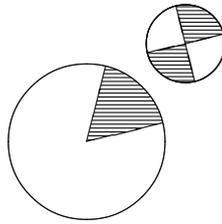


Hoverball Manual

Version 1.4 (2021.01.28)



Stefan Bornhofen
Matthias Bornhofen

www.hoverball.org

Contents

What is Hoverball?	5
1 The Program	6
1.1 Installation	6
1.2 Executable classes	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 The Game	9
2.1 Hoverball's world	9
2.2 Pucks	10
2.3 Vision and action	10
2.4 Energy	10
2.5 Communication	11
2.6 Rules of the game	11
3 The Simulator	12
3.1 The Simulator's three states	12
3.2 Qualities of the simulation	12
3.3 Simulator variables	13
3.4 Options	13
3.5 Orthonormal bases and Euler vectors	15
3.6 Network communication	15
3.7 Unit clients	16
3.8 Control clients	18
4 The Controller	20
4.1 Channels	20
4.2 Options	20
4.3 Control buttons	20

4.4	Screen	21
4.5	Zoom	21
4.6	Follow mode	21
4.7	Views	21
4.8	Debug	22
5	Programming	23
5.1	Session	23
5.2	Hovlets	24
5.3	Units	25
5.4	Teams	25
5.5	Graphical Debugging	26
5.6	Operator Overloading	27
A	Mathematical Specification	28
A.1	A puck's attributes	28
A.2	Propulsion	29
A.3	Friction	29
A.4	Universal equation of motion	30
A.5	Polarization	30
A.6	Collision and reflection	31
A.7	The sphere	33
A.8	Euler angle	34
A.9	Visual angle	35
A.10	Energy	35
B	Tables	37
B.1	Network protocols	37
B.2	Hoverball options	38
C	Example „Clumsy“	39

Introduction

What is Hoverball?

The **Hoverball** project pursues the approach of creating a playful real-time simulation for Multi Agent Systems. At the same time it was paid heed to designing the underlying simulation laws as well as the actual game rules as simple as possible for the purpose of axiomatic mathematics and aesthetics. In doing so, we expect that the development of efficient agents keeps plain in order to contribute to new fundamental insights in Artificial Intelligence.

The available abstract game concept lives up to these postulations. It would be difficult to find another simulation platform who offers such a clear, delightful and quick access to the world of Multi Agent Systems. In addition, real time allows human participation in the simulation. Though **Hoverball** represents basically a challenge for programmers creating efficient teams that compete against each other, it is also an easily accessible computer game for humans.

Hoverball was inspired by the existence of the RoboCup Soccer Simulator aiming at the research of Artificial Intelligence based on a soccer game. As a result, RoboCup has rather complex rules conforming to soccer and to the real world. However, **Hoverball** doesn't claim a suchlike reference to reality and offers a simpler concept: Two dimensional pucks forming two or more teams are placed on a spherical surface, bumping other little pucks in a playful sports contest in order to score by following simple rules.

As for the laws of the simulated world, the objective during the development of the game was to ground the simulator on a sound physical basis: motions, friction, the pucks' collisions — all computations are based on existing physical laws. Thanks to physicist Horst Wilhelm (Papenburg, Germany) we managed to meet this challenge, even though some game parameters differ from the values of respective physical constants.

The simulator is written in Java and consequently operating system independent as well as network-compatible. In the spirit of free software distribution **Hoverball** is an Open Source Project under the terms of the GNU General Public Licence.

Part 1

The Program

Hoverball is completely written in Java and runs on any operating system.

1.1 Installation

Visit **Hoverball** on the internet at

www.hoverhall.org

and download the archive `hoverball.1.4(2021.01.28).en.zip`. It contains:

<code>demo[.*]/</code>	... some examples of the usage of Hoverball .
<code>docs/</code>	... the Hoverball Interface Specification.
<code>src/</code>	... the source code.
<code>contact@...</code>	... for some feedback.
<code>COPYING</code>	... the license agreements.
<code>hoverball.jar</code>	... the program components of Hoverball .
<code>manual.pdf</code>	... this documentation.

To start **Hoverball** you only have to add the archive `hoverball.jar` to the *classpath* of your Java Platform. You can install `hoverball.jar` as Java Extension as well. You'll find out more about this subject in the documentation of your Java Platform.

1.2 Executable classes

The classes of the file `hoverball.jar` are condensed in a package called `hoverball` and in some sub packages. The following classes of this package can be executed as an external application:

<code>class hoverball.Simulator</code>	... Hoverball 's Simulator.
<code>class hoverball.Server</code>	... an independent Hoverball Server.
<code>class hoverball.Controller</code>	... a universal tool controlling the Simulator.

```

class hoverball.Session    ... a standard session.
class hoverball.Human      ... an interface for human players.

```

1.2.1 Simulator

Class: `hoverball.Simulator`

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

The Simulator is responsible for every aspect relating to the management of **Hoverball**'s world and represents the core piece of the program.

If the `:port` is indicated, the Simulator is opened at this port, otherwise it registers at the standard port 1234. (The Simulator's complete network address is composed of `host:port`. So if your computer is called `galileo` and if the Simulator is running at the standard port, his network address is `galileo:1234`.)

All details concerning the **Hoverball** Simulator are described in part 3 of this documentation.

1.2.2 Server

Klasse: `hoverball.Server`

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

The server is able to host multiple games (i.e. multiple simulator instances).

In order to be able to assign clients who are registering to the correct game, a „hashtag“ is transferred when registering, which identifies the game on the server. If the client does not send a hashtag when registering, the server creates a new game with a new hashtag and sends it back to the client.

Currently there is a server running under the network address `hoverball.net` on which **Hoverball** can be played over the Internet!

1.2.3 Controller

Class: `hoverball.Controller`

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

The Controller is a multi-functional tool for the piloting of the **Hoverball** Simulator. When it is connected to the Simulator, it can

- manage the list of players that have connected to the Simulator,
- change game parameters,
- start and stop the simulation,
- display **Hoverball**'s world on screen.

If `host:port#hash` is specified, the Controller automatically connects to the Simulator at this address. The Controller's functions are described in part 4 of this documentation.

1.2.4 Session

Class: `hoverball.Session`

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

The Session forms the base class for a fast and flexible execution of **Hoverball**. It is designed to be able to program with a constant **Hoverball** configuration during the development of a new team.

If this class is executed, it opens a Simulator – at the standard port 1234 or at another port if `:port` is specified — and a Controller connected to it, or on specification of `host:port#hash` it only opens a Controller which automatically connects to the Simulator at this address.

There is a brief introduction to team programming in Java in part 5 of this documentation.

1.2.5 Human player

Class: `hoverball.Human`

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

As the simulation runs in real time, you can easily get in the game by way of a human player. There is a simple method for that using **Hoverball**'s human player: Define your team and player name in the command line and pilot your puck across the sphere by using the **CRSR** keys (move forward and backward, turn to the left and to the right) and the keys **CTRL** and **SHIFT** (attract and shoot the ball)!

The color *color* is given as a hexadecimal six-digit number (e.g. **FFC800** for orange). If *team* is a color word (e.g. **orange**), this color is used for the player.

The following command gives a first insight into **Hoverball**'s world:

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

Click on [>] — and here we go!

