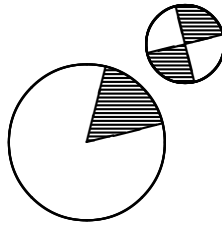


Hoverball Manual

Version 1.4 (2021.01.28)



Stefan Bornhofen
Matthias Bornhofen

www.hoverball.org

Inhaltsverzeichnis

Was ist Hoverball?	5
1 Das Programm	6
1.1 Installation	6
1.2 Ausführbare Klassen	6
1.2.1 Simulator	7
1.2.2 Server	7
1.2.3 Controller	7
1.2.4 Session	8
1.2.5 Human-Player	8
2 Das Spiel	9
2.1 Hoverball's Welt	9
2.2 Pucks	10
2.3 Sicht und Aktion	10
2.4 Energie	11
2.5 Kommunikation	11
2.6 Spielregeln	11
3 Der Simulator	12
3.1 Die drei Zustände des Simulators	12
3.2 Eigenschaften der Simulation	12
3.3 Simulator-Variablen	13
3.4 Optionen	13
3.5 Orthonormalbasen und Euler-Vektoren	15
3.6 Netzwerk-Kommunikation	16
3.7 Unit Clients	16
3.8 Control Clients	18
4 Der Controller	21
4.1 Channels	21
4.2 Options	21
4.3 Control-Buttons	21

4.4	Screen	22
4.5	Zoom	22
4.6	Follow-Mode	22
4.7	Views	22
4.8	Debug	23
5	Programmierung	24
5.1	Session	24
5.2	Hovlet	25
5.3	Unit	26
5.4	Team	26
5.5	Graphical Debugging	27
5.6	Operator Overloading	28
A	Mathematische Spezifikation	29
A.1	Attribute eines Pucks	29
A.2	Antrieb	30
A.3	Reibung	31
A.4	Allgemeine Bewegungsgleichung	31
A.5	Polarisation	32
A.6	Kollision und Reflexion	32
A.7	Sphäre	34
A.8	Euler-Winkel	36
A.9	Sichtwinkel	36
A.10	Energie	37
B	Tabellen	38
B.1	Netzwerk-Protokolle	38
B.2	Hoverball-Optionen	39
C	Beispiel „Clumsy“	40

Einführung

Was ist Hoverball?

Das Projekt **Hoverball** verfolgt den Ansatz, eine spielerische Echtzeit-Simulation für Multiagentensysteme zu schaffen, deren zugrundeliegende Weltgesetze sowie die eigentlichen Spielregeln ganz im Sinne der axiomatischen Mathematik so einfach wie möglich gestaltet sind und damit auch einem grundlegenden ästhetischen Anspruch genügen. Wir erwarten dadurch, dass die Entwicklung leistungsfähigster Agenten durchschaubar bleibt und zu neuen elementaren Erkenntnissen bei der Erforschung der Künstlichen Intelligenz beitragen kann.

Das vorliegende Spielkonzept wird diesen Forderungen gerecht: Kaum eine andere Simulationsplattform bietet einen so anschaulichen, reizvollen und niederschweligen Einstieg in die Welt der Multiagentensysteme. Die Echtzeit ermöglicht es zudem, als Mensch an der Simulation teilzunehmen und das Spiel als aktiver Mitspieler zu beeinflussen. Damit ist **Hoverball** einerseits ein Wettkampf zwischen Programmierern, die möglichst tüchtige Teams entwickeln und gegeneinander antreten lassen, und andererseits ein direkt zugängliches Computerspiel für Menschen.

Inspiziert wurde **Hoverball** durch den RoboCup Soccer Simulator, welche der Forschung zur Künstlichen Intelligenz auf der Basis eines Fußballspiels dient. Demzufolge besitzt RoboCup auch das einem Fußballspiel entsprechend komplizierte Regelwerk. **Hoverball** hat keinen Anspruch auf einen derartigen Realitätsbezug und schlägt ein einfacheres Spielkonzept vor: Auf eine Kugeloberfläche werden zweidimensionale Pucks gesetzt, die zwei oder mehr Teams bilden, sich in spielerisch-sportlichem Wettstreit kleinere Pucks zustoßen und innerhalb eines schlichten Regelsystems Punkte erzielen.

Was die Weltgesetze angeht, war es bei der Entwicklung von **Hoverball** wichtig, der Simulation ein physikalisch konsistentes Modell zugrunde zu legen: die Bewegungen, die Kollisionen der Pucks, die Reibung mit dem umgebenden Medium — alle Berechnungen basieren auf physikalisch realen Gesetzen. Mit Hilfe von Dipl.-Physiker Horst Wilhelm (Papenburg) ist es gelungen, diesen Anspruch zu erfüllen, wenn auch einige Spielparameter von den Werten der real existierenden Naturkonstanten abweichen.

Hoverball ist in Java geschrieben und somit plattformunabhängig und netzwerkfähig. Im Sinne der freien Verbreitung von Software ist **Hoverball** ein Open Source Projekt und untersteht der GNU General Public Licence.

Teil 1

Das Programm

Hoverball ist vollständig in Java geschrieben und läuft auf allen Betriebssystemen.

1.1 Installation

Besuche **Hoverball** im Internet unter

www.hoverhall.org

und lade das Archiv `hoverball.1.4(2021.01.28).de.zip` herunter. Es beinhaltet:

<code>demo[.*]/</code>	... Beispiele zur Anwendung von Hoverball .
<code>docs/</code>	... die Hoverball Interface Specification.
<code>src/</code>	... den Quellcode.
<code>contact@...</code>	... für Feedback.
<code>COPYING</code>	... die Lizenzbestimmungen.
<code>hoverball.jar</code>	... die Programmkomponenten von Hoverball .
<code>manual.pdf</code>	... genau diese Dokumentation.

Um **Hoverball** starten zu können, musst du die Datei `hoverball.jar` in den *classpath* der Java-Plattform einbinden. Du kannst `hoverball.jar` auch als Java-Extension installieren. Mehr zu diesen Themen findest du in der Dokumentation deiner Java-Plattform.

1.2 Ausführbare Klassen

Die Klassen der Datei `hoverball.jar` sind in einem Package namens `hoverball` bzw. weiteren Unterpackages zusammengefasst. Folgende Klassen dieses Packages können extern als Application aufgerufen werden:

<code>class hoverball.Simulator</code>	... der Simulator von Hoverball .
<code>class hoverball.Server</code>	... ein eigenständiger Hoverball -Server.

```
class hoverball.Controller    ... ein universelles Steuerungs-Tool für den Simulator.  
class hoverball.Session      ... eine Standard-Session.  
class hoverball.Human        ... eine Schnittstelle für einen menschlichen Spieler.
```

1.2.1 Simulator

Klasse: `hoverball.Simulator`

Aufruf: `java hoverball.Simulator [:port]`

Der Simulator ist verantwortlich für die Simulation der **Hoverball**-Welt und stellt somit das Kernstück des Programms dar.

Wird beim Aufruf `:port` angegeben, so wird der Simulator an diesem Port geöffnet, ansonsten meldet sich der Simulator am Standard-Port 1234 an. (Die vollständige Netzwerk-Adresse des Simulators setzt sich zusammen aus `host:port`. Heißt dein Computer etwa `galileo` und läuft der Simulator am Standard-Port, so lautet seine Netzwerk-Adresse `galileo:1234`.)

Der **Hoverball**-Simulator wird in Teil 3 dieser Dokumentation beschrieben.

1.2.2 Server

Klasse: `hoverball.Server`

Aufruf: `java hoverball.Server [:port]`

Der Server ist in der Lage, mehrere Spiele (d.h. mehrere Simulator-Instanzen) zu hosten.

Um sich anmeldende Clients dem richtigen Spiel zuordnen zu können, wird bei der Anmeldung ein „Hashtag“ übergeben, das das Spiel auf dem Server identifiziert. Sendet der Client bei der Anmeldung kein Hashtag, erzeugt der Server ein neues Spiel mit einem neuen Hashtag und sendet dieses an den Client zurück.

Aktuell läuft unter der Netzwerk-Adresse `hoverball.net` ein Server, auf dem **Hoverball** über das Internet gespielt werden kann!

1.2.3 Controller

Klasse: `hoverball.Controller`

Aufruf: `java hoverball.Controller [host][:port][#hash]`

Der Controller ist ein multifunktionales Tool zur Steuerung des **Hoverball**-Simulators. Ist er mit dem Simulator verbunden, so kann er

- die Liste der mit dem Simulator verbundenen Spieler verwalten,
- die Parameter der Simulation verändern,
- die Simulation starten und anhalten,
- die **Hoverball**-Welt auf dem Screen visualisieren.

Wird beim Aufruf *host:port#hash* angegeben, so verbindet sich der Controller automatisch mit dem Simulator an dieser Adresse.

Sämtliche Funktionen des Controllers werden in Teil 4 dieser Dokumentation beschrieben.

1.2.4 Session

Klasse: `hoverball.Session`

Aufruf: `java hoverball.Session [host][:port][#hash]`

Die Session bildet die Basis-Klasse für einen schnellen und flexiblen Aufruf von **Hoverball**. Sie ist dafür vorgesehen, während der Entwicklungsphase eines eigenen Teams mit einer gleichbleibenden **Hoverball**-Konfiguration programmieren zu können.

Wird diese Klasse extern aufgerufen, so öffnet sie einen Simulator — am Standard-Port 1234 oder bei Angabe von *:port* an diesem Port — und einen mit ihm verbundenen Controller, oder sie öffnet bei Angabe von *host:port#hash* nur einen Controller, der sich automatisch mit dem Simulator an dieser Adresse verbindet.

Einer kleinen Einführung in die Team-Programmierung in Java widmet sich Teil 5 dieser Dokumentation.

1.2.5 Human-Player

Klasse: `hoverball.Human`

Aufruf: `java hoverball.Human [team [name [color] [host][:port][#hash]]]`

Da das Spiel **Hoverball** in Echtzeit abläuft, kannst du leicht als menschlicher Spieler in das Spiel einsteigen. Eine einfache Lösung hierfür bietet **Hoverball**'s Human-Player: Definiere beim Aufruf deinen Team- und Spieler-Namen und steuere deinen Spieler mit den **CRSR**-Tasten (vorwärts und rückwärts fahren, links und rechts drehen) und den Tasten **CTRL** und **SHIFT** (Ball anziehen und schießen) über die Sphäre!

Die Farbe *color* wird als hexadezimale sechsziffrige Zahl angegeben (also etwa **FFC800** für orange). Ist *team* ein Farbwort (etwa **orange**), so wird diese Farbe für den Spieler verwendet.

Folgender Befehl verschafft dir einen ersten Einblick in die Welt von **Hoverball**:

```
java -cp hoverball.jar hoverball.Human orange name
```

Klicke auf [>] — und los geht's!

Teil 2

Das Spiel

Bei dem Spiel **Hoverball** handelt es sich um die Simulation eines abstrahierten Fußballspiels: Zwei Teams wetteifern um einen Ball und versuchen, durch gezielte Schüsse Punkte zu erzielen. „Abstrakt“ ist **Hoverball** deshalb, weil die virtuelle Umgebung auf einfachen geometrischen Formen und auf elementaren Naturgesetzen basiert.

Die Grundlage hierfür liefert die Mathematik, die das Modell der Welt mit ihren Eigenschaften eindeutig und störungsfrei definiert. Die mathematische Spezifikation ist im Anhang A dokumentiert. Vorerst darf hier aber die naive Vorstellung genügen.

