderlich.
endorphin umgeht diesen Nachteil, indem dem Benutzer lediglich die fertig eingestellten neuronalen Netzwerke zur Verfügung gestellt werden. Der Prozess der Kleinarbeit liegt bei der Herstellung des Produkts. Der Benutzer kann wie mit Bausteinen das Verhalten eines Charakters zusammenstellen.
Es sollte möglich sein, dieses Bausteinprinzip auch auf die Zusammenstellung einer Fitnesszuweisung anzuwenden. Man könnte so verschiedene Terme (z.B. Energiebedarf, Positionsgenauigkeit) zusammenstellen, manuell oder aufgrund von Vorschlägen der Software gewichten und den Evolutionsprozess starten. In diese Vorschläge könnten alle bis dahin gesammelten Erfahrungen im Umgang mit Evolutionsläufen fließen.

9.2 Fazit

Insgesamt muss festgestellt werden, dass das Thema mit all seinen Facetten und Details zu umfangreich für eine ausführliche Behandlung in sechs Monaten war. Vor allem im Bereich der Evolution der Bewegungen der virtuellen Charaktere mussten Abstriche gemacht werden, da alleine die Dauer der Simulationsläufe (zumeist ein Tag pro Lauf) die Anzahl der möglichen Versuche einschränkte.
Durch diesen Umfang wurde das Thema allerdings gleichzeitig auch interessant, abwechslungsreich und zu einer Herausforderung. Alle Lehrbereiche des Studiums (z.B. Englisch, höhere Programmiersprachen, Netzwerktechnik, Mathematik, Signalverarbeitung, Softwaredesign, Computergrafik, Virtuelle Umgebungen, Künstliche Intelligenz), wurden früher oder später benötigt.
Die „Überraschungen“ im Umgang mit den evolutionären Algorithmen ließen keine Langeweile aufkommen, boten immer wieder neue Herausforderungen und forderten die Kreativität.

Letztlich hinterlässt diese Thesis ein skalierbares und plattformunabhängiges System von Soft- und Hardware, welches sich zu weiteren Untersuchungen speziell an diesem Thema eignet, aber auch als Grundlage für Arbeiten, welche den Schwerpunkt mehr auf Simulation und virtuelle Umgebungen legen.

9.3 Zukünftige Arbeiten

Die folgenden Listen beschreiben zukünftige Arbeiten, welche zur Ergänzung, Verbesserung oder Vervollständigung dieser Thesis dienen könnten.

9.3.1 Hard- und Software

- **Xbox**
  Durch Portierung der Render-Engine und der Änderungen am Netzwerkprotokoll auf die Xbox können preiswert Projektionsleinwände mit mehr als zwei Beamern aufgebaut werden.

  Die Zustände der Eingabegeräte der Xbox (Controller) können abgefragt und als Netzwerkpakete versendet werden. Damit wäre eine Vereinfachung und Erweiterung der Navigation innerhalb einer Szene möglich.

- **Netzwerkprotokoll**
  Das Netzwerkprotokoll ist durch die Erweiterungen stabiler und zuverlässiger geworden. Es sind allerdings noch Verbesserungen und Optimierungen in der Geschwindigkeit und Universalität möglich.

9.3.2 Simulation

- **Hardwarebeschleunigung der Physik**
  Wenn die ersten Hardwarebeschleuniger für physikalische Simulationen erhältlich sind (PhysX-Chip, siehe Abschnitt 7.7.1 auf Seite 97) kann die Physik-Engine auf diese Hardware portiert werden.

- **TriTri-Kollision**
  Kollisionen zwischen zwei Meshes sind noch fehlerhaft implementiert (siehe Abschnitt 8.2.1.3 auf Seite 151). Dieser Fehler muss beseitigt werden, um die Physik universeller verwendbar zu machen.

- **Datenhandschuh**
  Durch die Anbindung der Datenhandschuhe an die Physik-Engine ergäben sich neue Möglichkeiten der Interaktion mit physikalisch simulierten Körpern. Man könnte so Gegenstände greifen, verschieben, drehen etc.

- **Kopplung der virtuellen und realen Welt**
  Bisher wird die Position der virtuellen physikalischen Objekte direkt an die Position realer, vom Tracker erfasster Gegenstände angeglichen. Dies resultiert in Situationen, in welchen Gegenstände sich zwangsweise durchdringen und die Physik-Engine zu drastischen Lösungen zwingen (siehe Abschnitt 8.2.3 auf Seite 153). Hier könnte ein Regelalgorithmus entwickelt werden, welcher mit Kräften und Drehmomenten versucht, die Position realer und virtueller Gegenstände abzugleichen.

- **Anwendungen**
  Durch die erstaunlich real wirkende Interaktion mit der virtuellen Umgebung und die grafischen Möglichkeiten der Render-Engine OGRE sind vollkommen neuartige Anwendungsfälle denkbar:

  – Durch Konfrontation mit virtuellen Repräsentationen angsterzeugender Situationen ist eine Therapie z.B. von Höhenangst, Flugangst denkbar.

  – Wie an den Beispielszenen „Billard“ und „Jenga“ zu sehen ist, sind neuartige Spielkonzepte oder neue Umsetzungen bekannter Spielkonzepte möglich. Mit zwei HMDs und Manipulatoren ist die Möglichkeit gegeben, gegeneinander oder kooperativ in der virtuellen Umgebung zu agieren.

  – Mit dem HMD als Kamera können Filmszenen „gedreht“ oder Szenen nachgestellt werden. Diese Anwendung könnte z.B. für kriminalistische Zwecke zur Rekonstruktion von Tathergängen dienen.

9.3.3 Virtuelle Charaktere

- **Biped**
  Zur besseren Simulation des natürlichen Vorbilds sollte das Biped zwei Freiheitsgrade an den Hüft- und Fußgelenken haben. Dadurch sollten weitere Verbesserungen der Fortbewegung möglich sein.

- **Steuerung der Muskelspannung**
  Ein bisher nicht modellierter Aspekt biologischer Muskelgruppen ist die Steuerung der Muskelspannung (siehe Abschnitt 4.3.2 auf Seite 43). Diese Information kann parallel zum Soll-Winkel eines Gelenks ausgegeben werden. Damit ist es möglich, im Bewegungsablauf Gliedmaßen von aktiver auf passive Bewegung zu „schalten“, damit sich z.B. die Fußstellung dem Boden anpassen kann.

- **Architektur des neuronalen Netzwerks**
  Die Architektur und Komplexität des neuronalen Netzwerks wurde bisher vorgegeben. Durch Methoden der Genetischen Programmierung könnte die Architektur des Netzwerks ebenso ein zu optimierender Aspekt werden wie dessen Gewichte.

- **Eingangsgrößen**
  Es sollte zusätzlich untersucht werden, ob andere Kombinationen von Eingabegrößen zu besseren Resultaten führen. Eine Analogie zum biologischen Vorbild der Muskeln wäre z.B. die simultane Eingabe von Position und Geschwindigkeit eines Gelenks (Ia- und II-Fasern, siehe dazu Abschnitt 4.3.2 auf Seite 43).

9.3.4 Evolution

- **Stufenweise Fitnessformel**
  In Abschnitt 9.1.2 auf Seite 192 wird deutlich, dass eine zu strenge Fitnessberechnung dem Beginn des Evolutionsvorgangs abträglich ist, eine zu lockere Berechnung hingegen zu unvollständigen Ergebnissen führt.

  Die Berechnung der Fitness sollte demnach stufenweise anhand der aktuellen Anzahl von Generationen und des Evolutionsfortschritts anpassbar sein. So kann man zunächst nur grob die Suchrichtung vorgeben und bei gutem Fortschritt die Kriterien immer weiter verfeinern.

- **Mehrkriterielle Fitnesszuweisung**
  Die in dieser Thesis verwendete Fitnessformel besteht aus mehreren Teilen, welche durch Gewichtungsfaktoren korrigiert werden, bevor alles zusammengefügt wird. Dabei müssen diese Faktoren gut gewählt sein, da sonst z.B. ein zu stark berücksichtigter Energieterm zu totaler Eliminierung jeglicher Bewegung führt.

  Durch mehrkriterielle Fitnesszuweisung entfällt diese Faktorenwahl. Dadurch vereinfacht sich die Formulierung geeigneter Bewertungskriterien für die evolutionäre Suche.

- **Anzahl Individuen**
  Weitergehende Arbeiten sollten untersuchen, wie der Evolutionsfortschritt und die Abdeckung des Suchraums vom Verhältnis der Anzahl der Individuen zu der Anzahl der Dimension des Suchraums abhängt. Ist es beispielsweise aus- oder unzureichend, wenn man die Lösung für ein Genotyp mit 206 Werten mit Hilfe von 250 Individuen sucht (siehe Abschnitt 8.5.1.4 auf Seite 172 und Abschnitt 8.5.5.2 auf Seite 189)?

- **Verkleinerung des Suchraums**
  [Beer u. a., 1999a,b] untersuchen in ihrer Arbeit eine Vielzahl von Konstellationen von CPG-Neuronen auf deren Eigenschaften und Verwendbarkeit. Sie schaffen eine abstrakte Beschreibung des zeitlichen Verhaltens von Mustergeneratoren durch sogenannte „dynamische Module“. Ein Modul ist definiert durch eine zeitlich begrenzte Zusammenfasung aller Neuronen, welche gerade ihren Zustand ändern, während andere Neuronen nahezu stabil bleiben. Das Schwingungsverhalten von CPGs ist mit diesem Verfahren als ein endlicher Automat zu beschreiben, dessen Zustände die jeweiligen dynamischen Module sind. Mit Hilfe dieser Methodik lassen sich stabile von instabilen Generatoren trennen und die genauen Umstände vorhersagen, unter welchen sich instabile Generatoren „festfressen“.

  Diese Erkenntnisse könnten von vorneherein dazu verwendet werden, funktionsfähige von funktionsunfähigen Netzwerken zu trennen und somit den Suchraum des evolutionären Algorithmus’ einzuschränken.

- **Assistenz bei der Konstruktion der Fitnesszuweisung**
  Wie in Abschnitt Abschnitt 9.1.2 auf Seite 192 vorgeschlagen, kann langfristig die Unterstützung des Benutzers bei der Konstruktion der Fitnesszuweisung durch ein Baukastenprinzip der verschiedenen Fitnessterme entwickelt werden.

