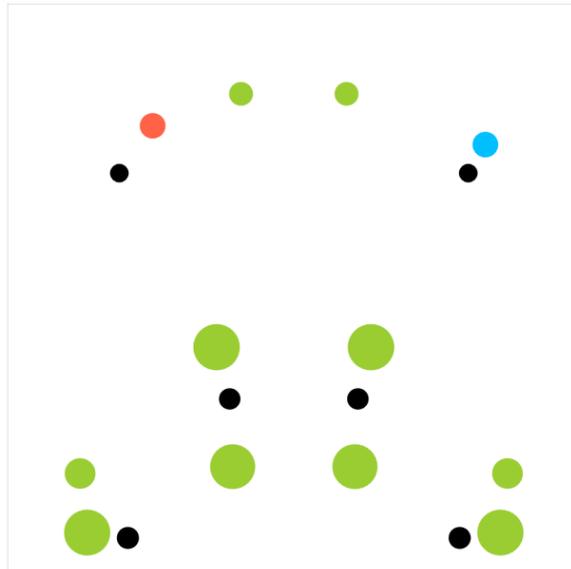


S · p · a · c · e · b · a · l · l · s



#### Halbwertszeit

Gordon Freeman  
Alyx Vance  
Eli Vance  
Isaac Kleiner  
Barney Calhoun  
Odessa Cabbage  
G-Man



#### Daleks

Dalek Sec  
Dalek Thay  
Dalek Caan  
Dalek Jast  
Davros

Spur  Geschwindigkeit  Beschleunigung  Untertitel  Video  Slomo

# Spaceballs

Künstliche Intelligenz in der Informatiklehre?

Jörg J. Buchholz

24. Januar 2015

# Inhaltsverzeichnis

<b>I</b>	<b>Bedienungsanleitung</b>	<b>4</b>
<b>1</b>	<b>Einführung</b>	<b>5</b>
<b>2</b>	<b>Spielablauf</b>	<b>6</b>
2.1	Tankstellen, Minen und Banden . . . . .	6
2.2	Anzeigen . . . . .	7
<b>3</b>	<b>Regeln</b>	<b>10</b>
<b>II</b>	<b>Unter der Haube</b>	<b>12</b>
<b>4</b>	<b>Blockschaltbild</b>	<b>13</b>
<b>5</b>	<b>Simulation in MATLAB</b>	<b>15</b>
5.1	spaceballs . . . . .	15
5.2	teams_einlesen . . . . .	16
5.3	team.xml . . . . .	18
5.4	spielfeld_erstellen . . . . .	19
5.5	schnitt_kreis_bande . . . . .	23
5.6	schnitt_kreis_kreis . . . . .	24
5.7	spielen . . . . .	25
5.7.1	video_auswaehlen . . . . .	33
5.8	kraft . . . . .	35
5.9	json_speichern . . . . .	36
5.10	spiel . . . . .	37
5.10.1	spiel.rot . . . . .	38
5.10.2	spiel.rot.mitarbeiter . . . . .	39
5.10.3	spiel.rot.video . . . . .	39
5.10.4	spiel.mat . . . . .	40
5.11	spielfeld . . . . .	41
<b>6</b>	<b>Visualisierung in HTML5</b>	<b>44</b>
6.1	spiel.js . . . . .	44

6.2	spiel.html	46
6.2.1	spiel.css	46
6.2.2	<head>	48
6.2.3	<body>	48
6.2.4	<script>	53
6.2.4.1	Blockschaltbild	53
6.2.4.2	teams_benennen	55
6.2.4.3	animieren	56
6.2.4.4	zeichnen	57
6.2.4.5	zeitbalken_zeichnen	59
6.2.4.6	spritbalken_zeichnen	60
6.2.4.7	umrandung_zeichnen	61
6.2.4.8	minen_zeichnen	62
6.2.4.9	tanken_zeichnen	63
6.2.4.10	rote_spur_zeichnen	63
6.2.4.11	rote_beschleunigung_zeichnen	64
6.2.4.12	roten_spieler_zeichnen	66
6.2.4.13	geschwindigkeiten_zeichnen	67
6.2.4.14	untertitel_darstellen	68
6.2.4.15	alle_videos_pausieren	68
6.2.4.16	videos_darstellen	69
6.2.4.17	game_over	69
6.3	Bilder	70
6.3.1	mitarbeiter.jpg	70
6.3.2	logo.png	70
6.4	Videos und Sound	70
6.5	1.mp4 ... 10.mp4	70
6.5.1	intro.wav	71
6.6	rot_intro.html	71
6.7	rot_videos.html	74

### III Schnitte und Kollisionen 78

#### A Kollision zweier Kreise 79

A.0.1	Beispiel	83
-------	----------	----

#### B Kollision eines Kreises mit einer Geraden 85

B.1	Vektorprojektion	85
B.2	Hessesche Normalform	86
B.2.1	Beispiele	88
B.3	Kollisionsanalyse	90
B.3.1	Beispiel	91

# **Teil I**

## **Bedienungsanleitung**

# 1 Einführung

Spaceballs ist ein Lehrprojekt im Rahmen des Informatikmoduls des Studiengangs ILST<sup>1</sup> der Abteilung Maschinenbau der Hochschule Bremen. Die etwa 50 Studierenden eines Jahrgangs teilen sich dazu in 10 Projektteams auf. Jedes Team programmiert im Laufe des Semesters eine KI (Künstliche Intelligenz) in MATLAB, die die autonome Bewegung ihres „Spaceballs“ steuert. Am Ende des Semesters tritt im Rahmen eines Turniers jeder Spaceball im Zweikampf gegen die Spaceballs der anderen Teams an.<sup>2</sup>

Die sich daraus ergebende Tabelle (Rangliste) fließt zu 50% in die Informatiknote ein; die verbleibenden 50% ergeben sich aus der Dokumentation, dem Design der Webseite und der Qualität und Originalität der Videos und Animationen.

Im Spiel werden – wie im Titelbild dieser Arbeit oder Abbildung 2.2 zu sehen – die beweglichen Spaceballs (rot und blau) genau wie die ortsfesten Tankstellen (grün) und Minen (schwarz) als ausgefüllte Kreise in der Ebene dargestellt. Das Spielfeld wird durch Banden begrenzt. Das Ziel des Spiels ist es, am Ende mehr Treibstoff als der Gegner zu haben. Dazu fliegt der Spaceball möglichst viele Tankstellen an und weicht dabei Minen, Banden und gegebenenfalls dem Gegner aus.

---

<sup>1</sup>Internationaler Studiengang Luftfahrtssystemtechnik und -management B.Eng. [1]

<sup>2</sup>Da das erste Team gegen neun andere Teams spielt, das zweite dann noch gegen acht, ... ergibt sich nach [2] die Gesamtanzahl der Spiele als Summe der Zahlen eins bis neun und damit zu  $\frac{9 \cdot 10}{2} = 45$ .

## 2 Spielablauf

Jeder Spaceball verfügt zur Steuerung über eine Düse, die in jede Richtung Schub erzeugen kann. Da die Masse<sup>1</sup> (und damit der Radius) von der aufgenommenen Treibstoffmenge bestimmt wird, der Schub aber konstant ist, hängt die Beschleunigung eines Spaceballs von seiner Größe ab. Kleinere Spaceballs lassen sich daher schneller beschleunigen, abbremsen und manövrieren; größere Spaceballs sind entsprechend schwerfälliger zu lenken.

Das Spiel endet, sobald eine der drei Bedingungen eintritt:

- Das Ende der Spielzeit (120 Sekunden) ist erreicht.
- Beide Spaceballs treffen aufeinander.
- Ein Spaceball berührt eine Mine oder eine Bande.

Nach Ablauf der Spielzeit oder wenn beide Spaceballs einander berühren, gewinnt der Spaceball mit dem meisten Treibstoff. Wenn ein Spaceball eine Mine oder eine Bande berührt, gewinnt sein Gegner. Ein Sieg am Ende der Spielzeit ergibt **einen** zusätzlichen Punkt in der Tabelle, die anderen Siege jeweils **zwei**. Da während der Schuberzeugung natürlich Treibstoff verbraucht wird und dieser außerdem mit der Zeit „verdunstet“, erhöht die geringere Punkteanzahl am berührungslosen Spielende die Wahrscheinlichkeit spannender Endphasen. Träges Herumlungern und Campen zahlt sich also nicht aus.

### 2.1 Tankstellen, Minen und Banden

Solange ein Spaceball eine Tankstelle berührt, fließt Treibstoff aus der Tankstelle in den Spaceball. Die Tankstelle wird dabei kleiner, der Spaceball wächst.

In Abbildung 2.1 ist zu sehen, dass der rote Spaceball zu Beginn des Tankens noch sehr viel kleiner als die Tankstelle ist, dass während des Tankens der Spaceball an Umfang zulegt und gleichzeitig die Tankstelle schrumpft und dass am Ende des Tankvorgangs die Tankstelle nicht mehr vorhanden und aller Treibstoff im Spaceball gespeichert ist.

---

<sup>1</sup>Natürlich haben nur dreidimensionale Körper eine Masse. Stellen Sie sich den Spaceball einfach als Zylinder mit infinitesimal kleiner Höhe vor. Seine Masse ist dann quadratisch vom Radius abhängig.

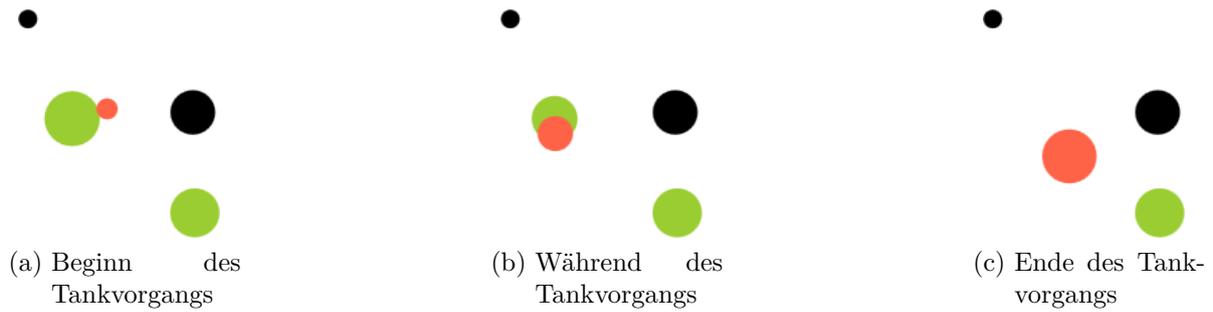


Abbildung 2.1: Ablauf eines Tankvorgangs

Wenn ein Spaceball eine Mine oder eine Bande berührt, endet das Spiel sofort mit dem Sieg des Gegners.

Wie im Titelbild oder in Abbildung 2.2 zu sehen ist, werden die Tankstellen und Minen vor Beginn jedes Spiels zufällig aber symmetrisch (links  $\leftrightarrow$  rechts) verteilt. Auf diese Weise ist jedes Spiel anders und trotzdem haben beide Spaceballs – die jeweils aus den oberen Ecken (rot links, blau rechts) starten – anfänglich immer die gleichen Chancen.

## 2.2 Anzeigen

Standardmäßig sind auf dem Bildschirm nur die sich zwischen den Tankstellen und Minen bewegendem Spaceballs selbst zu sehen. Während der Entwicklung der KI können durch Anklicken der entsprechenden Auswahlfelder unten in Abbildung 2.2 weitere Informationen angezeigt werden.

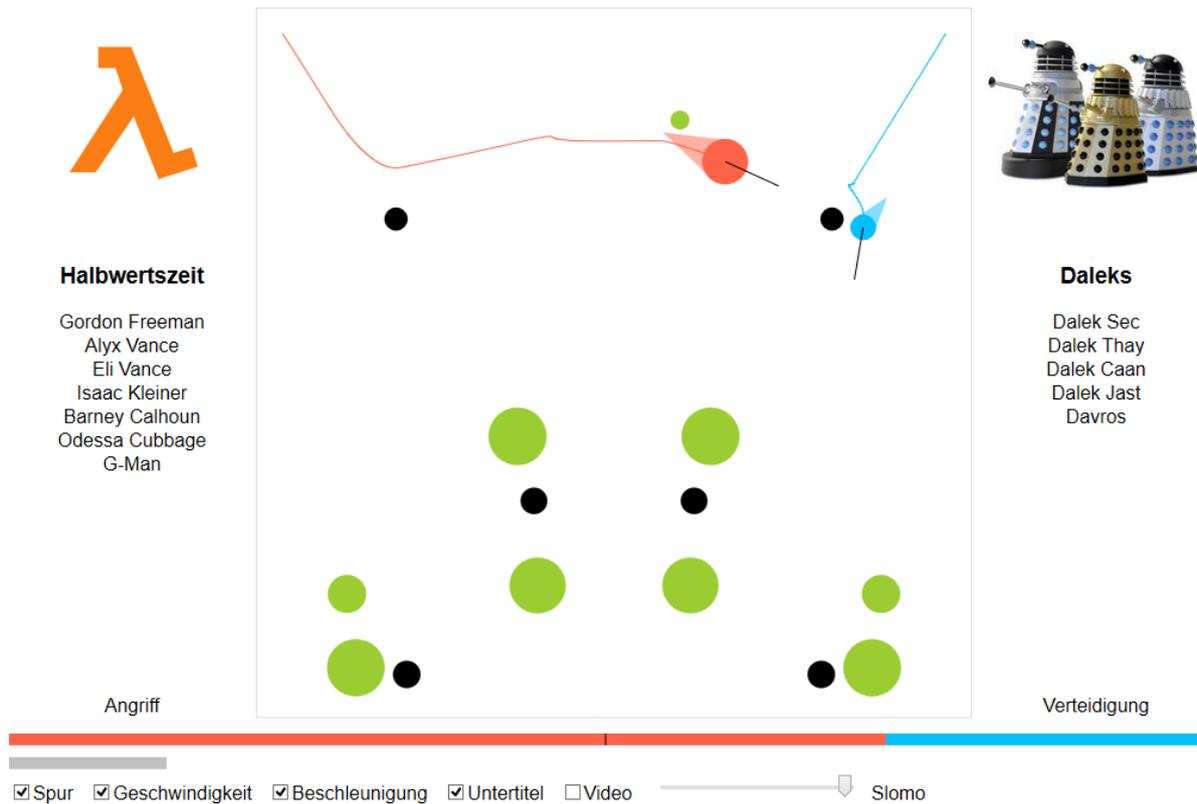


Abbildung 2.2: Mit Spur, Geschwindigkeit, Beschleunigung, Untertitel und Zeitlupe

Es werden dann beispielsweise die Spuren der bisher zurückgelegten Wege (rote und blaue Linien), die Geschwindigkeitsvektoren (Länge und Richtung der schwarzen Linien), die Schub- bzw. Beschleunigungsvektoren (rote und blaue Dreiecke<sup>2</sup>) und die Untertitel der Videos (hier: „Angriff“ und „Verteidigung“) dargestellt. In der abgebildeten Situation besitzt der rote Spaceball signifikant mehr Treibstoff als sein blauer Gegner, was auch an der größeren Länge des roten Treibstoffbalkens unten in Abbildung 2.2 sichtbar wird. Er verfolgt daher seinen Gegner, indem er seinen Beschleunigungsvektor auf den Gegner richtet, während der unterlegene blaue Spaceball versucht, senkrecht dazu nach rechts zu entkommen.<sup>3</sup>

**Achtung** Mit fortlaufender Animationszeit enthält die in jedem Schritt zu zeichnende Spur immer mehr (bis zu 7200) einzelne Geradenstücke. Wenn unser Rechner also nicht gerade der absolute HighTecFrontEndSuperGraphikGamerBolide ist, könnte dies den Browser durchaus ein wenig ausbremsen. Wir können dann einfach (testweise) die Spur ausschalten, um wieder in Echtzeit zu animieren.

<sup>2</sup>Die sichtbare Spitze des Dreiecks symbolisiert die Richtung eines Triebwerksstrahls. Der Schub(kraft)- und Beschleunigungsvektor hat dann natürlich die entgegengesetzte Orientierung.

<sup>3</sup>Da sich der Kurs der Spaceballs während der letzten Zeitschritte durch annähernd konstanten Schub stabilisiert hat, zeigen in dieser Momentaufnahme auch die Geschwindigkeitsvektoren schon etwa in die jeweiligen Beschleunigungsrichtungen.

Üblicherweise läuft die Simulation flüssig mit 60 Bildern pro Sekunde (60 fps); der Zeitlupenschieberegler (Slomo) steht dann ganz links. In Abbildung 2.2 befindet er sich ganz rechts; es wird dann ein Bild pro Sekunde angezeigt. Dies kann während der KI-Entwicklung sehr sinnvoll sein, um die Reaktionen der Spaceballs in jedem einzelnen Zeitschritt ganz genau analysieren zu können. Der Schieberegler lässt sich mit einer Auflösung von 100 ms verschieben.

Jedes Team erzeugt 6 bis 10 Video-Animationen der Größe  $800 \times 600$  Pixel mit einzeiligen Untertiteln, die ereignisgesteuert links und rechts vom Spielfeld angezeigt werden. Dabei sind die ersten sechs Videos für folgende Ereignisse reserviert:

1. Spaceball trifft auf eine Mine.
2. Spaceball trifft auf eine Bande.
3. Spaceball hat keinen Treibstoff mehr.
4. Spaceball gewinnt.
5. Spaceball verliert.
6. Spaceball tankt.

Da die ersten fünf Ereignisse spielentscheidend sind, können sie nur vom System geworfen werden. Die Ereignisse 6-10 können zu jedem Zeitpunkt von der KI ausgelöst werden. Allerdings gilt auch hier sicherlich: „Weniger ist mehr.“ Auch die Videountertitel können während der KI-Entwicklung sehr gut zur Analyse des Spielgeschehens verwendet werden.

# 3 Regeln

- Jeder Mitarbeiter führt sein persönliches Projekttagebuch unter [3]. Tragen Sie bitte **jede Woche** ausführlich ein, was Sie in der Woche zum Projekt beigetragen haben. Einträge lassen sich nicht nachträglich korrigieren. Am Ende des Semesters muss für jede Veranstaltungswoche mindestens ein Eintrag vorhanden sein.
- Jeder Mitarbeiter stellt an mindestens **fünf** Terminen seinen Projektstand in einer 15-minütigen Minipräsentation<sup>1</sup> dem Dozenten vor. Dazu tragen Sie dann bitte jeweils einen Terminvorschlag während der regulären Informatikveranstaltungszeiten unter <http://prof.red/contact/calendar.htm> ein. Ihr Terminvorschlag sollte dabei möglichst direkt im Anschluss vor oder nach einem anderen Termin liegen. Zusätzlich zu den Pflichtterminen können Sie natürlich jederzeit Hilfe- und Diskussionstermine vorschlagen. Bitte verwenden Sie als Titel des Terminvorschlags einfach nur Ihren Namen.
- Jedes Team erstellt
  - die MATLAB-Datei `kraft`, in der die KI die momentane Schubkraft(richtung) definiert
  - die XML-Datei `team.xml`, in der der Name des Teams und die Namen und Aufgaben aller Mitarbeiter definiert sind
  - für jeden Mitarbeiter eine JPG-Datei `mitarbeiter.jpg` mit einem Portrait (Kopfbild), auf dem der Mitarbeiter deutlich zu erkennen ist
  - die WAV-Datei `intro.wav` für die Vorstellung des Teams
  - die PNG-Datei `logo.png` als Teamlogo
  - sechs bis zehn Videos `1.mp4 ... 10.mp4` für besondere Ereignisse (Tanken, Gewonnen, Verloren, ...)
  - eine ausführliche, in  $\LaTeX$  (beispielsweise mit [4]) geschriebene Dokumentation des Projektes
  - eine Website, (beispielsweise bei [5]<sup>2</sup>), auf der das Projekt in allen Details beschrieben ist
- Bleiben Sie trotz der Wettbewerbssituation bitte fair und hilfsbereit den anderen Teams gegenüber.

---

<sup>1</sup>Das eigene Team darf dabei natürlich gerne anwesend sein.

<sup>2</sup>Trotz des für den deutschen Markt ziemlich schlampig recherchierten Namens einer der leistungsfähigsten Anbieter kostenloser Websites.

- Abgabetermin der für das Turnier benötigten Dateien (s. o.) ist am 20.1.2015. Das Turnier findet dann am 27.1.2015 statt. Die Dokumentation und die Website<sup>3</sup> müssen bis zum 30.1.2015 fertig sein.
- Die Teamnote setzt sich schließlich folgendermaßen zusammen
  - 50 % Turnierrangliste
  - 25 % Animationen, Portraits, Intro, Logo
  - 15 % Dokumentation
  - 10 % Website
- Verwenden Sie nur numerische und keine symbolischen Berechnungen, da das tausendfache Aufrufen der *symbolic engine* in `kraft` die Rechenzeit sehr stark vergrößert. Lösen Sie gegebenenfalls Gleichungen einmalig außerhalb von `Spaceballs` und benutzen Sie den Ergebnisterm in `kraft`. Die Verwendung symbolischer Ausdrücke wie `syms`, `solve`, `subs`, ... wird als kapitaler Fehler gewertet.

---

<sup>3</sup>Vielleicht möchten Sie auf der Website ja noch die Turnierspiele Ihrer KI einbinden.

# **Teil II**

## **Unter der Haube**

## 4 Blockschaltbild

Abbildung 4.1 zeigt das komplette Spaceball-Blockschaltbild.