Die Spieler von **Hoverball** sind Agenten, d.h. autonome Computerprogramme, die sensorische Informationen (wie zum Beispiel die momentane Sicht) verarbeiten und sich zu Aktionen (wie etwa eine bestimmte Bewegung) entschließen. Ziel ist nun, solche Agenten für ein eigenes Team zu programmieren, das gegen andere Teams im **Hoverball**-Spiel antritt.

2.1 Hoverball's Welt

Hoverball spielt sich in einer zweidimensionalen Ebene ab. Jedoch handelt es sich dabei nicht um die Euklidische Ebene, sondern um die Sphäre einer dreidimensionalen Kugel. In diese Sphäre können nun kreisrunde Pucks gesetzt werden, die sich in linearer Zeit innerhalb dieser sphärisch gekrümmten Ebene „freischwebend“ bewegen können. Die Pucks besitzen einen festen Radius und eine feste Masse, die über ihre Kreisfläche homogen verteilt ist.

In dieser virtuellen Welt herrschen die folgenden vier physikalischen Gesetze:

1. **Beschleunigte Bewegung** — *Grundgesetz der Mechanik*

Pucks können durch Wirken von Kräften beschleunigt und gedreht werden.

2. **Reibung** — *Gesetz der viskosen Reibung*

Sich bewegende Pucks werden ohne Einwirkung von äußeren Kräften langsamer. Hierzu stelle man sich vor, dass die Pucks bei ihrer Bewegung einen Äther verdrängen, der die ganze Sphäre homogen ausfüllt und die Bewegung der Pucks bremst.

3. **Kollision** — *Gesetz des elastischen Stoßes*

Kollidieren zwei Pucks, so werden sie ohne Energieverlust reflektiert, wobei die Reflexion wirkt

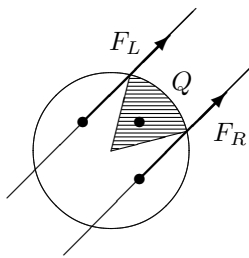
sich auch auf die Drehung der Pucks auswirkt, da für den Rand der Pucks eine Haftreibung von 100% gilt.

4. Polarisation — Gesetz von Coulomb

Jeder Puck hat einen Ladungspunkt, der sich positiv oder negativ polarisieren kann. Wie beim Elektromagnetismus stoßen sich gleich gepolte Ladungen ab und verschieden gepolte ziehen sich an. Die Wechselwirkung zwischen zwei Ladungen nimmt mit zunehmender Entfernung voneinander ab.

2.2 Pucks

Es gibt zwei verschiedene Typen von Pucks. Der *Ball* ist ein Puck mit sehr einfacher Struktur. Sein Ladungspunkt, im Bild mit Q bezeichnet, liegt im Schwerpunkt. Alle Bälle haben denselben Radius und dieselbe Masse, jedoch können sie unterschiedlich geladen sein.



Der Spieler, genannt *Unit*, ist ein komplexerer, etwas größerer Puck mit folgenden Eigenschaften:

- Sein Ladungspunkt Q liegt nicht im Mittelpunkt, sondern auf halbem Radius nach vorne verlagert. Die Unit kann sich innerhalb vorgegebener Schranken beliebig polarisieren und somit einen geladenen Ball mit unterschiedlicher Kraft anziehen oder abstoßen.
- Zusätzlich besitzt die Unit zwei Antriebspunkte, die sich links und rechts auf halben Radius an ihr befinden. An jeden Antriebspunkt kann eine Kraft F_L und F_R entlang der im Bild eingezeichneten parallelen Wirkungslinien angesetzt werden. Die Größe der beiden Kräfte sind innerhalb vorgegebener Schranken unabhängig voneinander wählbar.

Mit diesem Modell des Antriebs können nun alle notwendigen Bewegungen ausgeführt werden: Gleichgroße gleichgerichtete Kräfte führen zu einer reinen Vorwärts- oder Rückwärtsbewegung, gleichgroße entgegengesetzte Kräfte führen zu einer reinen Drehung.

Die äußeren Attribute wie Radius, Masse und die Schranken für Ladung und Antrieb sind für alle Units gleich.

2.3 Sicht und Aktion

Der algorithmische Ablauf für einen **Hoverball**-Spieler ist einfach: Der Spieler empfängt jeweils in kurzen Zeitabständen Informationen über seine momentane Sicht und kann daraufhin seine Aktion bezüglich Ladung und Antrieb versenden. Der Sichtbereich ist dabei eingeschränkt: Ein Spieler sieht nur das, was „vor ihm“ liegt. Für die gekrümmte, in sich geschlossene Ebene definieren wir dies als den Sphärensektor eines bestimmten Winkels, der direkt vor ihm beginnt — standardmäßig die vordere Halbsphäre (180°).

Um präziser navigieren zu können, sind auf der Sphäre sechs Knoten angebracht. Sie bewegen sich nicht und sind kein Hindernis, sondern bieten den Spielern lediglich eine Orientierungshilfe. Sie befinden sich genau an den Durchstoßpunkten der drei kartesischen Koordinatenachsen durch die Sphäre.

2.4 Energie

Antrieb und Ladung kosten Energie. Die Spieler sind hierfür mit einem Energiespeicher ausgestattet, den sie für ihre Aktionen verwenden. Dabei sind die oberen und unteren Schranken der Aktionen (Antrieb und Ladung) proportional abhängig vom verbleibenden Energievorrat des Spielers.

Zur Rückgewinnung der Energie werden die Spieler vom System mit einer konstanten Energiezufuhr versorgt.

2.5 Kommunikation

Den Spielern ist es zusätzlich möglich, miteinander zu kommunizieren. Es existiert ein Kommunikationskanal, auf dem jeder Spieler Nachrichten senden kann.

Eine Nachricht besteht aus einer endlichen Zeichenkette und bleibt solange „hörbar“, bis der entsprechende Spieler eine neue Nachricht sendet.

2.6 Spielregeln

Bei **Hoverball** treten zwei Teams zu je drei Spielern gegeneinander an. Es gibt einen geladenen *Shot-Ball* und zu jedem Team einen ungeladenen *Team-Ball*. Ein Team bekommt einen Punkt, wenn der Shot-Ball den eigenen Team-Ball berührt. Es ist allerdings den Spielern verboten, einen Team-Ball zu berühren. Geschieht dies doch, so wird dem Spieler als Penalty die komplette Energie entzogen und für einen gewissen Zeitraum nicht wieder aufgeladen. Ziel des Spiels ist es nun, innerhalb einer begrenzten Spielzeit möglichst viele Punkte zu erzielen.

Dadurch, dass nur der Shot-Ball geladen und somit auch nur dieser durch Polarisation führbar ist, erhält **Hoverball** seinen Fußball-Charakter: Die Spieler werden versuchen, den Shot-Ball zu erobern, um ihn auf die beweglichen „Tore“ zu schießen!

Etwas sonderbar erscheint zunächst die Tatsache, dass man bei **Hoverball** im Gegensatz zum Fußballspiel dann einen Punkt erhält, wenn der *eigene* Team-Ball angeschossen wird. Dies erlaubt jedoch die Verallgemeinerung von zwei auf beliebig viele spielende Teams, in welcher die Spieler aus jedem Team bestrebt sind, durch Schießen auf ihren eigenen Team-Ball Punkte zu erzielen. Ferner wären auch Spiele mit mehreren Shot-Balls, mit mehr als drei Spielern pro Team und viele weitere Spielvarianten denkbar.

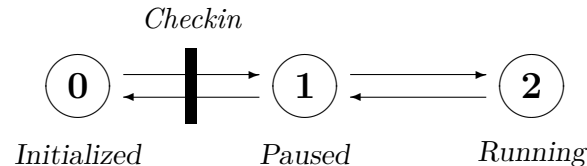
Teil 3

Der Simulator

Um mit dem Simulator umgehen zu können, musst du wissen, wie er konstruiert ist. In diesem Kapitel soll gezeigt werden, wie der Simulator als Automat funktioniert, wie er die Simulation realisiert und wie er die Parameter der **Hoverball**-Welt verwaltet. Schließlich soll die Netzwerk-Kommunikation genau vorgestellt werden.

3.1 Die drei Zustände des Simulators

Die Struktur des **Hoverball**-Simulators entspricht der Struktur eines endlichen Automaten mit drei Zuständen:



- | | |
|--------------------|--|
| <i>Initialized</i> | Der Simulator ist zurückgesetzt und offen für die Anmeldung von Spielern.
Beim Start befindet sich der Simulator in diesem Zustand. |
| <i>Paused</i> | Ein Spiel wurde erstellt, jedoch ist die Simulation angehalten. |
| <i>Running</i> | Die Simulation läuft. |

Grundsätzlich können sich Spieler jederzeit beim Simulator anmelden, jedoch werden neu registrierte Spieler immer erst beim nächsten *Checkin* mit ins Spiel aufgenommen.

3.2 Eigenschaften der Simulation

Wie bei allen Simulationsprogrammen kann die Simulation von **Hoverball** nicht kontinuierlich, sondern nur in diskreten Simulationsschritten erfolgen. Hierfür kann eine *Simulationsfrequenz* angegeben werden, die der Simulator — je nach Rechenaufwand — einzuhalten versucht.

Ausschlaggebend für die Präzision der Simulation ist ein *Präzisionsfaktor*, der ebenfalls als Parameter für die Simulation angegeben werden kann. Die Präzision ist also unabhängig von der Simulationsfrequenz!

Beachtenswert ist noch, dass die **Hoverball**-Zeit proportional an die reale Zeit (und ebenfalls nicht an die Frequenz) gebunden ist. Der Quotient aus **Hoverball**-Zeit durch Real-Zeit wird kurz als *Echtzeitfaktor* Λ („Lambda“) bezeichnet werden.

3.3 Simulator-Variablen

Der Simulator verwaltet eine Variablen-Tabelle, die man sich wie eine zunächst leere Hashtable vorstellen kann. Jedem beliebigen Schlüssel-String kann ein beliebiger Wert-String zugewiesen werden.

Einige Schlüssel sind dabei ausgezeichnet als **Hoverball**-Optionen, um die Parameter der Simulation bestimmen zu können. Solange eine Option nicht explizit definiert wird, verwendet der Simulator als Parameter ihren Default-Wert. Dies ist auch dann so, falls sich der Wert-String einer **Hoverball**-Option nicht eindeutig in eine Dezimalzahl konvertieren lässt (alle Parameter sind numerisch) oder der Wert die angegebenen Schranken verlässt.

3.4 Optionen

Die **Hoverball**-Optionen sind in fünf verschiedene Parameter-Klassen eingeteilt:

<code>simulator</code>	... Parameter, die das Verhalten des Simulators bestimmen.
<code>game</code>	... Parameter, die die Bedingungen des Spiels festlegen.
<code>world</code>	... Parameter der Hoverball -Welt.
<code>unit</code>	... Eigenschaften der Units.
<code>ball</code>	... Eigenschaften der Balls.

Die Schlüssel der **Hoverball**-Optionen haben nun die einheitliche Form "*class.parameter*".

Parameter der Klasse `simulator`

Option	Schranken	Default	Beschreibung
<code>simulator.frequency</code>	$0 < \cdot$	50	Frequenz der Simulation (in Hertz). Die Simulationsfrequenz beeinflusst <i>nicht</i> die Präzision der Simulation.
<code>simulator.time</code>	$\Lambda \quad 0 \leq \cdot$	1	Echtzeitfaktor (in Hertz). Quotient Hoverball -Zeit/Real-Zeit.
<code>simulator.precision</code>	$0 < \cdot$	1	Präzisionsfaktor. Ändert die Präzision und damit auch das Verhalten der Simulation. Achtung!