- **Interaktion**
  Die Möglichkeiten der virtuellen Umgebung können dazu genutzt werden, den virtuellen Charakter bei seinem Evolutionsvorgang zu unterstützen. Dies kann z.B. über Hilfestellungen mit dem Manipulator geschehen oder durch aktive Vorgabe von Bewegungen der Gliedmaßen mit Hilfe des Datenhandschuhs.

Teil IV

Anhang

A Quaternionen

In vielen Anwendungen findet man Quaternionen zur Beschreibung von Orientierungen vor. Dieses Kapitel bietet einen Überblick über die herkömmlichen „Techniken“ wie Rotationsmatrizen und Rotationsvorschriften und leitet über die dabei auftretenden Probleme zu der Verwendung von Quaternionen über.

A.1 Rotationsmatrizen

Die Orientierung eines Körpers im 3-dimensionalen Raum wird am einfachsten mit Hilfe einer 3×3-Rotationsmatrix dargestellt. Die drei Spaltenvektoren der Matrix sind die Einheitsvektoren $\vec{e}_{O,X}$, $\vec{e}_{O,Y}$ und $\vec{e}_{O,Z}$ der drei Achsen des rotierten Koordinatensystems O beschrieben im Weltkoordinatensystem W (siehe Abbildung A.1).

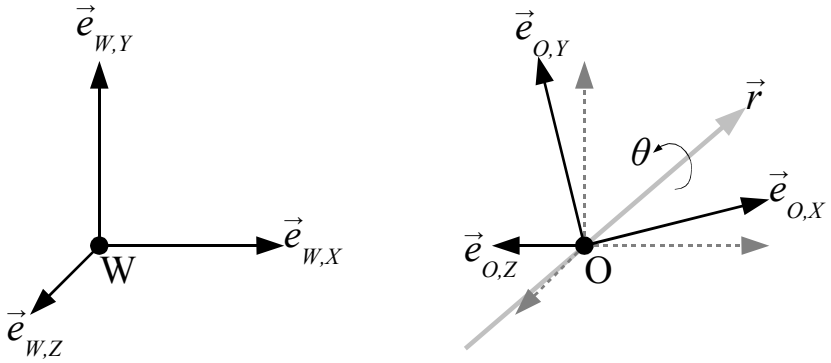


Abbildung A.1: Rotiertes Koordinatensystem

$$\mathrm{O} = \begin{pmatrix} \vec{e}_{O,X} & \vec{e}_{O,Y} & \vec{e}_{O,Z} \end{pmatrix} = \begin{pmatrix} e_{O,X,x} & e_{O,Y,x} & e_{O,Z,x} \\ e_{O,X,y} & e_{O,Y,y} & e_{O,Z,y} \\ e_{O,X,z} & e_{O,Y,z} & e_{O,Z,z} \end{pmatrix} \tag{A.1}$$

Diese Matrix besteht aus 9 Variablen, die voneinander abhängig sind. So müssen stets folgende Bedingungen erfüllt sein, damit die Rotationsmatrix die Orientierung eines rechtwinkligen Koordinatensystems beschreibt:

- Länge der Einheitsvektoren:

$$|\vec{e}_{O,X}| = |\vec{e}_{O,Y}| = |\vec{e}_{O,Z}| = 1 \tag{A.2}$$

- Rechtwinkligkeit:

$$\vec{e}_{O,X} \times \vec{e}_{O,Y} = \vec{e}_{O,Z} \tag{A.3}$$
$$\vec{e}_{O,Y} \times \vec{e}_{O,Z} = \vec{e}_{O,X} \tag{A.4}$$
$$\vec{e}_{O,Z} \times \vec{e}_{O,X} = \vec{e}_{O,Y} \tag{A.5}$$

Um die Einhaltung dieser Bedingungen zu gewährleisten, muss erhöhter Rechenaufwand und damit eine verringerte Performanz des Programms in Kauf genommen werden.

A.2 Rotationsabfolgen

Die Beschreibung einer Orientierung ist auch durch die Abfolge dreier Rotationen und der dabei verwendeten Drehwinkel möglich. Dabei gibt es zahlreiche Varianten:

Name	**Abfolge der Rotationen**	**Rotationswinkel**
RPY (Roll/Pitch/Yaw)	ursprüngliche X-Achse	α
(Rollen/Nicken/Gieren)	ursprüngliche Y-Achse	β
	ursprüngliche Z-Achse	γ
Z/Y/X Euler-Winkel	ursprüngliche Z-Achse	γ
	rotierte Y-Achse	β
	rotierte X-Achse	α
Z/Y/Z Euler-Winkel	ursprüngliche Z-Achse	γ
	rotierte Y-Achse	β
	rotierte Z-Achse	α

Tabelle A.1: Arten von Rotationsabfolgen

All diesen Rotationsverfahren sind zwei Probleme gemeinsam.

1. **Nicht unmittelbare Anwendbarkeit**
   Zur Rotation von Vektoren müssen die drei Winkel zunächst wieder in eine 3×3-Rotationsmatrix umgerechnet werden. Dazu sind relativ komplexe sin- und cos-Ausdrücke notwendig.

2. **Gimbal Lock**[1]
   Dieses Problem tritt bevorzugt auf, wenn zwei Achsen des zu rotierenden Koordinatensystems z.B. durch 90°-Rotation aufeinander oder gegenüber zu liegen kommen. In diesem Fall liegen Mehrdeutigkeiten in den Winkelangaben vor. Man kann zwei Winkel entgegengesetzt variieren, ohne die Ausrichtung des Koordinatensystems zu verändern. Umgekehrt lassen sich aus der Orientierung die Rotationswinkel nicht mehr eindeutig bestimmen.

Beiden Nachteilen begegnet man mit der Verwendung von Quaternionen.

[1] Der Begriff Gimbal-Lock stammt aus der Gyroskop-Technik. Wird ein Gyroskop 90° um eine seiner Achsen gedreht, so kann es passieren, dass sich die Kardanringe (engl.: „gimbal“), an denen die Kreisel aufgehängt sind, verklemmen oder ihre Vorzugsachsen tauschen.

A.3 Quaternionen

Quaternionen wurden 1843 von Sir William Rowan Hamilton als eine Erweiterung der komplexen Zahlen erdacht [Wikipedia Quaternions, 2005]. Ein Quaternion $\vec{\mathrm{q}} \in \mathbb{H}$ ist definiert als ein Quadrupel der reellwertigen Komponenten $\mathrm{q}_w, \mathrm{q}_x, \mathrm{q}_y, \mathrm{q}_z \in \mathbb{R}$, welche in Kombination mit den komplexen Basiselementen $\mathsf{i}, \mathsf{j}, \mathsf{k} \in \mathbb{C}$ dargestellt werden:

$$\vec{\mathrm{q}} := \begin{pmatrix} \mathrm{q}_w \\ \mathrm{q}_x\mathsf{i} \\ \mathrm{q}_y\mathsf{j} \\ \mathrm{q}_z\mathsf{k} \end{pmatrix} \tag{A.6}$$

Die Basiselemente stehen in folgendem Zusammenhang:

$$\mathsf{i}^2 = \mathsf{j}^2 = \mathsf{k}^2 = -1 \tag{A.7}$$

$$\mathsf{i} \cdot \mathsf{j} \cdot \mathsf{k} = -1 \tag{A.8}$$

$$\mathsf{i} \cdot \mathsf{j} = -(\mathsf{j} \cdot \mathsf{i}) = \mathsf{k} \tag{A.9}$$

$$\mathsf{j} \cdot \mathsf{k} = -(\mathsf{k} \cdot \mathsf{j}) = \mathsf{i} \tag{A.10}$$

$$\mathsf{k} \cdot \mathsf{i} = -(\mathsf{i} \cdot \mathsf{k}) = \mathsf{j} \tag{A.11}$$

Arthur Cayley (1821-1895) entdeckte, dass sich Quaternionen ideal zur Beschreibung von Orientierungen und Drehungen im 3-dimensionalen Raum verwenden lassen. Dazu betrachtet man die komplexen Basiselemente als drei aufeinander stehende Einheitsvektoren[2].

Gegeben sei eine Rotation um den Vektor $\vec{r}$ mit den Winkel Θ

$$\vec{r} := \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix}. \tag{A.12}$$

Dann ist das Einheitsquaternion $\vec{\mathrm{r}}$ definiert durch

$$\vec{\mathrm{r}} = \begin{pmatrix} \cos(\frac{\Theta}{2}) \\ r_x \sin(\frac{\Theta}{2})\mathsf{i} \\ r_y \sin(\frac{\Theta}{2})\mathsf{j} \\ r_z \sin(\frac{\Theta}{2})\mathsf{k} \end{pmatrix} \tag{A.13}$$

Quaternionen besitzen folgende Eigenschaften:

- **Betrag:**

$$|\vec{\mathrm{q}}| = \sqrt{\mathrm{q}_w^2 + \mathrm{q}_x^2 + \mathrm{q}_y^2 + \mathrm{q}_z^2} \tag{A.14}$$

- **Einheits-Quaternion:**

$$|\vec{\mathrm{q}}| = 1 \tag{A.15}$$

  Die folgenden Eigenschaften beziehen sich immer auf Einheits-Quaternionen.