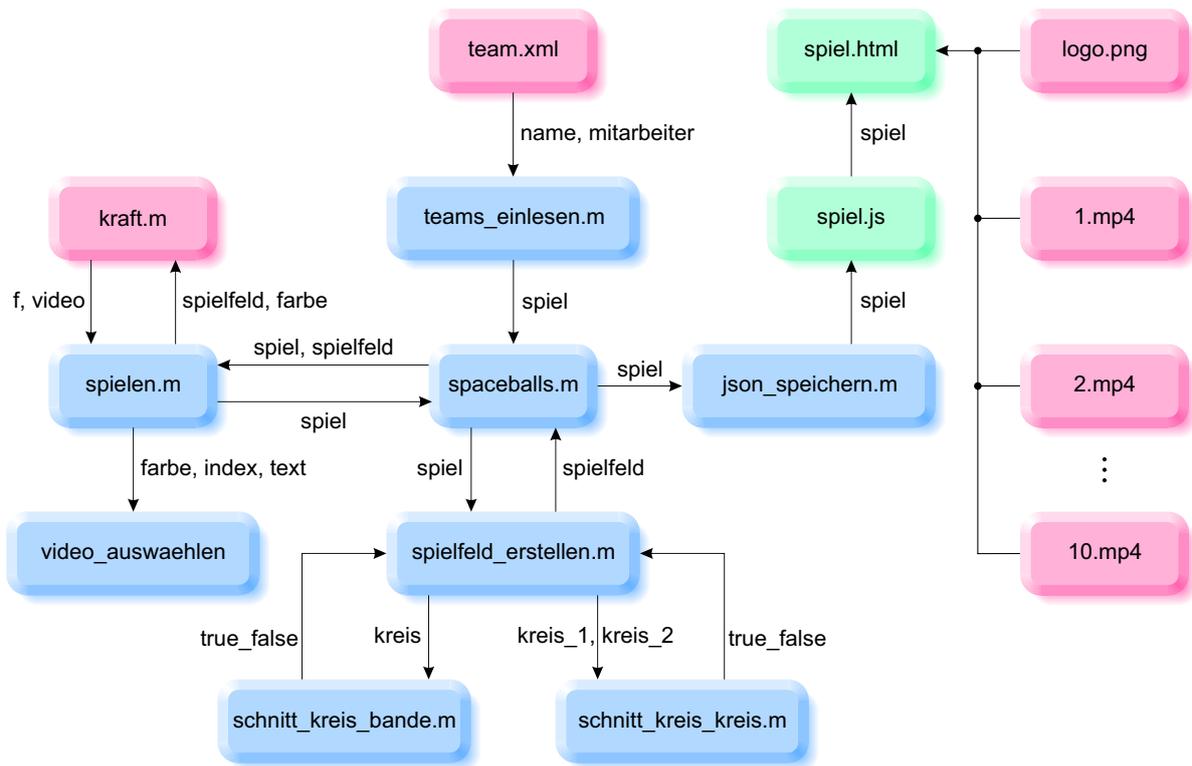


Abbildung 4.1: Blockschaltbild

Darin sind die MATLAB-Systemdateien in Blau, die von jedem Team zu erstellenden Dateien in Rot und die für die Visualisierung zuständigen HTML- und JavaScript-Dateien in Grün dargestellt.

Das zentral angeordnete `spaceballs`<sup>1</sup> ist das Hauptprogramm und kommuniziert mit den anderen blau dargestellten Unterprogrammen. Als erstes wird das Unterprogramm `teams_einlesen` aufgerufen, das die von beiden Teams jeweils erstellte Datei `team.xml` liest, die Informationen über die Teamnamen und Mitarbeiter in die Struktur `spiel` eintütet und diese wieder zurück an `spaceballs` übergibt.

<sup>1</sup>Die Endung `.m` der MATLAB-Dateien lassen wir hier und im Folgenden weg.

`spaceballs` übergibt `spiel` dann an das Unterprogramm `spielfeld_erstellen`, das die Tankstellen, Minen und Spaceballs zufällig, aber symmetrisch und ohne Überlappung, auf dem Spielfeld anordnet und dieses als die Struktur `spielfeld` an `spaceballs` zurückgibt. Um die Überlappungsfreiheit zu gewährleisten, ruft `spielfeld_erstellen` die beiden Unterprogramme `schnitt_kreis_bande` und `schnitt_kreis_kreis` auf, übergibt ihnen einen bzw. zwei Kreise als die Strukturen `kreis` bzw. `kreis_1` und `kreis_2` und erhält in Form der booleschen Variablen `true_false` die Information zurück, ob der Kreis die Bande, bzw. den anderen Kreis schneidet.

Als nächstes ruft `spaceballs` das Unterprogramm `spielen` auf, in dem die eigentliche Simulation des Spielablaufs stattfindet und übergibt die zuvor initialisierten Strukturen `spiel` und `spielfeld`.

`spielen` ruft dann in jedem Simulationsschritt das von beiden Teams jeweils erstellte Unterprogramm `kraft` auf, übergibt diesem den aktuellen Zustand des Spielfelds (`spielfeld`) und die Farbe (`farbe`) des Spaceballs des jeweiligen Teams.

In `kraft` haben die Teams jeweils ihre KI programmiert, die in jedem Zeitschritt aus dem momentanen Zustand des Spielgeschehens den gewünschten Schubvektor  $\mathbf{f}$  des eigenen Spaceballs berechnet und zurück gibt. Außerdem kann `kraft` eines der frei wählbaren Videos inklusive Untertitel in Form der Struktur `video` anfordern.

Im eingebetteten<sup>2</sup> Unterprogramm `video_auswaehlen` werden die angeforderten Videos auf ihre Gültigkeit hin überprüft und gegebenenfalls in der Struktur `spiel` abgespeichert.

`spielen` erkennt, wenn das Spiel beendet ist – entweder, wenn ein Spaceball eine Mine, Bande oder Gegner berührt hat, oder aber wenn die maximale Spielzeit abgelaufen ist – und übergibt dann das mit den kompletten Simulationsdaten gefüllte `spiel` zurück an das Hauptprogramm.

Dieses übergibt `spiel` schließlich an das Unterprogramm `json_speichern`, das die Simulationsdaten im JSON-Datenformat (JavaScript Object Notation) in die Datei `spiel.js` abspeichert.

Die Darstellung des Spielverlaufs geschieht dann sequenziell auf der Seite `spiel.html` im Browser. `spiel.html` liest dazu die abgespeicherten Spielverlaufsdaten aus `spiel.js` und die von den Teams erstellten Logo- und Videodateien `logo.png` und `1.mp4 ... 10.mp4` und verwendet HTML5, CSS3 und JavaScript, um den kompletten Spielverlauf in einer browserkontrollierten Echtzeitschleife zu animieren.

---

<sup>2</sup>Eingebettete Unterprogramme sind keine eigenständigen `m`-Dateien, weshalb der Block in Abbildung 4.1 keine Endung `.m` besitzt. Sie können auf alle Variablen des aufrufenden Programms zugreifen.

# 5 Simulation in MATLAB

## 5.1 spaceballs

spaceballs ist das MATLAB-Hauptprogramm, das den Ablauf der Simulation steuert und dazu seine Unterprogramme aufruft. Als erstes setzen wir allerdings alle Variablen zurück

```
clear variables
```

schließen etwaig geöffnete Fenster (obwohl das Programm selbst eigentlich keine Fenster öffnen sollte)

```
close all
```

und löschen das Kommandozeilenfenster:

```
clc
```

Wie in Abschnitt 4 beschrieben, übergeben wir dann die Kontrolle an das Unterprogramm `teams_einlesen`, das die Namen der Teams und Mitarbeiter einliest und in der Struktur `spiel` zurückgibt:

```
spiel = teams_einlesen;
```

In den nächsten Zeilen definieren wir weitere Konstanten in der Struktur `spiel`<sup>1</sup>:

```
spiel.dt = 1/60;  
spiel.t_end = 120;  
spiel.n_t = round (spiel.t_end/spiel.dt);  
  
spiel.n_mine = 6;  
spiel.n_tanke = 10;  
  
spiel.kreis_radius_min = 0.005;  
spiel.kreis_radius_max = 0.05;  
  
spiel.spieler_radius = 0.025;
```

---

<sup>1</sup>Erklärungen der einzelnen Größen unter `spiel`.

Wir übergeben dem Unterprogramm `spielfeld_erstellen` die Struktur `spiel` mit den bislang definierten Konstanten und erhalten von ihm die neue Struktur `spielfeld` zurück, die die anfänglichen Größen, Positionen, Geschwindigkeiten und Beschleunigungen der Spaceballs, Tankstellen und Minen enthält:

```
spielfeld = spielfeld_erstellen (spiel);
```

Nachdem das Spielfeld erstellt wurde, übergeben wir es zusammen mit den bisherigen Spieldaten an das Unterprogramm `spielen`, das die eigentliche Simulation durchführt und am Ende die gesamten Simulationsdaten (Positionen, Geschwindigkeiten, ...) zurück liefert:

```
spiel = spielen (spiel, spielfeld);
```

Die Simulation selbst ist jetzt abgeschlossen und wir speichern im Unterprogramm `json_speichern` alles, was wir für die Visualisierung brauchen, im JSON-Format in einer Datei ab:

```
json_speichern (spiel)
```

Abschließend rufen wir den Systembrowser<sup>2</sup> auf und weisen ihn an, die HTML-Datei `spiel.html` darzustellen:

```
web ('spiel.html', '-browser')
```

## 5.2 teams\_einlesen

Das Unterprogramm `teams_einlesen` besitzt keine Eingangsparameter und gibt die Struktur `spiel` zurück:

```
function spiel = teams_einlesen
```

Als erstes lesen wir die XML-Datei `team.xml` des roten Teams ein und speichern das zugehörige DOM (Document Object Model, [6]) im Objekt `team` ab:

```
team = xmlread ('teams/rot/team.xml');
```

Wir können nun mit den entsprechenden DOM-Methoden den Namen des roten Teams auslesen und im passenden Feld von `spiel` abspeichern. Dazu verwenden wir die Methode `getElementsByTagName`, um eine Liste aller Knoten mit dem Namen `team_name` zu erhalten. Der direkte Vergleich mit `team.xml` macht uns deutlich, dass es natürlich nur einen Knoten mit dem Namen `team_name` gibt. Die Liste enthält damit nur ein Element, auf das wir mit der Methode `item(0)` zugreifen können. Wir müssen dabei beachten, dass wir es hier innerhalb von MATLAB ärgerlicherweise mit nullbasierter

---

<sup>2</sup>Momentan funktioniert Firefox für die Spaceballsdarstellung besser als Chrome. Die übrigen Verdächtigen (IE, Opera, Safari) haben aktuell immer noch teilweise große Probleme, HTML5 vernünftig darzustellen. Auch auf Mobilgeräten sieht es noch ähnlich schlecht aus.

Java-Syntax zu tun haben; Das erste Element der Liste hat also den Index null. Mit der Methode `getTextContent` sprechen wir dann schließlich den Textinhalt des Knotens an. In `team.xml` ist dies natürlich der Text `Halbwertszeit`. Allerdings liefert uns `getTextContent` auch hier dummerweise nur eine Java-Zeichenkette, die wir erst noch mit `char` in eine MATLAB-Zeichenkette umwandeln müssen.

```
spiel.rot.name = char (team. ...
    getElementsByTagName ('team_name'). ...
    item(0). ...
    getTextContent);
```

Da es in `team.xml` mehrere `mitarbeiter` gibt, liefert uns der folgende Befehl eine Liste aller Mitarbeiter:

```
alle_mitarbeiter = team. ...
    getElementsByTagName ('mitarbeiter');
```

In einer Schleife über alle Mitarbeiter (die Methode `getLength` ermittelt die Anzahl der Elemente der Liste)

```
for i_mitarbeiter = 1 : alle_mitarbeiter.getLength
```

extrahieren wir als erstes den aktuellen Mitarbeiter, indem wir ihn über seinen Index ansprechen:

```
mitarbeiter = alle_mitarbeiter.item(i_mitarbeiter - 1);
```

Ein Mitarbeiter hat sowohl einen Namen als auch eine Aufgabe, die wir – wie schon beim Teamnamen – über die Methoden `getElementsByTagName`, `item` und `getTextContent` ansprechen und in das entsprechende Feld von `spiel` abspeichern. Als erstes ermitteln wir den Namen des Mitarbeiters:

```
spiel.rot.mitarbeiter(i_mitarbeiter).name = ...
    char (mitarbeiter. ...
    getElementsByTagName ('name'). ...
    item(0). ...
    getTextContent);
```

Dabei müssen wir wieder sorgfältig auf die null- bzw. einsbasierte Indizierung achten. In der gleichen Schleife – die ja mit `i_mitarbeiter = 1` beginnt – benutzen wir in der nullbasierten Java-Methode `item` ein `i_mitarbeiter - 1` als Index, damit als erstes das nullte Element angesprochen wird, während wir beim Abspeichern in das einsbasierte MATLAB-Feld direkt `i_mitarbeiter` verwenden. Anschließend setzen wir die gleichen Methoden für das Auslesen und Abspeichern der Aufgabe des aktuellen Mitarbeiters ein:

```
spiel.rot.mitarbeiter(i_mitarbeiter).aufgabe = ...
    char (mitarbeiter. ...
    getElementsByTagName ('aufgabe'). ...
    item(0). ...
    getTextContent);
```

```
end
```

Anschließend führen wir die entsprechenden Befehle zum Ermitteln der Daten des blauen Teams durch, indem wir in jeder Anweisung `rot` durch `blau` ersetzen:<sup>3</sup>

```
team = xmlread ('teams/blau/team.xml');
:
:
```

## 5.3 team.xml

XML-Dateien [7] eignen sich sehr gut, um hierarchisch strukturierte Informationen zu speichern. In der hier erstellten `team.xml` finden wir sowohl den Namen des gesamten Teams im Element `<team_name>` als auch die Klarnamen (keine Pseudonyme) und Aufgaben der einzelnen Teammitglieder in den Elementen `<name>` und `<aufgabe>` unterhalb des `<mitarbeiter>`-Knotens:

```
<?xml version="1.0" encoding="utf-8"?>
<team>
  <team_name>Halbwertszeit</team_name>
  <mitarbeiter>
    <name>Gordon Freeman</name>
    <aufgabe>Angriffsprogrammierung</aufgabe>
  </mitarbeiter>
  <mitarbeiter>
    <name>Alyx Vance</name>
    <aufgabe>Verteidigungsprogrammierung</aufgabe>
  </mitarbeiter>
  <mitarbeiter>
    <name>Eli Vance</name>
    <aufgabe>Versorgungsprogrammierung</aufgabe>
  </mitarbeiter>
  <mitarbeiter>
    <name>Isaac Kleiner</name>
    <aufgabe>Webseitenprogrammierung</aufgabe>
  </mitarbeiter>
  <mitarbeiter>
```

<sup>3</sup>Häufig müssen wir abwägen, ob die hier verwendete einfache Wiederholung und leichte Anpassung eines Programmabschnitts mittels Kopieren-und-Einfügen sinnvoller ist als das Schreiben eines eigenen Unterprogrammes, dem wir hier dann die Farbe `rot` oder `blau` über die Parameterliste übergeben würden. Hier kommt als Argument gegen ein Unterprogramm hinzu, dass die Farbe sowohl als Zeichenkette im Ordnernamen als auch als Feld in der Struktur `spiel` auftritt, was den Programmieraufwand erhöhen würde.

Als Faustregel könnte gelten, dass ein Unterprogramm angebracht sein mag, wenn wir einen Programmabschnitt mehr als einmal kopieren müssten.

```

    <name>Barney Calhoun</name>
    <aufgabe>Sounds & Videos</aufgabe>
</mitarbeiter>
<mitarbeiter>
    <name>Odessa Cabbage</name>
    <aufgabe>Dokumentation</aufgabe>
</mitarbeiter>
<mitarbeiter>
    <name>G-Man</name>
    <aufgabe>Projektleitung</aufgabe>
</mitarbeiter>
</team>

```

Das Wurzelement `<team>` umschließt den gesamten Baum; sein Name spielt hier keine Rolle.

## 5.4 spielfeld\_erstellen

Im Unterprogramm `spielfeld_erstellen` definieren wir die Anfangspositionen und -größen der Spaceballs, Tankstellen und Minen. Dazu bekommt das Unterprogramm über seine Parameterliste die Struktur `spiel` übergeben, in der es Informationen über die Anzahl und maximale bzw. minimale Größen der Spielobjekte findet. Am Ende gibt es die Struktur `spielfeld` zurück, in der die Spaceballs, Tankstellen und Minen beschrieben sind:

```
function spielfeld = spielfeld_erstellen (spiel)
```

Als allererstes initialisieren wir den Zufallszahlengenerator, der uns die zufälligen Positionen und Größen der Minen und Tankstellen liefert:<sup>4</sup>

```
rng shuffle
```

Die Gesamtzahl aller zu definierenden Kreise ergibt sich aus der Summe aller Minen und Tankstellen und beider Spaceballs. Zusätzlich brauchen wir noch zwei Kreise als Sicherheitszone um die Startpositionen der Spaceballs:

```
n_kreis = 4 + spiel.n_mine + spiel.n_tanke;
```

Da die Tankstellen und Minen gleich innerhalb einer Schleife definiert werden, ist es aus Geschwindigkeitsgründen sehr sinnvoll, den gesamten benötigten Speicherplatz für das `kreis`-Feld schon vorab anzufordern, um dem System die Möglichkeit zu geben, einen

<sup>4</sup>Würden wir den Generator nicht mit der aktuellen Uhrzeit initialisieren, würde jeder Neustart MATLABs wieder zu der gleichen Abfolge von Spielfeldern führen. Mit einer beliebigen Zahl als Argument des Befehls – beispielsweise `rng (42)` – können wir in der Entwicklungsphase immer wieder die gleichen „zufälligen“ Spielfelder erzeugen, um unterschiedliche Algorithmen miteinander zu vergleichen.

ausreichend großen zusammenhängenden Speicherbereich für das Feld zu reservieren und nicht immer wieder umspeichern zu müssen, wenn das Feld in der Schleife wächst. Dazu füllen wir das letzte Feldelement (`kreis(n_kreis)`) einfach mit einem willkürlichen Wert, der später in der Schleife ja dann durch den richtigen Wert überschrieben wird:

```
kreis(n_kreis).pos = [0; 0];
```

Als erstes definieren wir jetzt die Anfangsposition und -größe des roten Spaceballs (`kreis(1)`). Wir platzieren ihn in die linke obere Ecke des Spielfelds mit einem kleinen Sicherheitsabstand von den Banden:

```
kreis(1).pos(1) = 1.5*spiel.spieler_radius;
kreis(1).pos(2) = 1.5*spiel.spieler_radius;
kreis(1).radius = spiel.spieler_radius;
```

Wenn wir seine Positionskordinaten genau auf seinen Radius setzen würden, würde er mit seinem Rand genau die Banden berühren und das Spiel wäre sofort beendet.

Die erste Positionskoordinate (x-Koordinate) verläuft im Spielfeld von links nach rechts. Die zweite Positionskoordinate (y-Koordinate) verläuft – wie in Grafikanwendungen allgemein üblich – von **oben** nach **unten**. Beide Koordinaten haben einen normierten Wertebereich von 0.0 bis 1.0.

Der blaue Spaceball (`kreis(2)`) wird in die rechte obere Ecke gesetzt:

```
kreis(2).pos(1) = 1 - 1.5*spiel.spieler_radius;
kreis(2).pos(2) = 1.5*spiel.spieler_radius;
kreis(2).radius = spiel.spieler_radius;
```

Als nächstes definieren wir zwei Sicherheitszonen (Viertelkreise mit einem Radius von 10 Spaceballradien) um die beiden oberen Ecken, um zu verhindern, dass dort gleich Tankstellen oder Minen platziert werden, die zu ungerechten Anfangssituationen führen könnten:<sup>5</sup>

```
kreis(3).pos(1) = 0;
kreis(3).pos(2) = 0;
kreis(3).radius = 10*spiel.spieler_radius;

kreis(4).pos(1) = 1;
kreis(4).pos(2) = 0;
kreis(4).radius = 10*spiel.spieler_radius;
```

Jetzt erzeugen wir die Tankstellen und Minen in einer Schleife. Dazu initialisieren wir den Kreisindex auf den Wert fünf, da die ersten vier Indizes ja für Spaceballs und Sicherheitszonen reserviert sind und die erste Mine demnach den Index fünf hat:

```
i_kreis = 5;
```

<sup>5</sup>Beispielsweise könnten ungeschickt gesetzte Minen dazu führen, dass ein Spaceball seine Ecke überhaupt nicht verlassen kann. Und zu viele Tankstellen in der eigenen Ecke wären möglicherweise auch ein zu großer Anfangsvorteil.

In einer Schleife über alle restlichen Kreise

```
while i_kreis <= n_kreis
```

bekommt ein Kreis (Mine/Tankstelle) eine zufällige Position auf dem Spielfeld

```
kreis(i_kreis).pos = rand(1, 2);
```

und einen zufälligen Radius, der zwischen den vorher im Hauptprogramm definierten Minimal- bzw. Maximalwerten liegt:

```
kreis(i_kreis).radius = ...
    spiel.kreis_radius_min + ...
    (spiel.kreis_radius_max - spiel.kreis_radius_min)*rand;
```

Außerdem erzeugen wir gleich den dazu symmetrisch liegenden Kreis, indem wir den Kreis in die nächste freie Zelle kopieren

```
kreis(i_kreis + 1) = kreis(i_kreis);
```

und seine  $x$ -Koordinate an der senkrechten Symmetrieachse spiegeln:

```
kreis(i_kreis + 1).pos(1) = 1 - kreis(i_kreis).pos(1);
```

Als nächstes untersuchen wir, ob der gerade definierte Kreis<sup>6</sup> einen schon vorhandenen Kreis schneidet, um einander überlappende Minen oder Tankstellen zu verhindern. Dazu setzen wir ein binäres Flag zurück, das im Überlappingsfall gesetzt wird:

```
kreise_schneiden = false;
```

In einer Schleife über alle schon vorhandenen Kreise

```
for i_vorhandener_kreis = 1 : i_kreis - 1
```

untersuchen wir mit Hilfe des Unterprogramms `schnitt_kreis_kreis`, ob sich der gerade definierte Kreiskandidat mit einem schon vorher definierten Kreis überlappt:

```
if schnitt_kreis_kreis ( ...
    kreis(i_kreis), ...
    kreis(i_vorhandener_kreis))
```

Wenn dies der Fall ist, setzen wir das Überlappingsflag und brechen die Schleife ab, da es uns nicht interessiert, ob der Kandidat noch weitere Kreise schneidet:

```
    kreise_schneiden = true;

    break

end

end
```

<sup>6</sup>Den gespiegelten Kreis müssen wir nicht extra untersuchen, da sein Überlappingsverhalten in einer symmetrischen Welt natürlich dem seines Symmetriepartners entspricht.