Parameter der Klasse `game`

Option		Schranken	Default	Beschreibung
<code>game.duration</code>	T	$0 \leq \cdot$	420	Spieldauer.
<code>game.balls.shot</code>		$0 \leq \cdot \leq 99$	1	Anzahl der Shot-Balls.
<code>game.balls.team</code>		$0 \leq \cdot \leq 99$	1	Anzahl der Team-Balls pro Team.
<code>game.timeout</code>		$0 < \cdot$	1	Auszeit für erneutes Punkten.
<code>game.penalty</code>	Θ	$0 \leq \cdot$	10	Dauer eines Penalty.
<code>game.recharge</code>	κ	$0 \leq \cdot$	0.1	Energiezuwachsrate für Units.

Beachte: Die Parameter `game.balls.shot` und `game.balls.team` werden auf den nächsten ganzzahligen Wert gerundet. Alle Zeitangaben sind in **Hoverball**-Sekunden zu verstehen.

Parameter der Klasse `world`

Option		Schranken	Default	Beschreibung
<code>world.radius</code>	R	$0 < \cdot$	50	Radius der Sphäre. (siehe A.7)
<code>world.viscosity</code>	V	$0 \leq \cdot \leq 1$	0.1	Viskosität der Reibung. (siehe A.3)
<code>world.boundary</code>	b	$0 < \cdot$	0.1	Grenzschicht der Reibung. (siehe A.3)
<code>world.permittivity</code>	ε_0	$0 < \cdot$	0.000001	Feldkonstante. (siehe A.5)

Parameter der Klasse `unit`

Option		Schranken	Default	Beschreibung
<code>unit.radius</code>	r	$0 < \cdot$	2	Radius der Units.
<code>unit.mass</code>	m	$0 < \cdot$	4	Masse der Units.
<code>unit.charge.min</code>	Q^-	$\cdot \leq 0$	-1	Stärkste negative Ladung der Units.
<code>unit.charge.max</code>	Q^+	$0 \leq \cdot$	1	Stärkste positive Ladung der Units.
<code>unit.charge.pos</code>	λ_Q	$0 \leq \cdot \leq 1$	0.5	Positionskoeffizient der Ladung. (siehe A.5)
<code>unit.charge.cost</code>	c_Q	$0 \leq \cdot$	10	Energiekosten für die Ladung. (siehe A.10)
<code>unit.engine.min</code>	F^-	$\cdot \leq 0$	-50	Stärkste negative Antriebskraft der Units.
<code>unit.engine.max</code>	F^+	$0 \leq \cdot$	50	Stärkste positive Antriebskraft der Units.
<code>unit.engine.pos</code>	λ_F	$0 \leq \cdot \leq 1$	0.5	Positionskoeffizient der Antriebe. (siehe A.2)
<code>unit.engine.cost</code>	c_F	$0 \leq \cdot$	0.001	Energiekosten für den Antrieb. (siehe A.10)
<code>unit.energy.max</code>	E^+	$0 \leq \cdot$	1	Maximaler Energievorrat. (siehe A.10)
<code>unit.vision</code>	φ	$0 \leq \cdot \leq 2$	1.0	Sichtwinkel der Units. (siehe A.9) Dieser Wert wird noch mit π multipliziert.
<code>unit.message</code>	ℓ	$0 \leq \cdot$	20	Länge der Nachrichten der Units.

Beachte: Der Parameter `unit.message` wird auf den nächsten ganzzahligen Wert gerundet.

Parameter der Klasse ball

Option		Schranken	Default	Beschreibung
ball.radius	r	$0 < \cdot$	1	Radius der Balls.
ball.mass	m	$0 < \cdot$	1	Masse der Balls.
ball.charge	Q	$0 \leq \cdot$	1	Permanente Ladung der Shot-Balls.
ball.half-life		$0 < \cdot$	0.005	A team ball's charge half-life

Beachte: Die Default-Werte der **Hoverball**-Parameter sind also auf die Maße der Balls normiert!

3.5 Orthonormalbasen und Euler-Vektoren

Um die Sicht-Informationen des Simulators interpretieren zu können, musst du wissen, wie der Simulator die Positionen der Pucks auf der Sphäre verwaltet. Hierfür ist ein kleines Quäntchen Mathematik erforderlich, welches aber nicht über die elementare Lineare Algebra hinausgeht.

Der Simulator stützt seine Parametrisierung auf folgende Beobachtung: Wir können die sphärische Position eines Pucks samt Blickrichtung ein-eindeutig mit einer positiv orientierten Orthonormalbasis (x_1, x_2, x_3) des dreidimensionalen Raumes identifizieren. Dabei wählen wir die Zuordnung so, dass die Verlängerung des Vektors x_3 die Sphäre gerade im Mittelpunkt des Pucks durchstößt und der Vektor x_1 in die Blickrichtung des Pucks zeigt. Der Vektor x_2 , der sich ja aus x_1 und x_3 ableiten lässt, ist zunächst nur „schmückendes Beiwerk“, jedoch können wir anhand dieser Identifizierung der Positionen mit Orthonormalbasen nun die Matrizenrechnung zu Hilfe nehmen: Übersetzen wir die Orthonormalbasen in ihre zugehörigen 3×3 -Matrizen (Basisvektoren als Spaltenvektoren), so entspricht die Bewegung eines Pucks nun der Multiplikation seiner Positions-Matrix mit einer Drehmatrix, welche wiederum die Darstellung einer positiv orientierten Orthonormalbasis ist.

Für die Übermittlung einer Positions-Matrix ist es wünschenswert, nicht alle neun Koordinaten der Matrix angeben zu müssen, sondern sie mit so wenig Parametern wie möglich zu charakterisieren — es reichen drei. Euler erfand ein Verfahren, das die Lage einer positiv orientierten Orthonormalbasis in drei Elementardrehungen zerlegt. Der **Hoverball**-Simulator wendet ein ähnliches Verfahren an, jedoch basiert es auf anderen Elementardrehungen als bei Euler. Wir möchten trotzdem die Bezeichnung *Euler-Winkel* und *Euler-Vektor* (die Euler-Winkel als dreidimensionaler Vektor) für die Zerlegung einer Orthonormalbasis in unsere Elementardrehungen beibehalten.

Die Bedeutung der Euler-Winkel können wir uns mit einem naiven Gedankenspiel veranschaulichen. Sei $(\varphi_1, \varphi_2, \varphi_3)$ der Euler-Vektor, den du als Beobachter auf der Sphäre für die Position eines wahrgenommenen Pucks empfängst. Nun kannst du den Puck auf folgende Weise ausfindig machen:

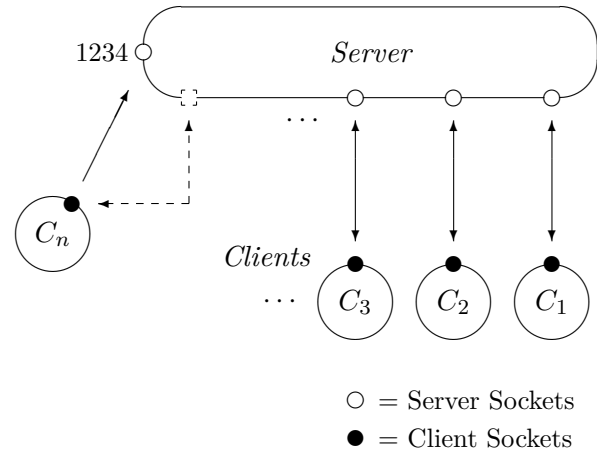
1. Drehe dich um φ_1 .
2. Laufe den sphärischen Winkel φ_2 (also die Strecke φ_2 mal den Sphärenradius) geradeaus.
Nun stehst du auf dem Mittelpunkt des Pucks.
3. Drehe dich um φ_3 .
Nun blickst du auch in dieselbe Richtung wie der Puck.

Eine exakte Definition der Euler-Winkel befindet sich im Anhang A unserer Dokumentation.

3.6 Netzwerk-Kommunikation

Die Netzwerk-Kommunikation von **Hoverball** findet im Server/Client-Stil statt: Der Simulator (*Server*) bedient die Spieler (*Clients*) auf Anfrage mit Sicht-Informationen und empfängt ihre Aktionen. Für diese Kommunikation wird TCP verwendet.

Das Vorgehen ist wie allgemein bekannt: Der Client richtet eine erste Nachricht an einen speziellen Anmelde-Socket des Simulators, welcher den ausgewiesenen Port 1234 (oder entsprechend) besitzt. Der Simulator registriert die Anmeldung, öffnet einen neuen Socket und antwortet bereits über diesen an den Client zurück. Somit kann der Client den neuen Server-Socket identifizieren und ab sofort alle weiteren Nachrichten an diesen richten, bis die Verbindung zwischen Client und Server getrennt wird.



Als Sprache für die Netzwerk-Nachrichten werden einfache Strings der Form

(command argument argument ...)

verwendet, wobei die Argumente atomar oder wiederum geklammerte Strings dieser Form sein können — je nach dem, was das Kommando als Argument verlangt. Argumente werden durch Leerzeichen voneinander getrennt. Soll ein Leerzeichen in einem Argument enthalten sein, etwa beim Team- oder Spielernamen, so muss das Argument in Anführungszeichen "..." gesetzt werden. (Um Anführungszeichen darzustellen, musst du \ " und für den Backslash \\ verwenden.)

Ein besonderes Argument für die Kommandos der Clients ist das Argument *. Es ist das Joker-Argument und bedeutet, dass der alte Wert beibehalten werden soll, falls dies sinnvoll ist. Der Simulator verwendet beim Verschicken seiner Nachrichten dieses Joker-Argument nicht.

3.7 Unit Clients

Verbindet sich ein Spieler mit dem Simulator, so sprechen wir von einem *Unit Client*.

Um die Unit Clients zu verwalten, weist der Simulator ihnen sogenannte *Channels* zu, über die er sie identifiziert. Der Begriff *Channel* suggeriert die nützliche Vorstellung, dass der Simulator mit den Units wie durch separate „Kanäle“ kommuniziert. Die Channels werden durch ein natürliches Zahlenpaar (t, n) identifiziert ($t, n \in \{1 \dots 99\}$). Die Zahl t entspricht der Teamnummer und die Zahl n der Spielernummer der Unit innerhalb ihres Teams. Die Zuweisung der Channels erfolgt bei der Anmeldung der Unit Clients und wird bis zur Trennung nicht mehr geändert. Das Einordnen der Spieler in Teams erfolgt nur über die Team-Namen: Spieler, die sich mit dem gleichen Team-Namen am Simulator anmelden, werden demselben Team zugeordnet.

Die Channel-Kennungen werden nun verwendet, um die Objekte von **Hoverball** für den Unit Client zu identifizieren. Ein Spieler „sieht“ einen anderen Spieler als (**unit** t n). So kann zum Beispiel durch einfaches Vergleichen der Variablen t auf die Team-Zugehörigkeit geschlossen werden. Dieses System wird nun auf die Balls und Nodes übertragen: Ein Ball wird als (**ball** t n) wahrgenommen, wobei die t 's der Bälle den Teamnummern der Spieler entsprechen. Die n 's der Bälle werden dagegen von 1 neu hochgezählt. Für die Shot-Balls ist $t = 0$. Die Knoten werden gemäß ihrer Anordnung mit (**node** 0 n) für $n = 1 \dots 6$ wie die Augen auf einem Spiel-Würfel durchgezählt.

Da die Units nicht den absoluten Zeitstand des Spiels kennen, ist es für sie nur wichtig zu wissen, wann ein Spielabschnitt beginnt (*Running*) und wann er endet (*Paused*). Hierfür legen wir fest, dass ein Spiel mit Empfangen der ersten Sicht als gestartet bzw. fortgesetzt gilt und dass es so lange läuft, bis ein Unterbrechungs-Kommando empfangen wird.