Part 2

The Game

Hoverball is the simulation of an abstract soccer game: Two teams compete for a ball and try to score by means of well directed shots. We call **Hoverball** „abstract“ because the virtual environment is reduced to simple geometric shapes, and the motions of the objects obey basic laws of nature.

For this purpose, mathematics gives us an important basis by providing for **Hoverball**'s world and its properties. Any curious reader can find the mathematical specifications that underlie the game and its dynamics in appendix A. For now, let's be satisfied with a more simplified presentation.

Hoverball players can be piloted by agents, that means autonomous computer programs, which process sensory information (e.g. their current vision) into actions (e.g. a certain movement). Now it's up to you to program computer agents for your own team in order to compete with other teams in the **Hoverball** Game.

2.1 Hoverball's world

The game **Hoverball** takes place on a two dimensional plane, except that it is not the Euclid plane but the surface of a three dimensional sphere. Circular pucks are put on it that are able to „hover“ across this bent plane in linear time. These pucks have a defined radius and mass distributed uniformly over their surface area.

The following four physical laws obtain:

1. **Accelerated Motion** — *Basic Law of Mechanics*

Pucks can be turned and accelerated by forces.

2. **Friction** — *Law of Viscous Friction*

Moving pucks slow down without being impacted by external forces. For this, imagine the pucks shifting some „ether“ that mantles the whole sphere homogenously.

3. **Collision** — *Law of Elastic Collision*

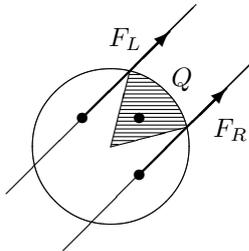
When two pucks crash they rebound without loss of energy. Reflection also affects the pucks' rotation as we premise a 100% stiction at their border.

4. Polarization — Coulomb's Law

Every puck holds a charging point that can polarize positively. Like electro magnetism, identically signed charges repel each other whereas differently signed charges attract. The interaction of two charges fades with increasing distance.

2.2 Pucks

There are two types of pucks. The *ball* is a puck with a very simple structure. Its charging point, called Q in the figure, is right in its center. All balls have the same radius and mass, but they may be differently polarized.



The player, called *unit*, is a more complex and slightly bigger puck:

- Its charging point Q doesn't reside in its center but at the midway to the front. The unit is able to polarize freely within default limits, and so it can attract or repel a charged ball with different forces.
 - What is more, a unit holds two propulsion points situated left and right at the midway between its center and its border. At these points it may apply forces F_L and F_R along the parallel action lines indicated in the figure, these forces being subject to default parameters.
- Equipped in such a manner, all essential movements are possible: For instance, rectified forces cause a mere forward or backward push, antipodal forces induce rotation.

General attributes like radius, mass or the limits of charge and propulsion are identical to all units.

2.3 Vision and action

The player's algorithmic process is quite simple: It receives information about its current vision in short intervals, and thereupon it may send its actions concerning charge and propulsion. However its field of vision is restricted: A player only sees what is „in front of it“. We define this term as the spherical pie of a certain angle starting straight ahead — per default the frontal hemisphere (180°).

To navigate more precisely, there are marked six spots, so-called *nodes*, on the sphere. They don't move and they aren't an obstacle, but they offer a reliable orientation to the players. The six nodes are situated exactly at the piercing points of the three Cartesian axes through the sphere.

2.4 Energy

Engine force and polarisation cost energy. Therefore, the players are equipped with a storage of energy to pay their action costs. Additionally, the lower and upper limits of actions (force and charge) proportionally depend on their remaining energy.

To gain back energy, the players are continuously recharged by the system with a constant energy recharge rate.

2.5 Communication

In addition, the players are able to communicate. There is a communication channel which each player can leave his messages on.

A message consists of a finite string of characters and remains „hearable“ until the sending player overwrites it.

2.6 Rules of the game

In **Hoverball** two competing teams of three players each are put on the sphere. Moreover, there is one unique polarized *shot ball* and one uncharged *team ball* per team. A team scores when the shot ball hits the own team ball. But it is prohibited to the players to touch any team balls themselves. Otherwise the player suffers from complete loss of energy for a certain penalty time period. Now the objective consists in scoring as much as possible within a limited game duration.

Since only the shot ball is charged and thus manageable by polarization, **Hoverball** gains its soccer-like character: Players will wage a fight for the shot ball in order to shoot at the mobile goals.

It may seem weird that, unlike in a soccer game, players score by shooting at their *own* goal. However, this allows an extension to more than two competing teams where every player is anxious to score by targetting at its own team ball. Again, there can be matches with multiple shot balls, more than three players per team and many other options.

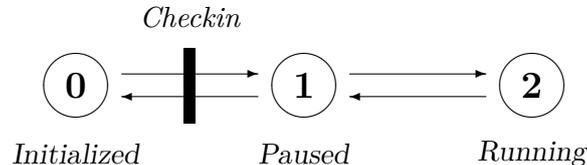
Part 3

The Simulator

To handle the Simulator, you have to know how it is designed. In this chapter we'll see how the Simulator operates as automaton, how it executes the simulation and how it administrates the parameters of **Hoverball**'s world. Lastly, a detailed presentation of its network communication will be given.

3.1 The Simulator's three states

The structure of the **Hoverball** Simulator is equivalent to a three-state automaton:



- Initialized* This is the default state. The Simulator is reset and accepts player registrations.
- Paused* A game has been created but is (still) paused.
- Running* The simulation is running.

In principle, players can register at the Simulator at any time, but newly-registered players are only accepted into the game after the next check in.

3.2 Qualities of the simulation

Like every simulation program, the **Hoverball** Simulator cannot operate continuously but only in discrete steps. Therefore, you can specify the *simulation frequency* the Simulator will try to keep — depending on the resources.

The decisiv parameter for the precision of simulation is a *precision factor*. Precision is thus independent of frequency!

Finally, note that the **Hoverball** time is proportional to real time (and not to frequency!). Let's call Λ („Lambda“) the ratio between **Hoverball** time and real time.

3.3 Simulator variables

The Simulator administrates a table of variables you can picture yourself as a normal, initially void hashtable allocating any value to a key string. Some strings are designated as **Hoverball** options in order to determine the Simulator's parameters. As long as an option isn't defined explicitly in the table, the Simulator uses its default value. It will react in the same way if the value exceeds its limits or cannot be converted into a decimal number (all parameters are numeric).

3.4 Options

The **Hoverball** options are divided into five parameter classes:

<code>simulator</code>	... parameters that determine the Simulator's behavior.
<code>game</code>	... parameters that define game conditions.
<code>world</code>	... parameters of Hoverball 's world.
<code>unit</code>	... unit properties.
<code>ball</code>	... ball properties.

The key strings of the **Hoverball** options have the consistent designation "*class.parameter*".

Parameters of the class `simulator`

Option	Limits	Default	Description
<code>simulator.frequency</code>	$0 < \cdot$	50	Simulation frequency (in Hertz). The Simulation frequency does <i>not</i> affect the calculating precision.
<code>simulator.time</code>	$\Lambda \quad 0 \leq \cdot$	1	Real time coefficient (in Hertz). Ratio Hoverball time/real time.
<code>simulator.precision</code>	$0 < \cdot$	1	Precision coefficient. This changes the precision and thus the simulation behavior. Attention!