Im nächsten Schritt untersuchen wir, ob der Kandidat seinen eigenen Symmetriepartner oder eine der Banden schneidet, was wir auch verhindern möchten. Nur wenn dies nicht der Fall ist und auch keiner der schon vorhandenen Kreise geschnitten wurde

```
if ...
    ~ schnitt_kreis_kreis ( ...
        kreis(i_kreis), ...
        kreis(i_kreis + 1)) && ...
    ~ schnitt_kreis_bande(kreis(i_kreis)) && ...
    ~ kreise_schneiden
```

akzeptieren wir den Kandidaten und seinen Symmetriepartner und erhöhen daher den Kreisindex um zwei:

```
    i_kreis = i_kreis + 2;

end

end
```

Wenn hingegen der Kandidat einen anderen Kreis oder eine Bande schneidet, wird der Index nicht erhöht und der nächste Kandidat überschreibt den aktuellen Kandidaten.

Nachdem alle Kreise definiert sind, speichern wir die Position und den Radius des ersten Kreises als roten Spaceball ab

```
rot = kreis(1);
```

und initialisieren seine Geschwindigkeit und Beschleunigung jeweils auf null:

```
rot.ges = [0 0];
rot.bes = [0 0];
```

Der zweite Kreis ist der blaue Spaceball:

```
blau = kreis(2);
blau.ges = [0 0];
blau.bes = [0 0];
```

Der dritte und vierte Kreis sind keine echten Objekte, sondern dienen ja nur dazu, die oberen Ecken von Tankstellen und Minen frei zu halten. Die Minen beginnen daher mit dem fünften Kreis und werden in einen eigenen Feld von Strukturen abgespeichert:

```
mine(1 : spiel.n_mine) = kreis(5 : spiel.n_mine + 4);
```

Die noch übrigen Kreise sind dann die Tankstellen:

```
tanke(1 : spiel.n_tanke) = kreis(spiel.n_mine + 5 : n_kreis);
```

Während der KI-Entwicklung kann es sehr sinnvoll sein, ein eindeutiges Anfangsszenario statt des zufällig erzeugten Spielfeldes zu definieren. Dazu können wir an dieser

Stelle Spaceballs, Minen und Tankstellen auf feste Werte setzen, um unsere Algorithmen immer wieder unter den gleichen Randbedingungen zu testen. Natürlich müssen wir diese Zeilen später wieder auskommentieren oder löschen.

Schließlich speichern wir die momentan noch lokalen Variablen in der Struktur `spielfeld` ab, die ja dann vom Unterprogramm an das Hauptprogramm zurückgegeben wird:

```
spielfeld.rot = rot;
spielfeld.blau = blau;
spielfeld.mine = mine;
spielfeld.tanke = tanke;
```

## 5.5 schnitt\_kreis\_bande

Das Unterprogramm `schnitt_kreis_bande` untersucht, ob ein über die Parameterliste übergebener Kreis eine Bande schneidet.

```
function true_false = schnitt_kreis_bande (kreis)
```

Dazu testen wir, ob die x- und y-Positionskoordinaten mindestens eine Radiuslänge Abstand von den Banden haben. Da für die vier Banden (links, oben, rechts und unten) jeweils andere Positionskomponente verwendet wird, bzw. der Radius mal addiert und mal subtrahiert wird, fragen wir die vier Fälle einzeln ab:

```
if ...
    kreis.pos(1) - kreis.radius <= 0 || ... % Linke Bande
    kreis.pos(2) - kreis.radius <= 0 || ... % Obere Bande
    kreis.pos(1) + kreis.radius >= 1 || ... % Rechte Bande
    kreis.pos(2) + kreis.radius >= 1      % Untere Bande
```

Wenn eine der Bedingungen erfüllt ist, sorgt das abweisende ODER (`||`) dafür, dass die anderen Bedingungen nicht mehr untersucht werden. In diesem Fall wird die Rückgabewariable auf `true` gesetzt:

```
    true_false = true;
```

Wenn keine Bedingung erfüllt ist, wird `false` zurückgegeben:

```
else

    true_false = false;

end
```

## 5.6 schnitt\_kreis\_kreis

Im Unterprogramm `schnitt_kreis_kreis` untersuchen wir, ob zwei über die Parameterliste übergebene Kreise einander schneiden:

```
function true_false = schnitt_kreis_kreis (kreis_1, kreis_2)
```

Dazu kontrollieren wir nach Abbildung 5.1, ob die Summe der Radien ( $r_1 + r_2$ ) der Kreise größer als der Abstand  $R$  ihrer Mittelpunkte ist.

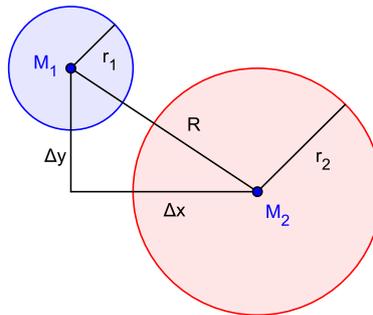


Abbildung 5.1: Schnitt zweier Kreise

Wenn dies der Fall ist

$$r_1 + r_2 \geq R \quad (5.1)$$

schneiden bzw. berühren die Kreise einander.

Den Abstand  $R$  können wir über den Satz des Pythagoras im in Abbildung 5.1 dargestellten Dreieck aus den Quadraten der Differenzen der  $x$ - bzw.  $y$ -Werte der Mittelpunktskoordinaten berechnen:

$$R^2 = (\Delta x)^2 + (\Delta y)^2 = (x_{M_1} - x_{M_2})^2 + (y_{M_1} - y_{M_2})^2 \quad (5.2)$$

Um das rechenintensive Wurzelziehen bei der Berechnung des Abstandes  $R$  aus Gleichung (5.2) zu vermeiden, quadrieren wir Gleichung (5.1), was ihre Aussage nicht verändert und erhalten mit Gleichung (5.2):

$$(r_1 + r_2)^2 \geq (x_{M_1} - x_{M_2})^2 + (y_{M_1} - y_{M_2})^2$$

was wir direkt in eine Abfrage gießen können:

```
if ...
    (kreis_1.radius + kreis_2.radius)^2 >= ...
    (kreis_1.pos(1) - kreis_2.pos(1))^2 + ...
    (kreis_1.pos(2) - kreis_2.pos(2))^2 && ...
    kreis_1.radius > 0 && ...
    kreis_2.radius > 0
```

```
    true_false = true;  
else  
    true_false = false;  
end
```

Die letzten beiden Bedingungen sind eigentlich eher philosophischer Natur („Kann ein Kreis mit einem Radius von null einen anderen Kreis schneiden?“, „Gibt es überhaupt Kreise mit einem Radius von null?“, „Ist das dann ein Schnittpunkt?“, ...). Sie sind aber an dieser Stelle sehr sinnvoll, um beispielsweise zu verhindern, dass aus einer leeren Tankstelle (mit verschwindendem Radius) noch Treibstoff bezogen werden kann.

## 5.7 spielen

Im Hauptprogramm `spielen` führen wir die eigentliche Simulation durch. Bevor wir dazu die Simulationsschleife anstarten, definieren wir ein paar Konstanten. Als erstes beschaffen wir uns die Funktionshandles [8] der Kraft-Unterprogramme beider Teams, um sie später bei der Beschleunigungsberechnung verwenden zu können. Dazu müssen wir leider umständlicher Weise (einmalig) in die entsprechenden Verzeichnisse und wieder zurück wechseln, da wir die beiden Unterprogramme wegen ihres gleichen Namens nicht im durchsuchbaren Pfad finden können:

```
cd teams/rot  
rot_kraft_handle = @kraft;  
cd ../..  
  
cd teams/blau  
blau_kraft_handle = @kraft;  
cd ../..
```

Als nächstes definieren wir die Treibstoffmassenströme, die beim Tanken, bei der Schuberzeugung und durch „Verdunstung“ pro Zeiteinheit fließen:

```
delta_tanken = 4e-6;  
delta_schub = 4e-7;  
delta_verdunstung = 4e-8;
```

Des Weiteren definieren wir den Triebwerksschub, mit dem die Spaceballs beschleunigt (und damit natürlich auch abgebremst) werden können

```
schub = 4e-5;
```

und initialisieren ein Flag, das gesetzt wird, wenn die Simulation vor Ablauf der maximalen Simulationsdauer beendet werden muss, weil eine Minen-, Banden- oder Gegnerberührung stattgefunden hat:

```
game_over = false;
```

Jetzt beginnen wir die Simulationsschleife über die vorher definierte Anzahl von Zeitschritten:

```
for i_t = 1 : spiel.n_t
```

In der Schleife berechnen wir als erstes die aktuelle Zeit und speichern sie in der Struktur `spielfeld` ab, die wir später an die KI-Unterprogramme übergeben:

```
    spielfeld.t = i_t * spiel.dt;
```

Außerdem schreiben wir die aktuelle Zeit zu jeder vollen Simulationssekunde zur Kontrolle<sup>7</sup> in MATLABs Befehlsfenster:

```
    if mod (spielfeld.t, 1) == 0
        clc
        disp (['t = ', num2str(spielfeld.t)])
    end
```

**Massen und Beschleunigungen** Als nächstes berechnen wir die aktuelle Massen und die Beschleunigungen beider Spaceballs. Dazu übergeben wir dem jeweiligen Kraftunterprogramm (angesprochen durch seinen Funktionshandle) die aktuelle `spielfeld`-Struktur und seine Farbe. Die KI berechnet daraus die aktuell sinnvolle Schubkraft und fordert gegebenenfalls ein darzustellendes Video an:

```
[rot_kraft, rot_video] = ...
    rot_kraft_handle (spielfeld, 'rot');

[blau_kraft, blau_video] = ...
    blau_kraft_handle (spielfeld, 'blau');
```

Um sicherzustellen, dass die KI keine reservierten Videos anfordert, führen wir einmalig einen „dummy“-Aufruf des eingebetteten Videoüberprüfungsunterprogrammes durch:

```
video_auswaehlen ('dummy', 0, '')
```

In jedem Zeitschritt verdunstet Treibstoff, bzw. wird für die Lebenserhaltungssysteme verbraucht. Damit lohnt es sich für einen Spaceball nicht, untätig abzuwarten. Wir ziehen dazu in jedem Zeitschritt eine gewisse Menge Treibstoff von der durch das Radiusquadrat repräsentierten Masse ab:

```
rot_masse = spielfeld.rot.radius^2 - delta_verdunstung;
```

<sup>7</sup>Wenn wir – beispielsweise zur Fehlersuche – während des Spiels im Unterprogramm `kraft` weitere Werte ausgeben, sollten wir den Befehl `clc` hier sinnvollerweise entfernen.

Als nächstes überprüfen wir, ob die KI Schub angefordert hat:

```
if norm (rot_kraft) > 0
```

Ist dies der Fall, berechnen wir eine normierte Kraft, die nur die Richtung der Anforderung berücksichtigt

```
rot_kraft = schub * rot_kraft / norm (rot_kraft);
```

und ziehen in diesem Zeitschritt die für den Schub benötigte Treibstoffmenge ab:

```
rot_masse = rot_masse - delta_schub;
```

```
end
```

Wenn dadurch die Treibstoffmenge aufgebraucht ist

```
if rot_masse < 0
```

wählen wir die dazu passenden Videos aus und setzen das Flag, das später zum Abbruch der Simulation führt:

```
video_auswaehlen ('rot', 3, 'Rot hat keinen Sprit mehr.')
```

```
video_auswaehlen ('blau', 4, 'Blau gewinnt.')
```

```
game_over = true;
```

Wenn der Spaceball aber noch genug Treibstoff besitzt

```
else
```

berechnen wir den neuen Radius (als Wurzel der „Masse“)

```
spielfeld.rot.radius = sqrt (rot_masse);
```

und die Beschleunigung als Quotient von Kraft und Masse (Newton:  $F = m \cdot a$ ):

```
spielfeld.rot.bes = rot_kraft / spielfeld.rot.radius^2;
```

```
end
```

Die gleichen Schritte wiederholen wir natürlich für den blauen Spaceball:

```
blau_masse = spielfeld.blau.radius^2 - delta_verdunstung;
```

```
:  
:
```

```
spielfeld.blau.bes = blau_kraft / spielfeld.blau.radius^2;
```

```
end
```

**Eulerschritt** Nachdem wir jetzt die Beschleunigungen dieses Zeitschrittes kennen, können wir die neue Geschwindigkeit mit einem einfachen Eulerschritt [9] erhalten:

```
spielfeld.rot.ges = ...
    spielfeld.rot.ges + spielfeld.rot.bes * spiel.dt;
```

In einem weiteren Eulerschritt berechnen wir aus der Geschwindigkeit die neue Position

```
spielfeld.rot.pos = ...
    spielfeld.rot.pos + spielfeld.rot.ges * spiel.dt;
```

und wiederholen das Ganze für den blauen Spaceball:

```
spielfeld.blau.ges = ...
    spielfeld.blau.ges + spielfeld.blau.bes * spiel.dt;

spielfeld.blau.pos = ...
    spielfeld.blau.pos + spielfeld.blau.ges * spiel.dt;
```

**Spieldatenmatrix** Als nächstes speichern wir die aktuellen Daten (Zeit, Position, Geschwindigkeit, Beschleunigung und Radius) beider Spaceballs in das Matrixfeld der Struktur `spiel` um, die wir später an das Hauptprogramm zurückgeben:

```
spiel.mat(i_t, 1) = spielfeld.t;

spiel.mat(i_t, 2 : 3) = spielfeld.rot.pos;
spiel.mat(i_t, 4 : 5) = spielfeld.rot.ges;
spiel.mat(i_t, 6 : 7) = spielfeld.rot.bes;
spiel.mat(i_t, 8) = spielfeld.rot.radius;

spiel.mat(i_t, 9 : 10) = spielfeld.blau.pos;
spiel.mat(i_t, 11 : 12) = spielfeld.blau.ges;
spiel.mat(i_t, 13 : 14) = spielfeld.blau.bes;
spiel.mat(i_t, 15) = spielfeld.blau.radius;
```

**Tankstellen** In den folgenden Programmabschnitten untersuchen wir, ob die Spaceballs Tankstellen, Minen, Banden oder einander berührt haben. Als erstes finden wir heraus, ob der Spaceball eine Tankstelle schneidet. Dazu starten wir eine Schleife über alle Tankstellen

```
for i_tanke = 1 : spiel.n_tanke
```

und ermitteln, ob der Spaceball die aktuelle Tankstelle schneidet:

```
if schnitt_kreis_kreis ...
    (spielfeld.tanke(i_tanke), spielfeld.rot)
```

Bevor wir in diesem Fall die neuen Radien von Spaceball und Tankstelle berechnen, wollen wir vorher noch eine für den Spielverlauf sinnvolle Anpassung vornehmen: Wenn der Spaceball tankt, vergrößert sich seine Masse. Damit würde auch beispielsweise ein vorher exakt berechneter Bremsweg zunehmen, was fatale Folgen hätte, wenn der Spaceball dadurch nicht mehr rechtzeitig vor einer Mine oder Bande zum Stehen käme. Wir argumentieren daher, dass seine kinetische Energie vor (Index 1) und nach (Index 2) dem Tankvorgang gleich bleiben soll:

$$E_{kin1} = \frac{1}{2}m_1v_1^2 = \frac{1}{2}m_2v_2^2 = E_{kin2} \quad (5.3)$$

Da das Radiusquadrat proportional zur Masse ist, können wir es in Gleichung (5.3) einsetzen

$$\frac{1}{2}r_1^2v_1^2 = \frac{1}{2}r_2^2v_2^2$$

den konstanten Faktor kürzen und auf beiden Seiten die Wurzel ziehen. Wir erhalten dann:

$$r_1v_1 = r_2v_2$$

oder

$$v_2 = \frac{r_1v_1}{r_2} \quad (5.4)$$

Durch Gleichung (5.4) wird die Geschwindigkeit während des Tankens automatisch verringert, so dass der berechnete Bremsweg wieder besser passt. Wir speichern also den Radius des Spaceballs vor dem Tanken zwischen

```
rot_radius_vorher = spielfeld.rot.radius;
```

berechnen seinen neuen Radius, indem wir zu seiner Masse eine konstante Menge Treibstoff addieren:

```
spielfeld.rot.radius = ...
    sqrt (spielfeld.rot.radius^2 + delta_tanken);
```

und passen seine Geschwindigkeit gemäß Gleichung (5.4) an:

```
spielfeld.rot.ges = ...
    rot_radius_vorher * ...
    spielfeld.rot.ges / ...
    spielfeld.rot.radius;
```

Die durch den Tankvorgang geänderten Größen müssen wir natürlich auch in der Spieldatenmatrix anpassen:

```
spiel.mat(i_t, 4 : 5) = spielfeld.rot.ges;
spiel.mat(i_t, 8) = spielfeld.rot.radius;
```

Durch das Tanken nimmt die Masse der Tankstelle um den gleichen Betrag ab, um den der Spaceball zugenommen hat:

```
tanke_masse = ...
    spielfeld.tanke(i_tanke).radius^2 - delta_tanken;
```

Wenn sie dadurch negativ werden würde

```
if tanke_masse < 0
```

setzen wir ihren Radius explizit auf null:

```
    spielfeld.tanke(i_tanke).radius = 0;
```

Solange die Tankstelle aber noch vorhanden ist

```
else
```

berechnen wir ihren neuen Radius als Wurzel ihrer neuen Masse:

```
    spielfeld.tanke(i_tanke).radius = ...
        sqrt (tanke_masse);
```

```
end
```

Schließlich starten wir das „Tankvideo“

```
    video_auswaehlen ('rot', 6, 'Tanke')
```

```
end
```

und wiederholen alles für den blauen Spaceball:

```
if schnitt_kreis_kreis ...
    (spielfeld.tanke(i_tanke), spielfeld.blau)
```

```
:
:
```

```
    video_auswaehlen ('blau', 6, 'Tanke')
```

```
end
```

```
end
```

Auch die Daten der Minen und Tankstellen speichern wir in die Spieldatenmatrix um:<sup>8</sup>

```
for i_mine = 1 : spiel.n_mine
```

```
    spiel.mat(i_t, 13 + 3*i_mine : 14 + 3*i_mine) = ...
        spielfeld.mine(i_mine).pos;
```

<sup>8</sup>In der momentanen Spielversion ist es natürlich eigentlich nicht notwendig, die Minendaten in jedem Simulationsschritt abzuspeichern, da sich ja überhaupt nicht ändern. Aber vielleicht wird es ja in einer späteren Version auch bewegte Minen und Tankstellen geben ...

```

    spiel.mat(i_t, 15 + 3*i_mine) = ...
        spielfeld.mine(i_mine).radius;

end

for i_tanke = 1 : spiel.n_tanke

    spiel.mat(i_t, 13 + 3*(spiel.n_mine + i_tanke) : 14 + 3*(spiel
        .n_mine + i_tanke)) = ...
        spielfeld.tanke(i_tanke).pos;

    spiel.mat(i_t, 15 + 3*(spiel.n_mine + i_tanke)) = ...
        spielfeld.tanke(i_tanke).radius;

end

```

**Minen** Im folgenden Programmabschnitt wollen wir – analog zur vorherigen Tankstellenberührung – ermitteln, ob ein Spaceball eine Mine berührt hat. Dazu starten wir eine Schleife über alle Minen

```
for i_mine = 1 : spiel.n_mine
```

und untersuchen einfach wieder, ob sich die Kreise von aktueller Mine und Spaceball überschneiden:

```
if schnitt_kreis_kreis ...
    (spielfeld.mine(i_mine), spielfeld.rot)
```

Wenn dies der Fall ist, wählen wir die passenden Videos aus und beenden das Spiel:

```

    video_auswaehlen ('rot', 1, 'Rot trifft eine Mine. ')

    video_auswaehlen ('blau', 4, 'Blau gewinnt. ')

    game_over = true;

end

```

Die Untersuchung, ob der blaue Spaceball eine Mine trifft, sieht natürlich genauso aus:

```

if schnitt_kreis_kreis ...
    (spielfeld.mine(i_mine), spielfeld.blau)

    video_auswaehlen ('blau', 1, 'Blau trifft eine Mine. ')

    video_auswaehlen ('rot', 4, 'Rot gewinnt. ')

```

```

    game_over = true;

end

end

```

**Banden** Für die folgende Bandenberührungsanalyse müssen wir noch nicht einmal eine Schleife starten, da alle vier Banden schon im Unterprogramm `schnitt_kreis_bande` abgearbeitet werden:

```

if schnitt_kreis_bande (spielfeld.rot)

    video_auswaehlen ('rot', 2, 'Rot trifft eine Bande. ')

    video_auswaehlen ('blau', 4, 'Blau gewinnt. ')

    game_over = true;

end

if schnitt_kreis_bande (spielfeld.blau)

    video_auswaehlen ('blau', 2, 'Blau trifft eine Bande. ')

    video_auswaehlen ('rot', 4, 'Rot gewinnt. ')

    game_over = true;

end

```

**Gegner und Spielzeitende** Jetzt gibt es noch zwei Fälle, die zu einem Spielende führen: wenn sich beide Spaceballs berühren oder wenn die Spielzeit abgelaufen ist:

```

if ...
    schnitt_kreis_kreis (spielfeld.rot, spielfeld.blau) || ...
    i_t == spiel.n_t

```

In beiden Fällen gewinnt der Spaceball mit dem meisten Treibstoff. Wenn also der rote Spaceball mehr Treibstoff besitzt

```

if spielfeld.rot.radius > spielfeld.blau.radius

```

wählen wir genauso die passenden Videos aus

```

    video_auswaehlen ('rot', 4, 'Rot gewinnt. ')
    video_auswaehlen ('blau', 5, 'Blau hat weniger Sprit. ')

```

wie im umgekehrten Fall:

```

else

    video_auswaehlen ('rot', 5, 'Rot hat weniger Sprit. ')
    video_auswaehlen ('blau', 4, 'Blau gewinnt. ')

end

```

In beiden Fällen wird das Spiel beendet:

```

game_over = true;

end

```

**Game over** Wenn eine der vorherigen Abfragen (Treibstoffmangel, Minen-, Banden- oder Gegnerberührung) das Spielende-Flag gesetzt hat

```

if game_over

```

speichern wir den aktuellen Simulationsschrittindex als Gesamtsimulationsschrittindex und brechen die Simulation ab:

```

    spiel.n_t = i_t;

    break

end

end

```

### 5.7.1 video\_auswaehlen

`video_auswaehlen` ist ein in `spielen` eingebettetes Unterprogramm, das also kein eigene `m`-Datei besitzt, daher nur innerhalb von `spielen` angesprochen werden kann und auf alle Variablen von `spielen` zugreifen kann:

```

function video_auswaehlen (farbe, index, text)

```

Seine Aufgabe ist es, sicherzustellen, dass die KI keine unerlaubten Videos auswählt, da diese während der Visualisierung für den Spielentscheid herangezogen werden.