Im Folgenden soll das Protokoll der Kommunikation zwischen Unit Client und Simulator beschrieben werden:

Unit Client to Server	Server to Unit Client
<p>(connect $\#[hash]$ <i>team unit color</i>)</p> <p>Verbindet mit dem Simulator.</p> <ul style="list-style-type: none"> • <i>hash</i> = Hashtag des Spiels • <i>team</i> = Name des Teams • <i>unit</i> = Name des Spielers • <i>color</i> = Farbcode der Form 0xRRGGBB (oder als Dezimalzahl) 	<p>(connect $\#[hash]$ t n <i>team unit color</i>)</p> <p>Akzeptiert die Verbindung. Liefert die Channel-Kennung und bestätigt die Argumente der Anmeldung und das Hashtag.</p> <p>(checkin X^*)</p> <p>Der Spieler wurde in ein Spiel mit den angegebenen Parametern eingecheckt. Er erfährt nicht alle Parameter, sondern nur (und in dieser Reihenfolge):</p> <ul style="list-style-type: none"> • Λ (simulator) • R V b ε_0 (world) • r m $Q^- \dots E^+ \varphi \ell$ (alles von unit) • r m Q (ball)

Unit Client to Server	Server to Unit Client
<p>(look)</p> <p>Erfragt die aktuelle Sicht.</p>	<p>(look $E \vartheta Puck^*$)</p> <p>Liefert die aktuelle Sicht.</p> <p>Knoten werden auch als Pucks wahrgenommen. Der Spieler sieht sich selbst.</p> <ul style="list-style-type: none"> • E = eigene Energie • ϑ = eigene Penalty-Zeit ◦ $Puck ::= (what \ t \ n \ x_1 \ x_2 \ x_3 \ [mess.])$ ◦ $what ::= \mathbf{unit} \mid \mathbf{ball} \mid \mathbf{node}$ • $what \ t \ n$ = Puck-Kennung • (x_1, x_2, x_3) = Relative Position des Pucks (Euler-Vektor) • $message$ = Nachricht (nur bei Units)
<p>(action $Q \ F_L \ F_R \ [message]$)</p> <p>Sendet eine Aktion.</p> <p>Die Parameter Q, F_L, F_R können mit * maskiert werden.</p> <ul style="list-style-type: none"> • Q = neue Ladung • F_L, F_R = neue Antriebskräfte • $message$ = neue Nachricht 	<p>(break)</p> <p>Das Spiel wurde unterbrochen.</p>
<p>(ping)</p> <p>Hält die Verbindung aufrecht.</p>	<p>(ping)</p> <p>Hält die Verbindung aufrecht.</p>
<p>(bye)</p> <p>Trennt die Verbindung zum Simulator.</p>	<p>(bye)</p> <p>Trennt die Verbindung zum Unit Client.</p>

3.8 Control Clients

Neben den Unit Clients gibt es noch die *Control Clients*, die die vollständigen Informationen des Simulators erhalten und den Simulator steuern können. Der **Hoverball**-Controller ist ein solcher Control Client.

Für die Control Clients ist eine erweiterte Syntax der Netzwerk-Sprache zulässig: Akzeptiert und versendet werden hier außer den zuvor beschriebenen einfachen Nachrichten auch zusammengesetzte Nachrichten der Form

$(command \ argument \ argument \ \dots)(command \ argument \ argument \ \dots) \dots$

welche in *einem* String verschickt werden können.

Im Folgenden soll das Protokoll der Kommunikation zwischen Control Client und Simulator genau beschrieben werden:

Control Client to Server	Server to Control Client
<p>(connect #<i>[hash]</i> <i>name</i>)</p> <p>Verbindet mit dem Simulator.</p> <ul style="list-style-type: none"> • <i>hash</i> = Hashtag des Spiels • <i>name</i> = Name des Control Clients 	<p>(connect #<i>[hash]</i> <i>name</i>)</p> <p>Akzeptiert die Verbindung und bestätigt den Anmelde-Namen und das Hashtag.</p> <p>Im Anschluss wird der Control Client über die aktuelle Konfiguration des Simulators informiert.</p>
<p>(set <i>key</i> [<i>value</i>])</p> <p>Setzt oder löscht eine Simulator-Variable.</p> <ul style="list-style-type: none"> • <i>key</i> = Schlüssel der Variable • <i>value</i> = Wert der Variable 	<p>(set <i>key</i> [<i>value</i>])</p> <p>Eine Simulator-Variable wurde gesetzt oder gelöscht.</p> <ul style="list-style-type: none"> • <i>key</i> = Schlüssel der Variable • <i>value</i> = Wert der Variable <p>(channel <i>t n team unit color host</i>)</p> <p>Ein Channel wurde geöffnet.</p> <ul style="list-style-type: none"> • <i>t n</i> = Channel-Kennung • <i>team</i> = Name des Teams • <i>unit</i> = Name des Spielers • <i>color</i> = Farbcode der Form 0xRRGGBB • <i>host</i> = Hostname des Unit Clients
<p>(channel <i>t n</i>)</p> <p>Schließt einen Channel.</p> <ul style="list-style-type: none"> • <i>t n</i> = Channel-Kennung 	<p>(channel <i>t n</i>)</p> <p>Ein Channel wurde geschlossen.</p> <ul style="list-style-type: none"> • <i>t n</i> = Channel-Kennung
<p>(state <i>q</i>)</p> <p>Ändert den Zustand des Simulators.</p> <ul style="list-style-type: none"> • <i>q</i> = neuer Zustand 	<p>(state <i>q</i>)</p> <p>Der Zustand des Simulators wurde geändert.</p> <ul style="list-style-type: none"> • <i>q</i> = neuer Zustand

Control Client to Server	Server to Control Client
<p>(view [time] Puck* Score*)</p> <p>Ändert die Spielsituation.</p> <p>Alle Argumente (außer den Kennungen) können mit * maskiert werden.</p> <ul style="list-style-type: none"> • time = Zeitstand ◦ Puck ::= (what t n x₁ x₂ x₃ v₁ v₂ v₃ r m Q [F_L F_R E ∅ message]) ◦ what ::= unit ball • what t n = Puck-Kennung • (x₁, x₂, x₃) = Absolute Position des Pucks (Euler-Vektor) • (v₁, v₂, v₃) = Geschwindigkeit des Pucks (Euler-Vektor) • r = Radius des Pucks • m = Masse des Pucks • Q = Ladung des Pucks • F_L, F_R = Antrieb (nur bei Units) • E = Energievorrat (nur bei Units) • ∅ = Penalty-Zeit (nur bei Units) • message = Nachricht (nur bei Units) ◦ Score ::= (score t score) • t = Team-Kennung • score = Punktzahl 	<p>(view time Puck* Score*)</p> <p>Liefert die aktuelle Spielsituation.</p> <p>(Beschreibung der Argumente siehe links)</p>
<p>(ping)</p> <p>Hält die Verbindung aufrecht.</p>	<p>(ping)</p> <p>Hält die Verbindung aufrecht.</p>
<p>(bye)</p> <p>Trennt die Verbindung zum Simulator.</p>	<p>(bye)</p> <p>Trennt die Verbindung zum Control Client.</p>

Teil 4

Der Controller

Dieses Kapitel möchte als kleine Referenz einen Überblick über die verschiedenen Funktionen des Controllers geben. Für die folgenden Abschnitte sei eine Verbindung zum **Hoverball**-Simulator, die wir sowohl über das Menu *Hovlet*▷*Connect* sowie über das Eingabefeld „Server:“ mit dem Button [-> <-] aufbauen können, vorausgesetzt.

4.1 Channels

Die Channel-Liste zeigt die mit dem Simulator verbundenen Unit Clients an. Sie kann mit dem Menu-Punkt *View*▷*Channels* eingeblendet werden.

Ein Doppelclick auf einen Channel veranlasst den Simulator, die Verbindung des entsprechenden Channels zu trennen. Wird ein Team ausgewählt, so wird die Verbindung zu allen Spielern des Teams getrennt.

4.2 Options

Mit dem Menu-Punkt *View*▷*Options* können das Options-Feld eingeblendet werden, womit wir auf die Parameter von **Hoverball** zugreifen können. Alle Parameter lassen sich auch während einer Simulation ändern, jedoch werden die Änderungen erst nach dem nächsten Checkin aktiv.

4.3 Control-Buttons

Die Buttons [<<|] [>>|] [>] steuern die Simulation:

[<<|] Setzt den Simulator zurück. Neue Spieler können verbunden werden.

[>>|] Setzt den Simulator zurück und checkt die verbundenen Spieler ein.

[>] Startet/stoppt die Simulation.

Beachte: Die Control-Buttons entsprechen *nicht* den drei Zuständen des Simulators!

4.4 Screen

Mit *View▷Screen* wird der Screen eingeblendet. Die Perspektive der Sphäre kann mit der Mouse gezogen werden. Auch können die Pucks mit der Mouse versetzt und gedreht werden, sofern sich die Simulation im *Paused*-Modus befindet. Als besonderes Feature wird auch der Full Screen Mode unterstützt.

Folgende Tastaturkürzel auf aktivem Screen vereinfachen die Bedienung des Controllers:

DEL	wie [<code><< </code>] — Initialisieren.
INS	wie [<code>>> </code>] — Einchecken.
SPACE	wie [<code>></code>] — Start/Stop.
PAUSE	wie SPACE, allerdings funktioniert PAUSE nur bei bereits bestehendem Spiel.
ENTER,ESC	Aktiviert/deaktiviert den Full Screen Mode.

4.5 Zoom

Die Sphäre kann im Screen gezoomt werden. Wähle den Zoom-Faktor im Menu *View▷Zoom* oder zoome per Hand bei aktivem Screen mit folgenden Tastaturkürzeln:

NUMPAD +	Zoomt die Sphäre heran.
NUMPAD -	Zoomt die Sphäre weg.

4.6 Follow-Mode

Der Screen kann so eingestellt werden, dass er einen bestimmten Puck „verfolgt“. Er dreht die Perspektive dann automatisch immer so, dass der zu verfolgende Puck im Vordergrund ist. Aktiviere den Follow-Mode mit dem Menu-Punkt *View▷Follow Mode*.

Folgende Tastaturkürzel auf aktivem Screen helfen dir beim Verfolgen:

PGUP, PGDN	Follow-Mode aktivieren und blättern.
HOME	Ball verfolgen.
END	Follow-Mode beenden.

4.7 Views

Der Controller kann Momentaufnahmen, genannt *Views*, der **Hoverball**-Welt speichern. Hierzu stehen ihm beliebig viele Speicherplätze zur Verfügung, wobei du auf neun Speicherplätze mit Tastaturkürzeln bei aktivem Screen zugreifen kannst:

CTRL-1...CTRL-9	Belegt Speicherplatz 1...9 mit dem momentanen View.
1...9	Ruft den View von Speicherplatz 1...9 ab.

Mit dem Menu-Punkt *View* ▷ *Copy to Clipboard* wird der aktuelle View als String ins System-Clipboard kopiert. Zusammen mit den `view(...)`-Methoden der Klasse `hoverball.Controller` kann dieser Spielstand nun bei der Entwicklung eines eigenen Teams als Startsituation verwendet werden.

4.8 Debug

Mit dem Menu *Debug* kann das Graphical Debugging der einzelnen Java-Units ein- und ausgeblendet werden. Wie das Graphical Debugging von Java-Units funktioniert und genutzt werden kann, wird in Abschnitt 5.5 beschrieben.

