TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

M A S T E R A R B E I T

Collision Avoidance in a Multi-Agent System

Ausgeführt am Institut für
Handhabungsgeräte und Robotertechnik
der Technischen Universität Wien

unter der Anleitung von
o.Univ.Prof. Dipl.-Ing. Dr. Dr.h.c.mult Peter Kopacek
und
Dipl.-Ing. Dr. Bernhard Putz

durch

Martin Stubenschrott, Bakk. techn.
Linzerstraße 429/3207
1140 Wien

Ort, Datum

Unterschrift

"Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning."

– Rick Cook

Abstract

The aim of collision avoidance is to find a path to an object's target without colliding with other static or moving obstacles. Furthermore, the length of this path should be minimized while still ensuring there is no crash with other objects. The demand for such a system is huge, as collision avoidance is essential for most robots but also for applications like autonomous vehicles.

This master's thesis goal is to design, to implement and to evaluate a collision avoidance algorithm for a multi-agent system with quickly moving objects. The obvious choice was to reuse an existing robot soccer framework from the IHRT institute of the Vienna University of Technology. While the IHRT has a long and successful tradition in playing robot soccer, the current system is lacking a proper collision avoidance module. Additionally – and even more important – robot soccer serves as a prime example for a multi-agent system because it needs intelligent interaction between robots.

The algorithm itself is divided in two parts: First we try to find information about the next collision for each robot and categorize it into one of three possible types (head-on, perpendicular or angular collision). Using this information, the algorithm aims to prevent the anticipated collision by using one of two strategies: The path of an individual robot is modified by either changing its direction or its speed.

The system is evaluated with two robots moving on predetermined and random paths. We count the number of collisions and calculate the average speed with and without using the collision avoidance module. Thus we find out whether the proposed algorithm works well and where its drawbacks are.

Zusammenfassung

Das Ziel von Kollisionsvermeidung ist es einen Pfad zum Ziel eines Objektes zu finden ohne mit anderen statischen bzw. beweglichen Hindernissen zu kollidieren. Zusätzlich soll dieser Pfad möglichst kurz sein ohne jedoch das Risiko einzugehen, mit anderen Objekten zu kollidieren. Der Bedarf an ein solches System ist hoch, da die meisten Roboter eine Kollisionsvermeidung benötigen, aber auch andere Anwendungen wie selbstfahrende Fahrzeuge profitieren davon.

Das Ziel dieser Diplomarbeit ist es, einen Algorithmus zur Kollisionsvermeidung in einem Multiagentensystem zu entwerfen, zu implementieren und zu testen. Die Wahl fiel darauf, ein existierendes Roboterfußball-Framework des IHRT Instituts der Technischen Universität Wien als Basis dafür zu verwenden. Denn obwohl das IHRT eine lange und erfolgreiche Tradition im Roboterfußballspielen hat, fehlt bisher ein ordentliches Kollisionsvermeidungsmodul. Außerdem – und noch viel wichtiger – kann man Roboterfußball als Paradebeispiel für ein Multiagentensystem ansehen da es intelligente Kommunikation zwischen den einzelnen Robotern benötigt.

Der Algorithmus wird dazu in zwei Teile gespalten: Als erstes versuchen wir möglichst viele Informationen über die nächste Kollision zu erhalten und kategorisieren diese in drei unterschiedliche Kollisionstypen (frontal, rechtwinkelig oder schräg). Auf Grund dieser Informationen versucht der Algorithmus die bevorstehende Kollision mit Hilfe von zwei Strategien zu vermeiden: Wir verändern den Pfad von einzelnen Robotern indem wir entweder deren Richtung oder deren Geschwindigkeit ändern.

Das System wird evaluiert, indem man zwei Roboter auf vordefinierte bzw. zufällige Pfade schickt. Nun zählen wir die Anzahl der Kollision und berechnen die Durchschnittsgeschwindigkeit sowohl mit ein- als auch ausgeschalteter Kollisionsvermeidung. Dadurch können wir herausfinden, ob der vorgeschlagene Algorithmus gut funktioniert und wo es noch Probleme gibt.

Acknowledgements

Writing a master's thesis is always something very special for every student. It often marks the end of his academic career and he has to show what he has learnt. Therefore it is always helpful when you have strong support from various people to facilitate this task.

First of all, I want to thank my parents because without their (not only) financial support, the years of being a student would have been much more difficult. Also my understanding and helpful girlfriend Petra always helped me to keep motivated, which surely wasn't an easy task.

Apart from this emotional and financial support, the entire IHRT team, especially Markus Würzl and Bernhard Putz assisted me a lot when I encountered technical difficulties.

And last but not least, special thanks go to the whole Open Source community whose free software helped me to make my life easier. I am especially grateful to Linus Torvalds for creating the Linux operating system, to Bram Moolenaar for creating the fabulous Vim text editor and to Donald Knuth for creating the LaTeX typesetting program. Without all this software, creating this master's thesis would have been much more painful.

Contents

Chapter 1

Introduction

Collision avoidance is usually part of a path planning problem where individual objects try to find a path to their targets without colliding with other static or moving obstacles while minimizing the needed detour. Early works in robot path planning problems date back to the late 1960's, but most research was done in the 1980's [Lau98]. At that time, robots were usually slow and most research focused on static problems, so there is an increasing need for research with fast robots in highly dynamic environments.

Collision-free path planning is needed in a wide ranges of scientific fields. Usually it is found everywhere where autonomous objects have to move around with artificial intelligence. This ranges from all sorts of robots to autonomous vehicles but is also important for the entertainment industry in order to create stunning computer games with realistic artificial intelligence.

This thesis implements and evaluates such a collision avoidance system for a multi-agent system of individual robots. The Vienna University of Technology already has an existing robot soccer team called *AUSTRO* which lacks, however, proper collision prevention. Instead of creating the system for an artificial world, implementing it for the existing robot soccer system was a logical step and has many advantages and interesting challenges:

- The framework is available and we can focus on implementing the advanced

motion planning.

- A robot soccer team is a typical example of a multi-agent system where all individual actors must work together intelligently to reach a common goal.

- Robot soccer is a highly dynamic game with quickly changing speeds and directions of the individual agents. This makes it hard to predict paths.

- Wrong sensor data
  The positions and angles of the robots are calculated by doing image processing on a video stream. This is prone to errors and we will have to see how it affects the performance of the collision prevention.

The proposed collision avoidance algorithm uses a geometric approach in a virtual 4D world where future robot positions and their (possible) collisions are calculated. Each robot searches for its next collision and ,,communicates" with the colliding robot to avoid the collision if necessary.

1.1 Multi-Agent systems

A multi-agent system (MAS) consists of multiple autonomous agents (also called entities) which interact by means of communication [Flo01]. This system can achieve goals which are difficult or impossible to reach by individual agents.

Robot soccer is a fitting example for a successful MAS as it is impossible for an individual robot to win any game. But even a full team of individual robots will not perform well unless they cooperate. For example, the strategy module of the *AUSTRO* team uses an advanced pass system where two robots work together [Wür05]. Instead of both heading towards the ball in an offense situation, only one is trying to catch the ball and is passing it to the second robot which is waiting in the center for the pass.

1.2 Robot soccer leagues

The first public robot soccer tournament was the *Micro-Robot World Cup Soccer Tournament* at KAIST, Korea, 1995. It was organized by Jong-Hwan Kim who later established the FIRA (Federation of International Robot-soccer Association) in 1997. This association holds annual events in various categories [FIR98]:

SimuroSot is a pure simulation league which consists of a soccer server and two clients with their game strategies.

KheperaSot is an autonomous robot soccer league that is played as a one-on-one soccer game with commercially available Kherpa robots ($130 \times 90\,\text{cm}$) and a tennis ball.

AndroSot is played with remote controlled robots.

MiroSot works with an external vision system and the small ($7.5 \times 7.5\,\text{cm}$) robots are controlled by a host computer which interprets the vision signals and calculates a strategy. This league will be explained more thoroughly in Chapter 2.

NaroSot can be compared to the MiroSot league, but the robots are even smaller ($4 \times 4\,\text{cm}$).

RoboSot has larger robots ($35 \times 35\,\text{cm}$) and can therefore work autonomously without a host computer.

HuroSot is FIRA's humanoid robot league where robots must walk on two legs.

It is noteworthy that many of those leagues are not just for playing soccer anymore, but also include other interesting challenges like a Marathon run for the HuroSot league. This thesis focuses, however, on the MiroSot robots only.

The second large robot soccer organization is the *RoboCup*, which was first announced in 1993 and had its first World Cup in 1997 which took place in Nagoya, Japan [KAK+97, RCB98]. RoboCup's aim is quite ambitious:

> "By 2050, develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer."

Currently RoboCup's leagues are quite similar to FIRA's, but two interesting leagues are added:

Four Legged Robot League: Similar to the humanoid robots but they can walk on four instead of just two robots, making it easier to keep the balance.

Rescue League: The aim is to find as many victims as possible in a designated area.

1.3 Applications

This collision avoidance system is implemented for robot soccer, which could make people think it is just needed for entertainment. This is not true at all, as there are a lot of more serious applications which need a reliable way to detect and prevent collisions:

- Automotive vehicles will become more and more important as over 90% of all traffic accidents are due to human errors [JJG02]. Many of them could be avoided if smart computer systems could replace tired, incautious or drunken drivers. But even for careful drivers, assisting technologies for detecting collisions can be helpful.
  At the CES 2008[1] General Motors have recently announced they are planning to develop a self-driving car within 10 years. Also the robot-cars participating in the annual DARPA Urban Challenge[2] are becoming better every year.

- Surveillance robots [AMT01] can be helpful to monitor large warehouses against thieves, but they need to have protective technology in order not to crash into expensive goods and destroy them.

[1] Consumer Electronics Show
[2] http://www.arpa.mil/grandchallenge/

- Airplanes have had assisting devices to detect mid-air collisions for a long time. In most countries, all aircraft with more than 30 passenger seats must have such a system, usually the *TCAS II* (Traffic alert and Collision Avoidance System) [TCA00].

1.4 Benefits

The benefits of well-working collision avoidance in robotics vary from application to application. Usually it is more important the more expensive and fragile the robots are. One obvious advantage is that by avoiding collisions, you can prevent damage to the robots. While this is an important factor, it is not that important for our RobySpeed robots which are very robust and comparatively cheap to other, bigger robots. In comparison, when the collision avoidance module of autonomous vehicles makes a mistake, every crash can cost tens of thousands of Euros and even lead to the death of the passengers.

Apart from this apparent benefit, robots can often get faster to their destinations without colliding, even when they need to take a small detour or have to slow down before the obstacle. There is, however, a trade-off between safety and speed. The shorter the detour is, the less time is needed to get to the destination, but that includes the risk of still colliding if there is a small calculation or measurement error. In robot soccer, we usually take the unsafer approach of a shorter detour, because speed is usually more important than occasional collisions with other robots.

While all these benefits sound great, some applications may also *want* to have a collision. For instance, our goal keeper should not slow down when trying to clear the situation, just because there might be a possible collision.

1.5 Problems

A collision avoidance system always has a lot of technical problems, for robot soccer it is even harder. The most difficult part are the enormous speeds and accelerations

with a lot of robots on a small field. To make it even worse, also the directions of robots can change instantly, making it very hard to predict a path.

Additionally, it is faced with the same problems as other applications which get their input from sensors. In our case it is a camera which is mounted above the field and transmits the images to a host computer where an image recognition software captures the positions and orientations of the robots. While the software works really well, there still are some errors and also a little latency which makes it harder than doing the same in a simulator with exact data.

1.6 Related work

Most path planning research has been done for static environments. Well known algorithms include *road-map*, *cell decomposition* and *potential field* methods. LaValle gives an exhaustive overview of them and many others in his book *Planning Algorithms* [LaV06]. The problem is, however, that it is difficult to extend those path planning algorithms to work in dynamic configurations with moving obstacles and high-speed robots as well.

This was shown in [Rog04] which implemented a simulator with static as well as dynamic obstacles, a ball and a robot – simulating robot soccer. It uses the potential field method, whose idea is based on electrostatic or magnetic fields:

- Target points have high attractive forces around them but the attraction diminishes linearly to the distance from the target. Also gyroscopic forces can be used to create swarming behaviors, like described in [CSMOS03].
- Obstacles have repulsive forces.
- These forces result in equations which are used to find paths with the maximum of attractive and the minimum of repulsive forces.

However the simulation showed that this algorithm cannot cope with the high speeds of the soccer robots which reach up to 4 m/s.

Fortunately, collision avoidance research for moving obstacles has been increasingly dealt with recently with many publications [FBT95, Rud97, FT02, RAP+04]. Most of them have a similar concept, which is also part of the proposed algorithm of this thesis: The space domain is extended to a space-time domain where temporal information plays an important part. The exact proposals on the implementation differ of course. For instance, Fox et al. present a *dynamic window approach* [FBT95] which reduces the search space to a dynamic window, which only consists of places reachable in a short time interval. They claim to be able to safely control their mobile robot *RHINO* (Fig. 1.1) with speeds of up to 0.95 m/s in dynamic environments. However, for even faster speeds it is probably beneficial to take the full search space into account, because the earlier a collision can be detected, the less corrections are necessary to avoid the collision.

Figure 1.1: RHINO

1.7 Outline

As this thesis is an extension to an existing robot soccer system, we give an overview of its software and hardware in Chapter 2. Furthermore, the current collision avoidance method of the *AUSTRO* project is investigated and we also look at how we can implement a better one on top of the existing code.