Wir haben drei Fälle zu unterscheiden: Wenn das Unterprogramm ganz am Anfang eines Simulationsschrittes (Abschnitt 5.7) mit dem Farbparameter `dummy` aufgerufen wird

```

if strcmp (farbe, 'dummy')

```

untersuchen wir, ob die KI gar kein Video oder – natürlich nur aus Versehen – ein Video mit einem reservierten Index ausgewählt hat:<sup>9</sup>

```
if isempty (rot_video.index ) || rot_video.index < 6
```

Wenn dies der Fall ist, setzen wir den Index explizit auf null:

```
    rot_video.index = 0;
end
```

Natürlich wiederholen wir die Untersuchung für den blauen Spaceball:

```
if isempty (blau_video.index ) || blau_video.index < 6

    blau_video.index = 0;

end
```

Wenn vom aufrufenden Programm (spielen) als Farbe nicht `dummy`, sondern `rot` ausgewählt wurde

```
elseif strcmp (farbe , 'rot')
```

sorgen wir dafür, dass sich die Indizes 1 - 5 auf jeden Fall durchsetzen und der Index 6 („Spaceball tankt“) nur, wenn die KI kein eigenes Video angefordert hat:

```
if rot_video.index == 0 || index < 6

    rot_video.index = index;

end
```

Auch die Untertitel werden nur für die Indizes 1 - 5 festgeklopft oder für den Index 6, wenn die KI keinen eigenen Untertitel vorgibt:

```
if strcmp (rot_video.text , '') || index < 6

    rot_video.text = text;

end
```

Natürlich wenden wir die gleichen Regeln auch auf den blauen Spaceball an:

```
elseif strcmp (farbe , 'blau')

    if blau_video.index == 0 || index < 6

        blau_video.index = index;

    end

end
```

<sup>9</sup>Laut Abschnitt 2.2 sind die Indizes 1 - 5 für „Gewinn und Verlust“ reserviert.

```

    end

    if strcmp (blau_video.text, '') || index < 6

        blau_video.text = text;

    end

end

```

Nachdem wir alle problematischen Videoanforderungen abgefangen haben, können wir die Videodaten gefahrlos in die Spieldatenmatrix speichern:

```

    spiel.rot.video{i_t} = rot_video;
    spiel.blau.video{i_t} = blau_video;

end

```

Es folgt jetzt noch der bei einem eingebetteten Unterprogramm notwendige Abschluss des aufrufenden Rahmenprogramms (spielen):

```

end

```

## 5.8 kraft

Im `kraft`-Unterprogramm programmiert jedes Team ihre KI. Diese bekommt über die Parameterliste den aktuellen Zustand des Spiels in Form der Struktur `spielfeld` und die Farbe des eigenen Spaceballs und berechnet daraus die gewünschte Kraft(richtung) `f` und gegebenenfalls das anzuzeigende Video:

```

function [f, video] = kraft (spielfeld, farbe)

```

Da im Turnier die einzelnen Teams zufällig einer Spaceballfarbe zugeordnet werden, ist es im KI-Unterprogramm notwendig, die über die Parameterliste übergebene Farbe auszuwerten, um herauszufinden, welche Spieldaten die eigenen und welche die des Gegners sind. Es wird dort also ein Programmabschnitt wie der folgende sinnvoll sein:

```

% Wenn meine Farbe rot ist,
if strcmp (farbe, 'rot')

    % dann bin ich der rote Spieler
    ich = spielfeld.rot;

    % und mein Gegner ist blau.
    gegner = spielfeld.blau;

```

```

% Wenn meine Farbe nicht rot ist,
else

    % dann bin ich der blaue Spieler
    ich = spielfeld.blau;

    % und mein Gegner ist rot
    gegner = spielfeld.rot;

end

```

Das `kraft` aufrufende Unterprogramm `spielen` erwartet sowohl für die Kraft `f` als auch für die Videostruktur einen gültigen Inhalt zurück geliefert zu bekommen. Um dies sicherzustellen, empfiehlt es sich, ganz oben im `kraft`-Unterprogramm etwas wie

```

f = [0 0];

video.index = 0;
video.text = '';

```

zu spendieren und die Größen dann im Verlauf des Programms gegebenenfalls mit neuen Werten zu überschreiben.

Um den Aufwand der KI überschaubar und für alle Teams gleich zu halten, darf die KI nur den aktuellen Spielzustand und keinerlei Informationen über den vergangenen Spielverlauf verwenden. Im `kraft`-Unterprogramm dürfen also definitiv keine statischen (persistenten oder globalen) Variablen oder Speicherzugriffe (Dateisystem, Web, ...) verwendet werden.<sup>10</sup>

Keine statischen Variablen oder Speicherzugriffe verwenden!

## 5.9 json\_speichern

Im Unterprogramm `json_speichern` speichern wir die Spieldaten in eine JSON-Datei (JavaScript Object Notation, [10]), um sie später bei der Visualisierung wieder verwenden zu können. Um uns die Arbeit zu erleichtern, verwenden wir dazu eine Bibliothek [11], die Qianqian Fang freundlicherweise auf dem MATLAB Central Server zur Verfügung gestellt hat.

Dazu übergeben wir dem Unterprogramm die `spiel`-Struktur

```

function json_speichern (spiel)

```

<sup>10</sup>Die Verwendung statischer Variablen oder von Speicherzugriffen wird als Täuschungsversuch gewertet.

und füllen in der bibliothekseigenen Optionsstruktur die beiden Felder `JSONP` und `FileName` aus. Das erste Feld erlaubt es uns, um das eigentliche JSON-Objekt herum noch eine Variablendeklaration zu basteln, so dass wir die Datei später unmittelbar als gültiges JavaScript einlesen können:

```
opt.JSONP = 'var spiel = ';
```

Mit dem zweiten Parameter definieren wir den Namen der JSON-Datei, in der wir die Spieldaten abspeichern wollen:

```
opt.FileName = 'spiel.js';
```

Schließlich verwenden wir den Bibliotheksbefehl `savejson`, um die Spieldaten in die JSON-Datei zu schreiben:

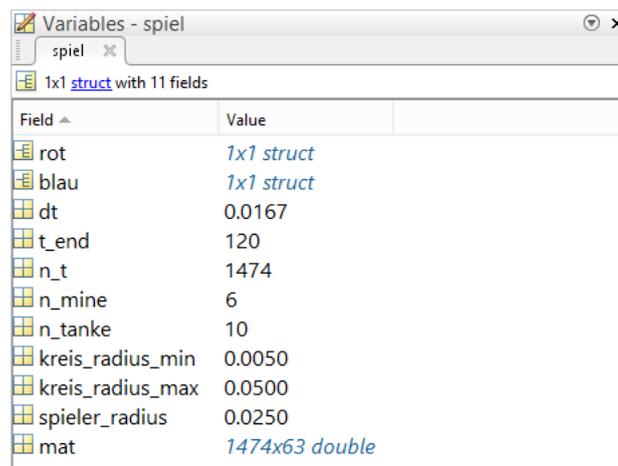
```
savejson('', spiel, opt);
```

**Achtung:** Das Verzeichnis `jsonlab`, in dem die Bibliothek liegt, muss im MATLAB-Pfad bekannt sein, damit das Unterprogramm `savejson` gefunden werden kann. Am komfortabelsten können wir dies manuell in der MATLAB-Systemumgebung unter dem Reiter `HOME/Set Path/Add with Subfolders/ ... /Save` erledigen.

## 5.10 spiel

Die Struktur `spiel` beinhaltet die Daten der gesamten Simulation. Sie wird, wie in Abbildung 4.1 dargestellt, von den Unterprogrammen `teams_einlesen` und `spielen` gefüllt und am Ende der Simulation von `json_speichern` als JSON-Datei `spiel.js` abgespeichert, um vom Visualisierungsprogramm `spiel.html` gelesen zu werden.

Die Struktur besitzt laut Abbildung 5.2 neben den beiden Teamstrukturen `rot` und `blau` sowohl skalare Daten (`dt`, ... `spieler_radius`) als auch die Spielverlaufsdaten in Form einer großen Matrix (`mat`).



Field	Value
rot	1x1 struct
blau	1x1 struct
dt	0.0167
t_end	120
n_t	1474
n_mine	6
n_tanke	10
kreis_radius_min	0.0050
kreis_radius_max	0.0500
spieler_radius	0.0250
mat	1474x63 double

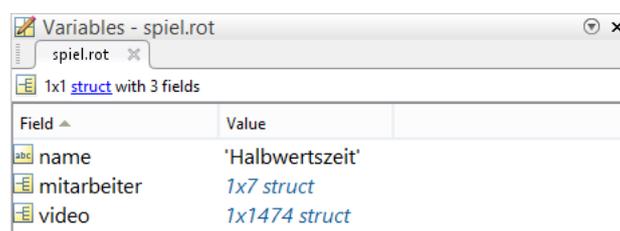
Abbildung 5.2: Struktur `spiel`

Im einzelnen haben die Felder folgende Bedeutung:

- rot** Struktur (Abbildung 5.3) mit dem Namen und den Mitarbeitern des roten Teams und den während der Simulation angeforderten Videos und Untertiteln
- blau** Struktur mit dem Namen und den Mitarbeitern des blauen Teams und den während der Simulation angeforderten Videos und Untertiteln
- dt** Simulationsschrittweite (in Sekunden). Im vorliegenden Fall wird mit 60 Hz (60 Simulationsschritten pro Sekunde) abgetastet. Es ergibt sich daher eine Simulationsschrittweite von  $dt = \frac{1}{60} = 0.01\bar{6}$  Sekunden.
- t\_end** Maximale Simulationszeit (in Sekunden). Jedes Spiel dauert im vorliegenden Fall höchstens zwei Minute. Viele Spiele enden vorzeitig nach wenigen Sekunden.
- n\_t** Anzahl der tatsächlich durchgeführten Simulationsschritte. Im vorliegenden Fall könnten maximal  $n_{t_{max}} = \frac{t_{end}}{dt} = \frac{120}{\frac{1}{60}} = 120 \cdot 60 = 7200$  Schritte simuliert werden. Die Zahl  $n_t = 1474$  zeigt uns allerdings, dass das Spiel tatsächlich nur  $t = n_t \cdot dt = \frac{1474}{60} = 24.5\bar{6}$  Sekunden gedauert hat.
- n\_mine** Anzahl der auf dem Spielfeld zufällig aber symmetrisch verteilten Minen
- n\_tanke** Anzahl der auf dem Spielfeld zufällig aber symmetrisch verteilten Tankstellen
- kreis\_radius\_min** Minimaler Radius der Minen und Tankstellen
- kreis\_radius\_max** Maximaler Radius der Minen und Tankstellen
- spieler\_radius** Anfänglicher Radius eines Spaceballs
- mat** Spieldatenmatrix (Abbildung 5.6) mit allen numerischen Daten der Simulation

### 5.10.1 spiel.rot

Das in Abbildung 5.3 dargestellte Feld `spiel.rot` beinhaltet den Namen des roten<sup>11</sup> Teams, die Namen und Aufgaben seiner Mitarbeiter (Abbildung 5.4) und das Strukturfeld `video` (Abbildung 5.5), in dem für jeden Zeitschritt die von der KI angeforderten Videos und Untertitel gespeichert sind.



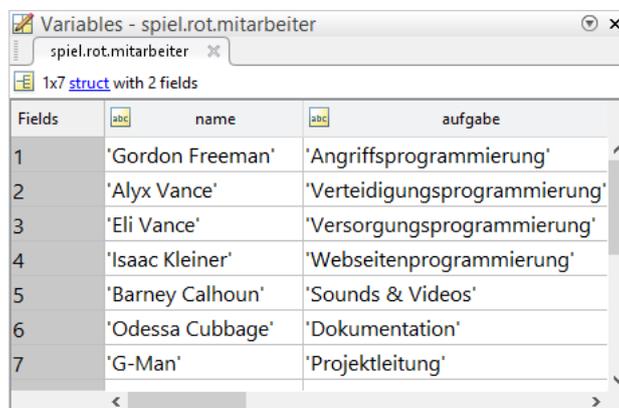
Field	Value
name	'Halbwertszeit'
mitarbeiter	1x7 struct
video	1x1474 struct

Abbildung 5.3: Feld `spiel.rot`

<sup>11</sup>Natürlich gibt es auch die entsprechenden Felder des blauen Teams, die wir hier aber aus Redundanzgründen nicht aufführen.

### 5.10.2 spiel.rot.mitarbeiter

Das Unterprogramm `teams_einlesen` liest die Namen und die Aufgaben der Teammitglieder aus der Datei `team.xml` und speichert sie in dem in Abbildung 5.4 dargestellten Feld von Strukturen.<sup>12</sup>

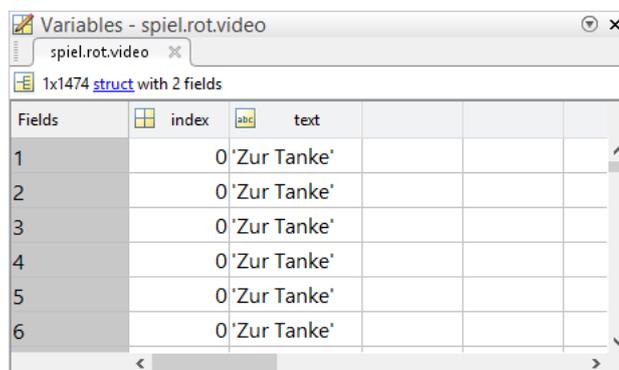


Fields	name	aufgabe
1	'Gordon Freeman'	'Angriffsprogrammierung'
2	'Alyx Vance'	'Verteidigungsprogrammierung'
3	'Eli Vance'	'Versorgungsprogrammierung'
4	'Isaac Kleiner'	'Webseitenprogrammierung'
5	'Barney Calhoun'	'Sounds & Videos'
6	'Odessa Cabbage'	'Dokumentation'
7	'G-Man'	'Projektleitung'

Abbildung 5.4: Feld `spiel.rot.mitarbeiter`

### 5.10.3 spiel.rot.video

Während jedes Simulationsschrittes kann die KI Videos und Untertitel definieren, die dann in der Visualisierung angezeigt werden. Wir speichern daher für jeden Simulationsschritt im Strukturfeld `video` in `spiel.rot` eine Videostruktur ab, die wiederum den Index des Videos als numerisches Feld (`index`) und den Untertitel als Zeichenkettenfeld (`text`) enthält. In Abbildung 5.5 sind die ersten sechs von 1474 Zellen dargestellt.



Fields	index	text
1	0	'Zur Tanke'
2	0	'Zur Tanke'
3	0	'Zur Tanke'
4	0	'Zur Tanke'
5	0	'Zur Tanke'
6	0	'Zur Tanke'

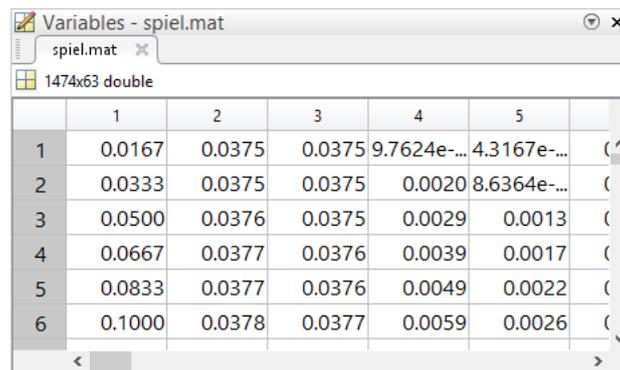
Abbildung 5.5: Feld `spiel.rot.video`

<sup>12</sup>Zum Unterschied zwischen einer Feldstruktur („Structure of arrays“) und Strukturfeldern („Array of structures“) hat sich Doug Hull von den Mathworks in [12] ein paar Gedanken gemacht. Auch die Kommentare auf der Seite sind teilweise ganz hilfreich.

Wenn die KI keinen Untertitel bzw. kein Video ausgeben möchte, gibt sie `text` als leere Zeichenkette (" , zwei einfache Hochkommata ohne Leerzeichen dazwischen) bzw. `index` als Zahl 0 zurück. Natürlich ist es – beispielsweise während der KI-Entwicklung – auch möglich, einen Untertitel, aber kein Video darzustellen (Abbildung 5.5).

### 5.10.4 spiel.mat

Das in Abbildung 5.6 ausschnittsweise dargestellte numerische Feld `spiel.mat` umfasst alle während der Simulation berechneten Positionen, Geschwindigkeiten, Beschleunigungen und Radien der Spaceballs, Minen und Tankstellen.



	1	2	3	4	5	
1	0.0167	0.0375	0.0375	9.7624e-...	4.3167e-...	( ^
2	0.0333	0.0375	0.0375	0.0020	8.6364e-...	(
3	0.0500	0.0376	0.0375	0.0029	0.0013	(
4	0.0667	0.0377	0.0376	0.0039	0.0017	(
5	0.0833	0.0377	0.0376	0.0049	0.0022	(
6	0.1000	0.0378	0.0377	0.0059	0.0026	(

Abbildung 5.6: Feld `spiel.mat`

Dabei stellt jede Zeile einen Zeitpunkt dar. Die Spalteninhalte zeigt Tabelle 5.1.

Spalte	Inhalt
1	Zeit ( <code>t</code> )
2 : 3	Position des roten Spaceballs ( <code>rot.pos</code> )
4 : 5	Geschwindigkeit des roten Spaceballs ( <code>rot.ges</code> )
6 : 7	Beschleunigung des roten Spaceballs ( <code>rot.bes</code> )
8	Radius des roten Spaceballs ( <code>rot.radius</code> )
9 : 10	Position des blauen Spaceballs ( <code>blau.pos</code> )
11 : 12	Geschwindigkeit des blauen Spaceballs ( <code>blau.ges</code> )
13 : 14	Beschleunigung des blauen Spaceballs ( <code>blau.bes</code> )
15	Radius des blauen Spaceballs ( <code>blau.radius</code> )
16 : 17	Position der ersten Mine ( <code>mine(1).pos</code> )
18	Radius der ersten Mine ( <code>mine(1).radius</code> )
...	Positionen und Radien der übrigen Minen
33	Radius der sechsten Mine ( <code>mine(6).radius</code> )
34 : 35	Position der ersten Tankstelle ( <code>tanke(1).pos</code> )
36	Radius der ersten Tankstelle ( <code>tanke(1).radius</code> )
...	Positionen und Radien der übrigen Tankstellen
63	Radius der zehnten Tankstelle ( <code>tanke(10).radius</code> )

Tabelle 5.1: Spalteninhalte in `spiel.mat` bei sechs Minen und zehn Tankstellen

Natürlich hätten wir aus Übersichtlichkeitsgründen auch die numerischen Spieldaten als Strukturen mit bezeichneten Feldern formulieren können; numerische Matrizen werden in MATLAB aber um Größenordnungen schneller verarbeitet und nehmen sowohl in MATLAB als auch in JSON signifikant weniger Platz ein.

## 5.11 spielfeld

In jedem Simulationsschritt übergibt das Simulationsunterprogramm `spielen` die Struktur `spielfeld` an das KI-Unterprogramm `kraft`. In der Struktur findet die KI die zum aktuellen Zeitpunkt vorherrschende Spielsituation, um darauf aufbauend ihre Entscheidungen zu treffen. Laut Abbildung 5.7 gibt es in `spielfeld` die aktuellen Daten des roten und des blauen Spaceballs, der Minen und der Tankstellen.

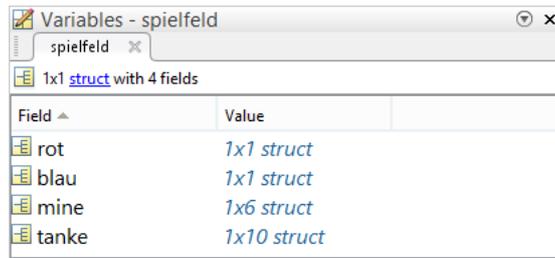


Abbildung 5.7: Struktur spielfeld

Wie wir in Abbildung 5.8 sehen, besteht die Struktur eines Spaceballs (hier exemplarisch der rote) aus den Feldern Position (**pos**,  $x$ - und  $y$ -Koordinate), Radius (**radius**, skalar), Geschwindigkeit (**ges**,  $x$ - und  $y$ -Koordinate) und Beschleunigung (**bes**,  $x$ - und  $y$ -Koordinate).

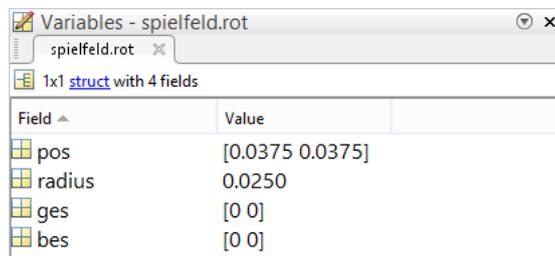


Abbildung 5.8: Feld spielfeld.rot

Abbildung 5.9 zeigt die Positionen und Radien der Minen im Strukturfeld mine.