Teil 5

Programmierung

Der **Hoverball**-Simulator kommuniziert in einem Netzwerk über TCP durch Versenden von Strings. Daher kann für die Programmierung von Spielern grundsätzlich jede Programmiersprache benutzt werden, die TCP unterstützt.

Mit *Java* oder alternative Sprachen wie *Groovy*, *Scala*, *Kotlin* oder *Jython*, die auf einer Java Virtual Machine laufen, können die Klassen von **Hoverball** auch direkt eingebunden werden. Dies bietet folgende Vorteile:

- Vordefinierte abstrakte Spieler- und Team-Klassen
(class `hoverball.Unit` und class `hoverball.Team`)
- Netzwerk-Kommunikation ohne Programmieren von Sockets und Parsen von Strings
(class `hoverball.Unit` und class `hoverball.Controller`)
- Vordefinierte, auf **Hoverball** zugeschnittene Mathematik-Bibliothek
(package `hoverball.math`)
- Visualisieren von Spieler-Strategien auf der Sphäre durch „Graphical Debugging“
(package `hoverball.debug`)
- Schnelle Start-Datei mit Einbindung der Spieler und Teams ins **Hoverball**-Framework
(class `hoverball.Session`)
- Bereitstellen eigener Teams innerhalb des Kosmos der Java Virtual Machine
(package `hoverball.team`)

Details hierzu finden sich in der Java API Specification von **Hoverball** (Verzeichnis `docs`). Im Folgenden sollen nur einige Aspekte genauer herausgestellt werden.

5.1 Session

Die Klasse `hoverball.Session` erlaubt das gleichzeitige Starten von Spielern und allen benötigten **Hoverball**-Komponenten wie den Simulator und den Controller und lässt dabei auf einfache Weise vielfältige Möglichkeiten der Konfiguration zu. Sie übernimmt zum Beispiel folgende Aufgaben:

- Öffnen und Konfigurieren von Simulator und Controller
- Öffnen von Teams und Spielern
- Anordnen der Fenster
- Automatisches Verbinden
- Start der Simulation

Das folgende Beispiel öffnet eine **Hoverball**-Session mit Simulator, Controller und einem Human Player, wobei der Screen des Controller sichtbar ist und den Human Player im Follow-Mode verfolgt.

```
import hoverball.*;

public class MySession extends Session
{
    public static void main (String[] args) { new MySession(); }

    public MySession ()
    {
        super("My Session");           // öffne Simulator und Controller
        Human human = new Human("Human"); // öffne Human Player
        add(human);                     // füge Human Player hinzu
        controller.show();              // zeige Controller an
        controller.viewer.show();       // zeige Controller's Screen an
        controller.follow(human);       // verfolge Human Player
        simulator.state(1);             // checkin!
    }
}
```

5.2 Hovlet

Die Struktur und Architektur von **Hoverball** basiert auf der elementaren Programmeinheit *Hovlet*. Jede Programm-Komponente von **Hoverball** — außer dem Simulator — ist ein Hovlet.

Hovlets sind kleine **Hoverball**-Programme, die sich als Unit Client oder als Control Client mit dem Simulator verbinden können. Sie sind ineinander verschachtelbar, so dass sie im nicht-trivialen Fall baumartige Hovlet-Strukturen abbilden können. (So können zum Beispiel Spieler-Hovlets in einem Team-Hovlet zusammengefasst werden.) Diese Hovlet-Bäume werden dann jeweils in einem Fenster (einem sog. *Browser*) angezeigt, die je nach Auswahl ein Hovlet des Baumes in den Vordergrund bringen. Natürlich können einzelne Hovlets auch in jeweils eigenen Browsern angezeigt werden.

Mit dem Wissen, dass alle **Hoverball**-Komponenten Hovlets sind, kannst du nun auch die Session aus dem vorigen Abschnitt besser verstehen: die Session ist Super-Hovlet für Controller, Teams und Spieler. (Im obigen Beispiel wurde neben dem Controller, welcher automatisch in die Session integriert ist, mit dem Befehl `add(human)` der Human-Player in die Hovlet-Struktur

eingefügt. Der Befehl `controller.show()` öffnet einen Browser für die komplette Session-Struktur und bringt den Controller in den Vordergrund.)

5.3 Unit

Der Ausgangspunkt für deine eigenen Spieler liefert die abstrakte Klasse `hoverball.Unit`. Sie stellt sozusagen die „leere Hülle“ eines Spielers dar, die nur noch mit einem Denk-Algorithmus gefüllt werden muss.

Zentral hierfür ist die Methode `loop()`: Sie wird von **Hoverball** aufgerufen, wenn das Spiel beginnt, und sollte erst wieder verlassen werden, wenn das Spiel unterbrochen wird oder beendet ist. Hierzu passend gibt es die Methode `look()`, welche `true` zurückliefert, solange das Spiel andauert, und gleichzeitig die aktuelle Sicht vom Simulator erfragt. Mit dem Befehl `action(...)` kann schließlich eine Aktion gesendet werden.

Somit ergibt sich folgende einfache Struktur für die Java-Unit:

```
public void loop()
{
    ...                // Initialisieren.
    while(look())
    {
        ...            // Denk-Prozess...
        action(...);   // Action!
    }
}
```

Ein Beispiel einer einfachen, allerdings ziemlich „tollpatschigen“ Unit ist im Anhang C wiedergegeben.

5.4 Team

Spieler können in Teams zusammengefasst werden. Die Idee hierzu ist einfach: Füge die Spieler in ein triviales Hovlet mit bündelnder Funktion ein.

Die Klasse `hoverball.Team` übernimmt diese Aufgabe. Der Programm-Code eines Java-Teams könnte etwa so aussehen:

```
import hoverball.*;

public class MyTeam extends Team
{
    public MyTeam ()
    {
        super("My Team");           // eröffne eigenes Team

        add(new MyUnit1());         // füge Spieler 1 hinzu
        add(new MyUnit2());         // füge Spieler 2 hinzu
    }
}
```

```

        add(new MyUnit3());    // füge Spieler 3 hinzu
    }
}

```

5.5 Graphical Debugging

Java-Units haben die Möglichkeit, graphische Elemente direkt auf der Sphäre darzustellen. Hierzu stellt das Package `hoverball.debug` ein paar Grundelemente, genannt *Debugs*, wie Linie, Kreisbogen und Text zur Verfügung, welche aber auch beliebig kombiniert und erweitert werden können. Es besteht auch die Möglichkeit, eigene Debugs zu definieren und auf den kompletten Befehlsvorrat der Grafik von *Java 2D* zurückzugreifen.

Die Java-Unit besitzt den Befehl `debug(...)`, welcher ein Debug-Element relativ zur Position der Unit auf den Screen zeichnet. Der Aufruf dieses Befehls kann überall in der Methode `loop()` erfolgen. Anders als die Nachrichten, die so lange gelten, bis sie überschrieben werden, „hält“ ein gesetztes *Debug* nur bis zum nächsten Aufruf von `look()`.

Folgende Human Unit markiert alle für sie sichtbaren Objekte:

```

import hoverball.*;
import hoverball.math.*;
import hoverball.debug.*;

public class Debuggy extends Human
{
    public Debuggy () { super("Debuggy"); }           // Constructor

    public void loop ()
    {
        while(look())
        {
            for (int i=0; i<pucks.length; ++i) {      // für alle Pucks tue:
                Puck puck = pucks[i];
                if (puck.X == null) continue;         // (wird nur gehört?)
                debug(new Circle(puck.X.c,Math.PI/18)); // Kreis um Puck
                debug(new Text(puck.X.c,puck.id,1),-1); // Identität an Puck
            }
        }
    }
}

```

Beachte: Units mit Graphical Debugging sind am anzeigenden Controller zu registrieren. Dies kann entweder mit der Methode `add(Unit,true)` der **Hoverball**-Session oder mit der Methode `debug(Unit,true)` des Controllers geschehen.

5.6 Operator Overloading

Während *Java* das Konzept des Operator Overloading bis heute ablehnt, bieten oben die genannten Sprach-Alternativen *Groovy*, *Scala*, *Kotlin* oder *Jython* dieses für **Hoverball** überaus nützliche Konzept an.

Die Klassen aus `hoverball.math` unterstützen das Operator Overloading insofern, als dass entsprechende Methoden bereits implementiert sind und „out of the box“ funktionieren. So kann etwa der *Java*-Programmausschnitt

```
Complex a = new Complex(3,1)
Complex b = new Complex(1,2)
Complex c = a.add(b).mul(0.5)

Vector v = new Vector(1,0,0);
Matrix M = puck[i].X.mul(puck[j].X.inv());
v = v.mul(M);
```

beispielsweise in *Groovy* geschrieben werden durch

```
a = new Complex(3,1)
b = new Complex(1,2)
c = (a+b)/2

v = new Vector(1,0,0)
M = puck[i].X * (~ puck[j].X)
v *= M
```

Hierbei sei angemerkt, dass die Matrizenmultiplikation in **Hoverball** entgegen der Standard-Notation der Mathematik *von links nach rechts* gelesen wird. Somit werden auch Befehle wie die unterste Befehlszeile der Code-Beispiele möglich.

An dieser Stelle sei ebenfalls darauf hingewiesen, dass es auch für Java Ansätze gibt, Operator Overloading zu ermöglichen, etwa durch das Projekt *Java-OO* von Artem Melentyev:

<http://amelentev.github.io/java-oo/>

Auch hierfür sind die entsprechenden Methoden in `hoverball.math` bereits definiert.

Anhang A

Mathematische Spezifikation

In diesem Abschnitt wollen wir uns genauer mit der **Hoverball** zugrundeliegenden Mathematik und Physik beschäftigen. Welche Formeln gelten, wenn ein Puck seine Motoren aktiviert oder einen Ball anzieht? Wie ist die Reibung der Pucks auf der Spielfläche modelliert? Was geschieht bei einer Kollision zweier Pucks? Diese und andere Fragen werden wir uns stellen und tief in den Kern der Simulation hineinschauen.

Da sich die Kugeloberfläche des Spielfeldes lokal durch einen zweidimensionalen Vektorraum approximieren lässt, vereinfachen wir zunächst die Situation und verlagern die Spielfläche auf \mathbb{R}^2 , den wir mit der komplexen Zahlenebene \mathbb{C} identifizieren.

A.1 Attribute eines Pucks

Welche Eigenschaften benötigt man zunächst, um den Zustand eines Pucks auf \mathbb{C} zu beschreiben?

Ein Puck besitzt die festen Attribute

- Radius $0 < r \in \mathbb{R}$,
- Masse $0 < m \in \mathbb{R}$,

und für die Fortbewegung in Abhängigkeit der Zeit t

- Position $x(t) \in \mathbb{C}$ (Scheibenmittelpunkt),
- Drehwinkel $\alpha(t) \in \mathbb{R}$ (Winkel im Bogenmaß bzgl. der x -Achse),
- Impuls $p(t) \in \mathbb{C}$,
- Drehimpuls $L(t) \in \mathbb{R}$.

Die aktuelle Geschwindigkeit errechnet sich durch

$$v(t) = \frac{p(t)}{m}$$

und die Drehgeschwindigkeit lautet

$$\omega(t) = \frac{L(t)}{J},$$

wobei $J = \frac{1}{2}mr^2$ das sogenannte *Trägheitsmoment* darstellt. Es ist bemerkenswert, dass wegen $\mathbb{C} \times \mathbb{R} = \mathbb{R}^3$ für eine eindeutige Positions- und Impulsangabe des Pucks jeweils drei reelle Parameter genügen, nämlich einmal das Tupel (x, α) und einmal (p, L) .