Chapter 3 builds upon this knowledge and introduces our idea of an improved collision avoidance system. First, we illustrate different types of collisions, later we explain our algorithm to detect possible collisions and finally we show how our algorithm actually can prevent a crash. After showing the concepts in a more theoretical way, this chapter concludes by an outline of the actual implementation.

The evaluation of the implemented system is described in Chapter 4 where we send two robots on certain paths with and without collision avoidance and measure the number of collisions. This allows us to detect weaknesses of the algorithm, but also shows where it works well.

In the final Chapter 5, we summarize the problem statement and how we solved it and also suggest possible future improvements for further research.

Appendix A lists the full source code which was implemented for this master's thesis.

Chapter 2

The AUSTRO Robot Soccer System

Robot soccer has a long tradition at the Technical University of Vienna. The first demonstration matches date back to the year 1997 [HFE07] and nowadays even two institutes are working on robot soccer, the ICT[1] and the IHRT[2]. The ICT developed an autonomous robot called *TinyPhoon*. If you are interested in that robot, please refer to [NM05]. For this thesis, however, the AUSTRO soccer team – which was developed by the IHRT – is much more relevant. They use fast robots called *RobySpeed* (see Section 2.1.1 for more details).

In the meantime, the *AUSTRO* team has participated in a lot of FIRA European Cups or FIRA World Cups with lots of success:

- World Champion 2004 in the MiroSot Middle and NaroSot league
- World Champion 2005 and 2006 in the Narosot league
- European Champion 2005 in the following leagues: MiroSot Middle, Large and X-Large

[1]Institute of Computer Technology, https://www.ict.tuwien.ac.at
[2] Institute of Handling Devices and Robotics, http://www.ihrt.tuwien.ac.at/

- European Champion 2006 in the following leagues: MiroSot Extended Middle and X-Large
- Many other titles like Olympic Gold 2004 or the First Place in the category Robot Parade in 2003.

2.1 Hardware

As this thesis focuses on the algorithmic part of collision avoidance, only a short overview of the hardware is given. If you are interested in more technical details, have a look at the diploma theses from Markus Würzl [Wür05] and Bernhard Putz [Put04].

The game is played on a 220×180 cm field with an orange golf ball. A camera is mounted above the field, and the images are transmitted to a host computer which interprets them and sends commands to the robots via a radio transmitter (Fig. 2.2).

2.1.1 The robots

The current robots are called RobySpeed (Fig. 2.1) and are an improvement of the previous models Roby-Go [Nov05] and Roby-Run.

In accordance with FIRA rules, this robot's size is limited to 7.5×7.5 cm. Despite its small size, RobySpeed can reach a top speed of nearly $4\,m/s$ with a maximum acceleration of $10\,m/s^2$.

2.2 Software

The *Team AUSTRO* software is a full framework for playing robot soccer. The system can roughly be categorized into four modules:

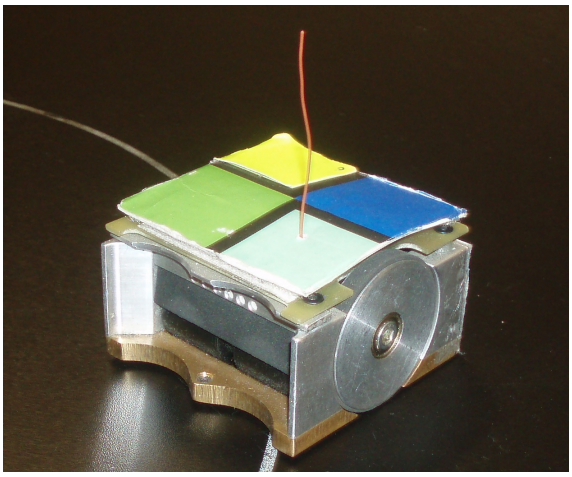

Figure 2.1: RobySpeed

1. Code interfacing with the hardware. This includes the frame grabber which needs to decode the images of the camera and the radio transmitter which is needed to send commands from the host computer to the robots.

2. The strategy module is responsible for sending movement commands to the robots, depending on the position of the ball and the other robots. It actually consists of multiple files, optimized for each league. Three robots playing on a small field usually need a different strategy than those playing a full 11 vs. 11 game.

3. Movement code for positioning the robots on specified coordinates or for changing the velocity of robots.

4. A VisualBasic GUI[3] with which you can control the whole system and start or stop the system.

This thesis focuses on the movement code, by adding a new `CollisionAvoid (whichrobot, x, y, endspeed)` function. It takes the same parameters as the existing `SuperPosition()` function used for positioning robots. But instead of moving the robot straight to the specified coordinates, it tries to get there without colliding with other robots.

[3]Graphical user interface

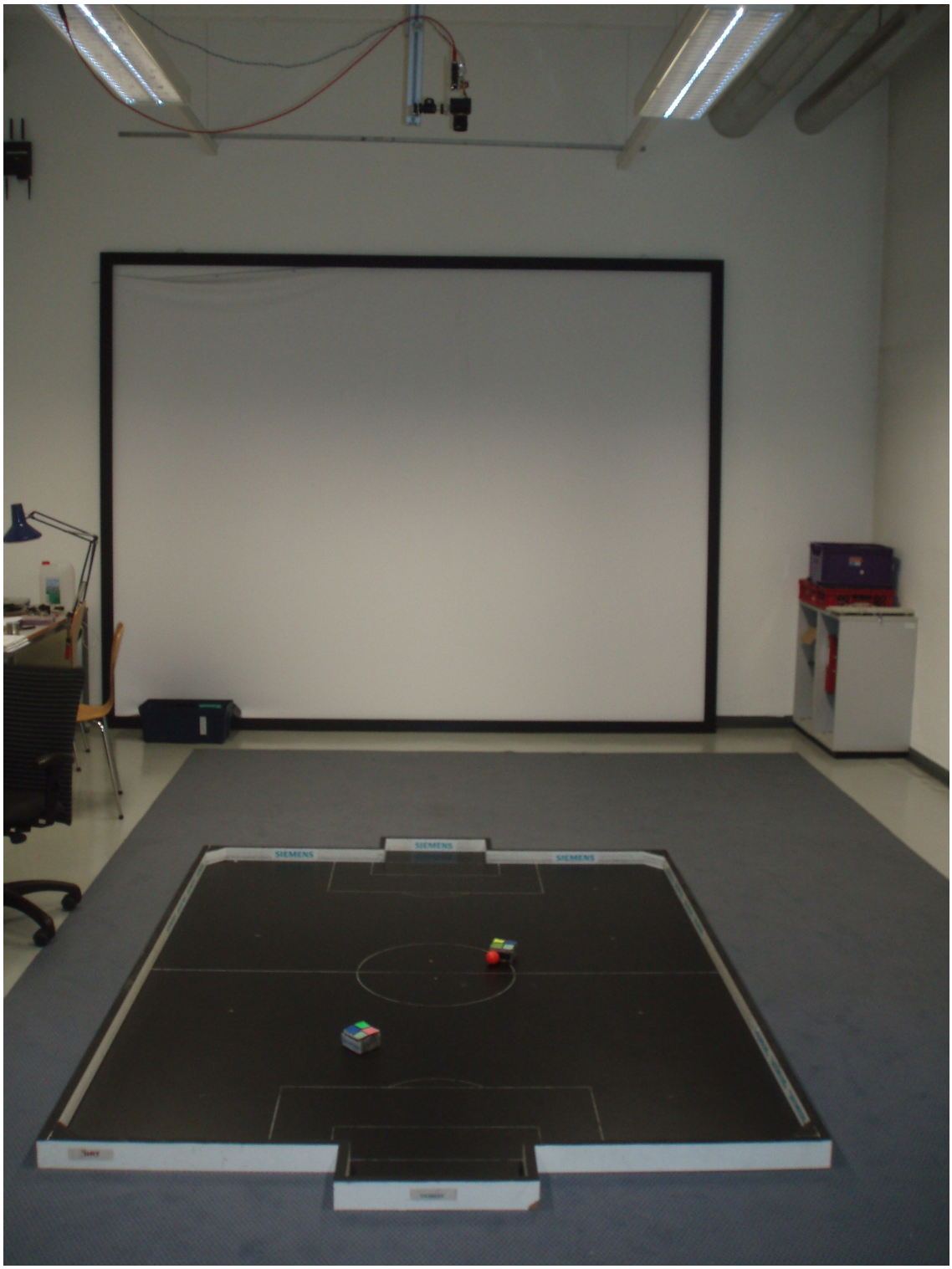

Figure 2.2: A camera is mounted above the field

2.2.1 Current collision avoidance system

In the current *AUSTRO* software version only a very primitive method is available called `AutoAvoid()`. It works by defining a safefy area between the robot and its target (illustrated by the gray rectangle in Fig. 2.3). Whenever an obstacle is *currently* within this area, the robot moves to a temporary point T.

Algorithm 1 AUTOAVOID (whichrobot, x, y)

1: {*Let T be a temporary point:*}
2: $T(x, y)$ = Transform target (x, y) {*into the coordinate system of whichrobot*}
3: **for all** other robots R **do**
4: $R(x, y)$ = Transform R {*into the coordinate system of whichrobot*}
5: **if** $R(x) > 0$ and $R(x) < x$ and $|R(y)| < safety\_margin$ **then**
6: $T(x) = R(x)$
7: $T(y) = R(y) + safety\_margin$
8: **end if**
9: **end for**
10: *Move robot to position T*

While this basic approach might work for very simple situations, it usually does not cope with more complex collisions. The main problems are:

- It does not search for the *next* collision, but it only handles the *last* one which it finds.
- It does not calculate where the other robot is going but only works for the current position.
- The temporary point T is always chosen statically in one y-direction. It would be better to choose T, depending on the exact position/direction of the other robot.
- The whole algorithm is speed independent. No matter if the robots move with 1 or 5 m/s, the detour is always the same, resulting in either a long way round or a crash.

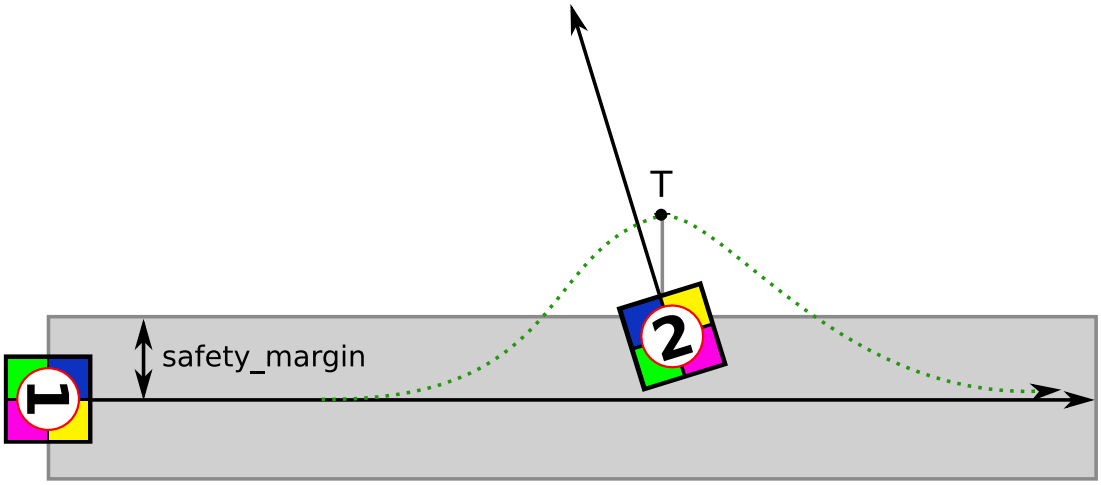

Figure 2.3: The `AutoAvoid` function always tries to avoid the collision by finding an immediate point T when an obstacle is *currently* within the gray area.

The authors of the current software version are aware of the limitations and suggest an algorithm which is very similar to the one which is described and elaborated in this paper. They call their idea the *SAW* algorithm (Simply Another Way), more details are described in [Wür05], pages 98–100.

Chapter 3

Collision Avoidance

The aim of all collision avoidance systems is to find collision-free paths to their targets for one or more objects. Depending on the application, it must handle static but often also dynamic obstacles.

There are at least two fundamentally different ways to implement such a system. The first approach is to build a mathematical model of the world and always try to find a path with the least resistance. The *potential field* and many other methods use this approach [Zel95], but it was shown in [Rog04] that at least the potential field method is unsuitable for high speed soccer robots. Therefore our proposed algorithm uses the second common approach that separates the system into a collision detection and a collision prevention part. First it tries to find out the position of a possible collision, and only if the collision is near enough we try to prevent crashing into the obstacle. Figure 3.1 shows a graphical overview of the algorithm.