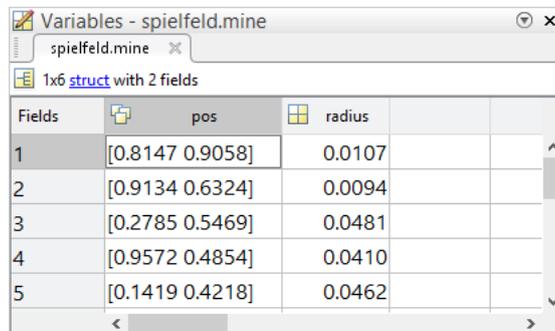


Abbildung 5.9: Feld spielfeld.mine

Die zehn Tankstellen des Strukturfelds tanke sind mit ihren Positionen und Radien in Abbildung 5.10 dargestellt.

The screenshot shows the MATLAB Variables window for a file named 'spielfeld.tanke'. It displays a 1x10 struct with two fields: 'pos' and 'radius'. The data is as follows:

Fields	pos	radius
1	[0.7922 0.9595]	0.0345
2	[0.6787 0.7577]	0.0384
3	[0.3922 0.6555]	0.0127
4	[0.7060 0.0318]	0.0175
5	[0.6948 0.3171]	0.0478
6	[0.0344 0.4387]	0.0222
7	[0.7655 0.7952]	0.0134
8	[0.4898 0.4456]	0.0341
9	[0.6797 0.6551]	0.0123
10	[0.3404 0.5853]	0.0151

Abbildung 5.10: Feld `spielfeld.tanke`

# 6 Visualisierung in HTML5

MATLAB ist nicht dazu ausgelegt, in Echtzeit zu rechnen, gleichzeitig zu visualisieren und synchron dazu noch mehrere Videos zu zeigen. Spannenderweise können aber moderne Browser wie Firefox und Chrome all dies in hoher Qualität und Geschwindigkeit leisten, wenn sie in HTML5 [13], CSS3 [14] und JavaScript [15] geschriebene Seiten darstellen.

Wir werden daher im Folgenden die Seite `spiel.html` programmieren, die den kompletten Ablauf eines Spiels gemäß Abbildung 4.1 aus der JSON-Datei `spiel.js` liest und in Echtzeit darstellt.

## 6.1 `spiel.js`

Die JSON-Datei `spiel.js` [10] dient der Kommunikation zwischen der Simulation in MATLAB und der Visualisierung in HTML5. In der Datei sind Informationen über das Team, die Positionen, Geschwindigkeiten, Beschleunigungen und Radien der Spaceballs, Minen und Tankstellen und die anzuzeigenden Videos gespeichert.

Wir erkennen als erstes die Deklaration der Variablen `spiel`, dann darin das Anlegen des Objekts `rot` mit den Schlüsseln `name` und `mitarbeiter` und schließlich in `mitarbeiter` ein Array mit mehreren<sup>1</sup> Objekten, die selbst wiederum die Schlüssel `name` und `aufgabe` besitzen:

```
var spiel = ({
  "rot": {
    "name": "Halbwertszeit",
    "mitarbeiter": [
      {
        "name": "Gordon Freeman",
        "aufgabe": "Angriffsprogrammierung"
      },
      :
      :
      {
        "name": "G-Man",
        "aufgabe": "Projektleitung"
      }
    ]
  }
});
```

---

<sup>1</sup>Hier und im Folgenden charakterisieren die zwei Zeilen mit den Doppelpunkten, dass wir dort Zeilen nicht dargestellt haben.

```

    }
  ],

```

Immer noch im Objekt `rot` wird dann das Array `video` angelegt, das für jeden Zeitschritt ein Objekt mit den Schlüsseln `index` und `text` besitzt:

```

    "video": [
      {
        "index": 0,
        "text": "Zur Tanke "
      },
      :
      :
      {
        "index": 5,
        "text": "Rot hat weniger Sprit. "
      }
    ]
  },

```

Die Daten des blauen Spaceballs haben einen gleichwertigen Aufbau:

```

    "blau": {
      :
      :
      {
        "index": 4,
        "text": "Blau gewinnt. "
      }
    ]
  },

```

Abschließend folgen die im Hauptprogramm `spaceballs` definierten Konstanten und die gesamten Spieldaten in der  $1474 \times 63$  Elemente umfassenden Matrix `mat` (vgl. Abbildung 5.6):

```

    "dt": 0.01666666667,
    "t_end": 120,
    "n_t": 1474,
    "n_mine": 6,
    "n_tanke": 10,
    "kreis_radius_min": 0.005,
    "kreis_radius_max": 0.05,
    "spieler_radius": 0.025,
    "mat": [
      [0.01666666667,0.03751627066, ... ,0.01507153728],
      [0.03333333333,0.0375488235, ... ,0.01507153728],
      :
    ]

```

```
:  
  [24.566666667,0.3331182553, ... ,0.00995747136]  
  ]  
}  
);
```

## 6.2 spiel.html

In der HTML5-Datei `spiel.html` [13] visualisieren wir die Bewegungen der Spaceballs, indem wir die Simulationsdaten aus der JSON-Transferdatei `spiel.js` einlesen und dann mit CSS3 und JavaScript in einer Echtzeitschleife darstellen.

Eine Webseite in HTML5 hat den folgenden Aufbau:

Die Dokumententypdeklaration sagt dem Browser, dass es sich um eine HTML5-Seite und nicht eine ältere Version handelt:

```
<!DOCTYPE html >
```

Die eigentliche Seite beginnt mit dem öffnenden `<html>`-Tag und endet mit dem schließenden `</html>`-Tag. Darin eingerahmt werden im `<head>` externe Fonts, Style Sheets und JavaScript-Dateien geladen. Im `<body>` befinden sich die Beschreibung der Seitenelemente und das JavaScript-Programm, das die Animation durchführt:

```
<html >  
  
<head > ... </head >  
  
<body > ... </body >  
  
</html >
```

### 6.2.1 spiel.css

In der Datei `spiel.css` [14] sind die Formatierungen der auf der HTML-Seite dargestellten Elemente definiert.

So sollen alle Textelemente die Schrift `Droid Sans` verwenden, die im `<head>` der Seite geladen wird:

```
body {  
  font-family: 'Droid Sans', sans-serif;
```

Die große mittige Überschrift soll hingegen in der Schrift `Play` und nicht fett gesetzt werden. Sie soll eine silberne Farbe erhalten und keine Begrenzungsleerräume enthalten. Außerdem vergrößern wir den Abstand der einzelnen Buchstaben kräftig auf fünf Pixel:

```
h1 {
  font-family: 'Play', sans-serif;
  font-weight: normal;
  color: silver;
  margin: 0px;
  letter-spacing: 5px;
}
```

Alle Textelemente der Tabelle (Überschrift, Logo, Spielfeld, Teaminformationen und Videos) sollen horizontal zentriert werden:

```
table {
  text-align: center;
}
```

Schließlich definieren wir noch vier Sonderformate für das Logo und die Teaminformationen (.x200), die Videos (.videos), das Spielfeld (.canvas) und die Videountertitel (.untertitel) und zwei Formate (.galerie und .silber) für die Tabellen in rot\_intro.html und rot\_videos.html:

```
.x200 {
  width: 200px;
  height: 200px;
  vertical-align: top;
  border: none;
}

.videos {
  width: 200px;
  height: 175px;
  vertical-align: bottom;
}

.canvas {
  width: 600px;
  height: 600px;
  vertical-align: bottom;
}

.untertitel {
  width: 200px;
  height: 25px;
  vertical-align: top;
}

.galerie {
  border-spacing: 25px;
}
```

```
.silber {  
    color: silver;  
}
```

### 6.2.2 <head>

Im Kopf der Seite laden wir zwei Webfonts herunter, die good guy Google freundlicher-weise kostenlos zur Verfügung stellt:

```
<head>  
  
  <link  
    href="http://fonts.googleapis.com/css?family=Play"  
    rel="stylesheet"  
    type="text/css">  
  
  <link  
    href="http://fonts.googleapis.com/css?family=Droid+Sans"  
    rel="stylesheet"  
    type="text/css">
```

In der nächsten Zeile verweisen wir auf die CSS-Datei `spiel.css`, die die Formatierung der Elemente definiert:

```
<link rel="stylesheet" type="text/css" href="spiel.css" />
```

Als Zeichensatz verwenden wir – wie im Internet üblich – Unicode in 8-Bit-Kodierung [16]:

```
<meta charset="utf-8" />
```

Nach der Definition des Seitentitels

```
<title>Spaceballs</title>
```

laden wir schließlich die JSON-Datei `spiel.js`, in der die Simulationsdaten abgespeichert sind:

```
<script src="spiel.js"></script>  
  
</head>
```

### 6.2.3 <body>

Im `<body>` der Seite beschreiben wir die Spielfeld- und Team-Elemente in einer Tabelle, zwei Fortschrittsbalken und ein paar Schalter und Schieberegler. Im Anschluss definieren wir das JavaScript, das die Echtzeit-Animation durchführt.

Die erste Zeile der dreispaltigen (und fünfzeiligen) Tabelle definiert die (im Titelbild sichtbare) Überschrift: „Spaceballs“. Zur optischen Auflockerung setzen wir zwischen jeweils zwei Buchstaben noch einen kleinen halbhohen Punkt:

```
<body>

  <table>
    <tr>
      <td colspan="3">
        <h1>
          S &sdot;
          p &sdot;
          a &sdot;
          c &sdot;
          e &sdot;
          b &sdot;
          a &sdot;
          l &sdot;
          l &sdot;
          s</h1>
        </td>
      </tr>
```

Das Attribut `colspan="3"` der Tabellenelementdefinition bedeutet dabei, dass sich das Element über alle drei Spalten erstreckt.

In der zweiten Tabellenzeile definieren wir in der ersten und dritten Spalte die beiden Logos, die wir aus den jeweils von den Teams erstellten Dateien `logo.png` einlesen und in der mittleren Spalte das Spielfeld, das sich – mit der aktuellen Zeile beginnend – über insgesamt vier Zeilen nach unten ausdehnt (`rowspan="4"`):

```
<tr>
  <td class="x200">
    
  </td>
  <td class="canvas" rowspan="4">
    <canvas
      id="canvas_spiel_feld"
      width="600"
      height="600"></canvas>
  </td>
  <td class="x200">
    
    </td>
</tr>

```

Dabei werden die Logos auf eine Größe von  $200 \times 200$  Pixeln beschränkt und das Spielfeld besitzt eine Größe von  $600 \times 600$  Pixeln. Auf diese Weise passt das gesamte Spiel gut in die in vielen Beamern noch übliche XGA-Auflösung von  $1024 \times 768$  Pixeln.

Für das Spielfeld nutzen wir die in HTML5 eingeführte Leinwand (`canvas`), auf die wir später (in jedem Animationsschritt) die Spaceballs, Minen und Tankstellen zeichnen werden.

In die dritte Tabellenzeile<sup>2</sup> schreiben wir die Namen der Teams und der Mitarbeiter:

```

<tr>
  <td class="x200">
    <h3 id="rot_name"></h3>
    <div id="rot_mitarbeiter"></div>
  </td>
  <td class="x200">
    <h3 id="blau_name"></h3>
    <div id="blau_mitarbeiter"></div>
  </td>
</tr>

```

Auch diese beiden Elemente haben aufgrund ihrer Klassenzugehörigkeit (`class="x200"` in `spiel.css`) eine Größe von  $200 \times 200$  Pixeln.

In der vierten Tabellenzeile laden wir für jedes Team zehn selbst erstellte Videos, die aber alle anfänglich das Attribut `hidden` bekommen, damit sie noch nicht angezeigt werden:

```

<tr>
  <td class="videos">
    <video
      id="rot_video_1"
      hidden
      src="teams/rot/videos/1.mp4"
      width="200">
    </video>
:
:
    <video
      id="rot_video_10"
      hidden

```

<sup>2</sup>Da sich die Leinwand ja über vier Zeilen erstreckt, brauchen wir hier und in den folgenden Zeilen nur zwei Spalten zu definieren, die dann automatisch links und rechts der Leinwand angeordnet werden.

```

        loop
        src="teams/rot/videos/10.mp4"
        width="200">
    </video>
</td>
<td class="videos">
    <video
        id="blau_video_1"
        hidden
        src="teams/blau/videos/1.mp4"
        width="200">
    </video>
:
:
    </td>
</tr>

```

Die spielentscheidenden Videos 1 - 5 dürfen eine Maximallänge von 10 Sekunden besitzen und werden nur einmalig – am Ende des Spiels – angezeigt. Den übrigen Videos – beispielsweise dem „Tanken“-Video – spendieren wir das Attribut `loop`. Auf diese Weise wird das Video immer dann angezeigt, wenn ein Spaceball tankt und stoppt, wenn nicht mehr getankt wird. Beim nächsten Tanken läuft das Video dann an der gleichen Stelle weiter, an der es vorher gestoppt hat und beginnt am Ende automatisch übergangslos wieder mit seinem Anfang.

In der fünften (und damit letzten) Tabellenzeile geben wir die vom System und von der KI definierbaren Videountertitel aus:

```

<tr>
  <td class="untertitel">
    <div id="rot_video_text"></div>
  </td>
  <td class="untertitel">
    <div id="blau_video_text"></div>
  </td>
</tr>
</table>

```

Die KI kann die Untertitel unabhängig von den Videos schreiben; während der KI-Entwicklungsphase können wir sie daher sehr gut für Diagnose- und Fehlererkennungszwecke verwenden.

Unter der Tabelle spendieren wir zwei weitere Leinwände. Auf die erste (lang und dünn) malen wir später<sup>3</sup> einen roten und einen blauen Balken, an deren Größen wir die momentane Treibstoffsituation der Spaceballs ablesen können:

<sup>3</sup>In HTML wird erst einmal nur die Leinwand definiert. Das Zeichnen der Inhalte geschieht dann später während der Animation in JavaScript.)



Dabei haben wir hier die Felder `geschwindigkeiten`, `beschleunigungen` und `untertitel` mit Hilfe des Attributs `checked` von Anfang an eingeschaltet. Die `&nbsp;` hinter den Feldbeschriftungen sind quick-and-dirty poor-man's-solutions, um ein wenig mehr Platz zwischen den Feldern zu erzwingen.

Als letztes benutzen wir in der gleichen Zeile noch einen ebenfalls in HTML5 eingeführten Schieberegler (`range`), mit dem wir später eine variable Zeitlupenfunktion einschalten können:

```
<input
  type="range"
  id="slomo"
  min="0"
  step="100"
  max="1000"
  value="0" />
Slomo
```

Die einzustellenden Verzögerungszeiten pro Animationsschritt betragen dabei null bis eine Sekunde mit einer Schrittweite von 100 Millisekunden.

Das in `<script>` beschriebene JavaScript ist dann der letzte große Block von `<body>`, womit das HTML-Dokument geschlossen werden kann:

```
<script> ... </script>

</body>

</html>
```

## 6.2.4 `<script>`

Während HTML die Elemente einer Seite definiert, beschreibt JavaScript deren Funktionalität.

### 6.2.4.1 Blockschaltbild

Abbildung 6.1 zeigt das Blockschaltbild des JavaScript-Programms mit all seinen Unterprogrammen und deren Aufrufhierarchie.

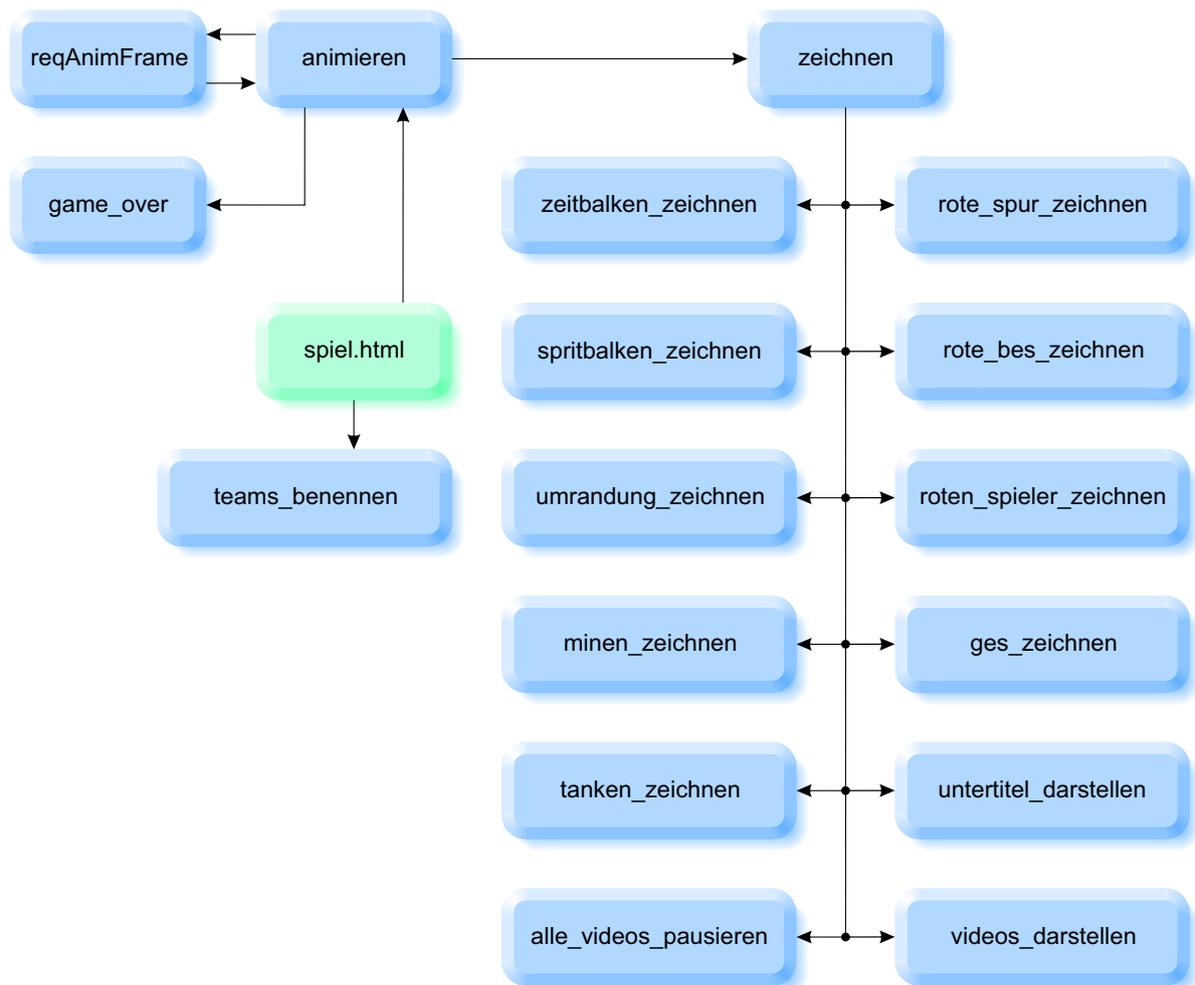


Abbildung 6.1: Blockschaltbild des JavaScript-Programms

Das grün dargestellte `spiel.html` symbolisiert das Hauptprogramm, das nach der Initialisierung der aktuellen Zeit

```
<script>
```

```
var i_t = 0
```

aus den drei in `<body>` definierten Leinwänden drei zweidimensionale Zeichenflächen (Kontexte<sup>4</sup>) ableitet, auf denen später die Objekte gezeichnet werden:

```
var spielfeld = canvas_spielfeld.getContext("2d")
```

```
var zeit = canvas_zeit.getContext("2d")
```

```
var sprit = canvas_sprit.getContext("2d")
```

<sup>4</sup>Die verschiedenen Befehle, um Rechtecke, Linien, Kreise, ... zu zeichnen, finden wir sehr übersichtlich dargestellt beispielsweise in [17].

Anschließend ruft das Hauptprogramm `spiel.html` gemäß Abbildung 6.1 das Unterprogramm `teams_benennen` auf, in dem die Namen der Teams und Mitarbeiter in die dafür vorgesehenen Tabellenelemente eingetragen werden

```
teams_benennen()
```

und übergibt die Kontrolle dann an das Unterprogramm `animieren`, das die Echtzeitanimation steuert:

```
animieren()
```

Das Unterprogramm `animieren` baut unter Zuhilfenahme des Systemunterprogramms `reqAnimationFrame` [18] eine Echtzeitschleife mit maximal 60 Hz auf, die in jedem Schritt das Unterprogramm `zeichnen` aufruft. Am Ende der Animation beendet ein Aufruf von `game_over` die Animation.

`zeichnen` wiederum zeichnet mit Hilfe der in Abbildung 6.1 dargestellten Unterprogramme `zeitbalken_zeichnen`, ..., `videos_darstellen` die animierten Objekte auf das Spielfeld und die anderen beiden Zeichenflächen.

#### 6.2.4.2 teams\_benennen

Im Unterprogramm `teams_benennen` lesen wir als erstes den Namen des roten Teams aus der Variablen `spiel`, die ja am Ende von `<head>` in der Datei `spiel.js` definiert wird und schreiben den Teamnamen in das dafür vorgesehene HTML-Element:

```
function teams_benennen() {
    rot_name.innerHTML = spiel.rot.name
```

Auch die Namen aller Mitarbeiter des roten Teams kopieren wir in einer Schleife aus der `spiel`-Variablen zeilenweise in das entsprechende HTML-Element:

```
for (
    var i_mitarbeiter = 0;
    i_mitarbeiter < spiel.rot.mitarbeiter.length;
    i_mitarbeiter++) {

    rot_mitarbeiter.innerHTML +=
        spiel.rot.mitarbeiter[i_mitarbeiter].name +
        "<br>"

}
```

Das Füllen der blauen Team- und Mitarbeiternamen geschieht analog, indem wir im Quelltext `rot` durch `blau` ersetzen.