Mit diesen Begriffen erhalten wir eine erste Bewegungsgleichung eines unbeschleunigten Pucks ohne Reibung. Ist $t \geq 0$ die in der Simulation verstrichene Zeit, so gilt

$$\begin{aligned} x'(t) &= v(t), & x(0) &= x_0 \in \mathbb{C} \\ \alpha'(t) &= \omega(t), & \alpha(0) &= \alpha_0 \in \mathbb{R}. \end{aligned} \tag{1}$$

A.2 Antrieb

Um sich nach seinen Wünschen fortzubewegen, muss der Puck seinen Impuls und Drehimpuls beeinflussen können. Zu diesem Zweck besitzt er zwei Motoren auf seiner Kreisfläche, mit denen er Kräfte wirken lässt, die den Puck beschleunigen. Die Motoren sind an einer festen Stellung und in einer bestimmten Richtung bezüglich der Position des Pucks angebracht. Wenn wir einen Puck auf dem Ursprung entlang der x -Achse orientieren, so befinden sich die beiden Motoren auf den Punkten

$$r_L = ri \lambda_F \in \mathbb{C} \quad \text{und} \quad r_R = -ri \lambda_F \in \mathbb{C} \quad (\text{mit } 0 \leq \lambda_F \leq 1)$$

und die auftretende Kraft wirkt stets parallel zur x -Achse. Daher besitzen die Pucks für ihre Motoren nur die Wahl zwischen reellen Zahlenwerten F_L und F_R . Die Kraft wirkt nach vorne, falls der Parameter positiv, und nach hinten, falls er negativ ist. Welchen Einfluss besitzen nun die Kräfte dieser Motoren auf die Beschleunigung des Pucks? Dazu untersuchen wir zunächst den allgemeinen Fall, dass eine Kraft F an einer beliebigen Stelle z des Pucks in eine beliebige Richtung wirkt (beide Größen aus der Sicht des Pucks). In diesem Fall gilt nämlich

$$p'(t) = F \cdot e^{i\alpha(t)} \quad \text{und} \quad L'(t) = T,$$

wobei sich das *Drehmoment* T im \mathbb{R}^3 im allgemeinen durch das dreidimensionale Vektorprodukt $T = F \times z$ errechnet. Wegen

$$(a_1, a_2, 0) \times (b_1, b_2, 0) = (0, 0, a_1b_2 - a_2b_1)$$

definieren wir in unserem zweidimensionalen Fall das „*kleine Vektorprodukt*“ $\times : \mathbb{R}^2 \times \mathbb{R}^2 \longrightarrow \mathbb{R}$ durch

$$(a_1, a_2) \times (b_1, b_2) = a_1b_2 - a_2b_1.$$

Im speziellen Fall der Motoren ergeben sich also die Gleichungen

$$\begin{aligned} p'(t) &= (F_L(t) + F_R(t)) \cdot e^{i\alpha(t)} \\ L'(t) &= (F_L(t) \times r_L) + (F_R(t) \times r_R) \\ &= (F_L(t) \times ir \lambda_F) + (F_R(t) \times (-ir \lambda_F)) \\ &= (F_L(t) - F_R(t)) \cdot r \lambda_F \end{aligned} \tag{2}$$

A.3 Reibung

Die Pucks bewegen sich in einem gewissen Medium, etwa Luft. Nach dem Modell der Luftreibung wirkt ihnen beim Fortbewegen in diesem Medium eine Kraft entgegen, die proportional zur Geschwindigkeit steigt, solange die erzeugte Strömung *laminar*, d.h. glattfließend ist (bei schnellen Bewegungen ist die Strömung turbulent und dann wächst der Widerstand quadratisch zur Geschwindigkeit, welches wir aber vernachlässigen wollen). Das Maß der Reibung ist abhängig von der *Viskosität* V des Mediums.

Einem auf der Fläche auftretenden Impuls $p \in \mathbb{C}$ steht also stets eine Gegenkraft $\theta p \cdot p$ entgegen, $0 \leq \theta p$, so dass wir die Reibung als folgende Bewegungsgleichung schreiben können:

$$p'(t) = -\theta p \cdot p(t).$$

Zur Bestimmung dieses Bremsfaktors θp orientieren wir uns am *Gesetz von Stokes* für Kugeln, welches besagt, dass

$$p'(t) = -\frac{6\pi V r}{m} \cdot p(t). \quad (3)$$

Im Falle des Drehimpulses L ziehen wir der Einfachheit halber das *Newtonsche Reibungsgesetz* zu Rate (Gegenkraft = $V \cdot$ Oberfläche des Körpers \cdot Geschwindigkeitsgefälle) und benutzen zur Berechnung des Geschwindigkeitsgefälles eine *Grenzschicht* b , d.h. einen Abstand zum Rand des Pucks, ab der das Medium nicht mehr von der Drehung mitgerissen wird. Dies ergibt die realistisch erscheinende Bewegungsgleichung

$$\begin{aligned} L'(t) &= -V \cdot 2\pi r \cdot \frac{r\omega(t)}{b} \\ &= -V \cdot 2\pi r \cdot \frac{rL(t)}{\frac{1}{2}mr^2b} \\ &= -\frac{4\pi V}{mb} \cdot L(t) = -\theta L \cdot L(t). \end{aligned} \quad (4)$$

A.4 Allgemeine Bewegungsgleichung

Wir sind nun in der Lage, die Bewegung eines Pucks in Abhängigkeit der Motorenwerte $f_L(t)$ und $f_R(t)$ auf der komplexen Ebene in einem System zweier Differentialgleichungen zu beschreiben. Dazu brauchen wir nur die bisher untersuchten Bewegungsgleichungen zusammenzuführen. Es gilt also ganz allgemein wegen (2), (3) und (4)

$$\begin{aligned} v'(t) &= -\theta p \cdot v(t) + \frac{1}{m}[f_L(t) + f_R(t)] \cdot e^{i\alpha(t)} \\ \omega'(t) &= -\theta L \cdot \omega(t) + \frac{1}{J}[f_L(t) - f_R(t)] \cdot r\lambda_F \end{aligned}$$

Und wegen (1)

$$\begin{aligned} x''(t) &= -\theta p \cdot x'(t) + \frac{1}{m}[f_L(t) + f_R(t)] \cdot e^{i\alpha(t)}, & x(0) &= x_0 \in \mathbb{C} \\ \alpha''(t) &= -\theta L \cdot \alpha'(t) + \frac{1}{J}[f_L(t) - f_R(t)] \cdot r\lambda_F, & \alpha(0) &= \alpha_0 \in \mathbb{R} \end{aligned}$$

A.5 Polarisation

Zusätzlich zu den beiden Motoren besitzen die Pucks die Möglichkeit, eine bestimmte Stelle elektrostatisch aufzuladen und andere aufgeladene Pucks anzuziehen oder abzustößen. Wenn wir uns einen Puck auf dem Ursprung entlang der x -Achse orientiert denken, so befindet sich der Polarisationspunkt auf

$$r_Q = r\lambda_Q \in \mathbb{C} \quad (\text{mit } 0 \leq \lambda_Q \leq 1)$$

für die Spieler und im Puck-Mittelpunkt ($r_Q = 0$) für die Bälle.

Welches Gesetz wird hier angewandt? Es handelt sich um die sogenannte *Coulombkraft*, die zwischen zwei elektrostatisch geladenen Körpern wirkt. Seien P_1 und $P_2 \in \mathbb{C}$ zwei Punkte mit der Ladung Q_1 bzw. $Q_2 \in \mathbb{R}$. Dann wirkt an P_1 eine Kraft der Größe

$$F_1 = \frac{1}{4\pi\epsilon_0} \cdot Q_1 Q_2 \cdot \frac{P_1 - P_2}{|P_1 - P_2|^3} \in \mathbb{C}$$

und an P_2 die Kraft

$$F_2 = -F_1.$$

Dabei nennt man $0 < \epsilon_0 \in \mathbb{R}$ die *Feldkonstante*. (Leider sind in der Realität die elektrostatischen Kräfte so gering, dass man keine nennenswerten Effekte erzielen kann. Sie reichen gerade aus, um einen Papierschnipsel an einen Luftballon zu kleben, aber niemals, um schwere Gegenstände zu bewegen.)

Für zwei polarisierte Pucks ergibt sich entsprechend

$$F_1 = \frac{1}{4\pi\epsilon_0} \cdot Q_1 Q_2 \cdot \frac{x_1 + r_{Q1} - x_2 - r_{Q2}}{|x_1 + r_{Q1} - x_2 - r_{Q2}|^3} \quad \text{und} \quad F_2 = -F_1.$$

Da die Ladungen aller Pucks untereinander wechselwirken, erhalten wir mit der Definition

$$F_i(t) = \frac{1}{4\pi\epsilon_0} \cdot \sum_{i \neq j} Q_i(t) Q_j(t) \frac{x_i(t) + r_{Q_i}(t) - x_j(t) - r_{Q_j}(t)}{|x_i(t) + r_{Q_i}(t) - x_j(t) - r_{Q_j}(t)|^3}$$

bei n Pucks ein System von $2n$ Differentialgleichungen der Gestalt

$$\begin{aligned} x_i''(t) &= -\theta p_i \cdot x_i'(t) + \frac{1}{m_i} [f_{Li}(t) + f_{Ri}(t)] \cdot e^{i\alpha_i(t)} + \frac{1}{m_i} F_i(t), & x_i(0) &= x_{i0} \in \mathbb{C} \\ \alpha_i''(t) &= -\theta L_i \cdot \alpha_i'(t) + \frac{1}{J_i} [f_{Li}(t) - f_{Ri}(t)] \cdot r_i \lambda_F & \alpha_i(0) &= \alpha_{i0} \in \mathbb{R} \end{aligned} \quad (i = 1 \dots n)$$

A.6 Kollision und Reflexion

Wenn sich zwei oder mehr Pucks auf dem Spielfeld befinden, kann es zu Zusammenstößen kommen. Seien P_1 und P_2 zwei kollidierende Pucks, d.h. sie liegen genau aneinander. Nun müssen unter Berücksichtigung der aktuellen Eigenschaften der beiden Pucks die Impulse p_1, p_2 und die Drehimpulse L_1, L_2 „umgelenkt“ und zu neuen Impulsen p'_1, p'_2, L'_1, L'_2 verrechnet werden.

Zu Beginn unserer Rechnung wollen wir die zweidimensionalen Vektoren entlang der „Kollisionsachsen“ orientieren. Seien R_1, R_2 die Vektoren von den Puck-Mittelpunkten zum Kollisionspunkt auf dem Rand der Pucks. Dann definieren wir den *normalen* und *tangentialen Einheitsvektor* als

$$e_\perp := \frac{R_1}{r_1} \quad \text{und} \quad e_\parallel := i \cdot e_\perp = i \cdot \frac{R_1}{r_1}.$$

Ferner bemerken wir den Zusammenhang zwischen dem kleinen Vektorprodukt und dem Skalarprodukt:

$$\forall a, b \in \mathbb{C} : \quad a \times b = \langle ia, b \rangle$$

Nun sind wir gerüstet für die folgende Rechnung, in der wir alle Multiplikationen zwischen Vektoren als Skalarprodukt zu verstehen wollen.