[2] Man beachte die Analogien in den Formeln A.3/A.9, A.4/A.10, und A.5/A.11

- **Konkatenation:**
  Die Hintereinanderausführung einer Rotation $\vec{a}$ und $\vec{b}$ wird mittels einer einfachen Multiplikation ausgeführt.

$$\vec{a} \cdot \vec{b} = \begin{pmatrix} a_w b_w - a_x b_x + a_y b_y + a_z b_z \\ (a_w b_x + a_x b_w + a_y b_z - a_z b_y)\mathsf{i} \\ (a_w b_y + a_y b_w + a_z b_x - a_x b_z)\mathsf{j} \\ (a_w b_z + a_z b_w + a_x b_y - a_y b_x)\mathsf{k} \end{pmatrix} \tag{A.16}$$

- **Invertierung:**

$$\vec{q}^{-1} = \begin{pmatrix} q_w \\ -q_x\mathsf{i} \\ -q_y\mathsf{j} \\ -q_z\mathsf{k} \end{pmatrix} = \begin{pmatrix} -q_w \\ q_x\mathsf{i} \\ q_y\mathsf{j} \\ q_z\mathsf{k} \end{pmatrix} \tag{A.17}$$

- **Rotation eines Vektors $\vec{v}$:**
  Nach Erweiterung des Vektors zu einem Quaternion mittels

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \longrightarrow \vec{v} = \begin{pmatrix} 0 \\ v_x\mathsf{i} \\ v_y\mathsf{j} \\ v_z\mathsf{k} \end{pmatrix} \tag{A.18}$$

  wird die Rotation durch zwei einfache Multiplikationen ausgeführt:

$$rot(\vec{v})_{\vec{q}} = \vec{q} \cdot \vec{v} \cdot \vec{q}^{-1} \tag{A.19}$$

- **Umwandlung in 3×3-Rotationsmatrix:**

$$R_{\vec{q}} = \begin{vmatrix} 1 - 2(q_y^2 + q_z^2) & 2(q_x q_y - q_w q_z) & 2(q_x q_z + q_w q_y) \\ 2(q_x q_y + q_w q_z) & 1 - 2(q_x^2 + q_z^2) & 2(q_y q_z - q_w q_x) \\ 2(q_x q_z - q_w q_y) & 2(q_y q_z + q_w q_x) & 1 - 2(q_x^2 + q_y^2) \end{vmatrix} \tag{A.20}$$

B
Abkürzungen

Abkürzung	Bedeutung
AABB	axis-aligned bounding box
AI	artificial intelligence
ANN	artificial neural network
API	application programmers interface
ATP	Adenosintriphosphat
BSP	binary space partitioning
BPTT	backpropgation through time
CPG	central pattern generator
CTNN	continuous time neural network
CTRNN	continuous time recurrent neural network
DGL	Differentialgleichung
DNS	Desoxyribonukleinsäure
DTNN	discrete time neural network
DTRNN	discrete time recurrent neural network
EP	evolutionäre Programmierung
ES	evolutionäre Strategien
GA	genetische Algorithmen
GP	genetische Programmierung
GUI	graphical user interface
HMD	head mounted display
IPC	inter-process communication
KI	künstliche Intelligenz
MoCap	motion-capture
LOD	level of detail
NFS	network file service
NN	neural network / neuronales Netzwerk
OABB	object-aligned bounding box
ODE	open dynamics engine
OGRE	object-oriented graphics rendering engine
OSG	open scenegraph
RTRL	real-time recurrent learning
SNNS	Stuttgarter neuronaler Netzwerk-Simulator
STL	standard template library
SUS	stochastic universal sampling

Fortsetzung auf der nächsten Seite

Abkürzung	**Bedeutung**
UDP	user datagram protocol
VRM	virtual register machine / virtuelle Registermaschine
XML	extensible markup language
ZMP	zero momentum point

C Verzeichnisse

C.1 Literaturverzeichnis

C.1.1 Animation

[Carlson 2004] Carlson, Wayne E.: *CGI History Timeline*. 2004. – URL `http://accad.osu.edu/~waynec/history/timeline.html`. – Zugriffsdatum: 29.8.2005

[Film Education 2005] Film Education: *Animation History*. 2005. – URL `http://www.filmeducation.org/primary/animation/history.html`. – Zugriffsdatum: 29.8.2005

[Howe 1999] Howe, Tom: *Jason and the Argonauts – CED Web Page*. 1999. – URL `http://www.cedmagic.com/featured/argonauts.html`. – Zugriffsdatum: 29.8.2005

[McLaughlin 2001] McLaughlin, Dan: *A rather incomplete but still fascinating history of animation*. 2001. – URL `http://animation.filmtv.ucla.edu/program/anihist.html`. – Zugriffsdatum: 29.8.2005

[Nostalgia Central 1998] Nostalgia Central: *Jason & the Argonauts – Nostalgia Central*. 1998. – URL `http://www.nostalgiacentral.com/movies/jasonargonauts.htm`. – Zugriffsdatum: 29.8.2005

C.1.2 Analyse

[Beer u. a. 1999a] Beer, Randall D. ; Chiel, Hillel J. ; Gallagher, John C.: Evolution and Analysis of Model CPGs for Walking: I. Dynamical Modules. In: *Journal of Computational Neuroscience* Bd. 7. Kluwer Academic Publishers, September 1999, S. 99–118. – URL `http://vorlon.ces.cwru.edu/~beer/Papers/JCN1.pdf`. – Zugriffsdatum: 29.8.2005

[Beer u. a. 1999b] Beer, Randall D. ; Chiel, Hillel J. ; Gallagher, John C.: Evolution and Analysis of Model CPGs for Walking: II. General Principles and Individual Variability. In: *Journal of Computational Neuroscience* Bd. 7. Kluwer Academic Publishers, September 1999, S. 119–147. – URL `http://vorlon.ces.cwru.edu/~beer/Papers/JCN2.pdf`. – Zugriffsdatum: 29.8.2005

[Bourquin 2004] Bourquin, Yvan: Tank Wars! Evolving Steering and Aiming Behaviour for Computer Game Agents / University of Sussex, Brighton, United Kingdom. URL `http://tecfa.unige.ch/perso/yvan/tanks/TankWars.pdf`. – Zugriffsdatum: 29.8.2005, 2004. – Forschungsbericht

[Endo u. a. 2004] Endo, Gen ; Morimoto, Jun ; Nakanishi, Jun ; Cheng, Gordon: An Empirical Exploration of a Neural Oscillator for Biped Locomotion Control. In: *Proceedings of the 2004 IEEE International Conference on Robotics & Automation*, URL http://www-2.cs.cmu.edu/~cga/walking/endo-icra04.pdf. – Zugriffsdatum: 29.8.2005, April 2004

[Mathayomchan und Beer 2002] Mathayomchan, Boonyanit ; Beer, Randall D.: Center-crossing recurrent neural networks for the evolution of rhythmic behavior. In: *Neural Computation* Bd. 14, URL http://vorlon.cwru.edu/~beer/Papers/CCNets.pdf. – Zugriffsdatum: 29.8.2005, 2002, S. 2043–2051

[Mojon 2004] Mojon, Stéphane: *Using nonlinear oscillators to control the locomotion of a simulated biped robot*, École Polytechnique Fédérale de Lausanne, Diplomarbeit, Februar 2004. – URL http://birg.epfl.ch/webdav/site/birg/shared/import/migration/diploma_report_mojon.pdf. – Zugriffsdatum: 29.8.2005

[Nakada u. a. 2004] Nakada, Kazuki ; Asai, Tetsuya ; Amemiya, Yoshihito: Design of an Artificial Central Pattern Generator with Feedback Controller. In: *Intelligent Automation and Soft Computing* Bd. 10. TSI Press, 2004, S. 185–192. – URL http://sapiens-ei.eng.hokudai.ac.jp/contents/downloads/paper/autosoft_2004_nakada.pdf. – Zugriffsdatum: 29.8.2005

[Paul 2003] Paul, Chandana: Bilateral Decoupling in the Neural Control of Biped Locomotion. In: *2nd International Symposium on Adaptive Motion of Animals and Machines*, URL http://www.ifi.unizh.ch/groups/ailab/people/chandana/papers/amam2003.pdf. – Zugriffsdatum: 29.8.2005, 2003

[Paul 2004] Paul, Chandana: Sensorimotor Control of Biped Locomotion based on Contact Information. In: *International Symposium on Intelligent Signal Processing and Robotics*, URL http://www.ifi.unizh.ch/groups/ailab/people/chandana/papers/ispr2004.pdf. – Zugriffsdatum: 29.8.2005, 2004

[Paul und Bongard 2001a] Paul, Chandana ; Bongard, J.C.: Making Evolution an Offer It Can't Refuse: Morphology and the Extradimensional Bypass. In: *Proceedings of the Sixth European Conference on Artificial Life*, URL http://www.ifi.unizh.ch/groups/ailab/people/chandana/papers/bongardPaulEcal2001.ps.gz. – Zugriffsdatum: 29.8.2005, 2001

[Paul und Bongard 2001b] Paul, Chandana ; Bongard, J.C.: The Road Less Travelled: Morphology in the Optimization of Biped Robot Locomotion. In: *Proceedings of The IEEE/RSJ International Conference on Intelligent Robots and Systems*, URL http://www.ifi.unizh.ch/groups/ailab/people/chandana/papers/paulBongardIROS2001.pdf. – Zugriffsdatum: 29.8.2005, 2001

[Reil und Husbands 2002] Reil, Torsten ; Husbands, Phil: Evolution of central pattern generators for bipedal walking in a real-time physics environment. In: *IEEE Transactions on Evolutionary Computation* Bd. 6, Urban & Fischer, April 2002, S. 159–168. – URL http://www.ingentaconnect.com/content/urban/591/2001/00000120/F0020003/art00048. – Zugriffsdatum: 29.8.2005

[Reil und Massey 2001] Reil, Torsten ; Massey, Colm: Biologically Inspired Control of Physically Simulated Bipeds. In: *Theory in Biosciences* Bd. 120, Urban & Fischer, De-

zember 2001, S. 327–339. – URL http://www.ingentaconnect.com/content/urban/591/2001/00000120/F0020003/art00048. – Zugriffsdatum: 29.8.2005

[Ruebsamen 2002] Ruebsamen, Gene D.: *Evolving intelligent embodied agents within a physically accurate environment*, California State University, Long Beach, Master's Thesis, 2002. – URL http://www.erachampion.com/ai/src/embodied_agents.pdf. – Zugriffsdatum: 29.8.2005

[Smith 1998] Smith, Russell L.: *Intelligent Motion Control with an Artificial Cerebellum*, University of Auckland, New Zealand, Dissertation, 1998. – URL http://q12.org/phd/thesis/thesis.pdf. – Zugriffsdatum: 29.8.2005

[Taylor 1999] Taylor, Tim: *From Artificial Evolution to Artificial Life*, School of Informatics, University of Edinburgh, Dissertation, Mai 1999. – URL http://www.tim-taylor.com/papers/thesis/thesis.pdf. – Zugriffsdatum: 29.8.2005

[Wiley 2003] Wiley, Keith B.: *Observations on the Evolution of Neural Networks for the Control of a Simulated Quadruped Robot*, University of New Mexico, Master's Thesis, 2003. – URL http://www.cs.unm.edu/~kwiley/Downloads/Quadruped_Evolver.pdf. – Zugriffsdatum: 29.8.2005

[Wischmann 2003] Wischmann, Steffen: *Entwicklung der Morphologie und Steuerung eines zweibeinigen Laufmodells*, Universität des Saarlandes, Saarbrücken, Diplomarbeit, 2003. – URL http://www.ais.fraunhofer.de/~steffen/docs/diplomArbeit.pdf. – Zugriffsdatum: 29.8.2005