Apart from these fundamental differences how to implement collision avoidance, it can additionally be handled at two stages in robot soccer:

1. **At the strategy level**
   The strategy involves sending movement commands to the robots, depending on the current situation. This usually involves following the ball, defending the goal, or waiting for a pass. If the strategy module can check for collisions, it

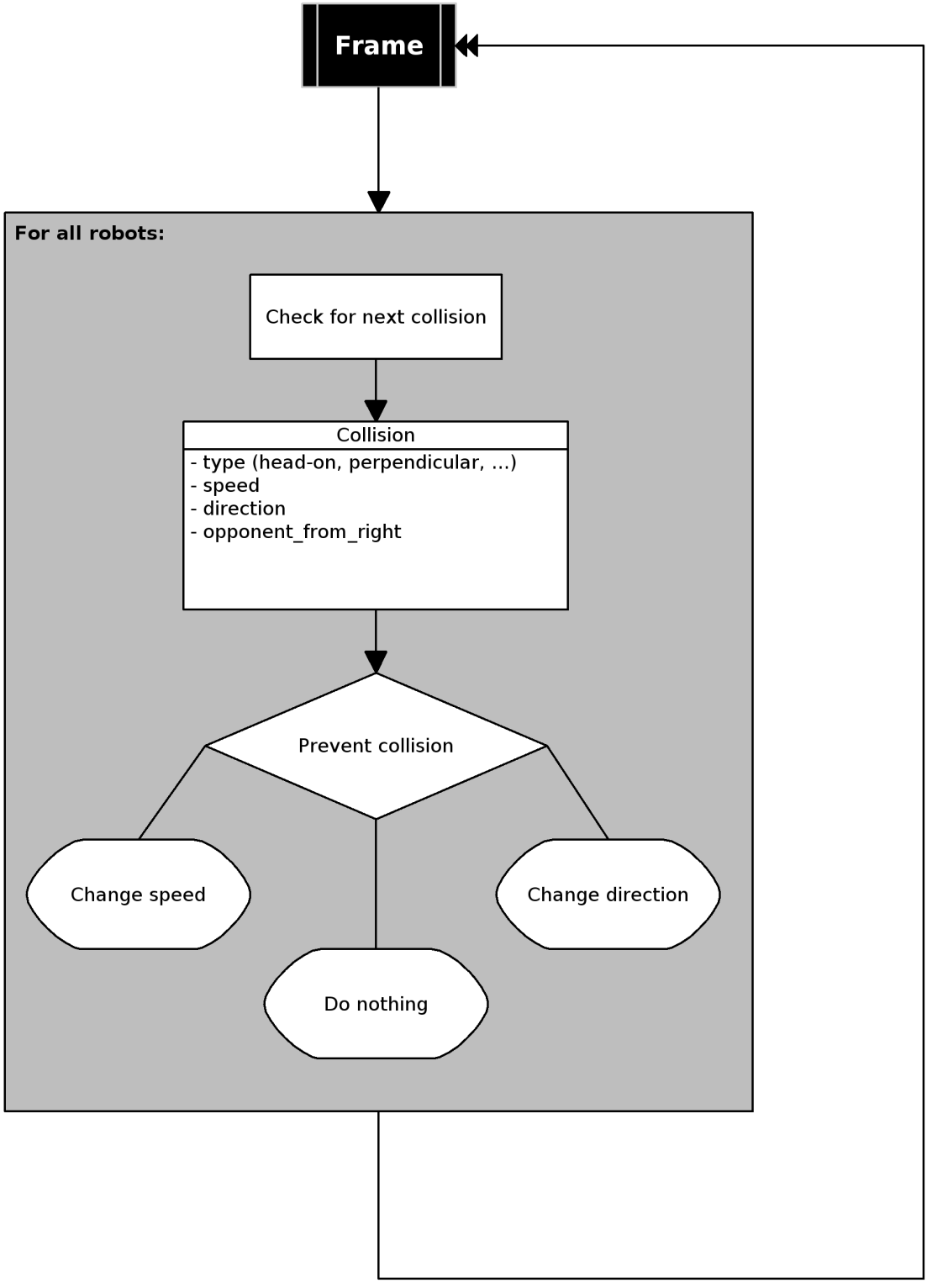

Figure 3.1: The basic concept of our algorithm

might change the strategy on the fly and send robots to other target positions which can be reached without the fear of a collision.

2. **By the movement code**
   It is easier to implement and evaluate collision avoidance at this stage as there is usually just one function to call which can be changed to include collision information. However, this does not mean that preventing crashes at this stage is less important, because even if we try to find collision-free paths at the strategy level, we still need to check for actual collisions in the movement code due to the high dynamics of robot soccer.

This thesis deals with collision avoidance at the movement level. Another interesting research field would be to adapt the strategy to also include collision information.

3.1 Different types of collisions

A collision is defined as a situation that two or more objects want to occupy the same position at the same time. While all of them should be avoided, it is necessary to differentiate between different types of collisions. Hwang et al. distinguish between five different collision types for any two straight-line moving objects [HCL99] (Figure 3.2).

This accuracy of discrimination makes sense for airplane collision avoidance systems, but can be simplified a bit for robot soccer. First, there is no distinction between an acute and obtuse collision, both are only detected as an angular collision. Second, a San-Diego collision of the object behind moving faster than the object ahead does not happen often in robot soccer. But even if it does, we can just use the same collision prevention algorithm as if the object ahead was a static obstacle. This leads us to three different types of collisions detected by our algorithm (let α be the relative angle between the direction vector of both robots as shown in Fig. 3.3):

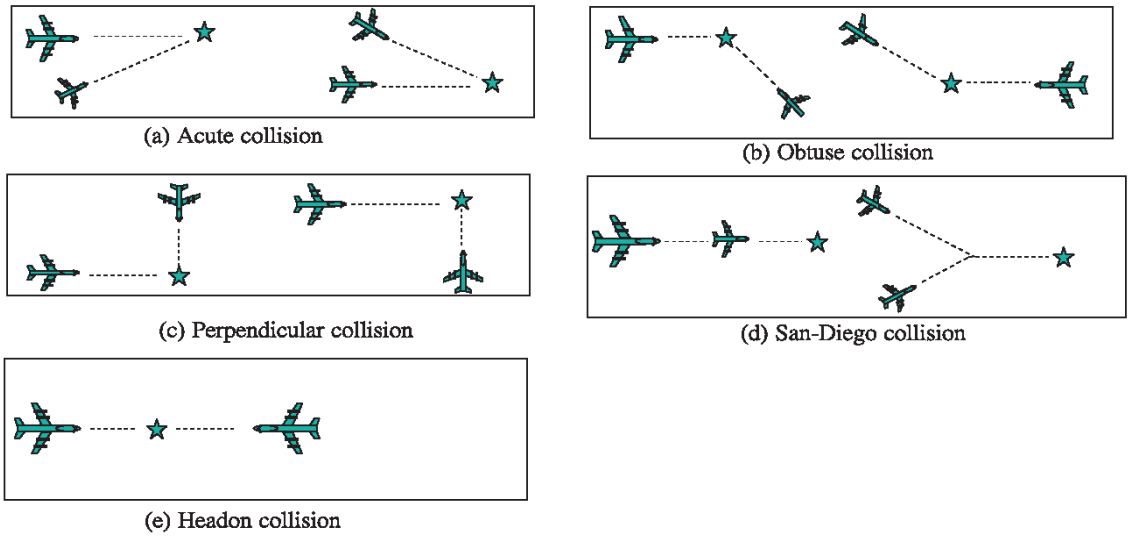

Figure 3.2: Collision types for two straight-line paths

3.1.1 Head-On collision

Head-on collisions must be detected very reliably as this is the most dangerous collision which can cause damage to our robot. Additionally, a head-on collision usually has the biggest impact on the speed and direction of the robot, which increases the time to reach its target.

A frontal collision is detected when α is either of:

$$150^\circ \leq \alpha \leq 210^\circ \tag{3.1}$$

$$\alpha \leq 30^\circ \vee \alpha \geq 330^\circ \tag{3.2}$$

The first equation is for a ,,true" head-on collision, and the later for the case of a rear-end collision which is handled like a head-on collision with a static obstacle.

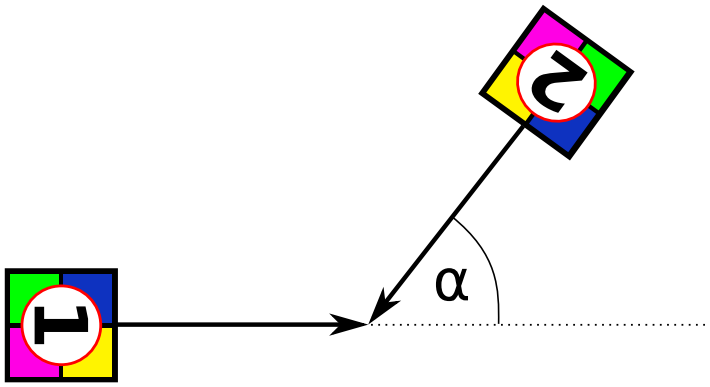

Figure 3.3: The collision angle α

3.1.2 Perpendicular collision

Perpendicular collisions happen when one robot hits another on the side in a (near-) orthogonal angle. This is true for those α values:

$$60^\circ \leq \alpha \leq 120^\circ \tag{3.3}$$

$$240^\circ \leq \alpha \leq 300^\circ \tag{3.4}$$

3.1.3 Angular collision

All other angles are recognized as angular collisions ignoring the difference of acute and obtuse angles as the collision handling is the same for both cases. Using these values, each collision type occurs in a third of all times if the movements of the robots are purely random.

3.2 Collision detection

Collision detection is defined by Stephen Cameron [Cam90] as:

α	detected collision type	remarks
$0 \leq \alpha \leq 30$	head-on	frontal collision
$30 < \alpha < 60$	angular	obtuse collision
$60 \leq \alpha \leq 120$	perpendicular	opponent coming from left
$120 < \alpha < 150$	angular	acute collision
$150 \leq \alpha \leq 210$	head-on	rear-end collision
$210 < \alpha < 240$	angular	acute collision
$240 \leq \alpha \leq 300$	perpendicular	opponent coming from right
$300 < \alpha < 330$	angular	obtuse collision
$330 \leq \alpha \leq 360$	head-on	frontal collision

Table 3.1: Different collision types, depending on the collision angle

> "Given two objects and desired motions, decide whether the objects will come into collision over a given time span."

The difficulty of this task varies, depending on the exact requirements of the application. Robot soccer with these very small cubic robots has the advantage that we can reduce the object's shape to a single point without affecting the accuracy of the algorithm noticeable.

Therefore our aim is to check when two robots occupy the same coordinates at the same time. If we knew the exact directions and velocities of both objects for the given time span we could integrate over the general motion equations [FBT95]:

$$x(t_n) = x(t_0) + \int_{t_0}^{t_n} v(t) \cdot cos\theta(t)dt \tag{3.5}$$

$$y(t_n) = y(t_0) + \int_{t_0}^{t_n} v(t) \cdot sin\theta(t)dt \tag{3.6}$$

However, robot soccer is such a dynamic game with quickly changing directions and velocities that we cannot reliably predict those values for a longer period of time. Because of that we have to replace the integral over time with simple constants for the speed and directions. This also benefits the real-time requirement of robot

soccer, as repeatably solving these integrals by approximation can entail a high computational cost.

This leads to our approach of dividing the collision detection into a multi-step process for each robot R_i which is repeated for all possibly colliding robots $R_j \in R_{1..n}$:

1. **Coordinate transformation**
   In order to make subsequent calculations easier, we transform R_j into the coordinate system of R_i (Figure 3.4). Now the position of R_i is the new point of origin and the x-axis reflects the direction in which this robot moves. Furthermore, positive values on the y-axis are located to the left of the robot and negative values to the right respectively. This also transforms the direction vector and we get the relative angles between the two robots which is important for checking the type of collision.

2. **Calculate intersection point**
   With this simplification we can obtain the intersection point with the parametric vector equation:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + t \cdot \begin{pmatrix} a \\ b \end{pmatrix} \tag{3.7}$$

   (x, y) is the point we want to find, (x_0, y_0) the position of R_j relative to our robot, t is an independent variable, and (a, b) the direction vector of R_j.

   A collision happens when the robot crosses the x-axis, which in turn means that y must be 0. By setting y to 0, we find out t, and can now get the x-position where the vectors of these two robots will cross.

   Careful readers will have noticed that this approach has two drawbacks:

   (a) Non-moving robots do not have a direction vector.
   We compensate for that by first checking the velocity v of the other robot. If v is close to zero[1] we just assume R_j to be a static obstacle which can be handled easily: If $|y_0| < safety\_margin$ we indicate a (head-on) collision

[1] In our experiments this was true for all $v \leq 0.1m/s$

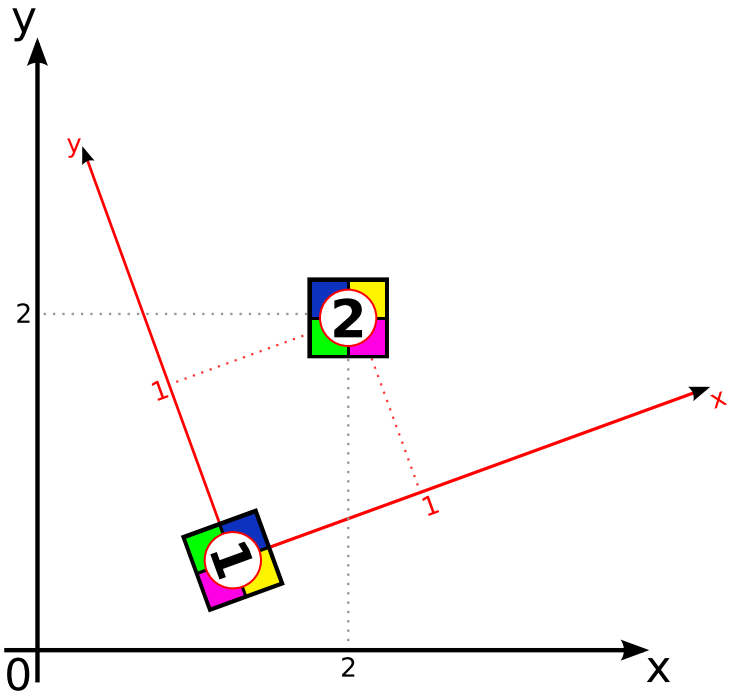

Figure 3.4: Coordinate transformation: Robot 2 is at position (2,2) in the absolute coordinate system, but at position (1,1) relative to robot 1

at position $(x_0, 0)$. For all other values we indicate no collision. We chose a safety margin of 10 cm for both directions but this can easily be adjusted.