Parameters of the class `game`

Option	Limits	Default	Description
<code>game.duration</code>	$T \quad 0 \leq \cdot$	300	Game duration. Specification in Hoverball time units.
<code>game.balls.shot</code>	$0 \leq \cdot \leq 99$	1	Number of shot balls.
<code>game.balls.team</code>	$0 \leq \cdot \leq 99$	1	Number of team balls per team.
<code>game.timeout</code>	$0 < \cdot$	1	Score timeout.
<code>game.penalty</code>	$\Theta \quad 0 \leq \cdot$	10	Penalty duration.
<code>game.recharge</code>	$\kappa \quad 0 \leq \cdot$	0.2	Energy recharge rate for units.

Note: `game.balls.shot` and `game.balls.team` will be rounded to the next integer value. All time specifications are given in **Hoverball** seconds.

Parameters of the class world

Option		Limits	Default	Description
<code>world.radius</code>	R	$0 < \cdot$	50	Sphere radius. (cf. A.7)
<code>world.viscosity</code>	V	$0 \leq \cdot \leq 1$	0.1	Friction viscosity. (cf. A.3)
<code>world.boundary</code>	b	$0 < \cdot$	0.1	Friction boundary layer. (cf. A.3)
<code>world.permittivity</code>	ε_0	$0 < \cdot$	0.000001	Electric field permittivity. (cf. A.5)

Parameters of the class unit

Option		Limits	Default	Description
<code>unit.radius</code>	r	$0 < \cdot$	2	Unit radius.
<code>unit.mass</code>	m	$0 < \cdot$	4	Unit mass.
<code>unit.charge.min</code>	Q^-	$\cdot \leq 0$	-1	Unit's highest negative charge.
<code>unit.charge.max</code>	Q^+	$0 \leq \cdot$	1	Unit's highest positive charge.
<code>unit.charge.pos</code>	λ_Q	$0 \leq \cdot \leq 1$	0.5	Position coefficient of the charge point. (cf. A.5)
<code>unit.charge.cost</code>	c_Q	$0 \leq \cdot$	10	Energy cost for charge. (cf. A.10)
<code>unit.engine.min</code>	F^-	$\cdot \leq 0$	-50	Unit's highest negative engine force.
<code>unit.engine.max</code>	F^+	$0 \leq \cdot$	50	Unit's highest positive engine force.
<code>unit.engine.pos</code>	λ_F	$0 \leq \cdot \leq 1$	0.5	Position coefficient of the propulsion points. (cf. A.2)
<code>unit.engine.cost</code>	c_F	$0 \leq \cdot$	0.001	Energy cost for engine force. (cf. A.10)
<code>unit.energy.max</code>	E^+	$0 \leq \cdot$	1	Unit's highest energy level. (cf. A.10)
<code>unit.vision</code>	φ	$0 \leq \cdot \leq 2$	1.0	Unit's vision angle. (cf. A.9) This value is multiplied by π again.
<code>unit.message</code>	ℓ	$0 \leq \cdot$	100	Length of unit messages.

Note: `unit.message` will be rounded to the next integer value.

Parameters of the class ball

Option		Limits	Default	Description
<code>ball.radius</code>	r	$0 < \cdot$	1	Ball radius.
<code>ball.mass</code>	m	$0 < \cdot$	1	Ball mass.
<code>ball.charge</code>	Q	$0 \leq \cdot$	1	A shot ball's permanent charge.
<code>ball.halflife</code>		$0 < \cdot$	0.005	A team ball's charge halflife

Note: The **Hoverball** default values are adjusted to the ball's dimensions!

3.5 Orthonormal bases and Euler vectors

To construe the Simulator’s vision information you have to know how it governs the pucks’ positions on the sphere. For this purpose, we need a bit of math but nothing that exceeds elementary linear algebra.

The Simulator bases its parametrization upon the following observation: We can identify a puck’s spherical position together with its direction of vision uniquely with a positively oriented three dimensional orthonormal base (x_1, x_2, x_3) . In doing this, we choose the correlation in such a way that the extension of the vector x_3 pierces the sphere exactly in the puck’s center, and that the vector x_1 points to its direction of vision. The vector x_2 seems dispensable as it can be derived from x_1 and x_3 , but due to this identification of positions and orthonormal bases we make use of matrix calculus: If we translate the orthonormal bases into their corresponding 3×3 matrices (using the base vectors as column vectors), a puck’s movement corresponds to a multiplication of its position matrix by a turn matrix described again by a positively oriented orthonormal base.

For the transmission of a position matrix we’d like not to specify all of its nine coordinates, but to make with a number as small as possible — three of them will do. Euler invented a way to decompose a positively oriented orthonormal base into three elementary turns. The **Hoverball** Simulator applies a similar method, but it uses other elementary turns than Euler. Anyway, we want to keep the terms *Euler angle* and *Euler vector* (Euler angles as three dimensional vectors) for **Hoverball**’s kind of decomposition.

The meaning of the Euler angles can be illustrated by a little thought game: Imagine yourself standing on the sphere and let $(\varphi_1, \varphi_2, \varphi_3)$ be an Euler vector you receive, representing the position of a perceived puck. Now you will be able to locate the puck by doing the following:

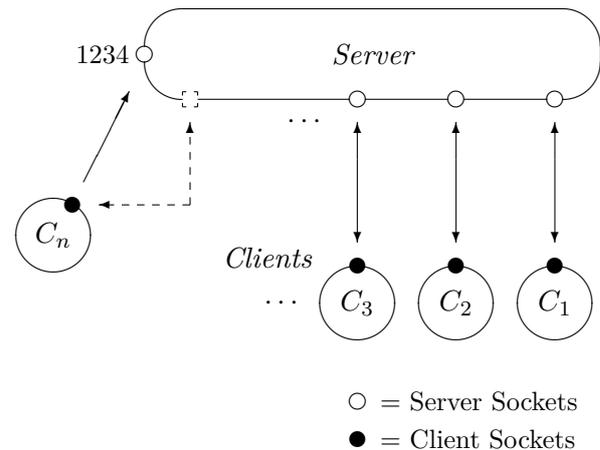
1. Turn around φ_1 .
2. Walk down the spherical angle φ_2 (i.e. the distance φ_2 multiplied by the sphere radius).
Now you are standing in the puck’s center.
3. Turn around φ_3 .
Now both of you are looking in the same direction.

You can find a detailed definition of the Euler angles in appendix A of our documentation.

3.6 Network communication

The **Hoverball** network communication operates in server-client style: The Simulator (*server*) serves the players (*clients*) on request with vision information and receives their actions. This is done via TCP.

The concrete approach is standard: The client addresses its first message to a special registration socket of the Simulator which is the default port 1234 (or likewise). The Simulator observes the registration, opens a socket and answers to the client by use of this new socket. As datagrams always imply their „sender“ (each datagram contains the socket address where it is sent from), the client is able to identify the new server socket and henceforth to address all following messages to it, until client and server get disconnected.



The network language consists of simple strings in the form

(command argument argument ...)

where the arguments can be terminal strings or other non-terminal parenthesized expressions — whatever is required. Terminal expressions are separated by space characters. If a space character is to be included in an expression, e.g. on declaring a team name or a player name, the expression has to be put into quotation marks "...". (If you want to produce quotation marks or backslashes you have to use \" and \\.)