[Wolff und Nordin 2003a] Wolff, Krister ; Nordin, Peter: An Evolutionary Based Approach for Control Programming of Humanoids. In: *Proceedings of the Third IEEE-RAS International Conference on Humanoid Robots*, URL http://fy.chalmers.se/~wolff/WN_Humanoids03.pdf. – Zugriffsdatum: 29.8.2005, 2003

[Wolff und Nordin 2003b] Wolff, Krister ; Nordin, Peter: Learning Biped Locomotion from First Principles on a Simulated Humanoid Robot using Linear Genetic Programming. In: *Proceedings of the Genetic and Evolutionary Computation Conference*, URL http://fy.chalmers.se/~wolff/WN_gecco03.pdf. – Zugriffsdatum: 29.8.2005, 2003

C.1.3 Physik

[Bourg 2002] Bourg, David M.: *Physics For Game Developers*. O'Reilly & Associates, 2002. – ISBN 0-596-00006-5

[De Loura 2000] De Loura, Mark A. (Hrsg.): *Game Programming Gems 1*. Charles River Media, 2000. – ISBN 1-58450-049-2

[Ďurikovič und Numata 2004] Ďurikovič, Roman ; Numata, Katsuhiro: Human Hand Model based on Ridid Body Dynamics / The University of Aizu, Software Department. URL http://fractal.dam.fmph.uniba.sk/~durikovic/publications/Pub03_05_soubory/G1430_Durikovic.pdf. – Zugriffsdatum: 29.8.2005, Mai 2004. – Forschungsbericht

[Eberly 2004] Eberly, David H.: *Game Physics*. Elsevier Inc., 2004. – ISBN 1-55860-740-4

[NovodeX AG 2005] NovodeX AG: *NovodeX Physics SDK Documentation*. Juni 2005. – URL `http://www.ageia.com/pdf/PhysicsSDK.pdf`. – Zugriffsdatum: 29.8.2005

[Smith 2002] Smith, Russell L.: *How to make new joints in ODE*. Februar 2002. – URL `http://ode.org/joints.pdf`. – Zugriffsdatum: 29.8.2005

[Smith 2004] Smith, Russell L.: *Open Dynamics Engine v0.5 User Guide*. Mai 2004. – URL `http://ode.org/ode-latest-userguide.pdf`. – Zugriffsdatum: 29.8.2005

[Weisstein 2002] Weisstein, Eric W.: *Moment of inertia*. 2002. – URL `http://scienceworld.wolfram.com/physics/MomentofInertia.html`. – Zugriffsdatum: 29.8.2005

[Wikipedia Physik 2005] Wikipedia, die freie Enzyklopädie: *Portal Physik*. Juni 2005. – URL `http://de.wikipedia.org/wiki/Portal_Physik`. – Zugriffsdatum: 29.8.2005

[Witkin und Baraff 2001] Witkin, Andrew ; Baraff, David: *Physically Based Modeling*. 2001. – URL `http://www.pixar.com/companyinfo/research/pbm2001`. – Zugriffsdatum: 29.8.2005

C.1.4 Neuronale Netzwerke

[Baev 1998] Baev, Konstantin V.: *Biological Neural Networks: Hierarchical Concept of Brain Function*. Birkhäuser, 1998. – ISBN 0-8176-3859-8

[Dayan und Abbott 2001] Dayan, Peter ; Abbott, L.F.: *Theoretical Neuroscience*. MIT Press, 2001. – ISBN 0-12-191650-3

[Elman 1990] Elman, J.L.: Finding Structure in Time. In: *Cognitive Science* Bd. 14, 1990, S. 179–211

[Jordan 1986] Jordan, M.I.: Attractor dynamics and parallelism in a connectionist sequential machine. In: *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, Erlbaum, 1986, S. 531–546

[Künzell 1996] Künzell, Stefan: *Motorik und Konnektionismus: Neuronal Netze als Modell interner Bewegungsrepäsentationen*. bps-Verlag, 1996. – ISBN 3-922386-53-9

[Netter 2001] Netter, Frank H.: *Neurologie*. Georg Thieme Verlag, 2001. – ISBN 3-13-123971-9

[Nicholls u. a. 2002] Nicholls, John G. ; Martin, A. R. ; Wallace, Bruce G.: *Vom Neuron zum Gehirn*. Spektrum Akademischer Verlag Heidelberg, 2002. – ISBN 3-8274-1347-8

[Pschyrembel 2001] Verlags, Wörterbuch-Redaktion des (Hrsg.): *Pschyrembel*. Walter de Gruyter GmbH & Co. KG, 2001 (254). – ISBN 3-11-016522-8

[Rosenblatt 1962] Rosenblatt, Frank: *Principles of Neurodynamics*. Spartan Books, 1962

[Wang und Blum 1995] Wang, Xin ; Blum, Edward B.: Dynamics and Bifurcation of Neural Networks. In: Arbib, Michael A. (Hrsg.): *The Handbook of Brain Theory and Neural Networks*. MIT Press, 1995, S. 339–343. – ISBN 0-262-01148-4

[Wikipedia Cerebellum 2005] Wikipedia, die freie Enzyklopädie: *Cerebellum*. Mai 2005. – URL `http://de.wikipedia.org/wiki/Cerebellum`. – Zugriffsdatum: 29.8.2005

[Wikipedia Junk DNA 2005] Wikipedia, the free Encyclopedia: *Junk DNA*. August 2005. – URL `http://en.wikipedia.org/wiki/Junk_DNA`. – Zugriffsdatum: 29.8.2005

[Zell 2000] Zell, Prof. D. A.: *Simulation neuronaler Netze*. 3. R. Oldenbourg Verlag München Wien, 2000. – ISBN 3-486-24350-0

C.1.5 Evolution

[Fogel u. a. 1966] Fogel, L.J. ; Owens, A.J. ; Walsh, M.J.: *Artificial intelligence through Simulated Evolution*. New York : John Wiley, 1966

[Holland 1975] Holland, John H.: *Adaption in Natural and Artificial Systems*. Ann Arbor: The University of Michigan Press, 1975

[Koza 1992] Koza, J.R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge : MIT-Press, 1992

[Michalewicz 1996] Michalewicz, Zbigniew: *Genetic Algorithms + Data Structures = Evolution Programs*. 3. Springer Verlag, 1996. – ISBN 3-540-60676-9

[Nissen 1994] Nissen, Volker: *Evolutionäre Algorithmen*. Deutscher Universitäts-Verlag, 1994. – ISBN 3-8244-0217-3

[Nissen 1997] Nissen, Volker: *Einführung in Evolutionäre Algorithmen*. Vieweg Verlag, 1997. – ISBN 3-528-05499-9

[Polheim 1999] Polheim, Hartmut: *Evolutionäre Algorithmen*. Springer Verlag, 1999. – ISBN 3-540-66413-0

[Rechenberg 1973] Rechenberg, Ingo: *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart : Frommann-Holzboog Verlag, 1973

[Schöneburg u. a. 1994] Schöneburg, Eberhard ; Heinzmann, Frank ; Feddersen, Sven: *Genetische Algorithmen und Evolutionsstrategien*. Addison-Wesley, 1994. – ISBN 3-89319-493-2

[Weicker 2002] Weicker, Karsten: *Evolutionäre Algorithmen*. Teubner, 2002. – ISBN 3-519-00362-7

[Wikipedia Evolution 2005] Wikipedia, die freie Enzyklopädie: *Evolution*. Juni 2005. – URL `http://de.wikipedia.org/wiki/Evolution`. – Zugriffsdatum: 29.8.2005

C.1.6 Sonstiges

[Görz u. a. 2000] Görz, Günther ; Rollinger, Claus-Rainer ; Schneeberger, Josef: *Handbuch der Künstlichen Intelligenz*. 3. R. Oldenbourg Verlag München Wien, 2000. – ISBN 3-486-25049-3

[Mittelbach und Goossens 2004] Mittelbach, Frank ; Goossens, Michel: *The LaTeXCompanion*. 2. Addison-Wesley, 2004. – ISBN 0-201-36299-6

[Pollak 2003] Pollak, Adrian: *Entwicklungsumgebung für VR-Applikationen*, Fachhochschule Gelsenkirchen, Master's Thesis, November 2003. – unveröffentlicht

[Russell und Norvig 1995] Russell, Stuart J. ; Norvig, Peter: *Artificial Intelligence: a modern approach*. Prentice Hall, 1995. – ISBN 0-13-103805-2

[Wikipedia Bifurkation 2005] Wikipedia, die freie Enzyklopädie: *Bifurkation (Mathematik)*. Juni 2005. – URL `http://de.wikipedia.org/wiki/Bifurkation_%28Mathematik%29`. – Zugriffsdatum: 29.8.2005

[Wikipedia Quaternions 2005] Wikipedia, die freie Enzyklopädie: *Quaternionen*. Juli 2005. – URL `http://de.wikipedia.org/wiki/Quaternionen`. – Zugriffsdatum: 29.8.2005

[Wikipedia XML 2005] Wikipedia, die freie Enzyklopädie: *Extensible Markup Language*. August 2005. – URL `http://de.wikipedia.org/wiki/XML`. – Zugriffsdatum: 29.8.2005

[Yergeau u. a. 2004] Yergeau, François ; Grosso, Paul ; Walsh, Norman ; Quin, Liam u. a.: *Extensible Markup Language (XML) 1.0*. 2004. – URL `http://www.w3.org/TR/REC-xml`. – Zugriffsdatum: 29.8.2005

C.2 Abbildungsverzeichnis

C.3 Tabellenverzeichnis

C.4 Programmcodeverzeichnis

C.5 Index

D
Farbtafeln

 [1]

(a) Höhlenmalerei

 [2]

(b) Schattentheater

 [3]

(c) Griechische Vase

 [4]

(d) Laterna Magica

Abbildung D.1: Bilder zur Geschichte der Animation (Teil 1)

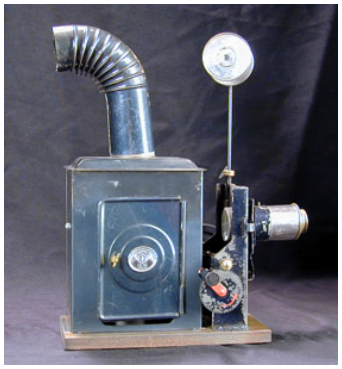
[1] http://www.sanford-artedventures.com/study/images/cave_painting_1.jpg
[2] http://www.doll.at/image1/china.jpg
[3] http://www.ceramicstoday.com/images/odd/olympics.jpg
[4] http://www.smca.at/presse/images/Laterna_Magica.jpg