(b) If both robots are driving parallel, there is no intersection point. Furthermore, even if they are just moving *nearly* parallel, the intersection point is not reliable. Therefore we do not calculate Equation 3.7 when the collision angle α is a head-on collision according to Table 3.1.3. Instead, we just set the possible collision in the middle of both robots if they are moving towards each other or otherwise indicate no collision.

3. **Handle intersection point**
   If this intersection point from step 2 is negative (only the x-value is important, y will always be 0 since we are working in our relative coordinate system), it

means that a possible collision would have happened in the past and we can move on to the next robot.
If the collision point is in front of our robot, we calculate the expected arrival times for both robots. If the time difference is below a certain threshold, we indicate a collision.

This algorithm is repeated for all possible obstacles and calculates the nearest collision, including information about the expected position and time of the collision and whether the moving obstacle is coming from the left or the right side.

3.3 Collision prevention

Section 3.2 dealt with finding the next collision. Given this information, the collision avoidance algorithm now has all the information it needs in order to find a collision free path to the robot's target. This thesis employs two different techniques: Whenever possible, it just changes the direction of the robots slightly, which is often enough to prevent a collision. Unfortunately, this technique is not applicable or useful in all situations, therefore the other solution is to just stop one robot, prioritizing the other.

3.3.1 Changing directions

The ideal solution of preventing the collision between two moving robots is to temporarily change the direction just a little, so that both robots can drive past the other as shown in Figure 3.5.

For that, we calculate temporary points P_1 and P_2 for both robots. We now redirect the robots to those points until the situation looks safe again. The exact position of these points depend on the speed of the robots. As a rule of thumb, we assume that the higher the speeds are, the higher the distance between P_i and the estimated collision point is.

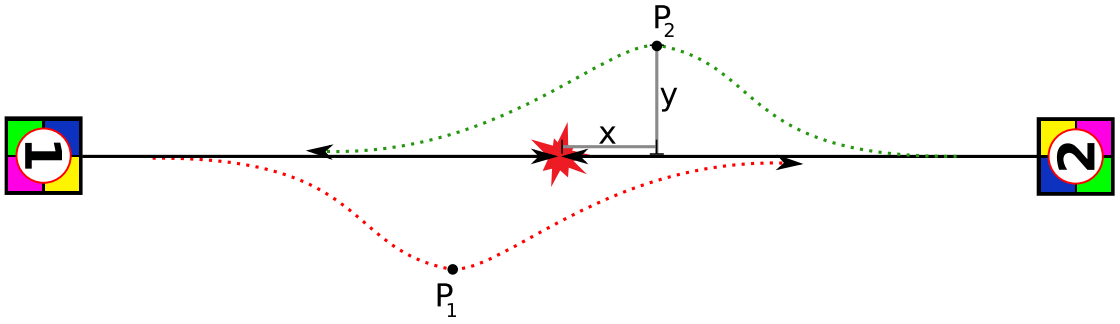

Figure 3.5: Avoiding a direct collision by changing directions of both robots

$$P_i = \begin{pmatrix} C_i(x) - x \\ \pm y \end{pmatrix} \tag{3.8}$$

$C_i(x)$ is the estimated distance of R_i to the collision point, x and y are speed dependent values with upper and lower limits (20 cm minimum and 80 cm maximum in our implementation).

Equation 3.8 also has two important characteristics:

1. The temporary point is always ahead of the estimated collision point. This is important for increased fault tolerance when the estimated position does not match the real collision point. Initially, $P_i(x)$ was set to 0 like in Figure 3.6. It indeed has a larger safety margin if the collision point was estimated

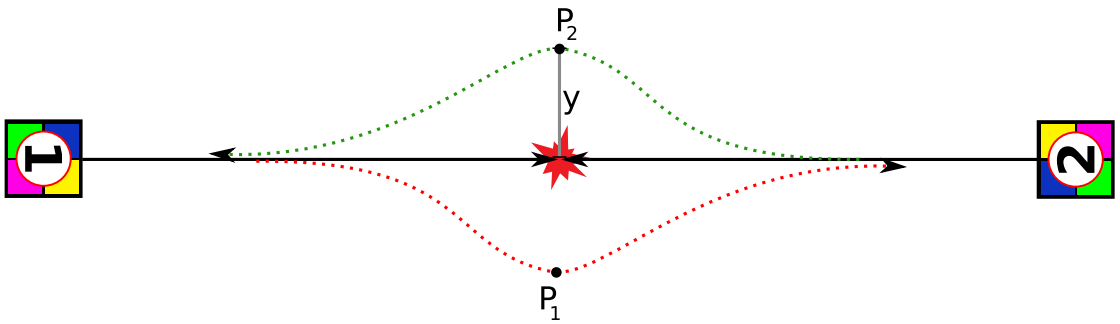

Figure 3.6: An alternative approach to changing directions

correctly. However, even small deviations decrease the distance between the red and the green path noticeably. The distance between these paths in Figure 3.5 is much more constant, which means it has a higher fault tolerance.

2. y can be positive or negative, depending in which direction we want to avoid the collision. If we can *safely* assume that the obstacle is to the right of the robot, we avoid the collision by moving to the left. In all other cases (like in Fig. 3.5 where the opponent is straight ahead), we always move to the right. With this simple assumption we can reliably avoid that both robots move to the same interim point and crash there.

An important feature of this algorithm is that we try to change the paths of *both* robots slightly instead of just changing the direction of one robot by a larger amount. First, it has the advantage that this keeps the value of y lower, resulting in less slowdowns of the robots and shorter paths. Second, it distributes the costs of moving along these extended paths equally between all involved robots. If this is not wanted because one robot – like an offender – should always try to move along the quickest path or if the other robot is uncontrollable (e.g. it is from the opposite team) one could, of course, also just reroute one robot with a larger safety margin.

This technique to change the robots' directions is used when the collision detection algorithm identifies a head-on collision, because in this scenario it is the most reliable solution. Unfortunately, for other collision types, changing the directions does not always result in safe paths. This is mostly the case when the collision angle α differs considerably from a head-on collision, like in the case of a perpendicular collision (Fig. 3.7).

A variation of this technique would be to let one robot move along its original path and only alter the path of the other robot like shown in Fig. 3.8. This only works if the real paths from the robots do not differ from the estimated one. In practice however, this is hardly the case. Therefore, the next section deals with a different approach for avoiding such collisions.

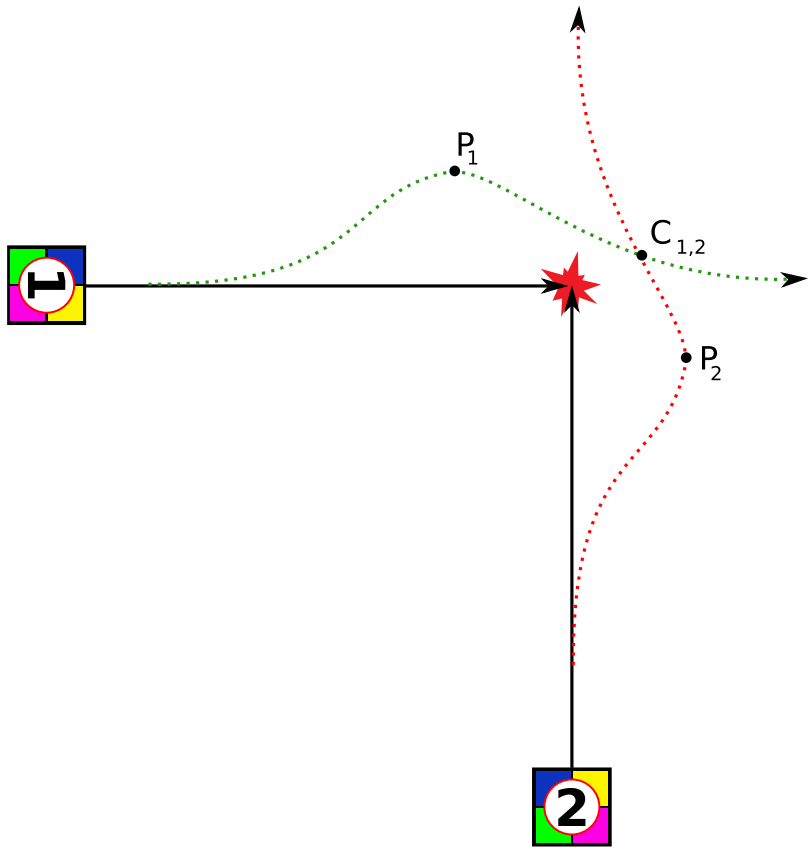

Figure 3.7: The algorithm as described in Section 3.3.1 for a perpendicular collision. Robots are likely to crash at position $C_{1,2}$

3.3.2 Changing speeds

In cases when changing directions do not yield good results, we use a different approach. The idea is to let one robot move along its planned path and temporarily stop the other robot until it is safe to resume its course (Fig. 3.9).

The coordinates of P_1 where R_1 will stop to give way to R_2 are defined by:

$$P_1 = \begin{pmatrix} C_1(x) - x \\ 0 \end{pmatrix} \tag{3.9}$$

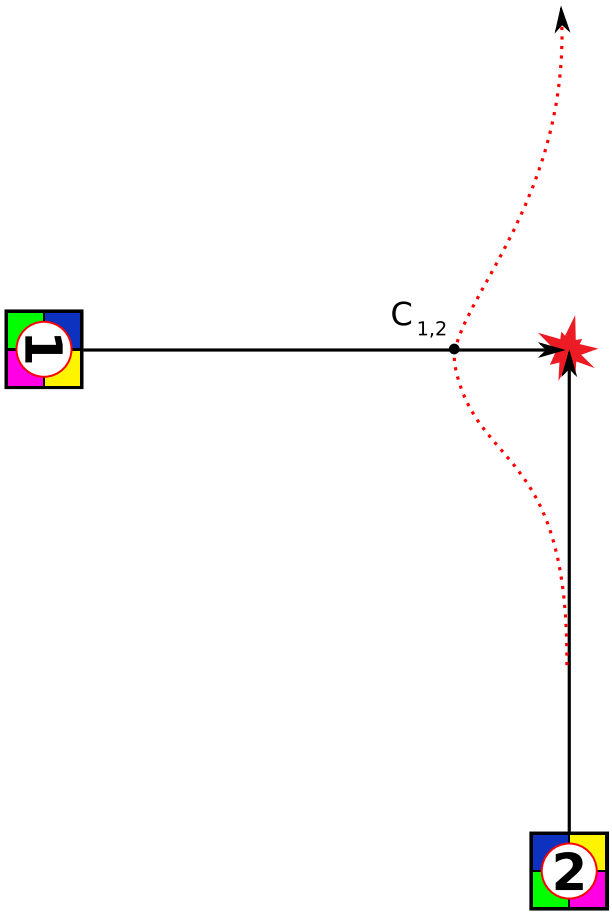

Figure 3.8: A different approach for changing directions in case of a perpendicular collision

Again the value of x depends on the robots' speeds because we need to stop earlier the faster the velocities are. Furthermore, in most cases it is not necessary to really stop one robot, but just to slow it down. This works implicitly, as we recalculate possible collisions and paths on each frame. Let us assume that we want to stop one robot from an initial speed of 3 m/s and it slowed down to 2 m/s several frames later.[2] When recalculating the intersection point and a collision seems unlikely, we do not need to stop the robot anymore, but can accelerate it again.

[2]Our camera outputs 60 frames per second, so each new frame just takes about 16 ms

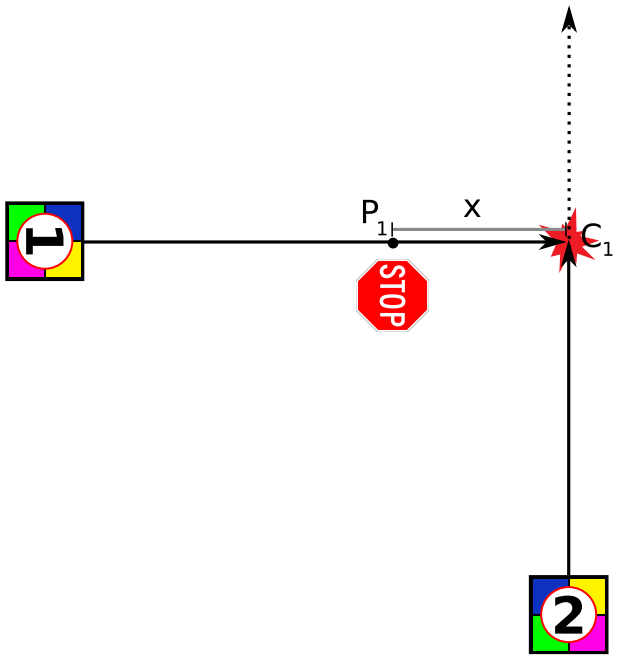

Figure 3.9: Avoiding a perpendicular collision by temporarily stopping one robot

3.3.3 Interaction with other agents

As the title of this thesis implies, we want to create a collision avoidance system for a multiple agent system, where collaboration between agents is important. They need to ,,communicate" with each other in order to intelligently avoid a possible collision.