A particular client's expression is the argument *. This wildcard signifies that the old value should be retained if this is useful. The Simulator doesn't employ the wildcard in its expressions.

3.7 Unit clients

When a player connects to the Simulator, we call it a *unit client*. The Simulator now attributes a so-called *channel* to each unit client for a non-ambiguous identification, denoted as the pair $(t, n \in \{1 \dots 99\})$ of natural numbers. The number t corresponds to the team number and n to the unit's player number within its team. This channel assignment happens once on the unit's registration and remains until disconnection. The player's integration into their team takes place only by means of the team name: Players registering with the same team name to the Simulator are assigned to the same team. The channel recognition is used by a unit to identify the **Hoverball** objects on the sphere. A player „sees“ another player as **(unit t n)**. Thus it can find out its team membership just by comparing the t -variables. This system is transferred to balls and nodes: A team ball is perceived as **(ball t n)** where t corresponds to the players' team number, but their n is counted again starting with 1. The t of a shot ball is 0. The nodes are enumerated as **(node 0 n)**, according to their position like the eyes on a die from $n = 1 \dots 6$.

As the units don't know the current time, it is only important for them to realize when a game period begins (*running*) and when it ends (*paused*). For this, we define that a game is started resp. resumed by reception of the first vision, and that it keeps running until a break message is received.

The communication protocol between unit client and Simulator is described below:

Unit Client to Server	Server to Unit Client
<p>(connect #[hash] team unit color)</p> <p>Connects to the Simulator.</p> <ul style="list-style-type: none"> • <i>hash</i> = hashtag of the game • <i>team</i> = team name • <i>unit</i> = player name • <i>color</i> = color code in the form 0xRRGGBB (or as decimal number) 	<p>(connect #[hash] t n team unit color)</p> <p>Accepts the connection. Delivers the channel identification and confirms the registration and the hashtag.</p>
	<p>(checkin X*)</p> <p>The player has been checked in a game with the indicated parameters.</p> <p>The player doesn't get all parameters, but only (in this order):</p> <ul style="list-style-type: none"> • Λ (simulator) • $R V b \varepsilon_0$ (world) • $r m Q^- \dots E^+ \varphi \ell$ (all of unit) • $r m Q$ (ball)
<p>(look)</p> <p>Requests the current vision.</p>	<p>(look $E \vartheta Puck^*$)</p> <p>Sends the current vision.</p> <p>Nodes are perceived as pucks, too. The player sees itself.</p> <ul style="list-style-type: none"> • E = own energy level • ϑ = own penalty time ◦ $Puck ::= (what t n x_1 x_2 x_3 [mess.])$ ◦ $what ::= \text{unit} \text{ball} \text{node}$ • $what t n$ = puck identification • (x_1, x_2, x_3) = the puck's relative position (Euler vector) • $message$ = puck's message (units only)
<p>(action $Q F_L F_R [message]$)</p> <p>Sends an action.</p> <p>The parameters Q, F_L, F_R can be replaced by *.</p> <ul style="list-style-type: none"> • Q = new charge value • F_L, F_R = new engine values • $message$ = new message 	
	<p>(break)</p> <p>The game has been suspended.</p>
<p>(ping)</p> <p>Holds the connection.</p>	<p>(ping)</p> <p>Holds the connection.</p>

Unit Client to Server	Server to Unit Client
(bye) Disconnects from the Simulator.	(bye) Disconnects from the unit client.

3.8 Control clients

Beside unit clients there are *control clients* that control the Simulator and receive its complete information. The **Hoverball** Controller is such a control client.

The control clients use an extended network syntax: Beside the above simple messages they accept and send messages as well in the form

(command argument argument ...)(command argument argument ...)...

which can be packaged in *one* string.

The communication protocol between control client and Simulator works as follows:

Control Client to Server	Server to Control Client
(connect #[<i>hash</i>] <i>name</i>) Connects to the Simulator. <ul style="list-style-type: none"> • <i>hash</i> = hashtag of the game • <i>name</i> = control client's name 	(connect #[<i>hash</i>] <i>name</i>) Accepts the connection and confirms the registered name and the hashtag. Subsequently the control client is informed about the current Simulator's configuration.
(set <i>key</i> [<i>value</i>]) Creates or deletes a Simulator's variable. <ul style="list-style-type: none"> • <i>key</i> = key string • <i>value</i> = variable value 	(set <i>key</i> [<i>value</i>]) A Simulator's variable has been created or deleted. <ul style="list-style-type: none"> • <i>key</i> = key string • <i>value</i> = variable value
	(channel <i>t n team unit color host</i>) A channel has been opened. <ul style="list-style-type: none"> • <i>t n</i> = channel identification • <i>team</i> = team name • <i>unit</i> = player name • <i>color</i> = color code in the form 0xRRGGBB • <i>host</i> = unit client's host name

Control Client to Server	Server to Control Client
<p>(channel $t n$)</p> <p>Closes a Channel.</p> <ul style="list-style-type: none"> • $t n$ = channel identification 	<p>(channel $t n$)</p> <p>A channel has been closed.</p> <ul style="list-style-type: none"> • $t n$ = channel identification
<p>(state q)</p> <p>Changes the Simulator's state.</p> <ul style="list-style-type: none"> • q = new state 	<p>(state q)</p> <p>The Simulator's state has been changed.</p> <ul style="list-style-type: none"> • q = new state
<p>(view [$time$] $Puck^*$ $Score^*$)</p> <p>Changes the game situation.</p> <p>All arguments (except identifications) can be replaced by $*$.</p> <ul style="list-style-type: none"> • $time$ = current time ◦ $Puck ::= (what\ t\ n\ x_1\ x_2\ x_3\ v_1\ v_2\ v_3\ r\ m\ Q\ [F_L\ F_R\ E\ \vartheta\ message])$ ◦ $what ::= \mathbf{unit} \mid \mathbf{ball}$ • $what\ t\ n$ = puck identification • (x_1, x_2, x_3) = puck's absolute position (Euler vector) • (v_1, v_2, v_3) = puck speed (Euler vector) • r = puck radius • m = puck mass • Q = puck charge • F_L, F_R = propulsion (for units only) • E = energy level (for units only) • ϑ = penalty time (for units only) • $message$ = message (for units only) ◦ $Score ::= (\mathbf{score}\ t\ score)$ • t = team identification • $score$ = score 	<p>(view $time\ Puck^*\ Score^*$)</p> <p>Delivers the current game situation.</p> <p>(cf. on the left for description)</p>
<p>(ping)</p> <p>Holds the connection.</p>	<p>(ping)</p> <p>Holds the connection.</p>
<p>(bye)</p> <p>Disconnects from the Simulator.</p>	<p>(bye)</p> <p>Disconnects from the control client.</p>

Part 4

The Controller

This chapter intends to give an overview of the Controller's different functions. Let's suppose a connection to the **Hoverball** Simulator that we can establish via the menu *Hovlet*▷*Connect* as well as via the input field „Server:“ and the button [-> <-].

4.1 Channels

The channel list indicates all unit clients connected to the Simulator. The list can be shown using the menu *View*▷*Channels*.