Bei der Reflexion wollen wir folgende Gesetze annehmen:

$$\begin{aligned} (1) \text{ Energieerhaltung: } & \frac{p_1^2}{m_1} + \frac{p_2^2}{m_2} + \frac{L_1^2}{J_1} + \frac{L_2^2}{J_2} = \frac{p_1'^2}{m_1} + \frac{p_2'^2}{m_2} + \frac{L_1'^2}{J_1} + \frac{L_2'^2}{J_2} \\ (2) \text{ Impulserhaltung: } & p_1 + p_2 = p_1' + p_2' \end{aligned}$$

Wir gehen ins *Schwerpunktsystem*. Der Schwerpunkt zweier Massenpunkte bewegt sich mit der Geschwindigkeit

$$v_s = \frac{p_1 + p_2}{m_1 + m_2}.$$

Diese Geschwindigkeit müssen wir jetzt, am Anfang der Rechnung, abziehen und ganz am Ende wieder dazuzählen. Im Schwerpunktsystem gilt nun

$$(3) \quad p_1 + p_2 = 0.$$

Wenn wir die Impulse der Pucks in die Komponenten $p_1 = p_{1\perp} + p_{1\parallel}$ und $p_2 = p_{2\perp} + p_{2\parallel}$ bezüglich des Koordinatensystems (e_\perp, e_\parallel) zerlegen, so gilt im Schwerpunktsystem zusätzlich

$$\begin{aligned} (4) \quad \Delta L_1 &= R_1 \times \Delta p_1 = r_1 e_\parallel \cdot \Delta p_1 = r_1 e_\parallel \cdot \Delta p_{1\parallel} \\ (5) \quad \Delta L_2 &= R_2 \times \Delta p_2 = (-r_2 e_\parallel) \cdot (-\Delta p_1) = r_2 e_\parallel \cdot \Delta p_{1\parallel} \quad [\text{beachte (2)}], \end{aligned}$$

wobei „ Δ “ stets die Differenz zwischen dem zu berechnenden neuen und dem alten Wert ausdrücken soll. Zuletzt fordern wir

$$(6) \quad \Delta p_{1\perp} = -2p_{1\perp} \quad (\Leftrightarrow p_{1\perp} = -p_{1\perp}'),$$

d.h. der Puck soll „frontal“ am anderen abprallen.

Wir setzen nun (2) – (6) in (1) ein und lösen nach $\Delta p_{1\parallel}$ auf. Mit $\mu := \frac{1}{m_1} + \frac{1}{m_2}$ folgt:

$$\begin{aligned} (1) \quad 0 &= \frac{p_1^2 - p_1'^2}{m_1} + \frac{p_2^2 - p_2'^2}{m_2} + \frac{L_1^2 - L_1'^2}{J_1} + \frac{L_2^2 - L_2'^2}{J_2} \\ \Rightarrow 0 &= \frac{2p_1 \Delta p_1 + \Delta p_1^2}{m_1} + \frac{2p_2 \Delta p_2 + \Delta p_2^2}{m_2} + \frac{2L_1 \Delta L_1 + \Delta L_1^2}{J_1} + \frac{2L_2 \Delta L_2 + \Delta L_2^2}{J_2} \\ (2), (3) \Rightarrow 0 &= \mu \cdot (2p_1 \Delta p_1 + \Delta p_1^2) + \frac{2L_1 \Delta L_1 + \Delta L_1^2}{J_1} + \frac{2L_2 \Delta L_2 + \Delta L_2^2}{J_2} \\ (4), (5) \Rightarrow 0 &= \mu \cdot (2p_1 \Delta p_1 + \Delta p_1^2) + \frac{2L_1 r_1 e_\parallel \Delta p_{1\parallel} + r_1^2 \Delta p_{1\parallel}^2}{J_1} + \frac{2L_2 r_2 e_\parallel \Delta p_{1\parallel} + r_2^2 \Delta p_{1\parallel}^2}{J_2} \end{aligned}$$

Nun ist wegen (6)

$$2p_1 \Delta p_1 + \Delta p_1^2 = (2p_{1\parallel} \Delta p_{1\parallel} + \Delta p_{1\parallel}^2) + (2p_{1\perp} \Delta p_{1\perp} + \Delta p_{1\perp}^2) = 2p_{1\parallel} \Delta p_{1\parallel} + \Delta p_{1\parallel}^2,$$

und somit gilt:

$$\begin{aligned}
0 &= \mu \cdot (2p_{1\parallel} \Delta p_{1\parallel} + \Delta p_{1\parallel}^2) + \frac{2L_1 r_1 e_{\parallel} \Delta p_{1\parallel} + r_1^2 \Delta p_{1\parallel}^2}{J_1} + \frac{2L_2 r_2 e_{\parallel} \Delta p_{1\parallel} + r_2^2 \Delta p_{1\parallel}^2}{J_2} \\
\Rightarrow 0 &= \left(\mu + \frac{r_1^2}{J_1} + \frac{r_2^2}{J_2} \right) \Delta p_{1\parallel}^2 + \left(2\mu \cdot p_{1\parallel} + \frac{2L_1 r_1}{J_1} e_{\parallel} + \frac{2L_2 r_2}{J_2} e_{\parallel} \right) \Delta p_{1\parallel} \\
&= \left[3\mu \cdot \Delta p_{1\parallel} + 2 \left(\mu \cdot p_{1\parallel} + \frac{L_1 r_1}{J_1} e_{\parallel} + \frac{L_2 r_2}{J_2} e_{\parallel} \right) \right] \cdot \Delta p_{1\parallel}
\end{aligned}$$

Da alle auftretenden Vektoren Vielfache von e_{\parallel} sind, ergibt diese Gleichung nun entweder $\Delta p_{1\parallel} = 0$ (d.h. keine Reibung am Puckrand und somit keine Übertragung von Rotation) oder

$$\Delta p_{1\parallel} = -\frac{2}{3} \left(p_{1\parallel} + \frac{L_1 r_1}{J_1 \mu} e_{\parallel} + \frac{L_2 r_2}{J_2 \mu} e_{\parallel} \right).$$

Wir entscheiden uns für die zweite Lösung. Schließlich gilt:

$$\begin{aligned}
p'_1 &= p_1 + \Delta p_{1\perp} + \Delta p_{1\parallel} \\
p'_2 &= p_2 - \Delta p_{1\perp} - \Delta p_{1\parallel} \\
L'_1 &= L_1 + \Delta L_1 \\
L'_2 &= L_2 + \Delta L_2
\end{aligned}$$

A.7 Sphäre

Wir verlagern das Spiel nun von der zweidimensionalen Ebene auf die Oberfläche einer Sphäre. Dabei versuchen wir, die Physik von **Hoverball** weitestgehend zweidimensional zu belassen. Welche Attribute des Pucks ändern sich durch die neuartige Topologie? Welche bereits entwickelten Formeln können wir übernehmen und welche müssen wir neu überdenken?

Da Pucks im Vergleich zur Sphäre „klein“ sind, können wir die Umgebung eines Pucks lokal in eine Ebene verwandeln. Der Puck besitzt also weiterhin die festen Attribute Radius $r > 0$ und Masse $m > 0$, sowie einen Impuls $p(t) \in \mathbb{C}$ und einen Drehimpuls $L(t) \in \mathbb{R}$. Dadurch lassen sich die Gesetze der Motoren, Reibung und Reflexion auf der Ebene anwenden. Aber wie parametrisieren wir die Position auf der Sphäre?

Hier kommt uns die Matrizenrechnung zu Hilfe: Eine *rechtshändige Orthonormalbasis* (RONB) des \mathbb{R}^3 ist ein System (x_1, x_2, x_3) von drei linear unabhängigen, senkrecht aufeinander stehenden, normierten Vektoren $\in \mathbb{R}^3$, so dass $x_1 \times x_2 = x_3$.

Schreiben wir die Vektoren in dieser Reihenfolge als Spaltenvektoren in eine Matrix, so erhalten wir ein-eindeutig ein Element der multiplikativen Gruppe der *speziellen orthogonalen Abbildungen* $SO(3)$.

Wir können die Position eines Pucks, die auf der Ebene dem zweidimensionalen x und eindimensionalen Drehwinkel α entspricht, auf der Sphäre nun eindeutig mit einem Element $X \in SO(3)$ identifizieren, indem wir uns vorstellen, dass der Puck entlang der X entsprechenden RONB „orientiert“ ist. Der

Mittelpunkt des Pucks befindet sich auf der Sphäre dann genau dort, wo der von x_3 erzeugte Strahl die Sphäre durchstößt.

Eine Bewegung eines Pucks auf der Sphäre entspricht nun der Multiplikation einer Drehmatrix $A \in SO(3)$ mit seiner aktuellen Position. Die zweidimensionalen Attribute der Bewegung des Pucks (v und ω) können auf der Sphäre beibehalten werden, denn wir konstruieren aus ihnen eine Drehmatrix S für festes $dt > 0$ sehr klein wie folgt (für $v(t) \neq 0$, sonst trivial):

$$S(v(t), \omega(t), dt) = D_z(\arg v(t)) \circ D_y\left(\frac{v(t)}{R} dt\right) \circ D_z(\omega(t) dt - \arg v(t))$$

wobei $D_y(\alpha)$ die Drehmatrix um die y -Achse um den Winkel α bedeute und D_z entsprechend definiert sei. Dann ist

$$X(t + dt) = X(t) \circ S(v(t), \omega(t), dt).$$

Da sich durch dieses Verfahren der Puck im Verhältnis zu seiner Geschwindigkeit dreht, muss anschließend $v(t)$ korrigiert und um $\omega(t) dt$ zurückgedreht werden.

Wir können also alle in der Ebene entwickelten Formeln der Bewegung auch auf der Sphäre benutzen (die Position ist neu modelliert) — bis auf eine: Bei der Polarisierung zweier Pucks ist nun zu berücksichtigen, dass sich die Anziehungskraft durch die neue kugelförmige Topologie ändert. Es gibt nämlich keine eindeutige Distanz zwischen zwei Objekten mehr, aber eine kürzeste Distanz. Weitere Distanzen entstehen durch beliebig ganzzahlige Sphärenumrundungen. Demnach erhalten wir bei der Polarisierung zum Beispiel den kuriosen Effekt, dass sich ein abgestoßener Ball auf der gegenüberliegenden Seite der Sphäre einfindet und dort verbleibt.

Sei R der Sphärenradius und seien P_1 und $P_2 \in \mathbb{R}^3$ Positionen zweier polarisierter Punkte auf der Sphäre mit der Ladung Q_1 bzw. $Q_2 \in \mathbb{R}$. Der kürzeste Abstand dieser Punkte zueinander beträgt

$$d = R \cdot \arccos\left(\frac{\langle P_1, P_2 \rangle}{|P_1||P_2|}\right)$$

Dann wirkt an P_1 eine Kraft der Größe

$$f_1 = \frac{1}{4\pi\epsilon_0} \cdot Q_1 Q_2 \cdot \left[\sum_{k \geq 0} \frac{1}{(d + 2\pi R \cdot k)^2} - \sum_{k > 0} \frac{1}{(d - 2\pi R \cdot k)^2} \right] \in \mathbb{R}$$

Die Summe $\sum_{k \geq 0} \frac{1}{(x+k)^2}$ ist nach der Funktionentheorie über die *Gammafunktion* Γ berechenbar:

$$\sum_{k \geq 0} \frac{1}{(x+k)^2} = \Psi'(x) \quad \forall x \in \mathbb{C} \setminus (-\mathbb{N}), \quad \text{wobei} \quad \Psi(x) = \frac{\Gamma'(x)}{\Gamma(x)}.$$

Wir erhalten also eine Anziehungskraft

$$f_1 = \frac{1}{4\pi\epsilon_0} \cdot Q_1 Q_2 \cdot \frac{1}{(2\pi R)^2} \left[\Psi'\left(\frac{d}{2\pi R}\right) - \Psi'\left(\frac{-d}{2\pi R}\right) - \frac{(2\pi R)^2}{d^2} \right] \quad \forall d \neq 2\pi R \cdot n, n \in \mathbb{Z}$$

und f_2 entsprechend.