This interaction is *required*, because we need to coordinate at least two robots in the case that one robot must give way to the other. Without any synchronization there would be a dead-lock when both robots wait for each other. Hence it is important that exactly one robot reduces its speed. But even when avoiding the collision by dodging to the side, it is important to not move sideways in the same direction.

Initially, the interaction between two robots was *explicit*. Whenever a collision was detected for the first time, one of the two involved robots took over the handling

of the collision. That robot had to handle the collision on its own until no collision was predicted anymore. The other robot could rely on that fact and continue to move on its original path. This strategy could solve the problem, so that not *both* robots stop for each other, but it was still quite inflexible.

Therefore we changed the interaction to be completely *implicit*. This is done by two simple rules:

1. **Dodge-to-the-right rule**
   This rule is used in the case when we want to avoid the collision by changing the direction of both robots. In a head-on collision situation it would be counterproductive if one robot dodged to the left and the other to the right. Therefore a very simple but effective rule is that both robots *always* dodge to the right unless an obstacle is static; then the robot is also allowed to move to the left if that is the better path.

2. **Give-way-to-the-right rule**
   The *give-way-to-the-right* rule is directly derived from one the most important traffic rules: The vehicle (or, in our case, the robot) coming from the right side has priority (Fig. 3.9) and the one to the left has to reduce the speed (or even stop completely) to let the other pass.

With these two simple rules we eliminated any need for explicit interaction between two robots while making the system even more reliable.

3.4 Integration in existing RoboSoccer system

The aim of this thesis is to describe and test a collision avoidance system for multiple agents. Therefore, we extended the existing framework with additional functions and data structures. We did not, however, change any existing strategy code to actually use those functions. For reference purposes, the complete source code is given in Appendix A.

3.4.1 New data structures

There is only one new data structure `Collision` which is defined as:

```
struct Collision
{
  CollisionType type;     // whether we have a head-on or angular collision
                          // possible values: NO_COLLISION, HEADON_COLLISION,
                          // PERPENDICULAR_COLLISION, ANGULAR_COLLISION
  int colliding_robot;    // the id of the robot with which we collide
  bool opp_controllable; // true for our own robots, false for other robots
  double time;            // time in milliseconds when the collision will occur
  double distance;        // distance in meters when the collision will occur
  double my_speed;
  double opp_speed;
  bool opp_from_right;    // true if the colliding robot is to our right side
}
```

`Collision` is used as the connection between the collision detection and the collision prevention part. Hence it is easy to exchange one of these two parts with another algorithm due to the standardized interface shown above.

3.4.2 New functions

The algorithm is divided into two distinct parts, which is the logical consequence of our two step approach illustrated in Figure 3.1:

NextCollision() is responsible for returning the next collision for the given robot. The return value is a `Collision` structure as described in Section 3.4.1. If no collision is detected, the *type* field is set to `NO_COLLISION` and the other members are undefined.

CollisionAvoidance() is the main function which is called whenever a robot is sent to a specific position with the intention to avoid a possible collision. It first obtains a `Collision` structure by calling `NextCollision()` and in the case of a collision it modifies the path of the robot by either setting an intermediate waypoint or by slowing down one of the robots.

Algorithm 2 NextCollision (whichrobot)

```
1: Collision collision
2: collision.type = NO_COLLISION
3: for all other robots R do
4:    angle, x, y = Transform R {into the coordinate system of whichrobot}
5:    if angle ≃ 0 then {head-on collision}
6:       collision_point = (x/2, y/2)
7:    else {angular or perpendicular collision}
8:       collision_point = (x + t · cos(angle), 0)
9:    end if
      {Find the time difference between both robots to the collision point}
10:   t_R = dist(collision_point, R) / v_R
11:   t_whichrobot = dist(collision_point, whichrobot) / v_whichrobot
12:   diff_time = |t_R − t_whichrobot|
13:   if diff_time > ε or diff_time > collision.time or t_R < 0 or t_whichrobot < 0 then
14:      continue with next robot
15:   end if
      {R is now the nearest collision, update the collision information}
16:   collision.time = diff_time
17:   collision.type = HEADON or PERPENDICULAR or ANGULAR
18:   collision.distance = collision_point.x
19:   ... {Setting the rest of the data structure's fields}
20: end for
21: return collision
```

Algorithm 3 CollisionAvoidance (whichrobot, x, y, endspeed)

```
1: collision = NextCollision(whichrobot)
2: if collision.type = HEADON then
3:    Change direction to the right
4: else if collision.type = (PERPENDICULAR or ANGULAR) and collision.opp_from_right then
5:    Stop robot
6: else {No collision}
7:    Call SuperPosition(whichrobot, x, y, endspeed)
8: end if
```

3.4.3 GUI integration

Sometimes it can be beneficial to disable the collision avoidance module temporarily. Therefore we added a check box to the existing graphical user interface (GUI) as shown in Figure 3.10.

An unchecked value does not mean that `CollisionAvoid()` is not called. Instead, it sets an internal flag to `false`. If this flag is false, `CollisionAvoid()` just forwards the arguments to `SuperPosition()` which is responsible for moving a robot to a specific position without caring about possible collisions.

Figure 3.10: Enable/disable collision avoidance globally with this checkbox

Chapter 4

Evaluation

Whenever a scientific system is implemented, the need for an evaluation suite emerges. This is required for numerous reasons:

- We can either verify or disprove that the algorithm and our corresponding implementation works.
- The results should be *at least* slightly better than using no collision avoidance at all. Otherwise we must admit that the algorithm (or implementation) does not work properly for the given task.
- Future improvements to the system can easily be compared.

Our robots can move from any position on the field to any other. Thus, we cannot test the system for all possible cases. Therefore, we need to test the most common collisions thoroughly but also have to test some random situations.

4.1 Tests

All tests are performed by two 7.5 × 7.5 cm *RobySpeed* robots on the standard soccer field for the *NaroSot* category (220 × 180 cm). We move both robots along certain paths, using three different functions:

1. SuperPosition()
   No collision avoidance at all is used and the robots move directly to their targets. *If* there were no intersecting paths, this would be the quickest way, as there are no detours and the robots are not slowed down unnecessarily.

2. AutoAvoid()
   This is the existing basic collision avoidance function from the *TEAM Austro* software (Algorithm 1). Its only possibility to avoid the collision is moving the robot to one side. It does never reduce a robot's speed.

3. CollisionAvoid()
   Our own, improved collision avoidance module as described in Chapter 3. It can either modify the robot's path or change its velocity, depending on which strategy has the greater chance of finding a collision free path in the given situation.

4.1.1 Moving along predefined paths

For the first type of tests, we define 15 fixed positions on the soccer field like shown in Figure 4.1. By moving the robots between specific positions, we can simulate all different collision types (head-on, perpendicular and angular) in a reproducible manner, which is important for a meaningful evaluation. The robots are moved with the `CoordinatedMove` function (Alg. 4).

In the following images we mark the start position of R_1 with a green circle and the start position of R_2 with a red circle and the paths with arrows of the corresponding color. At the end of a path, the robots return to their start positions and start their paths again. Sometimes the paths for the two robots have a different length, resulting in a different time needed for the path. We deliberately chose *NOT* to wait for the other robot to finish its path. This design decision ensures that we still test a certain collision type, but the exact timing and position varies between each possible collision slightly.

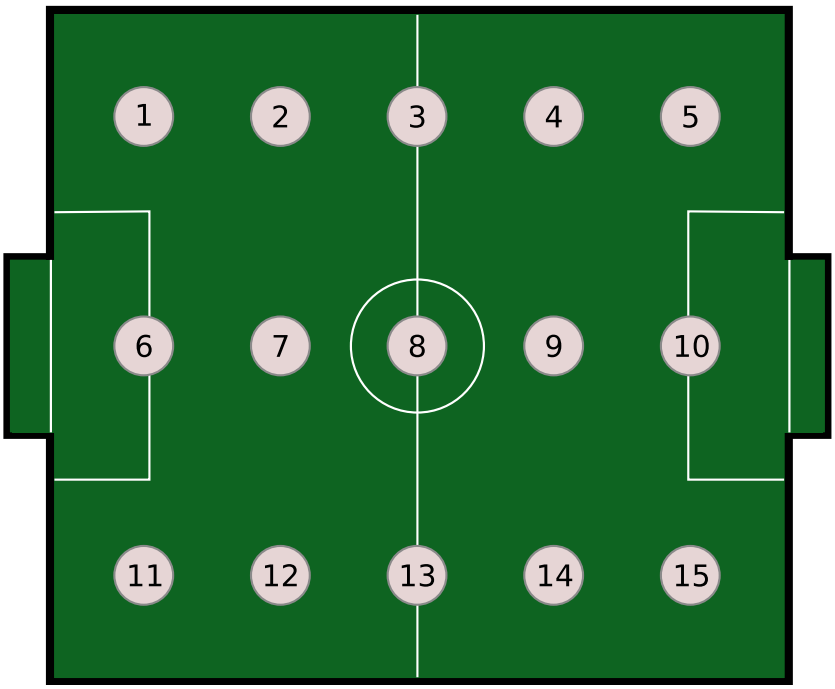

Figure 4.1: Possible targets for moving along predefined paths

Algorithm 4 CoordinatedMove ()

```
1: Initialize target_positions[1..15]
   {Set the paths for both robots}
2: path(R_1) = [6, 10] {The first robot is moving from left to right}
3: path(R_2) = [3, 13] {The second robot is moving from top to bottom}
4: current_index(R_1) = current_index(R_2) = 0; {start with the first waypoints}
5: for all robots R do
6:    if dist(R, target_positions[current_index(R)]) < ε then
7:       Increment current_index(R) or wrap to 0
8:    end if
9:    Move R to target_positions[current_index(R)]
10: end for
```

4.1.2 Moving along random paths

The tests from Section 4.1.1 can simulate all types of collisions in a reproducible manner, but real collisions are often more complex, involving even quicker path

changes and highly curved paths. We simulate this by moving both robots along *fully* random paths, like illustrated in Figure 4.2 and described in Algorithm 5.

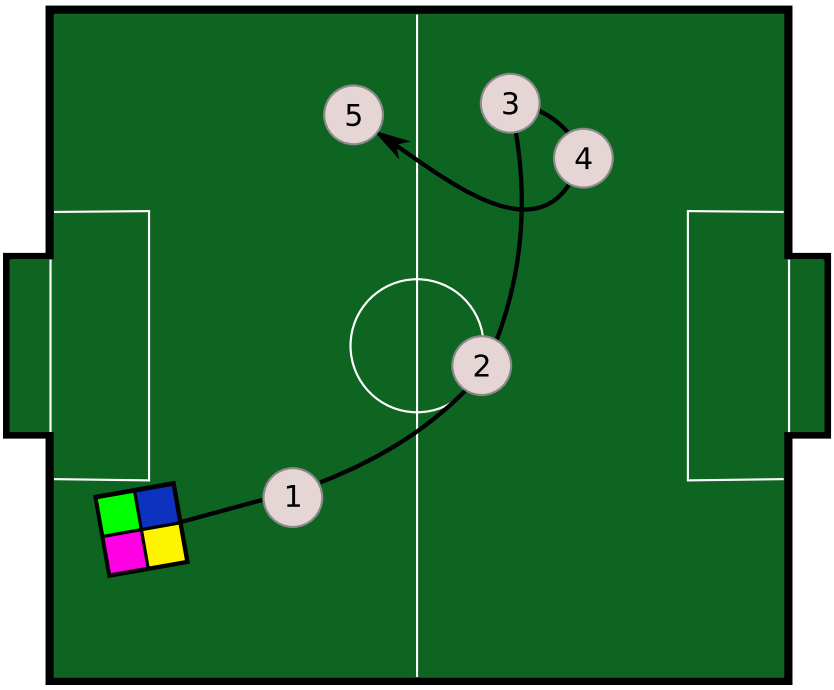

Figure 4.2: A fully random path can be quite complex

The drawback is that each time the evaluation function is run, the paths are different. This makes it harder to get significant data. We try to compensate for that by increasing the time for this test. Over time, the ,,luck" of getting easy or complex paths should even out.

This test is definitely the most difficult one for our algorithm, as the single paths between random points can be very short and also really hard to predict.

4.2 Evaluation criteria

All tests from the previous section are performed for a certain time span and with a variety of different maximum speeds. The higher the speeds, the shorter we run

Algorithm 5 RandomMove ()

```
1: Initialize target_position[] array for all robots with random positions
2: for all robots R do
3:    if dist(R, target_position[R]) < ϵ then
4:       target_position[R] ← new random position
5:    end if
6:    Move R to target_position[R]
7: end for
```

the tests in order to prevent our robots from damage. While they are built really robust, a head-on collision with 3 m/s could still cause expensive repairs.

While running the tests, we keep track of different properties to compare the individual collision avoidance functions:

4.2.1 Number of collisions

For many applications, the number of collision is the most important criteria for a successful collision avoidance system. Some may even require zero collisions for an acceptable system.

We differentiate between a *full* collision with heavy impact and noticeable path changes or speed losses and between a *,,half"* collision where two robots only touch slightly but are not really affected by that collision.

4.2.2 Average speed

For other applications with robust robots (like our *RobySpeed* robots) it can be more important to keep track of the average speed of the robots for a given path. Thus, the movement functions were adapted to keep track of the total distance which all robots have moved. Now the average speed can be easily calculated with this equation:

$$v = |\frac{d}{t}| \tag{4.1}$$

v is the average velocity and t the time required for distance d. d in turn is the shortest possible distance for the given path[1] and *not* the distance which the actual robot moved.