A double click on a channel makes the Simulator disconnect. If a team is selected, all of its players get disconnected.

4.2 Options

Using the menu *View*▷*Options* you can show the option panel which allows you to access the **Hoverball** parameters. All parameters are changeable during a simulation, but they only take effect after the next check in.

4.3 Control buttons

The buttons [<<|] [>>|] [>] pilot the simulation:

- [<<|] Reset the Simulator. New players can connect.
- [>>|] Reset the Simulator and check in the connected players.
- [>] Start/stop the simulation.

Note: The control buttons do *not* correspond to the Simulator's three states!

4.4 Screen

The screen is opened by means of *View* ▷ *Screen*. You can change the perspective by mouse dragging. Moreover, pucks can be relocated by the mouse when the simulation is paused. As a special feature, Full Screen Mode is also provided.

The following shortcuts simplify the handling:

DEL	like [<<] — Initialize.
INS	like [>>] — Check in.
SPACE	like [>] — Start/stop.
PAUSE	like SPACE, but PAUSE works only if a game is created.
ENTER, ESC	Enter/leave Full Screen Mode.

4.5 Zoom

The sphere can be zoomed. Select the zoom factor via the menu *View* ▷ *Zoom* or zoom by hand on active screen using the shortcuts:

NUMPAD +	Zoom in.
NUMPAD -	Zoom out.

4.6 Follow mode

The screen can be configured to „follow“ a certain puck. It changes the perspective automatically so that the tracked puck is always in the front. Enable the follow mode using the menu *View* ▷ *Follow Mode*.

These shortcuts on active screen will give you a hand:

PGUP, PGDN	Enable follow mode and browse.
HOME	Follow ball.
END	Quit follow mode.

4.7 Views

The Controller is able to save snapshots of **Hoverball**'s world, called *views*. For this purpose, it has any desired memory at its disposal, and you can access nine memory cells via the following shortcuts on active screen:

CTRL-1...CTRL-9	Save the current view in cell 1...9.
1...9	Fetch the view from cell 1...9.

The command *View*▷*Copy to Clipboard* copies the current view as a string into the operating system clipboard. You can now use this view together with the `view(...)` methods of the class `hoverball.Controller` during the development of your own team as starting situation.

4.8 Debug

The menu *Debug* can enable and disable the graphical debugging of each Java unit. Section 5.5 gives a detailing description of how to use graphical debugging.

Part 5

Programming

The **Hoverball**-Simulator communicates by sending strings in a network via TCP. Therefore, basically any programming language that supports TCP can be used for programming players.

With *Java* or alternative languages such as *Groovy*, *Scala*, *Kotlin* or *Jython*, which run on a Java Virtual Machine, you can include the classes from **Hoverball** directly. This offers the following advantages:

- Predefined abstract player and team classes
(class `hoverball.Unit` and class `hoverball.Team`)
- Network communication without programming sockets and parsing strings
(class `hoverball.Unit` and class `hoverball.Controller`)
- Predefined math library perfect fit to **Hoverball**
(package `hoverball.math`)
- Visualization of player strategies on the sphere by „Graphical Debugging“
(package `hoverball.debug`)
- Quick start file with the integration of players and teams in the **Hoverball** framework
(class `hoverball.session`)
- Provision of your own teams within the cosmos of the Java Virtual Machine
(package `hoverball.team`)

All details can be found in the Java API Specification from **Hoverball** (directory `docs`). In the following, only a few aspects will be highlighted in more detail.

5.1 Session

The class `hoverball.Session` allows the simultaneous execution of your players and all other required **Hoverball** components, offering a lot of configuration possibilities. For instance, it fulfills the following tasks:

- Opens and configures the Simulator and the Controller

- Opens team and player programs
- Coordinates the **Hoverball** Frames
- Connects automatically
- Starts the simulation

In the example below, a **Hoverball** Session opens a Simulator, a Controller and a human player, showing the Controller's screen and activating the follow mode on the human player.

```
import hoverball.*;

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

    public MySession ()
    {
        super("My Session");           // open Simulator and Controller
        Human human = new Human("Human"); // open Human Player
        add(human);                    // add Human Player
        controller.show();             // show Controller
        controller.viewer.show();      // show Controller's screen
        controller.follow(human);     // follow Human Player
        simulator.state(1);           // check in!
    }
}
```

5.2 Hovlets

The architecture of **Hoverball** is based on the fundamental programming unit *Hovlet*. Every component of **Hoverball** — except for the Simulator — is a hovlet.

Hovlets are small **Hoverball** programs that are able to connect to the Simulator as unit client or control client. They can be interlaced into each other building arborescent hovlet structures. (For example, player hovlets can be pooled to a team hovlet.) These hovlet trees are displayed by a single frame (so-called *browser*), that pops up a certain hovlet on request. Of course, hovlets can also be displayed seperately in individual browser windows.

This information lets you better understand the session code of the precedent section: The session is a super-hovlet for Controller, teams and players! (Besides the Controller which is automatically integrated to the session, the command `add(human)` adds the human player to the hovlet structure. The command `controller.show()` opens a browser for the complete session structure and pops up the Controller.)

5.3 Units

The abstract class `hoverball.Unit` is the starting point for your own players. It effectively represents the empty shell of a player that only has to be filled by a thinking algorithm.

For this purpose, use the essential method `loop()`: It is called up by **Hoverball** when the game starts and should only be left when the game is resumed or stopped. The method `look()` returns `true` as long as the game is in progress, and at the same time asks the Simulator for the current vision. The command `action(...)` finally sends an action to the Simulator.

The following simple structure arises for a Java unit:

```
public void loop()
{
    ...           // initialize.
    while(look())
    {
        ...       // thinking...
        action(...); // action!
    }
}
```

There is another example of an uncomplex and somewhat „clumsy“ Java unit in appendix C.

5.4 Teams

Players can be pooled together in teams. The idea is pretty simple: Add the players to a trivial bundling hovlet.

The class `hoverball.Team` performs that task. The programming code of a Java team may look like this:

```
import hoverball.*;

public class MyTeam extends Team
{
    public MyTeam ()
    {
        super("My Team");           // create the team

        add(new MyUnit1());        // add player 1
        add(new MyUnit2());        // add player 2
        add(new MyUnit3());        // add player 3
    }
}
```

5.5 Graphical Debugging

Java units are able to plot graphical elements directly on the sphere. The package `hoverball.debug` offers some base elements, called *debugs*, like lines, circular arcs and text that can be arbitrarily combined and extended. Moreover, it is possible to define private debugs accessing the whole graphical command pool of *Java 2D*.

The Java unit command `debug(...)`, which plots a debug element on screen relatively to the unit's position, can be called anywhere inside the method `loop()`. Unlike the messages, which are valid until they are overwritten, „survives“ a plotted debug until the next call of `look()`.

This human unit identifies each object visible to it:

```
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) {      // for all pucks do:
                Puck puck = pucks[i];
                if (puck.X == null) continue;        // (only heard?)
                debug(new Circle(puck.X.c,Math.PI/18)); // circle around puck
                debug(new Text(puck.X.c,puck.id,1),-1); // identity of puck
            }
        }
    }
}
```

Note: Units with Graphical Debugging are to be registered at the displaying Controller. This can be done either by the method `add(Unit,true)` of the **Hoverball** session, or by the Controller's method `debug(Unit,true)`.