 [5]

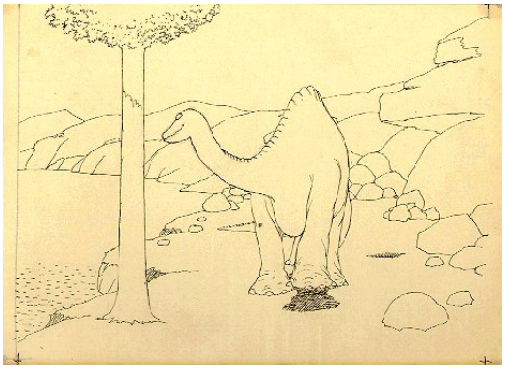
(a) Zoetrope

 [6]

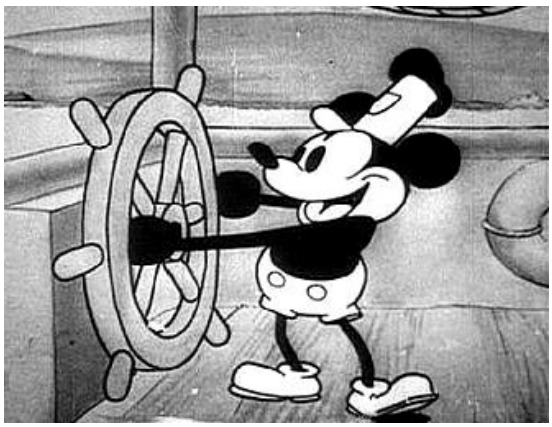
(b) Praxinoskop

 [7]

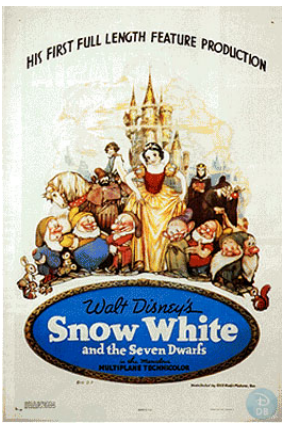
(c) Cinematoskop

 [8]

(d) Gertie the Dinosaur

 [9]

(e) Steamboat Willie



[10]

(f) Snow White and the Seven Dwarves

Abbildung D.2: Bilder zur Geschichte der Animation (Teil 2)

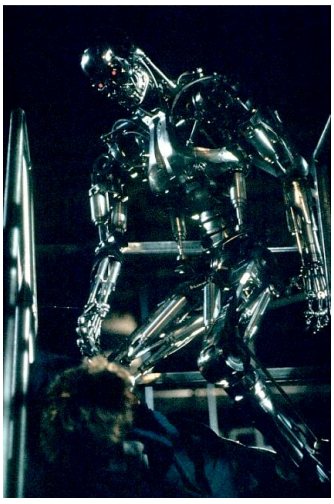
[5] http://courses.ncssm.edu/gallery/collections/toys/images/ZoetropeTopView0315.jpg
[6] http://courses.ncssm.edu/gallery/collections/toys/images/PraxTopVw0320.jpg
[7] http://courses.ncssm.edu/gallery/collections/toys/images/Cinematog322.jpg
[8] http://www.vegalleries.com/images/gertie285c.gif
[9] http://disneyshorts.toonzone.net/years/1928/graphics/steamboatwillie/steamboatwillie1thumb.jpg
[10] http://disney.go.com/vault/archives/characters/snow/snow2.jpg

 [11]

(a) Jason and the Argonauts

[12]

(b) Tron

 [13]

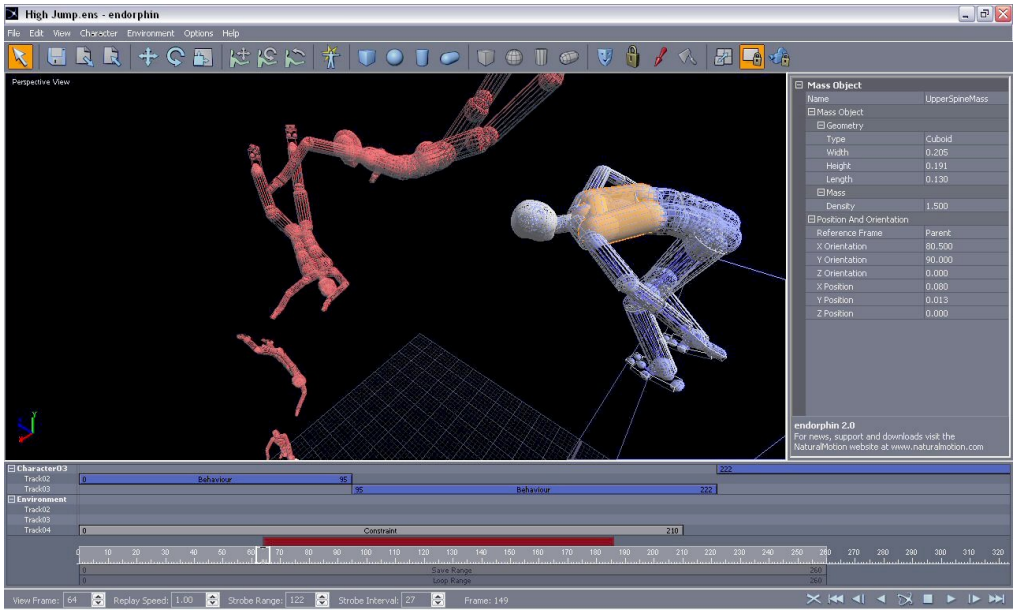
(c) T-800 (Stop-Motion)

 [14]

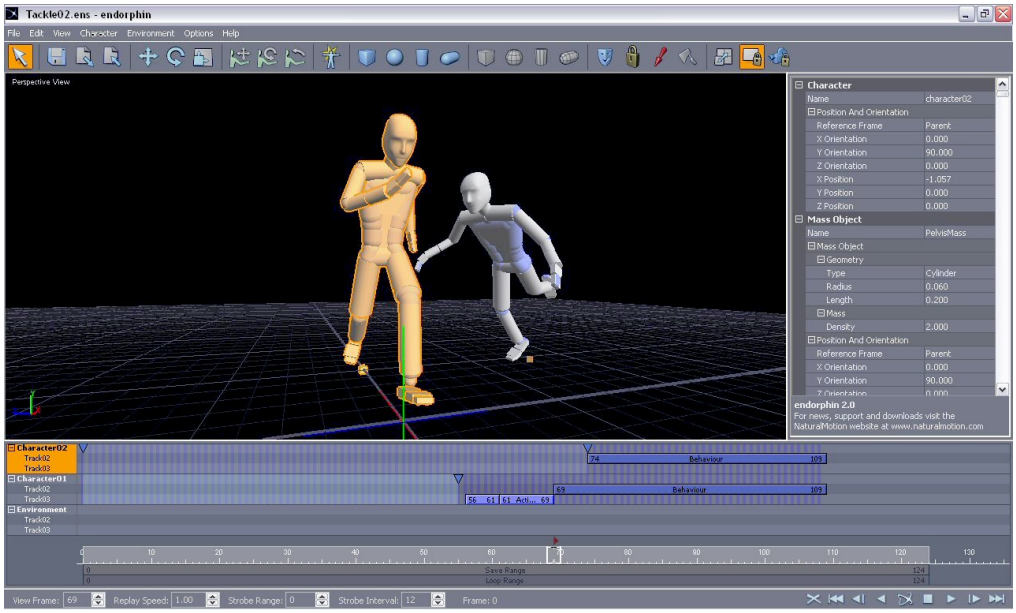
(d) T-1000 (Computeranimation)

Abbildung D.3: Bilder zur Geschichte der Animation (Teil 3)

[11] http://www.nostalgiacentral.com/images_movie/jason_03.jpg
[12] http://www.cybergeography.org/atlas/tron.jpg
[13] http://www.imdb.com/gallery/ss/0088247/Terminator_PUB10.jpg
[14] http://www.goingfaster.com/darkthoughts/t1000.jpg



15



16

Abbildung D.4: Benutzungsoberfläche von *endorphin*

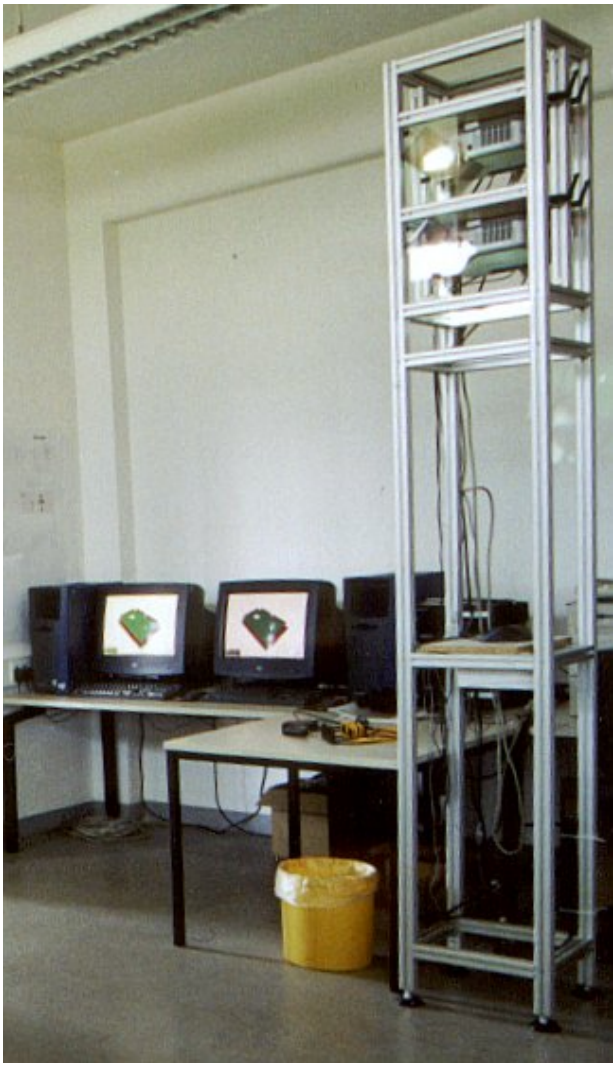
[15] http://www.naturalmotion.com/images/e20s2.jpg
[16] http://www.naturalmotion.com/images/e20s3.jpg