4.3 Results

4.3.1 Head-on collision

Head-on collisions were simulated by constantly moving two robots between two points (6 and 10 in our test setup) with different start positions like shown in Figure 4.3. Without any collision avoidance module, both robots would reliably crash into each other in the center of the field and stuck to each other. Therefore we only performed this test with the two collision avoidance algorithms.

Function	max speed	time	path length	collisions	avg speed
CollisionAvoid	1 m/s	5 min	304.0 m	0	0.51 m/s
	2 m/s	3 min	249.6 m	0	0.69 m/s
	3 m/s	2 min	294.4 m	0	1.23 m/s
AutoAvoid	1 m/s	5 min	307.2 m	0	0.51 m/s
	2 m/s	3 min	280.6 m	0	0.78 m/s
	3 m/s	2 min	339.2 m	0	1.41 m/s

Table 4.1: Test results for head-on collisions

Both collision avoiding functions could handle this test case without any collision (Table 4.1). The primitive `AutoAvoid` algorithm is, however, slightly faster than our `CollisionAvoid` algorithm. The reason is that our algorithm uses a larger safety margin.

We could of course optimize the safety margin for each individual test but this contradicts scientific evaluation where you should not optimize for individual test cases. Otherwise, the algorithm becomes biased on the test data and does not perform as well on unknown test cases. Furthermore, the larger safety margin should pay off in the other – more complex – tests.

[1] While not necessarily the fastest path, moving along straight lines between the way points results in the shortest possible path

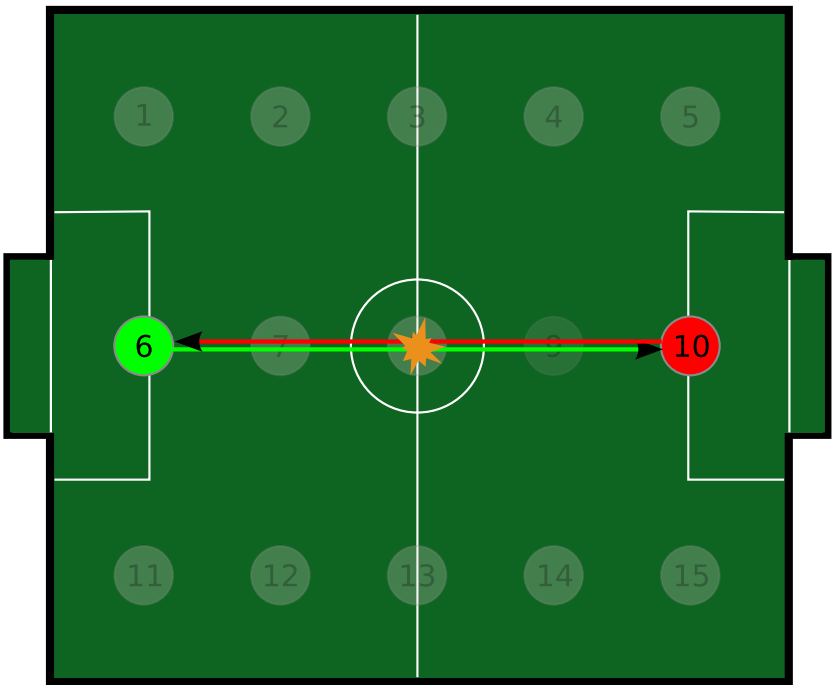

Figure 4.3: Head-on collisions are simulated by R_1 moving between target positions 6 and 10, and R_2 moving between positions 10 and 6

4.3.2 Perpendicular collision

Like in the previous test, perpendicular collisions were also simulated by moving two robots on straight lines. This time, however, one of the robots' paths was rotated by 90° in order to achieve a perpendicular crash situation like illustrated in Figure 4.4.

`CollisionAvoid` handled this test by far the best. It had the least crashes (12 in total) and also the highest average velocity. The basic `AutoAvoid` function performed only slightly better (35 crashes) than using no collision avoidance at all (`SuperPosition`, which had 44 crashes). The total distance did not vary significantly for each algorithm.

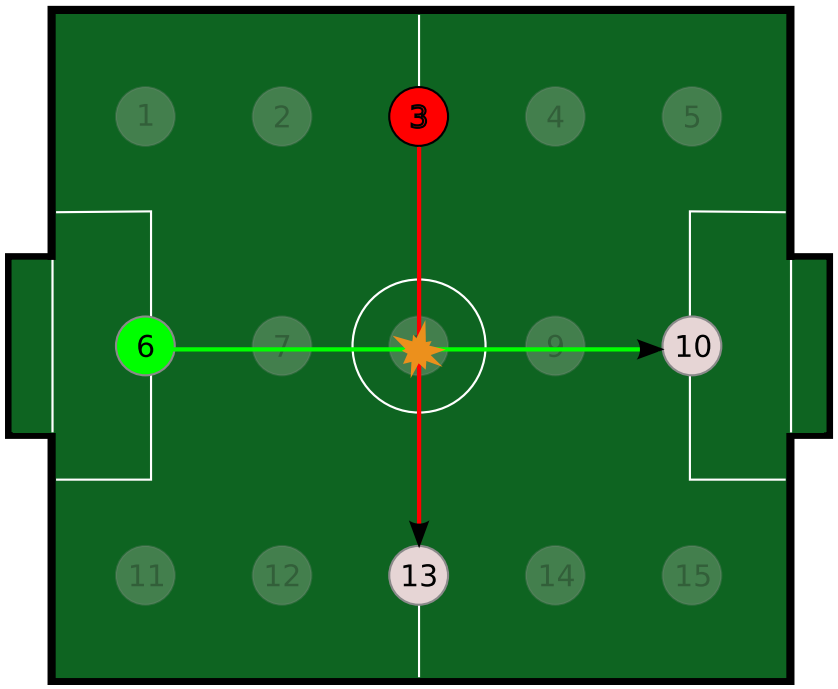

Figure 4.4: Perpendicular collisions are simulated by R_1 moving between target positions 6 and 10, and R_2 moving between positions 3 and 13

Function	max speed	time	path length	collisions	avg speed
CollisionAvoid	1 m/s	5 min	310.8 m	3.5	0.52 m/s
	2 m/s	3 min	260.4 m	5.5	0.72 m/s
	3 m/s	2 min	212.4 m	3	0.88 m/s
AutoAvoid	1 m/s	5 min	290.4 m	18	0.48 m/s
	2 m/s	3 min	267.4 m	6.5	0.74 m/s
	3 m/s	2 min	206.4 m	11	0.86 m/s
SuperPosition	1 m/s	5 min	318.8 m	16	0.53 m/s
	2 m/s	3 min	263.2 m	11.5	0.73 m/s
	3 m/s	2 min	193.6 m	17	0.80 m/s

Table 4.2: Test results for perpendicular collisions

4.3.3 Angular collision

Angular collisions were simulated by moving one robot on a straight line and the other one between the diagonally opposite corners of the soccer field (Figure 4.5).

As a result, the algorithm had to handle both acute as well as obtuse collisions.

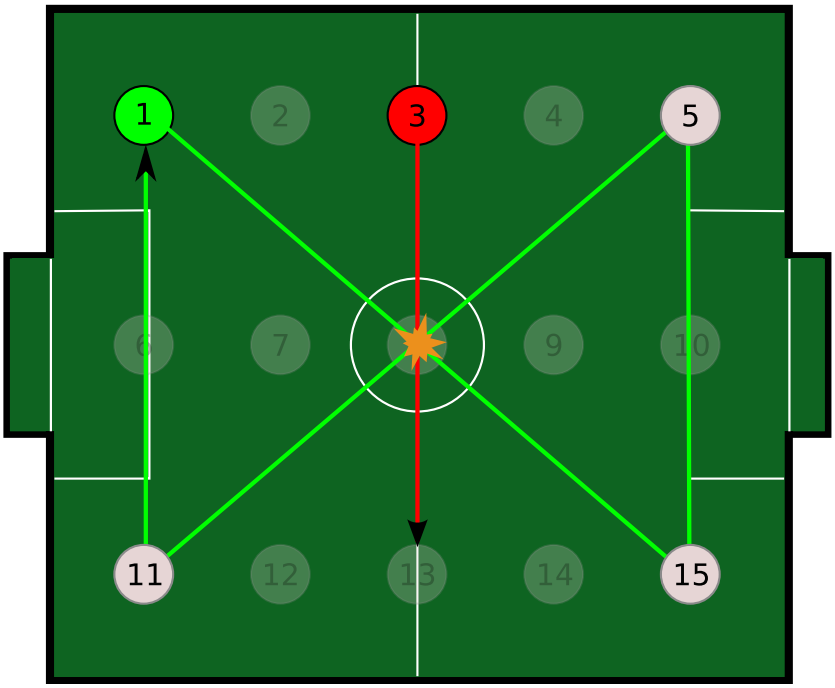

Figure 4.5: Test path for angular collisions are simulated by R_1 moving between target positions 1, 15, 5 and 11 and R_2 moving between positions 3 and 13

Function	max speed	time	path length	collisions	avg speed
CollisionAvoid	1 m/s	5 min	324.8 m	0	0.54 m/s
	2 m/s	3 min	272.8 m	0	0.76 m/s
	3 m/s	2 min	220.0 m	0.5	0.92 m/s
AutoAvoid	1 m/s	5 min	318.0 m	0	0.53 m/s
	2 m/s	3 min	258.4 m	7	0.72 m/s
	3 m/s	2 min	204.0 m	7	0.85 m/s
SuperPosition	1 m/s	5 min	312.4 m	11.5	0.52 m/s
	2 m/s	3 min	269.6 m	2	0.75 m/s
	3 m/s	2 min	215.2 m	3	0.90 m/s

Table 4.3: Test results for angular collisions

Our `CollisionAvoid` algorithm could handle this test case by far the best. We only observed one very light collision whereas `AutoAvoid` had a total of 14 crashes, which was only marginally better than 16.5 crashes without any collision avoidance

module. Furthermore, this test case showed that handling collisions can also lead to a significantly higher average speed.

4.3.4 Static obstacle

We also tested collisions with static obstacles by letting one robot rest in a fixed position and the other robot move on paths where it would hit the first robot from various directions (Fig. 4.6).

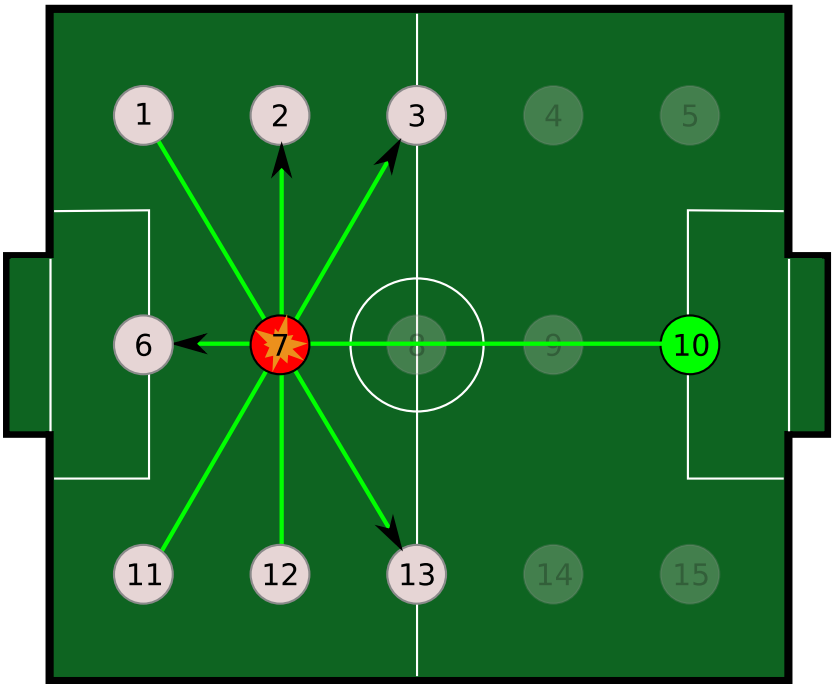

Figure 4.6: Test path for a collision with a static obstacle. For illustration purposes, only the important parts of the path are drawn. The full path for R_1 is: 10-6-11-12-2-1-13-11-3-5-10.

We again abstained from performing this test without collision avoidance as the test path contains direct collisions where both robots would get stuck to each other. Both collision avoidance functions could, however, handle this test case flawlessly without any collision (Table 4.4). `AutoAvoid` was again slightly faster due to a smaller safety margin.

Function	max speed	time	path length	collisions	avg speed
CollisionAvoid	1 m/s	5 min	134.8 m	0	0.45 m/s
	2 m/s	3 min	108.1 m	0	0.60 m/s
	3 m/s	2 min	84.3 m	0	0.70 m/s
AutoAvoid	1 m/s	5 min	141.5 m	0	0.47 m/s
	2 m/s	3 min	111.4 m	0	0.62 m/s
	3 m/s	2 min	89.8 m	0	0.75 m/s

Table 4.4: Test results for collisions simulated by moving R_1 on a path where it would hit the static obstacle R_2 from various directions.

4.3.5 Complex predefined paths

While the previous tests were quite simple and aimed at checking the performance for a single type of collision, we also performed an additional test which could test all types of collisions with a single path (Figure 4.7). Furthermore, the angles for each possible collision are quite diverse.