### 6.2.4.3 animieren

Das Unterprogramm `animieren` ist das Herz der Animation. Hier besorgen wir uns vom `windows`-Objekt das `RequestAnimationFrame`-Unterprogramm, das leider momentan noch in unterschiedlichen Browsern unterschiedliche Namen haben kann:

```
function animieren() {
    reqAnimFrame =
        window.RequestAnimationFrame ||
        window.mozRequestAnimationFrame ||
        window.webkitRequestAnimationFrame ||
        window.msRequestAnimationFrame ||
        window.oRequestAnimationFrame
```

Solange dann der in jedem Animationsschritt erhöhte Schrittzähler noch kleiner ist als die Gesamtzahl der Simulationsschritte

```
if (i_t < spiel.n_t) {
```

zeichnen wir – in jedem Animationsschritt – den momentanen Zustand des Spielfeldes, der Balken und Videos:

```
    zeichnen()
```

Wir erhöhen den Schrittzähler

```
    i_t += 1
```

und rufen das oben definierte `reqAnimFrame`-Unterprogramm auf:

```
    setTimeout(function () { reqAnimFrame(animieren) }, slomo.
        value)
}
```

Normalerweise reicht dazu ein einfaches `reqAnimFrame(animieren)` aus, bei dem wir vom Browser den nächsten Animationsrahmen<sup>5</sup> anfordern und ihm sagen, welches Unterprogramm er darin aufrufen soll. Dies ist natürlich das Unterprogramm `animieren` selbst, das dann wieder einen neuen Rahmen anfordert, der dann wieder `animieren` aufruft, ...

<sup>5</sup>Durch die Anforderung eines Animationsrahmens überlassen wir es dem Browser, etwaige Hardwarebeschleunigungen einzuschalten, die Animation anzuhalten, wenn das Browserfenster nicht mehr im Vordergrund liegt und vor allem dafür zu sorgen, dass die Animation nicht zu schnell läuft und damit unnötig viel Rechenzeit verbraucht. Momentan drosseln die Browser die Animationsfrequenz bei 60 Hz; und da wir auch unter Matlab mit 60 Hz simuliert haben, läuft die Animation in Echtzeit ab. Dabei gehen wir bei aktuellen Rechnern davon aus, dass sie es schaffen, die Kreise und Linien jeweils innerhalb von  $dt = \frac{1}{60} = 0.01\bar{6}$  Sekunden auf den Schirm zu bringen. Wenn der Rechner dies nicht packt, geht die Echtzeit verloren; die Animation verläuft dann langsamer.

Indem wir `reqAnimationFrame(animieren)` – wie oben geschehen – als anonyme Funktion in `setTimeout` schachteln, können wir eine einfache Zeitlupe realisieren. `setTimeout` wartet dabei die im Slomo-Schieberegler eingestellte Zeit (in Millisekunden) ab, bis es den nächsten Animationsrahmen anfordert.

Wenn der letzte Animationsschritt durchlaufen wurde (`i_t = spiel.n_t`) beenden wir die Animation und rufen das Aufräumunterprogramm `game_over` auf:

```
else {  
    game_over()  
}  
}
```

#### 6.2.4.4 zeichnen

In jedem Animationsschritt ruft `animieren` das Unterprogramm `zeichnen` auf. Wie in Abbildung 6.1 dargestellt, dient `zeichnen` als Rahmenprogramm, das die eigentlichen Zeichenunterprogramme nacheinander aufruft.

Als erstes führen wir dabei `zeitbalken_zeichnen` und `spritbalken_zeichnen` aus, die die aktuell verstrichene Zeit und die Treibstoffmengen der beiden Spaceballs anzeigen:

```
function zeichnen() {  
    zeitbalken_zeichnen()  
    spritbalken_zeichnen()  
}
```

Im vielen anderen Grafikoberflächen stellt das Löschen und Neuzeichnen von Grafikobjekten eine rechenzeittechnische Todsünde dar. Statt dessen werden dort nur die Positionskordinaten der sich bewegenden Objekte geändert; die unbewegten Objekte werden nicht angefasst. Im Gegensatz dazu müssen wir hier tatsächlich in jedem Animationsschritt die gesamte Zeichenfläche löschen

```
spielfeld.clearRect(  
    0,  
    0,  
    canvas_spielfeld.width,  
    canvas_spielfeld.height)
```

und dann alle Objekte neu zeichnen.

Die Reihenfolge der Zeichenbefehle definiert das „Einanderüberdecken“ der Objekte; ein später gezeichnetes Objekt überdeckt also ein vorher gezeichnetes Objekt. Da wir beispielsweise wollen, dass ein Spaceball beim Tanken als Ganzes über der Tankstelle sicht-

bar ist (Abbildung 2.1), zeichnen wir als erstes die Umrandung (`umrandung_zeichnen`), die Minen (`minen_zeichnen`) und die Tankstellen (`tanken_zeichnen`):

```
umrandung_zeichnen()  
  
minen_zeichnen()  
  
tanken_zeichnen()
```

Wenn der Nutzer das Auswahlfeld **Spuren** angewählt hat, zeichnen wir beide Spuren (`rote_spur_zeichnen`):

```
if (spuren.checked) {  
  
    rote_spur_zeichnen()  
  
    blaue_spur_zeichnen()  
  
}
```

Wenn das Beschleunigungsauswahlfeld ausgewählt ist, zeichnen wir die Beschleunigungsdreiecke (`rote_beschleunigung_zeichnen`):

```
if (beschleunigungen.checked) {  
  
    rote_beschleunigung_zeichnen()  
  
    blaue_beschleunigung_zeichnen()  
  
}
```

Jetzt können wir die Spaceballs selbst zeichnen, die ja „über“ den Tankstellen und Beschleunigungsdreiecken liegen sollen (`roten_spieler_zeichnen`):

```
roten_spieler_zeichnen()  
  
blauen_spieler_zeichnen()
```

In Abhängigkeit von den Auswahlfeldern zeichnen wir dann noch die Geschwindigkeitsvektoren (`geschwindigkeiten_zeichnen`)

```
if (geschwindigkeiten.checked) {  
  
    geschwindigkeiten_zeichnen()  
  
}
```

und stellen die Videos (`videos_darstellen`) und ihre Untertitel (`untertitel_darstellen`) dar:

```

if (videos.checked) {
    alle_videos_pausieren()
    videos_darstellen()
}

if (untertitel.checked) {
    untertitel_darstellen()
}
}

```

#### 6.2.4.5 zeitbalken\_zeichnen

Um die verstrichene Spielzeit anzuzeigen, lassen wir unter dem Spielfeld einen silbergrauen Balken von links nach rechts anwachsen (Abbildung 2.2). Dazu benutzen wir den Befehl `beginPath`, um auf der in Abschnitt 6.2.4.1 definierten 2D-Zeichenfläche `zeit` einen neuen Zeichenvorgang zu beginnen<sup>6</sup>:

```

function zeitbalken_zeichnen() {
    zeit.beginPath()

```

Als nächstes legen wir fest, dass der Balken eine silbergraue Füllfarbe haben soll

```

zeit.fillStyle = "Silver"

```

und definieren ein gefülltes Rechteck, dessen linke obere Ecke in der linken oberen Ecke  $(0, 0)$  der Zeichenfläche liegt, dessen aktuelle Breite  $b$  proportional zur aktuellen Zeit  $t$  ist:

$$b = \frac{t}{t_{max}} \cdot b_{max} \quad (6.1)$$

und dessen Höhe 10 Pixel beträgt:

```

zeit.fillRect(
    0,
    0,
    spiel.mat[i_t][0] / spiel.t_end * canvas_zeit.width,
    10)

```

<sup>6</sup>Häufig funktioniert die Aneinanderreihung mehrerer Zeichenbefehle unterschiedlicher Objekte auch ohne einen neuen `beginPath`-Befehl. Manchmal entstehen dadurch aber auch schwer nachzuvollziehende Zeichenfehler. Aus der Erfahrung heraus empfiehlt es sich daher, lieber zu viele als zu wenige einzelne Zeichenpfade zu definieren.

Die aktuelle Zeit  $t$  lesen wir dabei in der aktuellen Zeile (`i_t`) der ersten<sup>7</sup> Spalte der Spieldatenmatrix `spiel.mat` ab. Die Maximalbreite  $b_{max}$  des Balkens wählen wir so, dass sie genau der Breite der entsprechenden Zeichenfläche entspricht (`canvas_zeit.width`).

Um das Rechteck tatsächlich<sup>8</sup> zu zeichnen, müssen wir den `stroke`-Befehl verwenden:

```
zeit.stroke()
}
```

#### 6.2.4.6 spritbalken\_zeichnen

Bei der Anzeige der Treibstoffmengen lesen wir aus der Spieldatenmatrix zuerst die Radien beider Spaceballs (**nullbasierte** Spalte 7 und 14 in Tabelle 5.1) und bedenken dann, dass die Treibstoffmenge quadratisch vom Radius abhängt:

```
function spritbalken_zeichnen() {
    var sprit_rot = Math.pow(spiel.mat[i_t][7], 2)
    var sprit_blau = Math.pow(spiel.mat[i_t][14], 2)
```

Da wir den relativen Anteil beider Treibstoffmengen darstellen wollen, berechnen wir als nächstes die Gesamttriebstoffmenge:

```
var sprit_summe = sprit_rot + sprit_blau
```

Die Länge des roten Treibstoffbalkens ergibt sich dann analog zu Gleichung (6.1):

```
balken_laenge_rot = sprit_rot / sprit_summe *
    canvas_sprit.width
```

Wir leeren jetzt die gesamte Zeichenfläche

```
sprit.clearRect(0, 0, canvas_sprit.width, 10)
```

und zeichnen den roten Balken in der gerade berechneten Breite:

```
sprit.beginPath()
sprit.fillStyle = "Tomato"
sprit.fillRect(0, 0, balken_laenge_rot, 10)
sprit.stroke()
```

<sup>7</sup>JavaScript ist eine nullbasierte Sprache. Das bedeutet, dass die erste Spalte den Index 0 besitzt.

<sup>8</sup>Es ist einer der beliebtesten Anfängerfehler, `stroke` (oder `fill`) zu vergessen und sich (beliebig lange) darüber zu ärgern, dass ein Zeichenobjekt nicht dargestellt wird.

Der blaue Balken beginnt dann (mit seiner  $x$ -Koordinate) genau dort, wo der rote Balken aufhört (`balken_laenge_rot`) und hat eine Breite<sup>9</sup>, die ihn genau bis zum Ende der Zeichenfläche gehen lässt (`canvas_sprit.width - balken_laenge_rot`):

```
sprit.beginPath()

sprit.fillStyle = "DeepSkyBlue"

sprit.fillRect(
    balken_laenge_rot,
    0,
    canvas_sprit.width - balken_laenge_rot,
    10)

sprit.stroke()
```

Um bei kleinen Unterschieden leichter unterscheiden zu können, wer mehr Treibstoff besitzt, zeichnen wir noch eine Mittellinie auf der Mitte des Balkens. Beim Zeichnen von Linien bewegen wir den Zeichenstift (ohne zu zeichnen) mit einem `moveTo`-Befehl an den Anfangspunkt der Linie (oben) und zeichnen von dort mit `lineTo` eine Linie zum Endpunkt (unten):

```
sprit.beginPath()

sprit.moveTo(0.5 * canvas_sprit.width, 0)

sprit.lineTo(0.5 * canvas_sprit.width, 10)

sprit.strokeStyle = 'Black'

sprit.stroke()

}
```

#### 6.2.4.7 umrandung\_zeichnen

Die Spielfeldumrandung ist ein einfaches, nicht ausgefülltes, silbergraues Rechteck über die gesamte Zeichenfläche:

```
function umrandungen_zeichnen() {

    spielfeld.beginPath()
```

---

<sup>9</sup>Wir müssen an dieser Stelle verstehen, dass der dritte und vierte Parameter des `fillRect`-Befehls tatsächlich die Breite und die Höhe des Rechtecks (und **nicht** die Koordinaten seiner rechten, unteren Ecke) beschreiben.

```

    spielfeld.strokeStyle = "Silver"

    spielfeld.rect(
        0,
        0,
        canvas_spielfeld.width,
        canvas_spielfeld.height)

    spielfeld.stroke()
}

```

#### 6.2.4.8 minen\_zeichnen

Um die Minen zu zeichnen, starten wir eine nullbasierte(!) Schleife über alle Minen:

```

function minen_zeichnen() {

    for (i_mine = 0; i_mine < spiel.n_mine; i_mine++) {

```

Bei jedem Schleifendurchlauf beginnen wir einen neuen Zeichenpfad

```

    spielfeld.beginPath()

```

und verwenden dann den `arc`-Befehl, um einen Kreis zu definieren:

```

    spielfeld.arc(
        spiel.mat[i_t][15 + 3 * i_mine] * canvas_spielfeld.width,
        spiel.mat[i_t][16 + 3 * i_mine] * canvas_spielfeld.width,
        spiel.mat[i_t][17 + 3 * i_mine] * canvas_spielfeld.width,
        0,
        2 * Math.PI, true)

```

Dabei sind die ersten beiden Parameter des `arc`-Befehls die  $x$ - und  $y$ -Koordinate des Kreismittelpunktes. Der dritte Parameter definiert den Kreisradius und die übrigen beiden Parameter bestimmen den Anfangs- und Endwinkel des Kreisausschnitts. Bei einem Vollkreis läuft der Winkel natürlich im Bereich  $0 < \alpha < 2\pi$ .

Da wir das Spielfeld bei der Simulation mit einer normierten Höhe und Breite von eins angenommen haben, müssen wir die Koordinaten noch mit der Spielfeldgröße<sup>10</sup> skalieren.

Mit der richtigen Minenfarbe

```

    spielfeld.fillStyle = "Black"

```

füllen<sup>11</sup> wir abschließend den Kreis:

<sup>10</sup>Wir gehen ab jetzt zur Vereinfachung immer von einem quadratischen Spielfeld aus und verwenden daher auch in  $y$ -Richtung `canvas_spielfeld.width`.

<sup>11</sup>Wir nutzen hier `fill` statt `stroke`, um den Kreis zu füllen und nicht nur seinen Umfang zu zeichnen.

```
    spielfeld.fill()
  }
}
```

#### 6.2.4.9 tanken\_zeichnen

Das Zeichnen der Tankstellen geschieht völlig analog zum Zeichnen der Minen. Lediglich das Ermitteln der korrekten Indizes erfordert etwas mehr Überlegung, da der Index der ersten Tankstelle natürlich von der Anzahl der davor abgespeicherten Minen abhängt (Tabelle 5.1):

```
function tanken_zeichnen() {
  for (i_tanke = 0; i_tanke < spiel.n_tanke; i_tanke++) {
    spielfeld.beginPath()
    spielfeld.arc(
      spiel.mat[i_t][15 + 3 * (spiel.n_mine + i_tanke)] *
        canvas_spielfeld.width,
      spiel.mat[i_t][16 + 3 * (spiel.n_mine + i_tanke)] *
        canvas_spielfeld.width,
      spiel.mat[i_t][17 + 3 * (spiel.n_mine + i_tanke)] *
        canvas_spielfeld.width,
      0,
      2 * Math.PI, true)
    spielfeld.fillStyle = "YellowGreen"
    spielfeld.fill()
  }
}
```

#### 6.2.4.10 rote\_spur\_zeichnen

Die rote Spur ist ein Polygonzug, der am Startpunkt des roten Spaceballs beginnt und dann alle Zwischenpositionen bis zur aktuellen Position mit Geradenstücken verbindet. Wir bewegen den Zeichenstift dazu auf den roten Startpunkt

```
function rote_spur_zeichnen() {
```

```

spielfeld.beginPath()

spielfeld.moveTo(
  spiel.mat[0][1] * canvas_spielfeld.width,
  spiel.mat[0][2] * canvas_spielfeld.width)

```

und beginnen dann eine Schleife über alle bisherigen Zeitpunkte:

```

for (i_punkt = 1; i_punkt <= i_t; i_punkt++) {

```

In der Schleife definieren wir jeweils eine Linie zur nächsten Position:

```

  spielfeld.lineTo(
    spiel.mat[i_punkt][1] * canvas_spielfeld.width,
    spiel.mat[i_punkt][2] * canvas_spielfeld.width)

}

```

und vergessen nicht, die Gesamtlinie schließlich auch wirklich (in Tomatenrot) zu zeichnen:

```

  spielfeld.strokeStyle = 'Tomato'

  spielfeld.stroke()

}

```

Natürlich zeichnen wir auch die blaue Spur, indem wir im Quelltext rot durch blau ersetzen.

#### 6.2.4.11 rote\_beschleunigung\_zeichnen

Die Beschleunigung symbolisieren wir, wie in Abbildung 6.2 dargestellt, als gleichschenkeliges Dreieck, dessen Mittelsenkrechte  $\overline{M1} = 3r$  drei Radienlängen ( $\overline{M2} = \overline{M3} = r$ ) beträgt.<sup>12</sup>

---

<sup>12</sup>Auf diese Weise wächst und schrumpft der Triebwerksstrahl mit dem Spaceball, was eigentlich physikalisch nicht ganz in Ordnung ist, da ja der Schub konstant ist und die Beschleunigung bei einem fetten Spaceball ja sogar kleiner ist. Andererseits sieht ein immer gleich großes Dreieck bei ganz kleinen oder ganz großen Spaceballs ziemlich albern aus. Wir entscheiden uns daher bewusst für die unphysikalischere aber ästhetischere Variante.

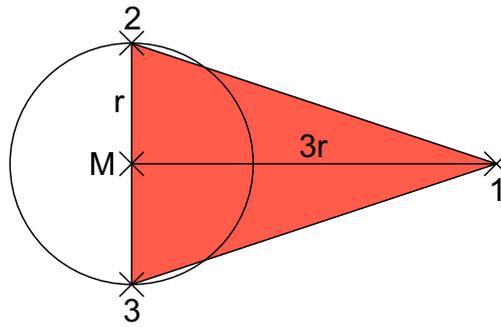


Abbildung 6.2: Beschleunigung als dreieckiger Triebwerksstrahl

Der Punkt 1 zeigt dabei in Richtung des Triebwerkstrahls. In Abbildung 6.2 hat der Beschleunigungsvektor also nur eine negative  $x$ -Komponente (nach links). Zur Definition der Dreieckskoordinaten bestimmen wir zuerst die aktuellen Beschleunigungskomponenten:

```
function rote_beschleunigung_zeichnen() {
    x_bes = spiel.mat[i_t][5]
    y_bes = spiel.mat[i_t][6]
```

Da uns vom Beschleunigungsvektor nur seine Richtung interessiert, bilden wir seinen Betrag

```
norm_bes = Math.sqrt(Math.pow(x_bes, 2) + Math.pow(y_bes, 2))
```

und normieren ihn auf seinen Einheitsvektor:

```
x_bes = x_bes / norm_bes
y_bes = y_bes / norm_bes
```

Als nächstes bestimmen wir die aktuellen Koordinaten des Mittelpunktes des roten Spaceballs und seinen Radius:

```
x_pos = spiel.mat[i_t][1] * canvas_spiel.feld.width
y_pos = spiel.mat[i_t][2] * canvas_spiel.feld.width

radius = spiel.mat[i_t][7] * canvas_spiel.feld.width
```

Jetzt können wir das Zeichnen des Dreiecks beginnen:

```
spiel.feld.beginPath()
```

Wir bewegen den Zeichenstift zum Ende des Triebwerkstrahls (Punkt 1 in Abbildung 6.2)

```
spiel.feld.moveTo(
    x_pos - 3 * x_bes * radius,
    y_pos - 3 * y_bes * radius)
```

und ziehen eine Linie zu Punkt 2:

```
spielfeld.lineTo(
  x_pos - y_bes * radius,
  y_pos + x_bes * radius)
```

Eine weitere Linie (über den Mittelpunkt) zu Punkt 3

```
spielfeld.lineTo(
  x_pos + y_bes * radius,
  y_pos - x_bes * radius)
```

und der Befehl zum Schließen des Polygonzugs zurück zu Punkt 1 definieren schließlich alle Seiten des Dreiecks:

```
spielfeld.closePath()
```

Als Farbe wählen wir die gleiche Farbe wie die des Spaceballs

```
spielfeld.fillStyle = "Tomato"
```

setzen aber die Deckkraft des Dreiecks auf 50%, so dass auch darunterliegende Objekte (Minen, Tankstellen, Gegner, Spuren) gut sichtbar sind:

```
spielfeld.globalAlpha = 0.5
```

Nachdem wir das Dreieck gefüllt haben, setzen wir die Deckkraft für die folgenden Objekte wieder auf 100% zurück:

```
spielfeld.fill()

spielfeld.globalAlpha = 1

}
```

Müssen wir noch erwähnen, wie das Beschleunigungsdreieck des blauen Spaceballs erzeugt wird?

#### 6.2.4.12 roten\_spieler\_zeichnen

Den roten (und natürlich auch den blauen) Spaceball zeichnen wir – genau wie jede einzelne Mine (Abschnitt 6.2.4.8) oder Tankstelle (Abschnitt 6.2.4.9) – als farbig gefüllten Kreis um seine aktuelle Position:

```
function roten_spieler_zeichnen() {

  spielfeld.beginPath()

  spielfeld.arc(
    spiel.mat[i_t][1] * canvas_spielfeld.width,
```

```

    spiel.mat[i_t][2] * canvas_spiel_feld.width,
    spiel.mat[i_t][7] * canvas_spiel_feld.width,
    0,
    2 * Math.PI, true)

    spiel_feld.fillStyle = "Tomato"

    spiel_feld.fill()
}

```

### 6.2.4.13 geschwindigkeiten\_zeichnen

Die Geschwindigkeitsanzeiger sind einfache schwarze Linien vom Mittelpunkt eines Spaceballs in Richtung des momentanen Geschwindigkeitsvektors. Der Betrag des Geschwindigkeitsvektors wird durch die Länge der Linie dargestellt.