(a) Steuercomputer

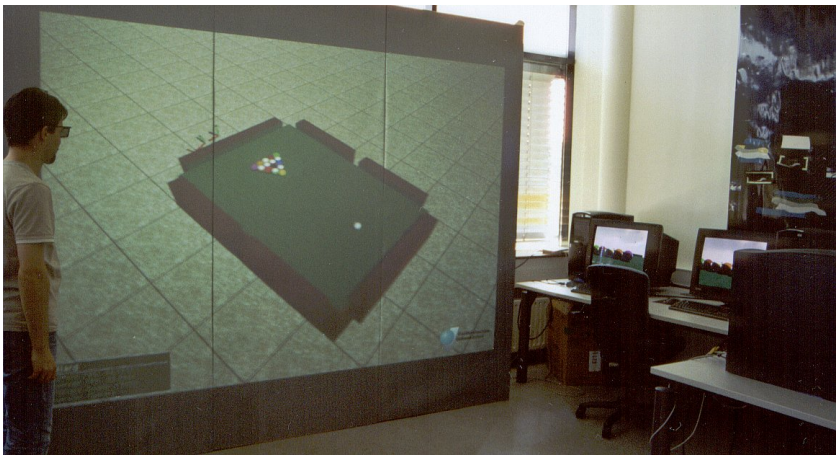
(b) HMD mit Render-Engines

(c) Tracker mit zweitem HMD

Abbildung D.5: Vorhandene Hardware des VUM-Labors (Teil 1)

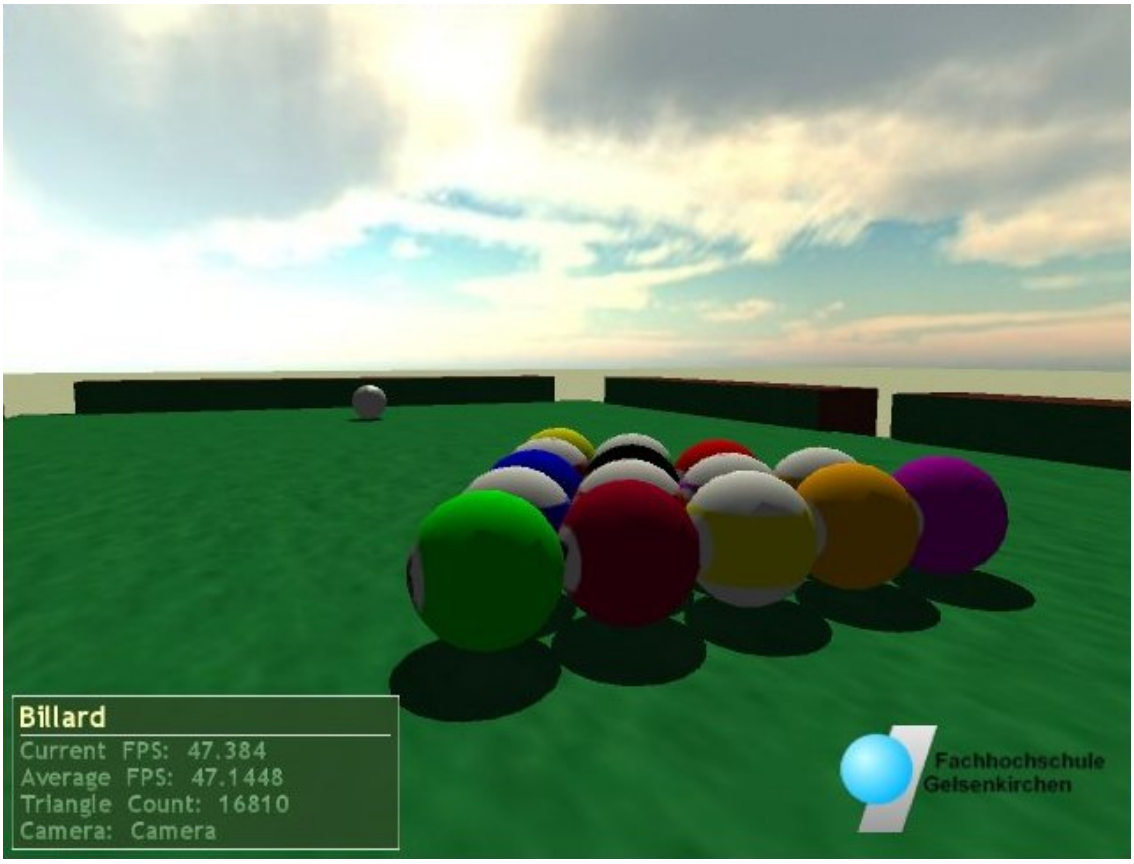


(a) 3D-Projektoren mit Render-Engines

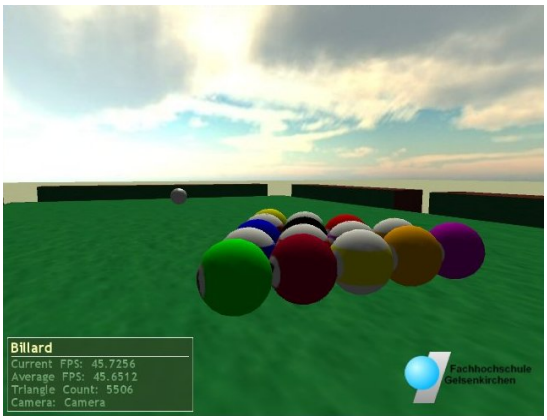


(b) 3D-Projektionsleinwand

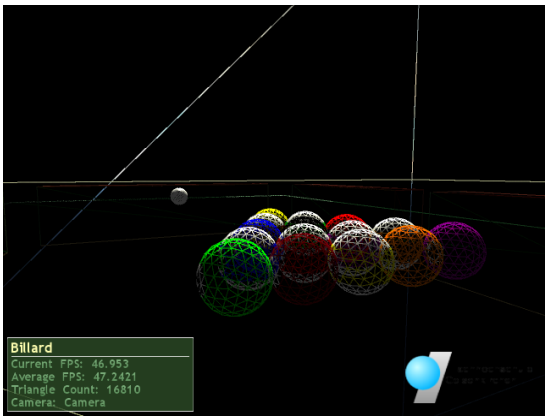
Abbildung D.6: Vorhandene Hardware des VUM-Labors (Teil 2)



(a) Normaler Modus



(b) Ohne Schattenwurf



(c) „Wireframe“-Modus

Abbildung D.7: Bildschirmfotos von `cerebellum` (Teil 1)

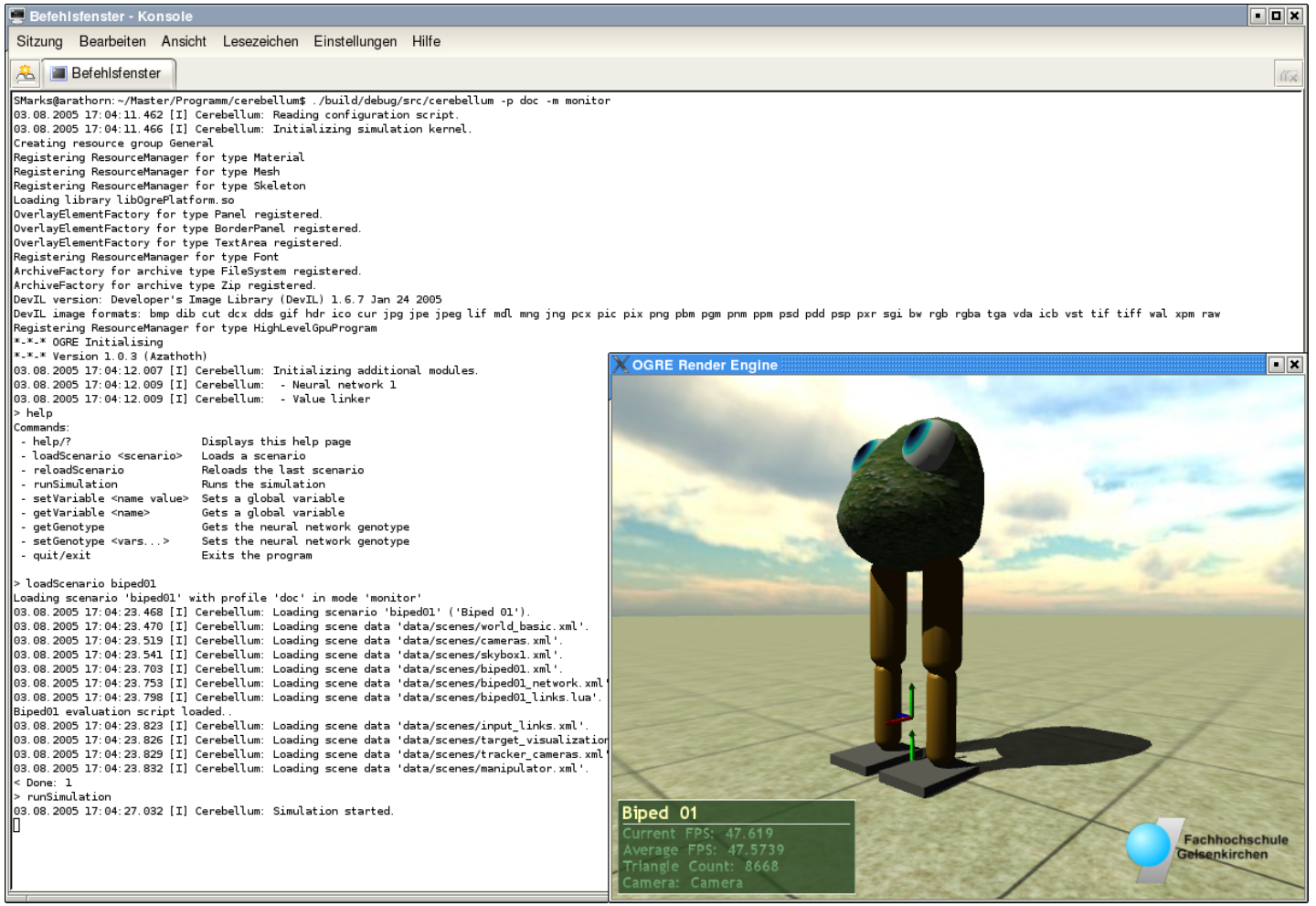


Abbildung D.8: Bildschirmfoto von `cerebellum` (Teil 2)