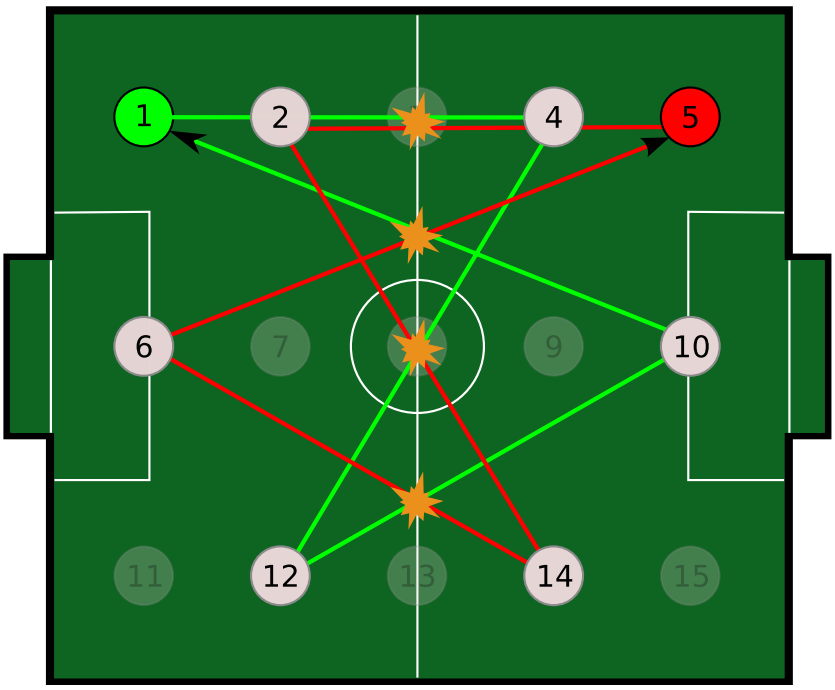

Figure 4.7: Complex path for testing different types of collisions with different impact angles. R_1 moves on path 1-4-12-10-1 and R_2 on path 5-2-14-6-5.

The results for this test (Table 4.5) clearly indicate that `CollisionAvoid` outperforms `AutoAvoid` in terms of the number of collisions (3.5 vs. 55). The difference in the average speed was not as distinct but still measurable.

Function	max speed	time	path length	collisions	avg speed
CollisionAvoid	1 m/s	5 min	281.9 m	3	0.47 m/s
	2 m/s	3 min	242.0 m	0.5	0.67 m/s
	3 m/s	2 min	181.2 m	0	0.76 m/s
AutoAvoid	1 m/s	5 min	282.2 m	22	0.47 m/s
	2 m/s	3 min	227.8 m	24	0.63 m/s
	3 m/s	2 min	182.2 m	9	0.76 m/s

Table 4.5: Test results for a complex path

4.3.6 Random paths

In our last test, we moved both robots to purely random destinations. This was done about 3 times longer than all other tests in order to get more significant data.

As expected, collision avoidance is much harder for this test as the path prediction works less reliably. Although our proposed algorithm has the fewest collisions (90 in 30 minutes total running time), it is still far away from handling all collisions properly. When running the test with `AutoAvoid` we counted 106.5 collisions, with `SuperPosition` we counted 150.5 collisions.

Function	max speed	time	path length	collisions	avg speed
CollisionAvoid	1 m/s	15 min	1059.8 m	25	0.59 m/s
	2 m/s	10 min	1033.1 m	43	0.86 m/s
	3 m/s	5 min	605.6 m	22	1.01 m/s
AutoAvoid	1 m/s	15 min	1145.3 m	36.5	0.64 m/s
	2 m/s	10 min	1062.2 m	38.5	0.89 m/s
	3 m/s	5 min	621.7 m	31.5	1.04 m/s
SuperPosition	1 m/s	15 min	1152.2 m	56.5	0.64 m/s
	2 m/s	10 min	1027.4 m	62	0.86 m/s
	3 m/s	5 min	613.7 m	32	1.02 m/s

Table 4.6: Test results for random paths

Chapter 5

Conclusions

In the previous chapters we have presented our algorithm for a collision avoidance algorithm in a multi-agent system. It was also implemented and evaluated within an existing robot soccer framework. The basic idea of the algorithm is to divide it into a collision detection and a collision prevention part. The collision detection is in turn a 3-step process which is repeated for all possible obstacles:

1. Transform the obstacle in the coordinate system of our robot.
2. Calculate the possible intersection point using estimated paths of the two robots.
3. Calculate the estimated arrival times at the intersection point. When the time difference is below a certain threshold we indicate a collision.

The collision could then be prevented by one of two strategies, depending on the type of collision:

- Change directions of both robots when there is a head-on collision
- Change the speed of one robot when there is either a perpendicular or angular collision

Overall, this system worked better than the previous `AutoAvoid` algorithm and certainly better than using no collision avoidance at all when it comes to the number of collisions. Counting the total number of collisions during all tests, we could avoid about two thirds of all collisions and ended up with a total number of 106 (Fig. 5.1). There is a big discrepancy in the individual tests though. On the one hand, all tests with fixed paths could be handled quite well. Therefore we deduce that the actual collision prevention part works in a satisfactory manner. On the other hand, the test with fully random paths still resulted in a lot of crashes. The problem is that the collision detection part has difficulties detecting quick path changes and adapting to them.

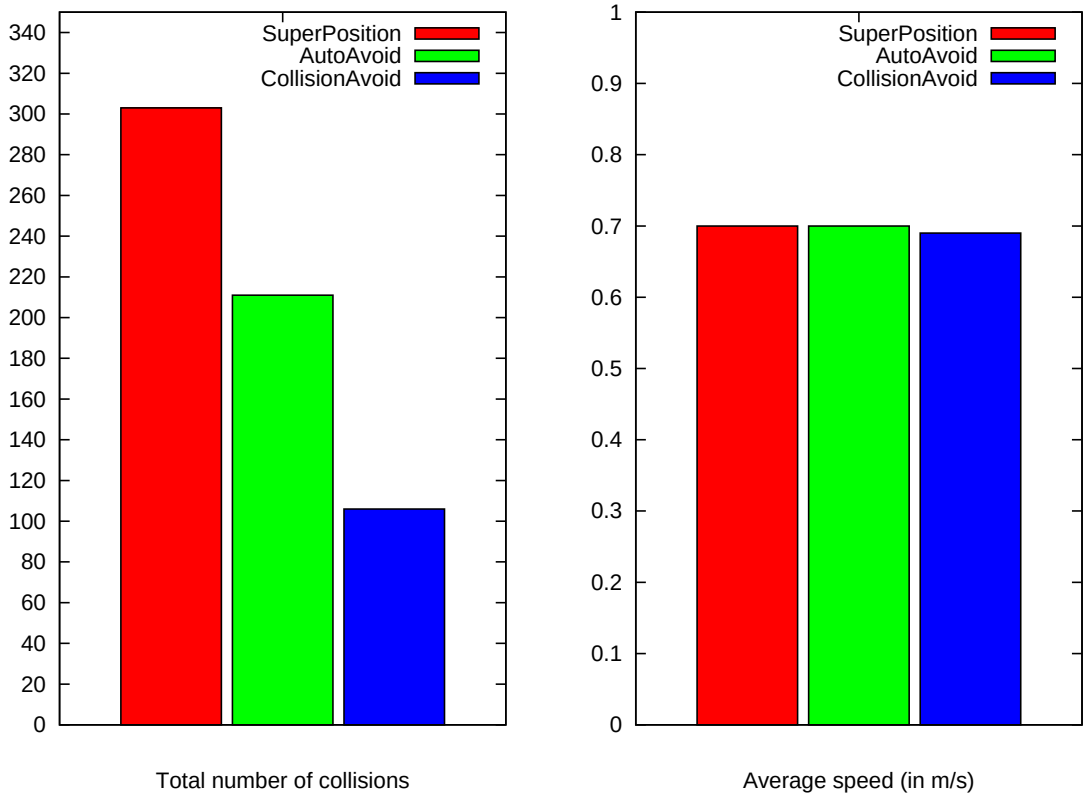

Figure 5.1: Statistics using data generated by running all tests. Values for `SuperPosition` were interpolated because it could not run all tests.

Initially, we also thought that successfully avoiding collisions could increase the average speed of the robots. This assumption, however, turned out to be false –

at least with our implementation. The average speed of all systems was close to 0.7 m/s and the remaining small deviations are statistically irrelevant as they could be measurement errors.

The bottom line is that while our approach successfully decreases the number of collisions it does not improve the overall speed of the objects. Therefore the benefits for robot soccer are negligible as the robots are really very robust, which they certainly showed by not suffering from any damages during the tests.

5.1 Future improvements

If somebody wants to build upon the results of this master's thesis, the main focus should be on researching a better path prediction module. One possibility would be to use a Kalman filter [WB95] but it is uncertain if that can keep up with instant path changes which occur often in robot soccer. Other interesting research areas include:

- Integrate collision information into the strategy module. While not always possible, sending a robot to a target where the path to it is less likely to have a collision is always better than trying to avoid imminent collisions.
- Test/expand the system to work with more than two robots. The collision finding algorithm already handles multiple robots well, but the collision prevention part does not check whether avoiding the collision with one robot actually results in a collision with another robot.
- Currently we try to avoid certain collisions by stopping one robot. It would be interesting to see, if actually *increasing* the speed of the other robot in certain cases would yield better results. This, however, is not easy with the current framework because it always tries to move and accelerate robots with the maximum possible (or user-defined) speed.

List of Figures

List of Tables

List of Algorithms

Appendix A: Source code

NextCollision()

```
// angle is the angle of whichrobot to its aim
// returns a (static) struct to the next collision
Collision NextCollision(int whichrobot, double angle)
{
  Collision collision;
  collision.type = NO_COLLISION;

  // TDP: point data structure, TDA: angle data structure
  TDP H, R;
  H.init();
  R.init();
  R.A.x = dat.hr[whichrobot].x;
  R.A.y = dat.hr[whichrobot].y;

  double my_vel = PredictRobotVelocity(whichrobot);
  if (my_vel < 0.1)
    my_vel = 0.1;

  // Home robots:
  for(unsigned int i = HT1; i < dat.nrofrobots; i++)
  {
    // don't check for collision with itself
    if (i == whichrobot)
      continue;

    H.A.x = dat.hr[i].x;
    H.A.y = dat.hr[i].y;
    // transform this robot into the coordinate system of whichrobot
    H.R = Transform(H.A, R.A, angle);

    double relative_angle = ATransform(dat.hr[i].a, dat.hr[whichrobot].a);
```

```
// this keeps our signs for the sin/cos functions like they should be
if (relative_angle > M_PI)
  relative_angle -= M_PI;

// calculate the time needed to the collision point
double opp_vel = PredictRobotVelocity(i);
double my_time = 1.0, opp_time = 1.0;

// find the collision point: 0 = y - t * sin() -> t = y/sin();
TDP collision_point;
collision_point.init();

double diff_angle = fabs(PredictRobotAngle(i) -
                         PredictRobotAngle(whichrobot));
// -> if the 2 robots are parallel, the angle is either 0, PI or 2*PI
// if they stand 90 degree to each other, it is PI/2 or PI+PI/2
if (fabs(diff_angle - M_PI) < M_PI/6) // check for frontal collision
{
  // possible collision point is in the middle of the two robots
  collision_point.A.x = (H.A.x + R.A.x) / 2;
  collision_point.A.y = (H.A.y + R.A.y) / 2;
  collision_point.R = Transform(collision_point.A, R.A, angle);

  if (H.R.x < 0) // opponent is driving in the wrong direction
    opp_time = -1.0;
}
else
{
  double to_go = -H.R.y / fabs(sin(relative_angle));
  collision_point.R.x = H.R.x + to_go * cos(relative_angle);
  collision_point.R.y = 0.0;
  collision_point.A = BackTransform(collision_point.R, R.A, angle);

  // check if opponent not already past collision point
  double vx = dat.hr[i].PredictorX.v();
  double vy = dat.hr[i].PredictorY.v();
  // opponent driving in "wrong" direction
  DPoint future_position = H.A;
  future_position.x += vx;
  future_position.y += vy;

  if(dist(collision_point.A,H.A) < dist(collision_point.A,future_position))
      opp_time = -1.0;
}

my_time = collision_point.R.x / my_vel;

// continue, if we already have a collision that happens earlier
if (collision.type != NO_COLLISION && my_time > collision.time)
  continue;
```

```
if (opp_vel > 0.1)
{
  opp_time *= dist(H.R, collision_point.R) / opp_vel;
}
else // robot not (really) moving
{
  // if the robot is not moving, but already at the collision point
  if (fabs(H.R.y) < 0.1)
    opp_time = my_time;
  else
    opp_time = -1;
}

// if the collision point is in the past of one robot
// don't assume a collision
if (my_time < 0.0 || opp_time < -0.1)
  continue;

// check the type of the collision
//
// -> if the 2 robots are parallel, the angle is either 0, PI or 2*PI
// if they stand 90 degree to each other, it is PI/2 or PI+PI/2
if (fabs(diff_angle - M_PI) < M_PI/6 || opp_vel <= 0.1) // head-on
{
  // we can't use the intersection point, as the lines are too
  // parallel
  if (H.R.x > 0.0 && fabs(H.R.y) < 0.1) // the robot is ahead of us
    collision.type = HEADON_COLLISION;
}
else
{
  if (my_time < 1.0 && fabs(my_time - opp_time) < 0.3)
  {
    // finally set the collision type if there is a possible
    // collision
    if (fabs(diff_angle - M_PI2) < M_PI/6 ||
        fabs(diff_angle - M_PI - M_PI2) < M_PI/6 )
      collision.type = PERPENDICULAR_COLLISION;
    else
      collision.type = ANGULAR_COLLISION;
  }
}

collision.is_opponent_team = false;
collision.distance = collision_point.R.x;
collision.time = my_time;
collision.my_speed = my_vel;
collision.opp_speed = opp_vel;
collision.colliding_robot = i;
```

```
    collision.opp_from_right = H.R.y < 0.1; // 0.1 as safety margin
  }

  return collision;
}
```