5.6 Operator Overloading

While *Java* rejects the concept of operator overloading to this day, the language alternatives *Groovy*, *Scala*, *Kotlin* or *Jython* mentioned above offer this for **Hoverball** extremely useful concept.

The classes from `hoverball.math` support Operator Overloading insofar as the corresponding methods are already implemented and work „out of the box“. For example, the *Java* snippet

```
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);
```

can be written in *Groovy* like

```
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
```

It should be noted here that the matrix multiplication in **Hoverball** is read *from left to right*, contrary to the standard notation in mathematics. This means that commands such as the bottom command line of the code examples are also possible.

At this point it should be pointed out that there are also approaches for Java to enable Operator Overloading, for example through the *Java-OO* project by Artem Melentyev:

```
http://amelentev.github.io/java-oo/
```

The corresponding methods for this are also defined in `hoverball.math`.

Appendix A

Mathematical Specification

In this chapter we want to concentrate on the mathematics and physics that underlie **Hoverball**. What formulas are used when a puck activates its engines or attracts a ball? How is friction modeled? What happens on a collision between two pucks? We will study these and other questions by looking deeply into the simulation kernel.

As the sphere surface can be locally approximated by a two dimensional vector space, let's first simplify the situation and shift the playing area to \mathbb{R}^2 that we identify with the field of complex numbers \mathbb{C} .

A.1 A puck's attributes

What properties are required to describe a puck's status on \mathbb{C} ?

A puck has the steady attributes

- radius $0 < r \in \mathbb{R}$,
- mass $0 < m \in \mathbb{R}$,

and for its movement depending on time t

- position $x(t) \in \mathbb{C}$ (puck's disc center),
- angular position $\alpha(t) \in \mathbb{R}$ (in radian relative to the x -axis),
- momentum $p(t) \in \mathbb{C}$,
- angular momentum $L(t) \in \mathbb{R}$.

The current velocity is calculated by

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

and the angular velocity is

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

where $J = \frac{1}{2}mr^2$ means the so-called *momentum of inertia*. Note that due to $\mathbb{C} \times \mathbb{R} = \mathbb{R}^3$ we only need three real parameters for a definite specification of positions and momenta, namely the tuples (x, α) and (p, L) .

With this terminology we get the first equation of motion of a non-accelerated puck without friction. If $t \geq 0$ is the elapsed simulation time, we have

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

A.2 Propulsion

In order to move at will, a puck has to influence its momentum and angular momentum. For this purpose it holds two engines by the use of which it produces accelerating forces. The engines are mounted at a defined location and in a defined direction relative to the puck's position. When we place the puck at the origin aligned to the x -axis, the engines are situated at the points

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

and the appearing forces always act parallel to the x -axis. That's why the pucks only choose among real number values F_L and F_R . The force pushes the puck forward if the value is positive, and backward if it is negative. What is the impact of these engine forces on acceleration? At first we should inspect the universal case of a force F acting in an arbitrary direction at an arbitrary position z on the puck's surface (the two values relative to the puck). For in this case we obtain

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

where the *rotational inertia* T in \mathbb{R}^3 is generally calculated by the three dimensional vector product $T = F \times z$. Because of

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

we define the „small vector product“ $\times : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}$ in our two dimensional case by

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

In the special case of the puck's engines, the following equations arise:

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

A.3 Friction

The pucks move in a certain medium like air. According to the model of air friction, motion in this medium is countered by a force that increases proportionally to velocity, as long as the generated flow is *laminar*, i.e. unruffled (we disregard that on fast motions the flow gets turbulent, and the drag has a squared growth rate). The quantum of friction depends on the medium *viscosity* V .

An occurring momentum $p \in \mathbb{C}$ is thus opposed by a reaction force $\theta p \cdot p$, $0 \leq \theta p$, so that we can write friction as follows:

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

To determine the slowing coefficient θp we abide by *Stokes' Law* for the drag on a sphere, stating that

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

To simplify matters in the case of angular momentum L , we consult *Newton's Law of Viscosity* (reacting force = $V \cdot$ body surface \cdot velocity gradient) and use a *boundary layer* to calculate the velocity gradient b . The boundary layer describes the distance to the puck's border where the medium is no longer carried along by rotation. This leads to the lifelike equation of motion

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

A.4 Universal equation of motion

We are now in the position to describe a puck's motion on the complex plane, depending on its engine forces $f_L(t)$ and $f_R(t)$ by a system of two differential equations. For this, we only have to bring together the two equations of motion we examined so far. There is altogether on account of (2), (3) and (4)

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

and due to (1) we obtain

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

A.5 Polarization

In addition to the engines, the players are able to polarize a defined point of its surface area which attracts or repels other polarized pucks. When we place the puck at the origin aligned to the x -axis, the charging point is situated at

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

for players and at the center ($r_Q = 0$) for balls. Which law is applied here? It is the so-called *Coulomb force* that acts between two electrostatically charged bodies. Let P_1 and $P_2 \in \mathbb{C}$ be two points with charges Q_1 resp. $Q_2 \in \mathbb{R}$. Then we observe a force operating at P_1 whose value is

$$F_1 = \frac{1}{4\pi\varepsilon_0} \cdot Q_1 Q_2 \cdot \frac{P_1 - P_2}{|P_1 - P_2|^3} \in \mathbb{C}$$

and at P_2 we have

$$F_2 = -F_1.$$

The constant $0 < \varepsilon_0 \in \mathbb{R}$ is the *electric field constant*. (Unfortunately, in reality electrostatic forces are so small that you cannot obtain any notable effects. They are just sufficient to stick some little scraps of paper to a balloon, but never strong enough to move heavy objects.)

For two polarized pucks, we get

$$F_1 = \frac{1}{4\pi\varepsilon_0} \cdot Q_1 Q_2 \cdot \frac{x_1 + r_{Q1} - x_2 - r_{Q2}}{|x_1 + r_{Q1} - x_2 - r_{Q2}|^3} \quad \text{and} \quad F_2 = -F_1.$$

Since all charges of n pucks interact, attending to the definition

$$F_i(t) = \frac{1}{4\pi\varepsilon_0} \cdot \sum_{i \neq j} Q_i(t) Q_j(t) \frac{x_i(t) + r_{Q_i}(t) - x_j(t) - r_{Q_j}(t)}{|x_i(t) + r_{Q_i}(t) - x_j(t) - r_{Q_j}(t)|^3}$$

we obtain a system of $2n$ differential equations in the form

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

A.6 Collision and reflection

When two or more pucks move on the sphere, collisions can occur. Let P_1 and P_2 be two colliding pucks, i.e. they just adjoin to one another. Now we have to redirect their momenta p_1, p_2 and angular momenta L_1, L_2 in consideration of their current properties by converting them into new momenta p'_1, p'_2, L'_1, L'_2 .