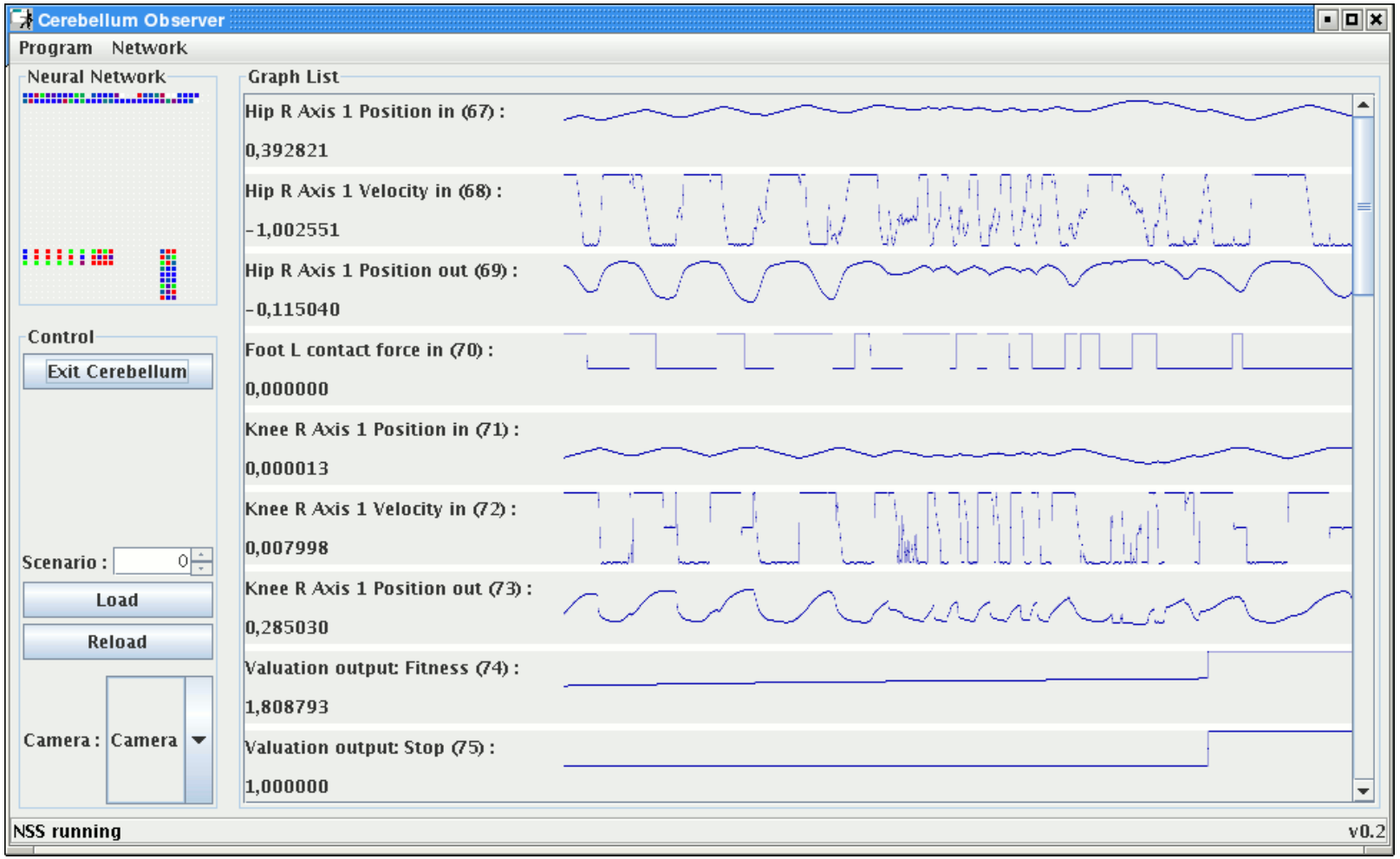


Abbildung D.9: Bildschirmfoto von `CerebellumObserver`

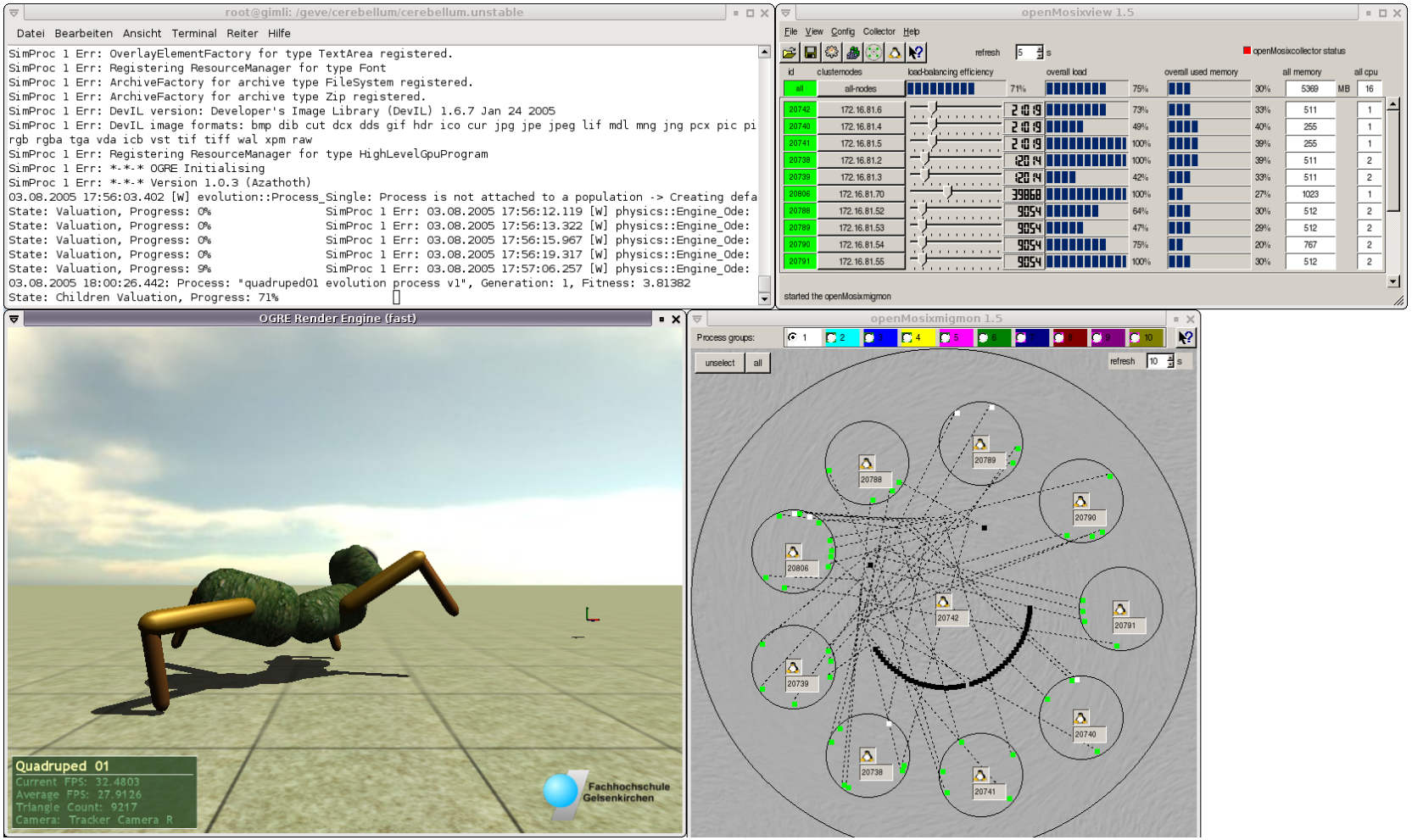
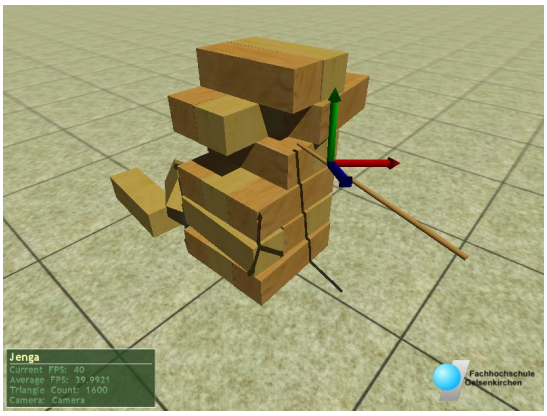


Abbildung D.10: Bildschirmfoto von `evolver`

Abbildung D.11: Virtuelle Autokarosserie

(a) Fallende Objekte (b) Würfelstapel (c) Kistenstapel

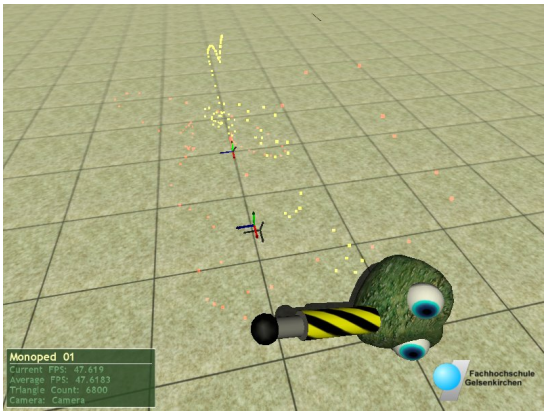
Abbildung D.12: Testszenen für die Physik-Engine (Teil 1)



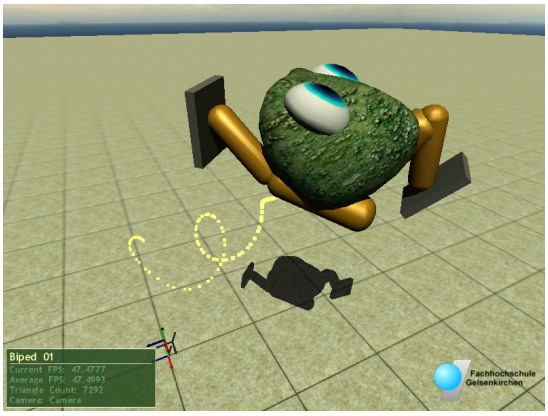
(a) Jenga

(b) Billard

Abbildung D.13: Testszenen für die Physik-Engine (Teil 2)