A.8 Euler-Winkel

Wir wollen nun versuchen, die Positions-Matrizen der Pucks mit möglichst wenig Parametern zu kodieren. Die Lage einer RONB besitzt drei Freiheitsgrade. Es ist also möglich, eine RONB in drei Elementar-Drehungen zu zerlegen. Die hieraus resultierenden Winkel wollen wir als *Euler-Winkel* bezeichnen, auch wenn unser Zerlegungsverfahren sich von dem von Euler erfundenen Verfahren etwas unterscheidet.

Für die Identifizierung der RONB's mit ihren zugehörigen Euler-Winkeln wählen wir nun folgende Funktion:

$$\chi : \mathbb{R}^3 \longrightarrow SO(3), \quad \chi(\varphi_1, \varphi_2, \varphi_3) = D_z(\varphi_1) \circ D_y(\varphi_2) \circ D_z(\varphi_3)$$

Die Funktion χ ist surjektiv und auf ihrer Einschränkung $\chi|[-\pi, \pi) \times [0, \pi] \times [-\pi, \pi)$ bis auf die beiden ausgearteten Fälle ($\varphi_2 = 0$ und $\varphi_2 = \pi$) sogar bijektiv. Somit gibt es eine „Umkehrfunktion“

$$\chi^{-1} : SO(3) \longrightarrow \mathbb{R}^3$$

zur Kodierung von Positions-Matrizen, für die man sich leicht ein geeignetes Berechnungsverfahren überlegen kann.

A.9 Sichtwinkel

Um die Sicht der Spieler einzuschränken, existiert das Attribut „*Sichtwinkel*“ $\varphi \in [0, 2\pi]$, das ihre Sicht wie Scheuklappen nur in einer gewissen Winkelbreite zulässt. Auf einer zweidimensionalen Ebene sähe der Puck also alle die Pucks vor ihm, die in das vom Intervall $[-\frac{\varphi}{2}, \frac{\varphi}{2}]$ aufgespannte Winkelsegment „hineinragen“.

Dazu berechnen wir das Winkelintervall $[\delta^-, \delta^+]$ (mit $0 < \delta^+ - \delta^- < \pi$ und $-2\pi \leq \delta^+ + \delta^- < 2\pi$), in dem sich der beobachtete Puck befindet, und er ist sichtbar, wenn

$$\delta^- \leq \frac{\varphi}{2} \quad \text{und} \quad \delta^+ \geq -\frac{\varphi}{2}.$$

Auf der Sphäre verfahren wir nach dem gleichen Prinzip. Die unterschiedliche Topologie bewirkt hier jedoch, dass die Strahlen, die die Sicht der Pucks begrenzen, auf der gegenüberliegenden Seite der Sphäre wieder zusammenführen und die Spieler somit ein Kugelsegment zu sehen bekommen.

Wie errechnen sich die Werte δ^- und δ^+ ? Wir vereinfachen die Berechnung durch Drehen des schauenden Pucks um die eigene Achse, so dass er den beobachteten Puck gerade vor sich liegen hat. Diesen Drehwinkel γ müssen wir nach der Berechnung dazuzählen. Dann legen wir vom schauenden Puck — relative Position stets $E = (e_1, e_2, e_3)$ — zwei Tangenten links und rechts an den beobachteten Puck — seine relative Position sei nun $X = (\star, \star, x)$, Radius r — und interessieren uns für die Berührungspunkte $v^-, v^+ \in \mathbb{R}^3$.

Diese erhalten wir durch das Gleichungssystem

- (1) $\|v\| = 1$
- (2) $R \cdot \arccos \langle v, x \rangle = r \iff \langle v, x \rangle = \cos \frac{r}{R} =: c$
- (3) $v \times x \perp v \times e_3,$

dessen Lösung wegen $x_2 = 0$ lautet:

$$\begin{aligned} v_3 &= \frac{x_3}{c} \\ v_1 &= \frac{c^2 - x_3^2}{cx_1} \\ v_2^\pm &= \pm \sqrt{1 - v_1^2 - v_3^2} \end{aligned}$$

Nun ist

$$\begin{aligned} \delta^- &= \arg(v_1 + iv_2^-) + \gamma \quad \text{und} \\ \delta^+ &= \arg(v_1 + iv_2^+) + \gamma \end{aligned}$$

A.10 Energie

Die Modellierung der Energie ist sehr einfach gehalten. Die Aktionen der Spieler kosten Energie. Dabei berechnet sich der Energieverbrauch für das Aufbringen der Antriebskraft F durch die Formel $c_F \cdot |F|$ mit einer bestimmten Antriebskostenkonstanten c_F . Die Energiekosten für das Aufbringen der Ladung Q betragen $c_Q \cdot |Q|$ mit einer bestimmten Ladungskostenkonstanten c_Q .

Jeder Spieler verfügt über einen Energievorrat $0 \leq E(t) \leq E^+$. Dieser wird für seine Aktionen aufgewendet. Zusätzlich wird der Energievorrat jedes Spielers vom System kontinuierlich mit einer festen Energiezuwachsrate κ wieder aufgeladen.

Wir erhalten für den Energievorrat $E(t)$ eines Spielers also folgende Differentialgleichung:

$$E'(t) = -c_F \cdot |F_L(t)| - c_F \cdot |F_R(t)| - c_Q \cdot |Q(t)| + \kappa$$

Die effektiven unteren und oberen Schranken für den Antrieb sowie für die Polarisierung eines Spielers hängen linear von seinem Energievorrat $E(t)$ ab. Es gelten die Gleichungen:

$$\begin{aligned} F^- \cdot \frac{E(t)}{E^+} &\leq F_L(t), F_R(t) \leq F^+ \cdot \frac{E(t)}{E^+} \\ Q^- \cdot \frac{E(t)}{E^+} &\leq Q(t) \leq Q^+ \cdot \frac{E(t)}{E^+} \end{aligned}$$

Die tatsächlichen Aktionswerte $F_L(t)$, $F_R(t)$ und $Q(t)$ werden angepasst, wenn sie die effektiven Schranken überschreiten.

Anhang B

Tabellen

B.1 Netzwerk-Protokolle

Unit Client to Server	Server to Unit Client
(connect #[hash] team unit color)	(connect #[hash] t n team unit color)
(look)	(checkin X*)
(action Q F _L F _R [message])	(look E ∅ Puck*)
	(break)
(ping)	(ping)
(bye)	(bye)

Puck ::= (what t n x₁ x₂ x₃ [message]) what ::= unit | ball | node

Control Client to Server	Server to Control Client
(connect #[hash] name)	(connect #[hash] name)
(set key [value])	(set key [value])
	(channel t n team unit color host)
(channel t n)	(channel t n)
(state q)	(state q)
(view [time] Puck* Score*)	(view time Puck* Score*)
(ping)	(ping)
(bye)	(bye)

Puck ::= (what t n x₁ x₂ x₃ v₁ v₂ v₃ r m Q [F_L F_R E ∅ message]) what ::= unit | ball
Score ::= (score t score)

B.2 Hoverball-Optionen

Option		Schranken	Default	Beschreibung
simulator.frequency		$0 < \cdot$	50	Frequenz der Simulation (Hertz)
simulator.time	Λ	$0 \leq \cdot$	1	Echtzeitfaktor (Hertz)
simulator.precision		$0 < \cdot$	1	Präzisionsfaktor
game.duration	T	$0 \leq \cdot$	300	Spieldauer
game.balls.shot		$0 \leq \cdot \leq 99$	1	Anzahl der Shot-Balls
game.balls.team		$0 \leq \cdot \leq 99$	1	Anzahl der Team-Balls pro Team
game.timeout		$0 < \cdot$	1	Auszeit für wiederholtes Punkten
game.penalty	Θ	$0 \leq \cdot$	10	Dauer eines Penalty
game.recharge	κ	$0 \leq \cdot$	0.2	Energiezuwachsrate der Units
world.radius	R	$0 < \cdot$	50	Radius der Sphäre
world.viscosity	V	$0 \leq \cdot \leq 1$	0.1	Viskosität der Reibung
world.boundary	b	$0 < \cdot$	0.1	Grenzschicht der Reibung
world.permittivity	ε_0	$0 < \cdot$	0.000001	Feldkonstante
unit.radius	r	$0 < \cdot$	2	Radius der Units
unit.mass	m	$0 < \cdot$	4	Masse der Units
unit.charge.min	Q^-	$\cdot \leq 0$	-1	Stärkste negative Ladung der Units
unit.charge.max	Q^+	$0 \leq \cdot$	1	Stärkste positive Ladung der Units
unit.charge.pos	λ_Q	$0 \leq \cdot \leq 1$	0.5	Positionskoeffizient der Ladung
unit.charge.cost	c_Q	$0 \leq \cdot$	10	Energiekosten für die Ladung
unit.engine.min	F^-	$\cdot \leq 0$	-50	Stärkste negative Antriebskraft der Units
unit.engine.max	F^+	$0 \leq \cdot$	50	Stärkste positive Antriebskraft der Units
unit.engine.pos	λ_F	$0 \leq \cdot \leq 1$	0.5	Positionskoeffizient der Antriebe
unit.engine.cost	c_F	$0 \leq \cdot$	0.001	Energiekosten für den Antrieb
unit.energy.max	E^+	$0 \leq \cdot$	1	Maximaler Energievorrat
unit.vision	φ	$0 \leq \cdot \leq 2$	1.0	Sichtwinkel der Units ($\cdot \pi$)
unit.message	ℓ	$0 \leq \cdot$	100	Länge der Nachrichten der Units
ball.radius	r	$0 < \cdot$	1	Radius der Balls
ball.mass	m	$0 < \cdot$	1	Masse der Balls
ball.charge	Q	$0 \leq \cdot$	1	Permanente Ladung der Shot-Balls
ball.halflife	Q	$0 \leq \cdot$	0.005	Halbwertszeit der Ladung der Team-Balls

Anhang C

Beispiel „Clumsy“

Die folgende Java-Unit „Clumsy“ verfolgt eine einfache Strategie, um den Shot-Ball gegen ihren Team-Ball zu schießen:

```
import hoverball.*;
import hoverball.math.*;

public class Clumsy extends Unit
{
    public Clumsy ()
    {
        super(null,"Clumsy",0x8888FF); // Name "Clumsy", Farbe hellblau
    }

    public void loop ()
    {
        Sphere sphere = new Sphere(option("world.radius")); // Parameter auslesen...
        double Qmax = option("unit.charge.max");
        double Fmax = option("unit.engine.max");

        while (look())
        {
            Puck ball = puck(BALL,0,1); // das ist der Ball
            Puck goal = puck(BALL,self.t,1); // das ist das "Tor"

            if (ball == null) action(0,-Fmax,Fmax); // sieht keinen Ball? drehe!
            else { // sieht Ball:
                Vector a = (goal != null)? Vector.vec(goal.X.c,ball.X.c) : // berechne
                           Vector.vec(ball.X.c,self.X.c); // Drehachse
                Matrix X = Matrix.mul(ball.X,Matrix.rot(a,(self.r+ball.r)/sphere.rad));
                Complex x = sphere.warp(X.c); // X ist die Schussposition

                double l = -x.arg() + Math.max(0.2,1-10*Math.abs(x.arg())); // drehe und
                double r = x.arg() + Math.max(0.2,1-10*Math.abs(x.arg())); // fahre zu X
                double q = (x.abs() < ball.r)? 0.5 : 0; // falls auf X: Schuss!

                action(q*Qmax,l*Fmax,r*Fmax);
            }
        }
    }
}
```