Wir beginnen also ein neues Grafikobjekt

```

function geschwindigkeiten_zeichnen() {

    spiel_feld.beginPath()

```

und bewegen den Zeichenstift zum Mittelpunkt des roten Spaceballs:

```

    spiel_feld.moveTo(
        spiel.mat[i_t][1] * canvas_spiel_feld.width,
        spiel.mat[i_t][2] * canvas_spiel_feld.width)

```

Dann definieren wir eine Linie zum Ende seines Geschwindigkeitsanzeigers, indem wir seinen (normierten) Geschwindigkeitsvektor zu seinem (normierten) Positionsvektor addieren:

```

    spiel_feld.lineTo(
        (spiel.mat[i_t][1] + spiel.mat[i_t][3]) * canvas_spiel_feld.
            width,
        (spiel.mat[i_t][2] + spiel.mat[i_t][4]) * canvas_spiel_feld.
            width)

```

Nachdem wir die entsprechenden Anfangs- und Endpunkte des Geschwindigkeitsanzeigers des blauen Spaceballs definiert haben

```

    spiel_feld.moveTo(
        spiel.mat[i_t][8] * canvas_spiel_feld.width,
        spiel.mat[i_t][9] * canvas_spiel_feld.width)

    spiel_feld.lineTo(
        (spiel.mat[i_t][8] + spiel.mat[i_t][10]) * canvas_spiel_feld.
            width,

```

```
(spiel.mat[i_t][9] + spiel.mat[i_t][11]) * canvas_spiel.feld.  
width)
```

können wir beide Linien (obwohl sie nicht zusammenhängen) in einem Rutsch zeichnen:

```
spiel.feld.strokeStyle = 'Black'  
  
spiel.feld.stroke()  
  
}
```

#### 6.2.4.14 unternitel\_darstellen

Im Unterprogramm `unternitel_darstellen` kopieren wir einfach nur die von der KI oder vom System angeforderten Untertitel in die entsprechenden HTML-Elemente:

```
function unternitel_darstellen() {  
  
    rot_video_text.innerHTML = spiel.rot.video[i_t].text  
  
    blau_video_text.innerHTML = spiel.blau.video[i_t].text  
  
}
```

#### 6.2.4.15 alle\_videos\_pausieren

Bevor wir gegebenenfalls ein neues Video darstellen können, pausieren und verstecken wir in jedem Animationsschritt erst einmal alle Videos beider Teams:

```
function alle_videos_pausieren() {  
  
    rot_video_1.pause()  
    rot_video_1.hidden = true  
  
:  
:  
  
    rot_video_10.pause()  
    rot_video_10.hidden = true  
  
    blau_video_1.pause()  
    blau_video_1.hidden = true  
  
:  
:  
}
```

#### 6.2.4.16 videos\_darstellen

Wenn die KI oder das System im aktuellen Animationsschritt ein bestimmtes Video angefordert hat, machen wir es sichtbar und „drücken seine Abspieltaste“:

```
function videos_darstellen() {  
  
    if (spiel.rot.video[i_t].index == 1) {  
  
        rot_video_1.hidden = false  
        rot_video_1.play()  
  
    }  
  
    :  
    :  
  
    if (spiel.rot.video[i_t].index == 10) {  
  
        rot_video_10.hidden = false  
        rot_video_10.play()  
  
    }  
  
    if (spiel.blau.video[i_t].index == 1) {  
  
        blau_video_1.hidden = false  
        blau_video_1.play()  
  
    }  
  
    :  
    :  
}
```

Dabei läuft ein Video automatisch an der Stelle weiter, an der es vorher pausiert wurde. Die von der KI anforderbaren Videos 5-10 beginnen automatisch von vorne, wenn sie an ihrem Ende angelangt sind.

#### 6.2.4.17 game\_over

Wenn die gesamte Darstellung des Spielverlaufs beendet ist, wird das Unterprogramm `game_over` aufgerufen, in dem gegebenenfalls Aufräum- und Schreibearbeiten für weiterführende Programm erledigt werden können. Momentan macht das Unterprogramm gar nichts:

```
function game_over() {  
}  
  
</script>  
  
</body>
```

## 6.3 Bilder

Jedes Team erstellt in ihrem Bilderordner ein Logo und Bilder ihrer Mitarbeiter. Die Bilder des roten Teams befinden sich im Ordner `spaceball/teams/rot/bilder`. Die Bilder des blauen Teams befinden sich im Ordner `spaceball/teams/blau/bilder`.

### 6.3.1 mitarbeiter.jpg

Jedes Team erstellt für jeden ihrer Mitarbeiter ein Kopfbild, auf dem der Mitarbeiter eindeutig zu erkennen ist. Die Bilder werden (für das rote Team) auf der Seite `rot_intro.html` dargestellt und müssen vom Typ JPG sein. Die Dateinamen ergeben sich direkt aus den in `team.xml` definierten Mitarbeiternamen.

Beispiel: `Gordon Freeman.jpg`

Die Bilder müssen eine einheitliche Größe von  $200 \times 200$  Pixeln haben und werden (für das rote Team) auf der Seite `rot_intro.html` dargestellt.

### 6.3.2 logo.png

Jedes Team erstellt ein Logo, das auf der Seite `rot_intro.html` und während des Spiels über dem eigenen Teamnamen dargestellt wird. Das Logo muss eine PNG-Datei mit dem Namen `logo.png` sein und eine Größe von  $200 \times 200$  Pixeln besitzen.

## 6.4 Videos und Sound

### 6.5 1.mp4 ... 10.mp4

Jedes Team erstellt<sup>13</sup> mindestens sechs und maximal zehn Videos, die das System oder die KI während der Animation darstellen lassen kann. Die Videos müssen MP4-Dateien

---

<sup>13</sup>Autodesk[19] stellt beispielsweise seine 3D-Animationswerkzeuge 3ds Max und Maya für Studierende kostenlos zur Verfügung.

sein und die Namen 1.mp4 ... 10.mp4 besitzen. Alle Videos (des roten Teams) befinden sich im Ordner `spaceball/teams/rot/videos`. Die ersten sechs Videos sind laut Abschnitt 2.2 für bestimmte Ereignisse reserviert; die restlichen vier können für beliebige andere Ereignisse genutzt werden. Videos dürfen maximal 10 Sekunden lang sein und müssen eine einheitliche Größe von  $800 \times 600$  Pixeln, also ein Standardverhältnis von 4 : 3 haben.

Alle Videos zusammen können wir uns auf der Seite `rot_videos.html` ansehen.

### 6.5.1 intro.wav

Jedes Team erstellt eine maximal 10 Sekunden lange Sounddatei, die beim Anzeigen von `rot_intro.html` abgespielt wird. Die Datei muss vom Typ WAV sein, `intro.wav` heißen und sich (für das rote Team) im Ordner `spaceball/teams/rot/sounds` befinden.

## 6.6 rot\_intro.html

Die Seite `rot_intro.html` stellt das Logo, die Mitarbeiter des (roten) Teams und ihre Aufgaben vor (Abbildung 6.3).



Abbildung 6.3: Vorstellung des Teams

Den mit dem Kopf der Animationsseite `<head>` identischen Kopf der Seite stellen wir hier nicht nochmals dar.

Im `<body>` der Seite starten wir automatisch (`autoplay`) die Sounddatei `intro.wav` an<sup>14</sup>

```
<body>

  <audio
    id="audio_intro"
    src="teams/rot/sounds/intro.wav"
    autoplay
    onended="wechseln()">
  </audio>
```

und stellen in einer zweizeiligen und vierspaltigen Tabelle unter dem Teamnamen

```
<table class="galerie">
  <tr>
    <td colspan="4">
      <h1 id="team_name"></h1>
    </td>
  </tr>
```

zuerst das Logo und dann die Mitarbeiter mit ihrem Bild<sup>15</sup>, ihrem Namen und ihrer Aufgabe dar:

```
<tr>
  <td>
    
  </td>
  <td id="mitarbeiter_1" hidden>
    
    <div id="name_mitarbeiter_1"></div>
    <div id="aufgabe_mitarbeiter_1" class="silber"></div>
  </td>
```

<sup>14</sup>Im `onended`-Attribut können wir ein Unterprogramm angeben, das am Ende der Sounddatei aufgerufen wird. Wir können dadurch dann beispielsweise automatisch auf die Seite `rot_videos.html` wechseln.

<sup>15</sup>Da die Bilder der Mitarbeiter erst später vom JavaScript definiert werden, das HTML-Element aber aus formalen Gründen schon eine Quelle benötigt, geben wir hier erst einmal das Logo als Bild der Mitarbeiter an.

```

:
:

    </tr>
  </table>

```

Die einzelnen Zellen sind anfänglich versteckt und werden erst per JavaScript sichtbar gemacht, wenn es den jeweiligen Mitarbeiter tatsächlich in `team.xml` gibt.

Das JavaScript

```
<script>
```

ruft als erstes das Unterprogramm `team_darstellen` auf

```
team_darstellen()
```

und weist dann das Laufzeitsystem an, in 10 Sekunden definitiv das Unterprogramm `wechseln` aufzurufen:

```
setTimeout(function () { wechseln() }, 10000);
```

Auf diese Weise wird die Seite nie länger als 10 Sekunden angezeigt; egal wie lange die Sounddatei eigentlich dauern würde.

Im Unterprogramm `team_darstellen` zeigen wir als erstes den Teamnamen an

```
function team_darstellen() {
    team_name.innerHTML = spiel.rot.name

```

und untersuchen dann für jeden einzelnen der maximal sieben Mitarbeiter, ob es für ihn einen Namen in `team.xml` gibt.

Wenn dies der Fall ist

```
if (spiel.rot.mitarbeiter[0].name) {
```

machen wir seine Tabellenzelle sichtbar, laden sein Bild und zeigen seinen Namen und seine Aufgabe an:

```

    mitarbeiter_1.hidden = false

    im_mitarbeiter_1.src =
      "teams/rot/bilder/" +
      spiel.rot.mitarbeiter[0].name +
      ".jpg"

    name_mitarbeiter_1.innerHTML =
      spiel.rot.mitarbeiter[0].name

```

```
        aufgabe_mitarbeiter_1.innerHTML =
            spiel.rot.mitarbeiter[0].aufgabe
    }
:
:
}
```

Das Unterprogramm `wechseln` wird aufgerufen, wenn die Sounddatei zu Ende ist oder wenn vorher 10 Sekunden abgelaufen sind. Wir wechseln dann automatisch zu der Seite `rot_videos.html`:

```
function wechseln() {
    window.location = "rot_videos.html"
}
```

```
</script>
</body>
</html>
```

## 6.7 rot\_videos.html

Da während eines Spiels mit Sicherheit nicht alle Videos eines Teams angesprochen werden – schließlich kann man ja beispielsweise nicht gleichzeitig gewinnen und verlieren – und diese im Spiel ja auch nur mit  $200 \times 150$  Pixeln angezeigt werden, stellen wir auf der Seite `rot_videos.html` (und der entsprechenden blauen Seite) alle Videos nacheinander in ihrer ganzen epischen Breite und Schönheit mit  $800 \times 600$  Pixeln vor (Abbildung 6.4).

## Halbwertszeit



Gewonnen

Abbildung 6.4: Vorstellung der Videos

Den mit dem Kopf der Animationsseite `<head>` identischen Kopf der Seite stellen wir hier nicht nochmals dar.

Im `<body>` der Seite öffnen wir eine dreizeilige, einspaltige Tabelle, fügen den Teamnamen als Überschrift ein

```
<body>
  <table class="galerie">
    <tr>
      <td>
        <h1 id="team_name"></h1>
      </td>
    </tr>
```

und laden das erste Video:

```
<tr>
  <td>
    <video
      id="video"
      src="teams/rot/videos/1.mp4"
      width="800"
      height="600"
      onended="naechstes_video_starten()"
      onerror="kein_video_mehr()">
    </video>
  </td>
</tr>
```

Mit dem `onended`-Attribut lassen wir jedes Video an seinem Ende ein Unterprogramm aufrufen, das seinen Nachfolger anstartet. Das `onerror`-Attribut definiert das Unterprogramm, das aufgerufen wird, wenn kein weiteres Video mehr vorhanden ist.

In der dritten Tabellenzeile stellen wir den Untertitel des Videos dar:

```
<tr>
  <td>
    <div id="untertitel"></div>
  </td>
</tr>
</table>
```

Das JavaScript

```
<script>
```

legt in seinem Hauptprogramm den Teamnamen fest

```
team_name.innerHTML = spiel.rot.name
```

setzt den Videoindex auf das erste Video

```
var video_index = 1
```

definiert die Liste der Untertitel

```
var untertitel_liste = [
  "Mine",
  "Bande",
  "Treibstoffmangel",
  "Gewonnen",
  "Verloren",
  "Tanken",
  "Video 7",
  "Video 8",
  "Video 9",
  "Video 10"
]
```

und startet das Unterprogramm zum Darstellen des ersten Videos:

```
naechstes_video_starten()
```

Im Unterprogramm `naechstes_video_starten` bestimmen wir aus dem Videoindex den Dateinamen des aktuellen Videos

```
function naechstes_video_starten() {  
    video.src = "teams/rot/videos/" + video_index + ".mp4"
```

und starten es an:

```
video.play()
```

Außerdem lesen wir den aktuellen Videountertitel aus der Liste und stellen ihn dar:

```
untertitel.innerHTML = untertitel_liste[video_index - 1]
```

Schließlich erhöhen wir den Videoindex, damit beim nächsten Aufruf des Unterprogrammes das nächste Video dargestellt wird:

```
video_index += 1  
}
```

Das Unterprogramm `kein_video_mehr` wird aufgerufen, wenn das `onerror`-Attribut des Videos-Objektes feuert. Dies ist beispielsweise dann der Fall, wenn kein Video mit dem Namen `7.mp4` gefunden werden kann. Wir rufen dann beispielsweise die Spielseite auf:

```
function kein_video_mehr() {  
    window.location = "spiel.html"  
}
```

```
</script>  
</body>  
</html>
```

# **Teil III**

## **Schnitte und Kollisionen**

# A Kollision zweier Kreise

In diesem Abschnitt wollen wir berechnen, ob – und wenn ja, wo – sich zwei Kreise, die sich mit jeweils konstanter Geschwindigkeit in der Ebene bewegen, berühren.

Dazu reicht es interessanterweise nicht aus, nur zu untersuchen, ob sich die Geraden, auf denen sich die Kreise bewegen, schneiden. In Abbildung A.1 bewegt sich der rote Kreis mit dem Radius  $r_1$  und dem Mittelpunkt  $M_1$  auf der Geraden, die durch die beiden Punkte  $A$  und  $B$  verläuft. Dabei befindet er sich zum (normierten) Zeitpunkt  $\lambda = 0$  im Anfangspunkt  $A$  und zum Zeitpunkt  $\lambda = 1$  im Endpunkt  $B$ . Entsprechend bewegt sich der blaue Kreis mit dem Radius  $r_2$  und dem Mittelpunkt  $M_2$  auf der Geraden, die durch die beiden Punkte  $C$  und  $D$  verläuft. Er befindet sich zum Zeitpunkt  $\lambda = 0$  im Anfangspunkt  $C$  und zum Zeitpunkt  $\lambda = 1$  im Endpunkt  $D$ . In Abbildung A.1 sehen wir die Momentaufnahme der Bewegung zum Zeitpunkt  $\lambda = 0.54$ .

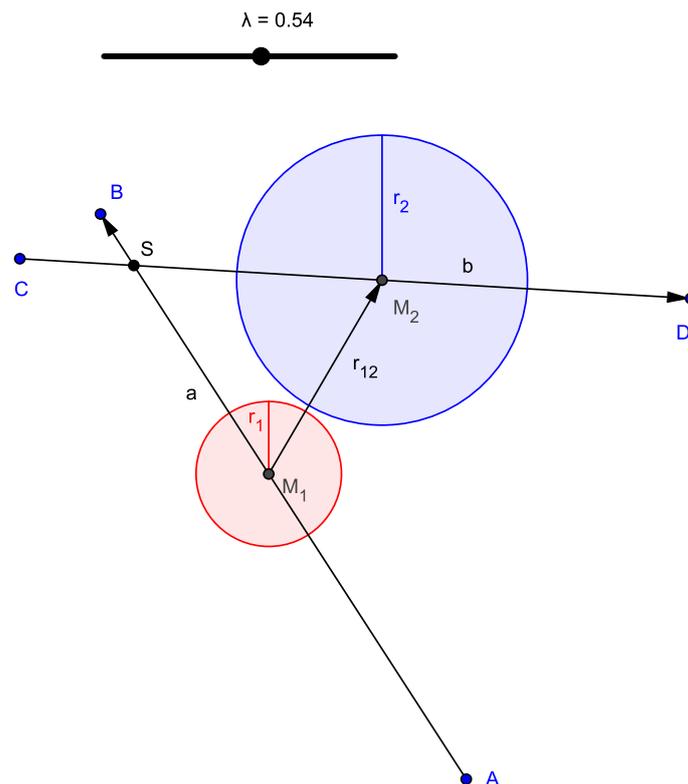


Abbildung A.1: Keine Kollision, obwohl sich die Geraden im Punkt  $S$  schneiden

Wie wir deutlich erkennen, schneiden sich die beiden Bewegungsgeraden im Punkt  $S$ .

Trotzdem berühren sich die beiden Kreise zu keinem Zeitpunkt. Wenn der Mittelpunkt des blauen Kreises (relativ früh) den Geradenschnittpunkt passiert, befindet sich der rote Kreis noch weit vom Geradenschnittpunkt entfernt in der Nähe seines Anfangspunktes  $A$ . Wenn dann (relativ spät) der rote Kreis den Geradenschnittpunkt passiert, befindet sich der blaue Kreis schon weit vom Geradenschnittpunkt entfernt in der Nähe seines Endpunktes  $D$ . Näher als zu dem in Abbildung A.1 dargestellten Zeitpunkt kommen sich die Kreise niemals.

In der in Abbildung A.2 und Abbildung A.3 dargestellten Anordnung der Anfangs- und Endpunkte hingegen berühren sich die Kreise zweimal. Zum ersten Mal treffen sie zum Zeitpunkt  $\lambda=0.56$  (Abbildung A.2), aufeinander und dann verlassen sie einander zum Zeitpunkt  $\lambda=0.94$  (Abbildung A.3) wieder [20].

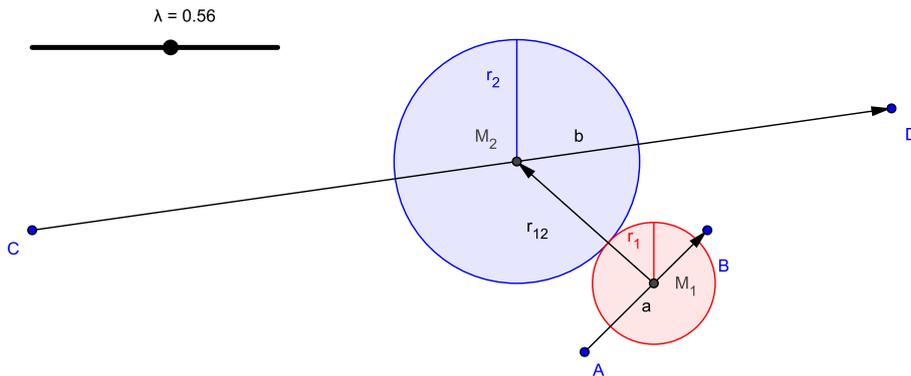


Abbildung A.2: Erster Berührungspunkt zum Zeitpunkt  $\lambda=0.56$

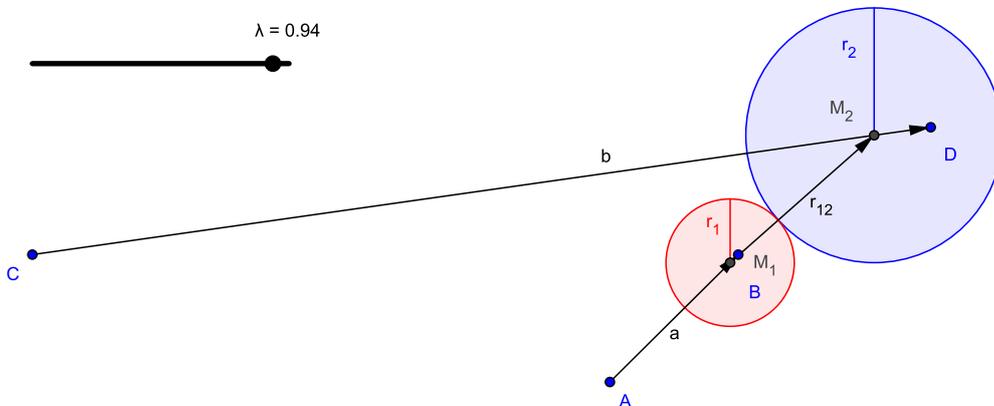


Abbildung A.3: Zweiter Berührungspunkt zum Zeitpunkt  $\lambda=0.94$

Zwischen den beiden Berührzeitpunkten überlappen sich die Kreise. Unser Ziel ist es nun, die Zeitpunkte  $\lambda_1$  und  $\lambda_2$  der beiden Berührungen und die Orte der Mittelpunkte beider Kreise zu den Berührzeitpunkten zu berechnen. Dazu definieren wir die Gleichung der ersten Geraden in Parameterform:

$$\mathbf{o}_1 = \mathbf{o}_A + \lambda \cdot \mathbf{a} \tag{A.1}$$

Dabei sind

- $\mathbf{o}_1$  der Ortsvektor<sup>1</sup> zum Mittelpunkt  $M_1$  des ersten Kreises auf der ersten Geraden
- $\mathbf{o}_A$  der Ortsvektor zum Anfangspunkt  $A$  der ersten Geraden
- $\mathbf{a} = \mathbf{o}_B - \mathbf{o}_A$  der Richtungsvektor der ersten Geraden vom Anfangspunkt  $A$  zum Endpunkt  $B$
- $\lambda$  der (Zeit-)Parameter der ersten (und zweiten) Geraden

Die zweite Gerade definieren wir entsprechend:

$$\mathbf{o}_2 = \mathbf{o}_C + \lambda \cdot \mathbf{b} \quad (\text{A.2})$$

Dabei sind

- $\mathbf{o}_2$  der Ortsvektor zum Mittelpunkt  $M_2$  des zweiten Kreises auf der zweiten Geraden
- $\mathbf{o}_C$  der Ortsvektor zum Anfangspunkt  $C$  der zweiten Geraden
- $\mathbf{b} = \mathbf{o}_D - \mathbf{o}_C$  der Richtungsvektor der zweiten Geraden vom Anfangspunkt  $C$  zum Endpunkt  $D$
- $\lambda$  der (Zeit-)Parameter der (ersten und) zweiten Geraden