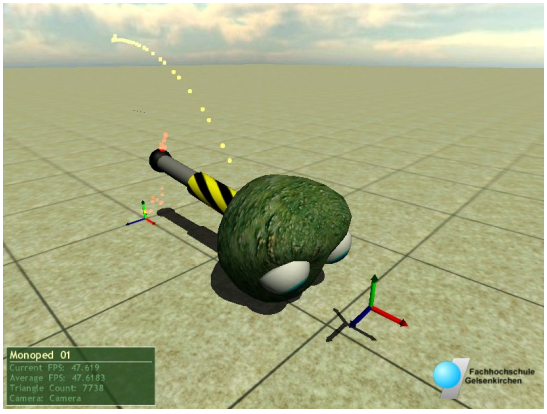
(a) Monoped



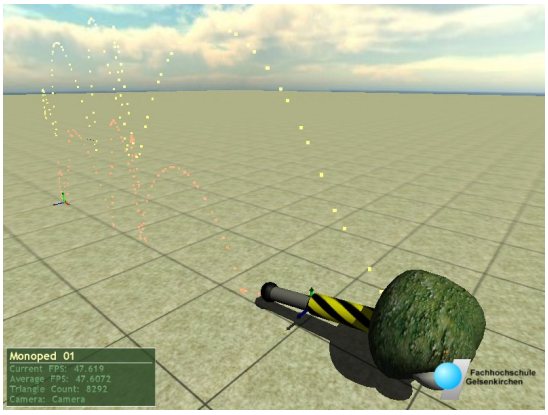
(b) Biped

(c) Quadruped

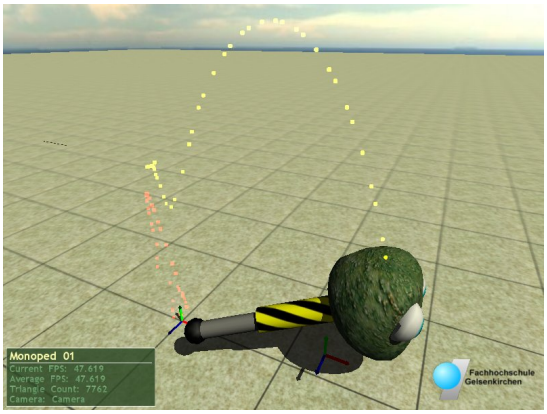
Abbildung D.14: Fehler in der physikalischen Simulation



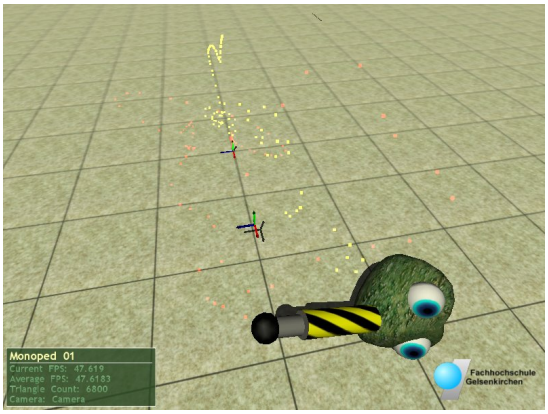
(a) Evolutionslauf 3



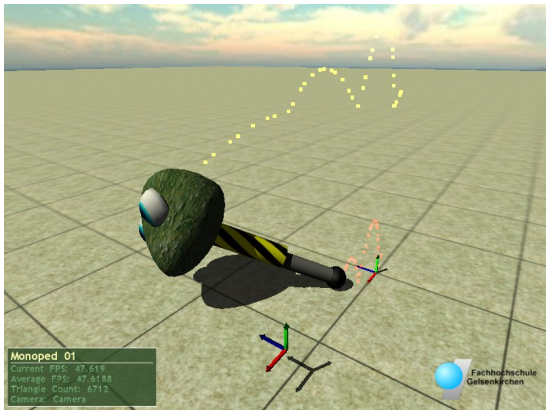
(b) Evolutionslauf 9



(c) Evolutionslauf 10

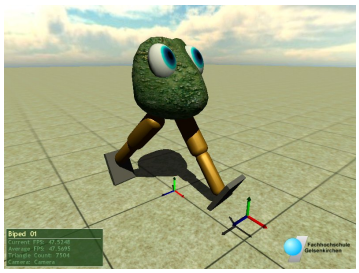


(d) Evolutionslauf 11

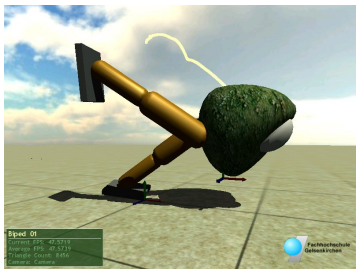


(e) Evolutionslauf 12

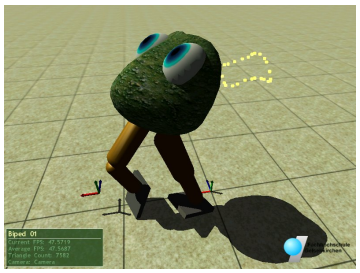
Abbildung D.15: Ergebnisse der Evolutionsläufe des Monoped



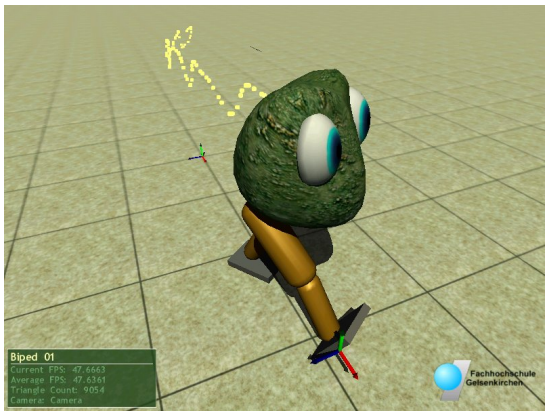
(a) Evolutionslauf 6



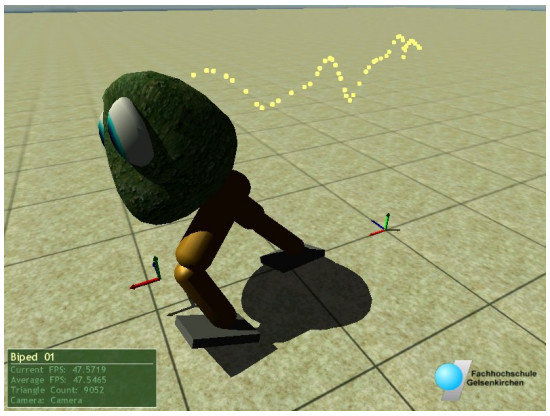
(b) Evolutionslauf 7



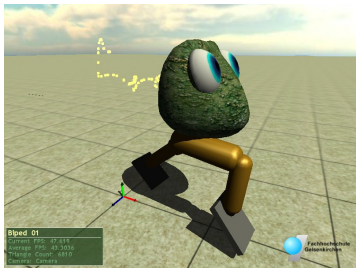
(c) Evolutionslauf 23



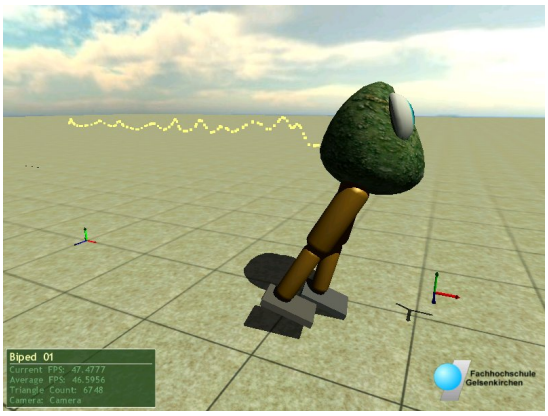
(d) Evolutionslauf 20



(e) Evolutionslauf 24



(f) Evolutionslauf 25

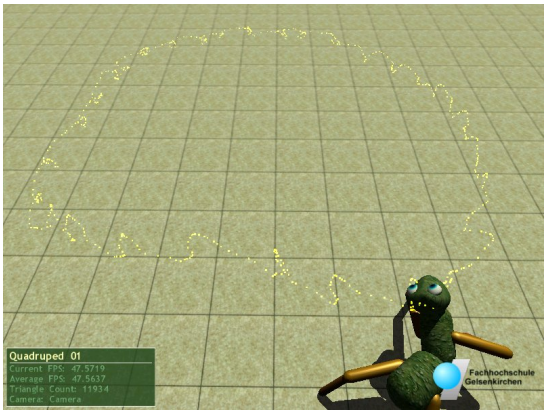


(g) Evolutionslauf 26

Abbildung D.16: Ergebnisse der Evolutionsläufe des Biped (Erfolgreiche Fortbewegungen in großen Bildern)

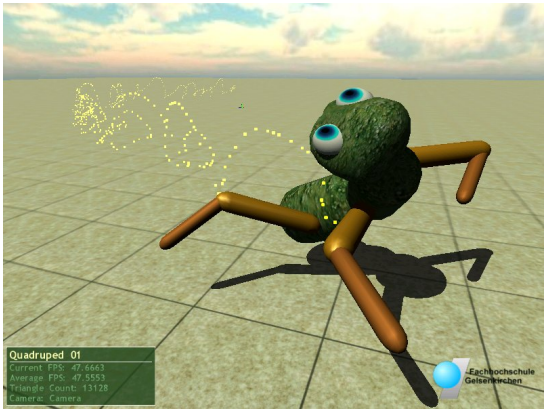
(a) Evolutionslauf 1

(b) Evolutionslauf 2

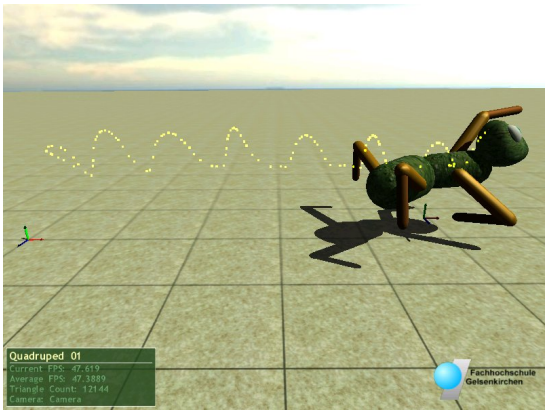


(c) Evolutionslauf 3

(d) Evolutionslauf 4



(e) Evolutionslauf 5



(f) Evolutionslauf 6

Abbildung D.17: Ergebnisse der Evolutionsläufe des Quadruped