CollisionAvoid()

```
// currently only activated on dat.movetype = TEST or dat.movetype = MOUSE
void CollisionAvoid(int whichrobot, double x, double y, double endspeed)
{
  // our robot
  TDP R; R.init(); // TDP = point structure
  R.A.x = dat.hr[whichrobot].x;
  R.A.y = dat.hr[whichrobot].y;

  // the target point
  TDP target; target.init();
  target.A.x = x; // target.A.{x,y} is the absolute coordinate system
  target.A.y = y;

  const double FOUND_TARGET_DIST = 0.1;

  // the robot is near enough the aim, don't check for collision anymore
  if (!dat.useCollisionAvoid || dist(R.A, target.A) < FOUND_TARGET_DIST)
  {
    // Positioning without collision detection
    SuperPosition(whichrobot, x, y, endspeed);
    return;
  }

  // the angle between the selected robot and its target
  TDA angle; angle.init();
  angle.A = atan2((target.A.y - R.A.y),(target.A.x - R.A.x));
  if (angle.A < 0)
    angle.A = angle.A + 2*M_PI;

  // find the next collision
  Collision collision = NextCollision(whichrobot, angle.A);

  // Middle Point, if we need to circumnavigate an obstacle we drive to
  // this point, until the way is free to the original target
  TDP temp; temp.init();
```

```
temp.A.x = target.A.x;
temp.A.y = target.A.y;

if (collision.type != NO_COLLISION)
{
  target.R = Transform(target.A, R.A, angle.A);
  temp.R.x = target.R.x;
  temp.R.y = target.R.y;

  // look ahead exactly one second
  // there is no reason to check collisions more than 1 sec in advance
  // due to the high dynamics of robot soccer and the high speeds
  if (collision.time <= 1.0)
  {
    // the lower avoid is, the closer we try to avoid the collision
    // but the more prone we are to collisions
    double avoid = (collision.my_speed + collision.opp_speed) / 10;
    if (avoid > 0.5)
      avoid = 0.5;
    else if (avoid < 0.1)
      avoid = 0.1;

    // code vor avoiding the collision:
    switch (collision.type)
    {
      case HEADON_COLLISION:
        temp.R.x = collision.distance - avoid;
        if (temp.R.x < 0.2) // always at least 20 cm in advance
          temp.R.x = 0.2;

        // dodge to the left or right
        temp.R.y = collision.opp_from_right ? avoid : -avoid;
        break;

      case PERPENDICULAR_COLLISION:
      case ANGULAR_COLLISION:
        if (collision.opp_from_right)
        {
          temp.R.x = 0.0;
          temp.R.y = 0.0;
          endspeed = 0.0;
        }
        else // the collision is handled by the other robot
          ;
        break;
    }
    temp.A = BackTransform(temp.R, R.A, angle.A);
  }
}
```

```
  // Final positioning without collision detection
  SuperPosition(whichrobot, temp.A.x, temp.A.y, endspeed);
}
```

CoordinatedMove()

```
// move the robots in a certain pattern
void CoordinatedMove_55 (bool wait)
{
  int robots[] = {HT1, HT2};
  int num_robots = 2;

  double border_safety_margin = 0.2;
  double min_x = border_safety_margin;
  double max_x = dat.PlayGroundDimX - border_safety_margin * 2;
  double min_y = border_safety_margin;
  double max_y = dat.PlayGroundDimY - border_safety_margin * 2;

  /*
   * the position indices where the robots should go:
   *
   * ####################
   * #0   1   2   3   4#
   * #-+      |      +-#
   * #5|  6  (7)  8  |9#
   * #-+      |      +-#
   * #10 11  12  13  14#
   * ####################
   */

  static DPoint positions[15];
  static pos_initialized = false;
  if (!pos_initialized)
  {
    for (int i = 0; i < 15; i++)
    {
      int col = i % 5;
      int row = i / 5;
      positions[i].x = min_x + col * (max_x - min_x) / 4;
      positions[i].y = min_y + row * (max_y - min_y) / 2;
    }
    pos_initialized = true;
  }
```

```
    // the robots will go to these positions in order
    // make sure to only uncomment one path at a time, and terminate
    // the paths with -1
    static int robot_position_indices[2][50] = {
      {5,9,-1}, {12,2,-1}                    // +-like collision
      // {5,9,-1}, {9,5,-1}                  // frontal collision
      // {6,-1}, {9,5,10,11,1,0,12,10,2,4,-1} // static obstacle
      // {0,14,4,10,-1}, {12,2,-1}           // angular collision
      // {0,3,11,9,0,-1},{4,1,13,5,4,-1}     // complex path
      // {2, 5, 12, 9, -1}, {3, 6, 13, -1}
      // {5, 9, 14, 10, -1}, {11,13,3, -1}
      // {5, 9, 14, 12, 2, -1}, {3, 1, 11, 13, -1}
      // {0,4,9,5,10,14,9,5,-1}, {14,4,2,12,10,0,2,12,-1}
      // {0, 13, -1}, {14,1, -1}
      // {1,13,11,3,-1}, {2,8,12,6,-1}
    };
    static int current_robot_position_index[] = { 0, 0 };
    static bool robot_attarget[] = {true, true};

    // when waiting for the other robot, we need to wait 20
    // frames (~0.3sec) for the other to really come to rest
    static wait_counter = 20;
    if (wait)
    {
      for (int k = 0; k < num_robots; k++)
        if (!robot_attarget[k])
          goto go;

      wait_counter--;
      if (wait_counter <= 0)
      {
        for (int k = 0; k < num_robots; k++)
        {
          current_robot_position_index[k] = 0;
          robot_attarget[k] = false;
        }
        wait_counter = 20;
      }
    }

  go:
    // go there
    double distance = 0.05;
    for (int i = 0; i < num_robots; i++)
    {
      double wanted_x = positions[robot_position_indices[i]
          [current_robot_position_index[i]]].x;
      double wanted_y = positions[robot_position_indices[i]
          [current_robot_position_index[i]]].y;
```

```
      CollisionAvoid(robots[i], wanted_x, wanted_y, 0.);
      if (wait && robot_attarget[i])
        continue;

      if (DistFunk55(dat.hr[robots[i]].x, dat.hr[robots[i]].y,
            wanted_x, wanted_y) < distance)
      {
        int saved_index = current_robot_position_index[i];
        current_robot_position_index[i]++;

        if (robot_position_indices[i][current_robot_position_index[i]]==-1)
        {
          current_robot_position_index[i] = 0;
          robot_attarget[i] = true;
        }

        // update total distance
        double future_x = positions[robot_position_indices[i]
            [current_robot_position_index[i]]].x;
        double future_y = positions[robot_position_indices[i]
            [current_robot_position_index[i]]].y;

        dat.DDebug[dat.useCollisionAvoid ? 18 : 24] +=
          DistFunk55(wanted_x, wanted_y, future_x, future_y);
      }
    }
  }
```

RandomMove()

```
// this function moves a robot to a randomly calculated place
// When it already is there, find another random place and go there
// NOTE: it just works for 5x5 size field
void RandomMove(int whichrobot)
{
  static DPoint positions[MAX_NR_OF_ROBOTS];
  static bool valid_pos[MAX_NR_OF_ROBOTS] =
    { false, false, false, false, false, false,
      false, false, false, false, false };
  static bool initialized = false;

  // avoid going nearer than 40 cm to the border
```

```
    const double AVOID_BORDER = 0.4;

    // if we are already at the random position, calculate a new one
    if (DistFunk55(dat.hr[whichrobot].x, dat.hr[whichrobot].y,
      positions[whichrobot].x, positions[whichrobot].y) < 0.1)
        valid_pos[whichrobot] = false;

    if (!valid_pos[whichrobot])
    {
      int current_x, current_y;
      if (initialized)
      {
        current_x = positions[whichrobot].x;
        current_y = positions[whichrobot].y;
      }
      positions[whichrobot].x = AVOID_BORDER + (rand() /
          (double)RAND_MAX) * (PLAYGROUND_DIM_X_55 - 2*AVOID_BORDER);
      positions[whichrobot].y = AVOID_BORDER + (rand() /
          (double)RAND_MAX) * (PLAYGROUND_DIM_Y_55 - 2*AVOID_BORDER);
      valid_pos[whichrobot] = true;

      if (initialized)
      {
        dat.DDebug[dat.useCollisionAvoid ? 18 : 24] +=
          DistFunk55(current_x, current_y,
          positions[whichrobot].x, positions[whichrobot].y);
      }
      else
        initialized = true;
    }

    // now go there
    CollisionAvoid(whichrobot, positions[whichrobot].x,
        positions[whichrobot].y, 0);
  }
```

Bibliography

[AMT01] N. Achour, N.K. M'Sirdi, and R. Toumi. Reactive Path Planning with Collision Avoidance in Dynamic Environments. In *10th IEEE International Workshop on Robot and Human Interactive Communication*, pages 62–67, Bordeaux, Paris, France, 2001.

[Cam90] S. Cameron. Collision detection by four–dimensional intersection testing. *IEEE Transaction on Robotics and Automation*, 6(3):291–302, 1990.

[CSMOS03] Dong Eui Chang, Shawn C. Shadden, Jerrold E. Marsden, and Reza Olfati-Saber. Collision avoidance for multiple agent systems. 2003.

[FBT95] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. Technical Report IAI-TR-95-13, 1, 1995.

[FIR98] FIRA — Federation of International Robot-soccer Association. http://www.fira.net/, 1998.

[Flo01] Adina Magda Florea. Introduction to Multi-Agent Systems. In *Continuous Education Program on Intelligent Agents Technology and Knowledge Processing*, pages 49–64, 2001.

[FT02] Atsushi Fujimori and Shinsuke Tani. A Navigation of Mobile Robots with Collision Avoidance for Moving Obstacles. *Industrial Technology, 2002. IEEE ICIT '02*, 1(4):1–6, 2002.

[HCL99] Kao-Shing Hwang, Hung-Jen Chao, and Jy-Hsin Lin. Collision-Avoidance Motion Planning Amidst Multiple Moving Objects. *J. Inf. Sci. Eng.*, 15(5):715–736, 1999.

[HFE07] History and future events. http://www.roboterfussball.at/info/gesch-eng.html, 2007.

[JJG02] Jonas Jansson, Jonas Johansson, and Fredrik Gustafsson. Decision Making for Collision Avoidance Systems, 2002.

[KAK+97] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: The robot world cup initiative. In *AGENTS '97: Proceedings of the first international conference on Autonomous agents*, pages 340–347, New York, NY, USA, 1997. ACM Press.

[Lau98] Jean-Paul P. Laumond. *Robot Motion Planning and Control*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[LaV06] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[NM05] Gregor Novak and Stefan Mahlknecht. TINYPHOON: A Tiny Autonomous Mobile Robot. volume 4, pages 1533–1538, June 2005.

[Nov05] Gregor Novak. Roby-go, a prototype for cooperating mirosot soccer-playing robots. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, 9(4):409–414, 2005.

[Put04] Bernhard Putz. *Navigation mobiler, kooperativer Roboter*. PhD thesis, Wien, 2004.

[QHH03] Huang Qianwei, Ma Hongxu, and Zhang Hui. Collision-avoidance mechanism of multi agent system. In *Robotics, Intelligent Systems and Signal Processing*, volume 2, pages 1036–1040, 2003.

[RAP+04] S. H. Rhee, M. B. Ahmad, S.-J. Park, K.-J. Beak, and J. A. Park. Collision avoidance of two moving objects using the anticipated path. In

S. Cao, C.-L. I, and J.-A. Tsai, editors, *Mobile Service and Application. Edited by Cao, Shumin; I, Chih-Lin; Tsai, Jiann-An. Proceedings of the SPIE, Volume 5283, pp. 88-96 (2004).*, volume 5283 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, pages 88–96, March 2004.

[RCB98] RoboCup : Brief of History. http://www.robocup.org/overview/23.html, 1998.

[Rog04] Radu Rogojanu. Mobile Robots' Path Planning - Potential Field Method in Known and Unknown Environments. Master's thesis, Romania, 2004.

[RS94] John Reif and Micha Sharir. Motion planning in the presence of moving obstacles. *J. ACM*, 41(4):764–790, 1994.

[Rud97] M. Rude. Collision Avoidance by Using Space-Time Representations of Motion Processes. *Autonomous Robots*, 4(1):101–119, 1997.

[TCA00] Introduction to TCAS II Version 7. http://www.arinc.com/downloads/tcas/tcas.pdf, 2000.

[WB95] Greg Welch and Gary Bishop. An Introduction to the Kalman Filter. Technical report, Chapel Hill, NC, USA, 1995.

[Wür05] Markus Würzl. *Ein Beitrag zur Robotik speziell im Bereich der Multiagenten- und Expertensysteme.* PhD thesis, Wien, 2005.

[Zel95] J. S. Zelek. Dynamic path planning. pages 1285–1290 (vol. 2), 1995.