At the beginning let's align the two dimensional vectors along the coordinate axes. If R_1, R_2 are the vectors pointing from the pucks' center to the collision point at their borders, we define the *normal* and *tangential unit vector* as

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

Moreover, we can observe the correlation between the small vector product and the dot product:

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

Now we are set up for the following calculation in which we want to consider all vector multiplications as dot products. We want to suppose the laws

$$\begin{aligned} (1) \text{ Conservation of Energy:} & \quad \frac{p_1^2}{m_1} + \frac{p_2^2}{m_2} + \frac{L_1^2}{J_1} + \frac{L_2^2}{J_2} = \frac{p_1'^2}{m_1} + \frac{p_2'^2}{m_2} + \frac{L_1'^2}{J_1} + \frac{L_2'^2}{J_2} \\ (2) \text{ Conservation of Momentum:} & \quad p_1 + p_2 = p_1' + p_2' \end{aligned}$$

We translate the problem to the *barycentric system*. The barycenter of two mass points moves with the velocity

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

which we have to subtract at the beginning and add at the end of the calculation. In the barycentric system is

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

If we decompose the pucks' momenta into the components $p_1 = p_{1\perp} + p_{1\parallel}$ and $p_2 = p_{2\perp} + p_{2\parallel}$ in respect of the coordinate system (e_\perp, e_\parallel) , we can use additional laws that are true in barycentric systems:

$$(4) \quad \Delta L_1 = R_1 \times \Delta p_1 = r_1 e_\parallel \cdot \Delta p_1 = r_1 e_\parallel \cdot \Delta p_{1\parallel}$$

$$(5) \quad \Delta L_2 = R_2 \times \Delta p_2 = (-r_2 e_\parallel) \cdot (-\Delta p_1) = r_2 e_\parallel \cdot \Delta p_{1\parallel} \quad [\text{regard (2)}],$$

where „ Δ “ always means the difference between the new and the old value. Finally we stipulate that

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

i.e. the puck rebounds „head-on“. We put (2) – (6) into (1) and solve for $\Delta p_{1\parallel}$. Then we get applying $\mu := \frac{1}{m_1} + \frac{1}{m_2}$:

$$(1) \quad 0 = \frac{p_1^2 - p_1'^2}{m_1} + \frac{p_2^2 - p_2'^2}{m_2} + \frac{L_1^2 - L_1'^2}{J_1} + \frac{L_2^2 - L_2'^2}{J_2}$$

$$\Rightarrow 0 = \frac{2p_1 \Delta p_1 + \Delta p_1^2}{m_1} + \frac{2p_2 \Delta p_2 + \Delta p_2^2}{m_2} + \frac{2L_1 \Delta L_1 + \Delta L_1^2}{J_1} + \frac{2L_2 \Delta L_2 + \Delta L_2^2}{J_2}$$

$$(2), (3) \Rightarrow 0 = \mu \cdot (2p_1 \Delta p_1 + \Delta p_1^2) + \frac{2L_1 \Delta L_1 + \Delta L_1^2}{J_1} + \frac{2L_2 \Delta L_2 + \Delta L_2^2}{J_2}$$

$$(4), (5) \Rightarrow 0 = \mu \cdot (2p_1 \Delta p_1 + \Delta p_1^2) + \frac{2L_1 r_1 e_\parallel \Delta p_{1\parallel} + r_1^2 \Delta p_{1\parallel}^2}{J_1} + \frac{2L_2 r_2 e_\parallel \Delta p_{1\parallel} + r_2^2 \Delta p_{1\parallel}^2}{J_2}$$

Because of (6) we obtain the following formula:

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

and hence

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

As all appearing vectors are multiples of e_{\parallel} , the equation results either in $\Delta p_{1\parallel} = 0$ (i.e. no stiction at the pucks' border and thus no transmission of rotation) or

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

We opt for the second solution and eventually apply

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

A.7 The sphere

Now we reshift the game from the two dimensional plane back onto the sphere surface. In doing so, we try to keep the physics of **Hoverball** as two dimensional as possible. Which attributes change because of the new topology? Which already developed formulas can we retain and which ones have to be reviewed?

As pucks are „small“ in comparison to the sphere, we can transform the puck's neighborhood locally into a plane. Hence the puck maintains the determined attributes radius $r > 0$ and mass $m > 0$ as well as momentum $p(t) \in \mathbb{C}$ and angular momentum $L(t) \in \mathbb{R}$, and all laws about planar propulsion, friction and reflection apply. But how do we parametrize a puck's position on the sphere?

The matrix calculus comes in handy: A *right-handed orthonormal basis* (RONB) of the \mathbb{R}^3 is a system (x_1, x_2, x_3) of three linearly independent, pairwise orthogonal unit vectors $\in \mathbb{R}^3$ so that $x_1 \times x_2 = x_3$.

If we consider the vectors in this order as column vectors of a matrix, we get a unique element of the multiplicative group of the special orthogonal matrices $SO(3)$. The puck's position which corresponds on the plane to the two dimensional x and the one dimensional angular position α can be identified on the sphere with an element $X \in SO(3)$, if we imagine the puck „oriented“ along the X -associated RONB. The puck's center on the sphere is thus situated exactly where the ray generated by x_3 pierces the sphere.

A movement on the sphere corresponds to a multiplication of the current position by a turn matrix $A \in SO(3)$. The two dimensional attributes of motion (v and ω) can be maintained for we build of them a turn matrix S for a definite small $dt > 0$ as follows ($v(t) \neq 0$, else trivial):

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

where $D_y(\alpha)$ indicates the turn matrix around the y -axis by the angle α , and D_z is respectively defined. Then the puck moves by

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

As this method turns the puck in comparison to its velocity, we have to adjust $v(t)$ and turn it back by $\omega(t) dt$.

We can thus also use all planar formulas of motion on the sphere (position is remodeled) — except one: As to the polarization of two pucks we have to take into account that the attraction changes owing to the new topology. That is to say that there is no definite distance between two objects any more, but a shortest distance, other distances result from any often repeated orbits around the sphere. This gives rise to the odd behavior that a repelled ball moves to the opposite side of the sphere and remains there.

Given the sphere radius R and positions P_1 und $P_2 \in \mathbb{R}^3$ of two polarized points on the sphere with charges Q_1 resp. $Q_2 \in \mathbb{R}$, the shortest distance between these points is

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

Then there is a force acting at P_1

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

According to function theory, the sum $\sum_{k \geq 0} \frac{1}{(x+k)^2}$ can be calculated via the Gamma function Γ :

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

Hence we get an attraction

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

and f_2 symmetrically.

A.8 Euler angle

How can we encode the pucks' position matrices with the least number of parameters? The location of a RONB has three degrees of freedom. It is thus possible to decompose a RONB into three elementary turns. The resulting angles are called *Euler angles*, even if our decomposition method differs slightly from the Euler one.

To indentify the RONB's with their corresponding Euler angles we choose the following function:

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

The function χ is surjective and on its restriction $\chi|[-\pi, \pi) \times [0, \pi) \times [-\pi, \pi)$ even one-to-one except for the two degenerate cases ($\varphi_2 = 0$ and $\varphi_2 = \pi$). There is thus an „inverse function“

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

for the encoding of position matrices whose method of calculation can be easily found.

A.9 Visual angle

To restrict a player's field of vision there is the attribute „visual angle“ $\varphi \in [0, 2\pi]$ allowing a blinker-like vision only within a certain angular interval. On a two dimensional plane a puck would see all pucks in front of it that overlap the angular segment spanned by the interval $[-\frac{\varphi}{2}, \frac{\varphi}{2}]$.

For this we calculate the angular interval $[\delta^-, \delta^+]$ (with $0 < \delta^+ - \delta^- < \pi$ and $-2\pi \leq \delta^+ + \delta^- < 2\pi$), in which the observed puck is situated, and it is visible if

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

On the sphere we proceed in the same way: Every player only perceives objects within restraining borders, however the different topology has the effect that the rays which limit a puck's field of vision intersect again on the opposite side of the sphere, so that player see a sphere segment. How do we calculate the values δ^- and δ^+ ? We simplify the matter by turning the observing puck so that the observed one lies straight ahead. We shall add the angle γ at the end of the calculus. Then we draw two tangents from the observing puck — its relative position is always $E = (e_1, e_2, e_3)$ — left and right adjacent to the observed puck — let its relative position be $X = (\star, \star, x)$, radius r — and are interested in the boundary points $v^-, v^+ \in \mathbb{R}^3$. We obtain them by means of the following system of equations

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

whose solution is on account of $x_2 = 0$:

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

Now we conclude by

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

A.10 Energy

The modeling of the energy is kept very simple. The actions of the players cost energy. The energy consumption for applying the driving force is calculated using F the formula $c_F \cdot |F|$ with a certain drive cost constant c_F . The energy costs for applying the charge are $Q \ c_Q \cdot |Q|$ with a certain charge cost constant c_Q .

Each player has an energy supply $0 \leq E(t) \leq E^+$. This is used for his actions. In addition, the energy supply of each player is continuously recharged by the system with a fixed energy increase rate κ .

We get the following differential equation for the energy reserve $E(t)$ of a player:

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

The effective lower and upper limits for the drive and for the polarization of a player depend linearly on his energy reserve $E(t)$. The equations apply:

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

The actual action values $F_L(t)$, $F_R(t)$ and $Q(t)$ are adjusted if they exceed the effective limits.

Appendix B

Tables

B.1 Network protocols

Unit client to Server	Server to Unit client
(connect #[<i>hash</i>] <i>team unit color</i>)	(connect #[<i>hash</i>] <i>t n team unit color</i>)
(look)	(checkin <i>X</i> *)
(action <i>Q F_L F_R [message]</i>)	(look <i>E ∅ Puck</i> *)
(ping)	(break)
(bye)	(ping)
	(bye)
<hr/>	
<i>Puck</i> ::= (<i>what t n x₁ x₂ x₃ [message]</i>)	<i>what</i> ::= unit ball node
<hr/>	
Control client to Server	Server to Control client
(connect #[<i>hash</i>] <i>name</i>)	(connect #[<i>hash</i>] <i>name</i>)
(set <i>key [value]</i>)	(set <i>key [value]</i>)
(channel <i>t n</i>)	(channel <i>t n team unit color host</i>)
(state <i>q</i>)	(channel <i>t n</i>)
(view [<i>time</i>] <i>Puck</i> * <i>Score</i> *)	(state <i>q</i>)
(ping)	(view <i>time Puck</i> * <i>Score</i> *)
(bye)	(ping)
	(bye)
<hr/>	
<i>Puck</i> ::= (<i>what t n x₁ x₂ x₃ v₁ v₂ v₃ r m Q [F_L F_R E ∅ message]</i>)	<i>what</i> ::= unit ball
<i>Score</i> ::= (<i>score t score</i>)	

B.2 Hoverball options

Option		Limits	Default	Description
simulator.frequency		$0 < \cdot$	50	Simulation frequency (Hertz)
simulator.time	Λ	$0 \leq \cdot$	1	Real time coefficient (Hertz)
simulator.precision		$0 < \cdot$	1	Precision coefficient
game.duration	T	$0 \leq \cdot$	300	Game duration
game.balls.shot		$0 \leq \cdot \leq 99$	1	Number of shot balls
game.balls.team		$0 \leq \cdot \leq 99$	1	Number of team balls per team
game.timeout		$0 < \cdot$	1	Timeout for repeated scoring
game.penalty	Θ	$0 \leq \cdot$	10	Penalty duration
game.recharge	κ	$0 \leq \cdot$	0.2	Energy recharge rate for units
world.radius	R	$0 < \cdot$	50	Sphere radius
world.viscosity	V	$0 \leq \cdot \leq 1$	0.1	Friction viscosity
world.boundary	b	$0 < \cdot$	0.1	Friction boundary layer
world.permittivity	ε_0	$0 < \cdot$	0.000001	Electric field permittivity
unit.radius	r	$0 < \cdot$	2	Unit radius
unit.mass	m	$0 < \cdot$	4	Unit mass
unit.charge.min	Q^-	$\cdot \leq 0$	-1	Unit's highest negative charge
unit.charge.max	Q^+	$0 \leq \cdot$	1	Unit's highest positive charge
unit.charge.pos	λ_Q	$0 \leq \cdot \leq 1$	0.5	Position coefficient of the charging point
unit.charge.cost	c_Q	$0 \leq \cdot$	10	Energy cost for charge
unit.engine.min	F^-	$\cdot \leq 0$	-50	Unit's highest negative engine force
unit.engine.max	F^+	$0 \leq \cdot$	50	Unit's highest positive engine force
unit.engine.pos	λ_F	$0 \leq \cdot \leq 1$	0.5	Position coefficient of the propulsion points
unit.engine.cost	c_F	$0 \leq \cdot$	0.001	Energy cost for engine force
unit.energy.max	E^+	$0 \leq \cdot$	1	Unit's highest energy level
unit.vision	φ	$0 \leq \cdot \leq 2$	1.0	Unit's vision angle ($\cdot \pi$)
unit.message	ℓ	$0 \leq \cdot$	100	Length of unit messages
ball.radius	r	$0 < \cdot$	1	Ball radius
ball.mass	m	$0 < \cdot$	1	Ball mass
ball.charge	Q	$0 \leq \cdot$	1	A shot ball's permanent charge
ball.halflife		$0 < \cdot$	0.005	A team ball's charge halflife

Appendix C

Example „Clumsy“

The following Java unit called „Clumsy“ pursues a simple strategy in order to shoot the shot ball at its team ball:

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

public class Clumsy extends Unit
{
    public Clumsy ()
    {
        super(null,"Clumsy",0x8888FF); // name "Clumsy", color light blue
    }

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

        while (look())
        {
            Puck ball = puck(BALL,0,1); // that's the ball
            Puck goal = puck(BALL,self.t,1); // that's the goal

            if (ball == null) action(0,-Fmax,Fmax); // does not see ball? turn!
            else { // does see ball:
                Vector a = (goal != null)? Vector.vec(goal.X.c,ball.X.c) : // calculate
                    Vector.vec(ball.X.c,self.X.c); // axis
                Matrix X = Matrix.mul(ball.X,Matrix.rot(a,(self.r+ball.r)/sphere.rad));
                Complex x = sphere.warp(X.c); // X is shot position

                double l = -x.arg() + Math.max(0.2,1-10*Math.abs(x.arg())); // turn and
                double r = x.arg() + Math.max(0.2,1-10*Math.abs(x.arg())); // go to X
                double q = (x.abs() < ball.r)? 0.5 : 0; // if at X: shoot!

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