Wenn wir  $\lambda$  als (normierte) Zeit auffassen, können wir die Richtungsvektoren  $\mathbf{a}$  und  $\mathbf{b}$  als die Geschwindigkeitsvektoren interpretieren, mit denen sich die Kreise auf ihren jeweiligen Geraden bewegen. Wir suchen nun also das gemeinsame  $\lambda$ , an dem sich beide Kreise berühren. Dazu berechnen wir den Vektor  $\mathbf{r}_{12}$ , der die beiden Kreismittelpunkte miteinander verbindet, als Differenz der Mittelpunktsortsvektoren:

$$\mathbf{r}_{12} = \mathbf{o}_2 - \mathbf{o}_1 \quad (\text{A.3})$$

Wenn wir jetzt die beiden Geradengleichungen Gleichung (A.1) und Gleichung (A.2) in Gleichung (A.3) einsetzen, erhalten wir:

$$\mathbf{r}_{12} = \mathbf{o}_C + \lambda \mathbf{b} - (\mathbf{o}_A + \lambda \mathbf{a}) = \mathbf{o}_C - \mathbf{o}_A + \lambda (\mathbf{b} - \mathbf{a}) \quad (\text{A.4})$$

Zur Vereinfachung definieren wir die beiden Vektordifferenzen in Gleichung (A.4) als neue Differenzvektoren  $\mathbf{e}$  und  $\mathbf{f}$ :

$$\begin{aligned} \mathbf{e} &= \mathbf{o}_C - \mathbf{o}_A \\ \mathbf{f} &= \mathbf{b} - \mathbf{a} \end{aligned} \quad (\text{A.5})$$

Der Mittelpunktabstandsvektor vereinfacht sich dann zu:

---

<sup>1</sup>Vektoren schreiben wir hier und im Folgenden **fett** und nicht *kursiv*. Der Skalar  $a$  ist also der Betrag des Vektors  $\mathbf{a}$ .

$$\mathbf{r}_{12} = \mathbf{e} + \lambda \mathbf{f} \quad (\text{A.6})$$

Wenn sich die beiden Kreise berühren, ergibt sich der skalare Abstand der beiden Mittelpunkte einerseits als Betrag  $r_{12}$  des Vektors  $\mathbf{r}_{12}$  und andererseits als Summe der beiden Kreisradien (vgl. Abbildung A.2):

$$|\mathbf{r}_{12}| = r_{12} = r_1 + r_2 \quad (\text{A.7})$$

Um nun das  $\lambda$  zum Berührzeitpunkt in Abhängigkeit von bekannten Größen zu berechnen, verwenden wir folgenden Satz:

Das Quadrat des Betrags eines Vektors  $\mathbf{x}$  ist gleich dem Skalarprodukt des Vektors mit sich selbst:

$$|\mathbf{x}|^2 = x^2 = \left( \sqrt{x_1^2 + x_2^2 + \dots} \right)^2 = x_1^2 + x_2^2 + \dots = \mathbf{x} \cdot \mathbf{x} \quad (\text{A.8})$$

Wir wenden Gleichung (A.8) auf Gleichung (A.7) an, setzen Gleichung (A.6) ein und erhalten:

$$|\mathbf{r}_{12}|^2 = (\mathbf{e} + \lambda \mathbf{f}) \cdot (\mathbf{e} + \lambda \mathbf{f}) = r_{12}^2 \quad (\text{A.9})$$

Das Skalarprodukt können wir ausmultiplizieren und sortieren:

$$\lambda^2 \mathbf{f} \cdot \mathbf{f} + 2\lambda \mathbf{e} \cdot \mathbf{f} + \mathbf{e} \cdot \mathbf{e} = r_{12}^2 \quad (\text{A.10})$$

Da die einzelnen Skalarprodukte in Gleichung (A.10) natürlich Skalare sind, können wir sie weiter vereinfachen, indem wir folgende Ersetzungen durchführen:

$$\begin{aligned} \alpha &= \mathbf{f} \cdot \mathbf{f} \\ \beta &= \mathbf{e} \cdot \mathbf{f} \\ \gamma &= \mathbf{e} \cdot \mathbf{e} - r_{12}^2 \end{aligned} \quad (\text{A.11})$$

Gleichung (A.10) wird dann zu einer einfachen quadratischen Gleichung in  $\lambda$

$$\alpha \lambda^2 + 2\beta \lambda + \gamma = 0$$

die nach der vereinfachten Mitternachtsformel [21] genau dann reelle Lösungen hat, wenn die Diskriminante  $\delta$  positiv ist:

$$\delta = \beta^2 - \alpha \gamma > 0 \quad (\text{A.12})$$

Die beiden Lösungen lauten dann:

$$\lambda_{1,2} = \frac{-\beta \pm \sqrt{\delta}}{\alpha} \quad (\text{A.13})$$

Die beiden Zeitpunkte  $\lambda_1$  und  $\lambda_2$  können wir schließlich in Gleichung (A.1) und Gleichung (A.2) einsetzen, um die Orte der Kreismittelpunkte bei Berührung zu berechnen.

### A.0.1 Beispiel

Die Zahlenwerte für die in Abbildung A.2 und Abbildung A.3 dargestellte Situation [20] lauten:

$$\begin{aligned}\mathbf{o}_A &= \begin{bmatrix} 10 \\ 1 \end{bmatrix} \\ \mathbf{o}_B &= \begin{bmatrix} 12 \\ 3 \end{bmatrix} \\ \mathbf{o}_C &= \begin{bmatrix} 1 \\ 3 \end{bmatrix} \\ \mathbf{o}_D &= \begin{bmatrix} 15 \\ 5 \end{bmatrix} \\ r_1 &= 1 \\ r_2 &= 2\end{aligned}$$

Die Richtungsvektoren ergeben sich dann zu:

$$\begin{aligned}\mathbf{a} &= \mathbf{o}_B - \mathbf{o}_A = \begin{bmatrix} 12 \\ 3 \end{bmatrix} - \begin{bmatrix} 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix} \\ \mathbf{b} &= \mathbf{o}_D - \mathbf{o}_C = \begin{bmatrix} 15 \\ 5 \end{bmatrix} - \begin{bmatrix} 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 14 \\ 2 \end{bmatrix}\end{aligned}$$

Nach Gleichung (A.5) berechnen wir die Differenzvektoren

$$\begin{aligned}\mathbf{e} &= \mathbf{o}_C - \mathbf{o}_A = \begin{bmatrix} 1 \\ 3 \end{bmatrix} - \begin{bmatrix} 10 \\ 1 \end{bmatrix} = \begin{bmatrix} -9 \\ 2 \end{bmatrix} \\ \mathbf{f} &= \mathbf{b} - \mathbf{a} = \begin{bmatrix} 14 \\ 2 \end{bmatrix} - \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 12 \\ 0 \end{bmatrix}\end{aligned}$$

und nach Gleichung (A.7) die Länge des Mittelpunktabstandsvektors:

$$r_{12} = r_1 + r_2 = 1 + 2 = 3$$

Jetzt können wir nach Gleichung (A.11) die Koeffizienten der quadratischen Gleichung ermitteln:

$$\begin{aligned}\alpha &= \mathbf{f} \cdot \mathbf{f} = \begin{bmatrix} 12 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} 12 \\ 0 \end{bmatrix} = 144 \\ \beta &= \mathbf{e} \cdot \mathbf{f} = \begin{bmatrix} -9 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 12 \\ 0 \end{bmatrix} = -108 \\ \gamma &= \mathbf{e} \cdot \mathbf{e} - r_{12}^2 = \begin{bmatrix} -9 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} -9 \\ 2 \end{bmatrix} - 3^2 = 76\end{aligned}$$

Die quadratische Gleichung in  $\lambda$  lautet also:

$$144\lambda^2 - 108\lambda + 76 = 0$$

Um zu überprüfen, ob es überhaupt reelle Berührungspunkte gibt, untersuchen wir die Diskriminante nach Gleichung (A.12):

$$\delta = \beta^2 - \alpha\gamma = (-108)^2 - 144 \cdot 76 = 720$$

Da die Diskriminante positiv ist, gibt es zwei Berührzeitpunkte:

$$\begin{aligned}\lambda_{1,2} &= \frac{-\beta \pm \sqrt{\delta}}{\alpha} = \frac{-(-108) \pm \sqrt{720}}{144} \\ \lambda_1 &= \frac{108 - \sqrt{720}}{144} = 0.5637 \\ \lambda_2 &= \frac{108 + \sqrt{720}}{144} = 0.9363\end{aligned}$$

Die Mittelpunkte der Kreise zu den Berührzeitpunkten finden wir durch Einsetzen der Berührzeitpunkte in die Geradengleichungen (Gleichung (A.1) und Gleichung (A.2)):

$\lambda = \lambda_1$  :

$$\begin{aligned}\mathbf{o}_{11} &= \mathbf{o}_A + \lambda_1 \cdot \mathbf{a} = \begin{bmatrix} 10 \\ 1 \end{bmatrix} + 0.5637 \cdot \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 11.1273 \\ 2.1273 \end{bmatrix} \\ \mathbf{o}_{21} &= \mathbf{o}_C + \lambda_1 \cdot \mathbf{b} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} + 0.5637 \cdot \begin{bmatrix} 14 \\ 2 \end{bmatrix} = \begin{bmatrix} 8.8913 \\ 4.1273 \end{bmatrix}\end{aligned}$$

$\lambda = \lambda_2$  :

$$\begin{aligned}\mathbf{o}_{12} &= \mathbf{o}_A + \lambda_2 \cdot \mathbf{a} = \begin{bmatrix} 10 \\ 1 \end{bmatrix} + 0.9363 \cdot \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 11.8727 \\ 2.8727 \end{bmatrix} \\ \mathbf{o}_{22} &= \mathbf{o}_C + \lambda_2 \cdot \mathbf{b} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} + 0.9363 \cdot \begin{bmatrix} 14 \\ 2 \end{bmatrix} = \begin{bmatrix} 14.1087 \\ 4.8727 \end{bmatrix}\end{aligned}$$

Diese beiden Berührungspunkte können wir sehr schön veranschaulichen, wenn wir in [20] die beiden berechneten Berührzeitpunkte einstellen.

# B Kollision eines Kreises mit einer Geraden

Bevor wir die Kollision eines Kreises mit einer Geraden analysieren, wollen wir als mathematisches Werkzeug die Projektion eines Vektors auf einen anderen Vektor (B.1) und die Darstellung einer Geraden in Hessescher Normalform (B.2) verstehen. Die eigentliche Kollisionsanalyse findet dann in B.3 statt.

## B.1 Vektorprojektion

Um den in Abbildung B.1 dargestellten Projektionsvektor<sup>1</sup>  $\mathbf{b}_a$  von  $\mathbf{b}$  auf  $\mathbf{a}$  zu berechnen, ermitteln wir zuerst seine Länge  $b_a = |\mathbf{b}_a|$ .

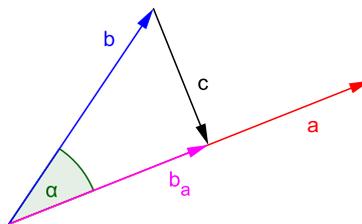


Abbildung B.1: Projektionsvektor  $\mathbf{b}_a$  des Vektors  $\mathbf{b}$  auf den Vektor  $\mathbf{a}$

Dazu berücksichtigen wir, dass das Skalarprodukt zweier Vektoren gleich dem Produkt der beiden Beträge mit dem Kosinus des eingeschlossenen Winkels ist:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \cdot |\mathbf{b}| \cdot \cos \alpha = a \cdot b \cdot \cos \alpha \quad (\text{B.1})$$

Der Kosinus ist andererseits der Quotient von Ankathete und Hypotenuse im rechtwinkligen Dreieck in Abbildung B.1:

$$\cos \alpha = \frac{b_a}{b} \quad (\text{B.2})$$

Wenn wir Gleichung (B.2) in Gleichung (B.1) einsetzen, erhalten wir eine weitere Veranschaulichung des Skalarproduktes als Produkt der Länge  $b_a$  des Projektionsvektors mit

---

<sup>1</sup>Wir schreiben hier und im folgenden Projektionsvektoren, indem wir den Vektor, auf den projiziert wird, als Index verwenden.

der Länge  $a$  des Vektors, auf den projiziert wird:

$$\mathbf{a} \cdot \mathbf{b} = a \cdot b \cdot \frac{b_a}{b} = a \cdot b_a \quad (\text{B.3})$$

Gleichung (B.3) lösen wir nach der gesuchten Länge des Projektionsvektors auf:

$$b_a = \frac{\mathbf{a} \cdot \mathbf{b}}{a} \quad (\text{B.4})$$

Ein Vektor  $\mathbf{x}$  ist bekanntlich gleich dem Produkt seiner Länge  $x$  mit seinem Einheitsvektor  $\mathbf{x}^0$ , der selbst eine Länge von eins besitzt und nur die Richtung (und Orientierung) angibt:

$$\mathbf{x} = x \cdot \mathbf{x}^0 \quad (\text{B.5})$$

Da nun der gesuchte Projektionsvektor  $\mathbf{b}_a$  die gleiche Richtung wie der Vektor  $\mathbf{a}$  besitzt, können wir statt seines Einheitsvektors  $\mathbf{b}_a^0$  auch den Einheitsvektor  $\mathbf{a}^0$  verwenden, den wir gemäß Gleichung (B.5) erhalten, indem wir  $\mathbf{a}$  durch seinen Betrag  $a$  teilen:

$$\mathbf{b}_a = b_a \cdot \mathbf{b}_a^0 = b_a \cdot \mathbf{a}^0 = b_a \frac{\mathbf{a}}{a} \quad (\text{B.6})$$

Wenn wir jetzt noch die Länge des Projektionsvektors aus Gleichung (B.4) in Gleichung (B.6) einsetzen, erhalten wir:

$$\mathbf{b}_a = \frac{\mathbf{a} \cdot \mathbf{b}}{a} \cdot \frac{\mathbf{a}}{a} = \frac{\mathbf{a} \cdot \mathbf{b}}{a \cdot a} \cdot \mathbf{a} = \frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{a} \cdot \mathbf{a}} \cdot \mathbf{a} \quad (\text{B.7})$$

In der letzten Umwandlung in Gleichung (B.7) haben wir die in Gleichung (A.8) beschriebene Tatsache verwendet, dass wir statt des Betragsquadrates auch das Skalarprodukt eines Vektors mit sich selbst verwenden können. Die letzte Form von Gleichung (B.7) hat den Charme, dass wir nur noch zwei Skalarprodukte berechnen (was in MATLAB ja komfortabel mit dem Befehl `dot` möglich ist), die beiden Skalare durcheinander dividieren und den sich daraus ergebenden Skalar mit dem Vektor  $\mathbf{a}$  multiplizieren müssen.<sup>2</sup>

## B.2 Hessesche Normalform

In der Hesseschen Normalform können wir die in Abbildung B.2 dargestellte grüne Gerade über ihren Abstand  $d$  vom Ursprung  $O$  und ihren vom Ursprung weg zeigenden Einheitsnormalenvektor  $\mathbf{n}$  beschreiben.

<sup>2</sup>Natürlich können wir in Gleichung (B.7) nicht einzelne Vektoren „herauskürzen“. Die Skalarprodukte in Zähler und Nenner sind ja keine „normalen“ Produkte. Sie binden stärker als die Division und müssen daher unbedingt zuerst ausgeführt werden. Allgemein müssen wir bei mehreren „Produkten“ von Vektoren genau definieren, welches die Skalarprodukte sind. Ein Ausdruck wie  $\mathbf{a} \cdot \mathbf{b} \cdot \mathbf{c}$  ist streng genommen ohne Klammern überhaupt nicht definiert. Einerseits ist  $(\mathbf{a} \cdot \mathbf{b}) \cdot \mathbf{c}$  ein Vektor in Richtung  $\mathbf{c}$  (das Skalarprodukt  $\mathbf{a} \cdot \mathbf{b}$  wirkt hier nur als zusätzlicher skalarer Faktor); andererseits ist  $\mathbf{a} \cdot (\mathbf{b} \cdot \mathbf{c})$  ein ganz anderer Vektor in Richtung  $\mathbf{a}$ , bei dem das Skalarprodukt  $\mathbf{b} \cdot \mathbf{c}$  als skalarer Faktor fungiert.

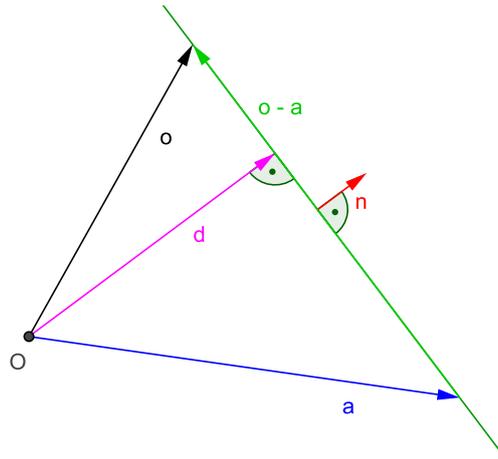


Abbildung B.2: Hessesche Normalform

Dazu definieren wir den schwarzen, allgemeinen, variablen Ortsvektor  $\mathbf{o}$  zur Geraden und einen blauen Ortsvektor  $\mathbf{a}$  zu einem beliebigen aber festen Punkt auf der Geraden. Der grüne Differenzvektor  $\mathbf{o} - \mathbf{a}$  ist dann ein Richtungsvektor der Geraden. Da der Einheitsnormalenvektor  $\mathbf{n}$  senkrecht auf der Geraden und damit auf dem Differenzvektor steht und das Skalarprodukt zweier senkrecht aufeinander stehender Vektoren verschwindet, gilt:

$$(\mathbf{o} - \mathbf{a}) \cdot \mathbf{n} = 0$$

Wir multiplizieren aus, sortieren und erhalten:

$$\mathbf{o} \cdot \mathbf{n} = \mathbf{a} \cdot \mathbf{n} \quad (\text{B.8})$$

In Gleichung (B.3) haben wir gezeigt, dass sich das Skalarprodukt  $\mathbf{a} \cdot \mathbf{b}$  auch als Produkt der Länge  $b_a$  des Projektionsvektors  $\mathbf{b}_a$  mit der Länge  $a$  des Vektors  $\mathbf{a}$ , auf den projiziert wird, schreiben lässt. Dies bedeutet, dass wir das Skalarprodukt der rechten Seite von Gleichung (B.8) umschreiben können

$$\mathbf{o} \cdot \mathbf{n} = a_n \cdot n \quad (\text{B.9})$$

wobei  $a_n$  der Betrag des Projektionsvektors  $\mathbf{a}_n$  ist, der sich ergibt, wenn wir  $\mathbf{a}$  auf  $\mathbf{n}$  projizieren. In Abbildung B.2 wird deutlich, dass die Projektion von  $\mathbf{a}$  auf  $\mathbf{n}$  dem Abstandsvektor  $\mathbf{d}$  entspricht, der ja auch senkrecht auf der Geraden steht und damit kollinear zu  $\mathbf{n}$  ist:<sup>3</sup>

$$\mathbf{a}_n = \mathbf{d}$$

Der Betrag  $a_n$  des Projektionsvektors  $\mathbf{a}_n$  ist dann also gleich dem Betrag  $d$  des Abstandsvektors  $\mathbf{d}$ . Wenn wir jetzt noch beachten, dass der Einheitsnormalenvektor  $\mathbf{n}$  definitionsgemäß einen Betrag von  $n = 1$  besitzt, vereinfacht sich Gleichung (B.9) zu:

$$\mathbf{o} \cdot \mathbf{n} = d \quad (\text{B.10})$$

<sup>3</sup>Der Abstandsvektor  $\mathbf{d}$  ist auch ein Normalenvektor der Geraden. Der Einheitsnormalenvektor  $\mathbf{n}$  ist damit der normierte Abstandsvektor:  $\mathbf{n} = \frac{\mathbf{d}}{d}$

In der Hesseschen Normalform können wir den Abstand eines Punktes zur Geraden besonders einfach berechnen. Dazu verschieben wir die Gerade mit dem Ursprungsabstand  $d$  und dem Einheitsnormalenvektor  $\mathbf{n}$  parallel zu sich selbst in den Punkt  $P$ . Wir erhalten dann die in Abbildung B.3 grün gestrichelt dargestellte Gerade, die natürlich noch den gleichen Normalenvektor, aber einen anderen Abstand vom Ursprung hat.

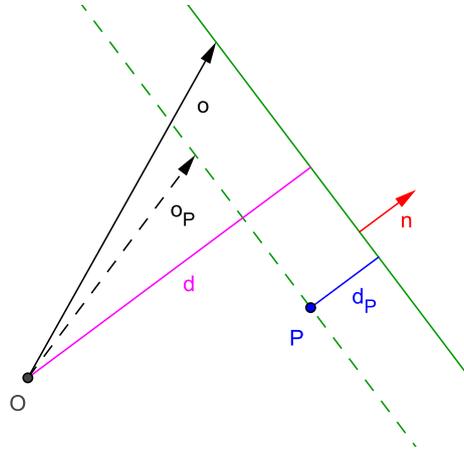


Abbildung B.3: Abstand des Punktes  $P$  von der Geraden  $\mathbf{o} \cdot \mathbf{n} = d$

Da wir die Gerade ja genau um den gesuchten Abstand  $d_P$  des Punktes von der Geraden verschoben haben, hat die verschobene Gerade den Abstand  $d - d_P$  vom Ursprung und besitzt damit gemäß Gleichung (B.10) die Gleichung:

$$\mathbf{o}_P \cdot \mathbf{n} = d - d_P \quad (\text{B.11})$$

Gleichung (B.11) können wir nach dem gesuchten Abstand auflösen und erhalten:

$$d_P = d - \mathbf{o}_P \cdot \mathbf{n} \quad (\text{B.12})$$

Da  $\mathbf{o}_P$  dabei der allgemeine Ortsvektor zu einem beliebigen Punkt auf der verschobenen Geraden ist, können wir jetzt den Ortsvektor zu jedem Punkt der Geraden (also auch zum Punkt  $P$  selbst) für  $\mathbf{o}_P$  in Gleichung (B.12) einsetzen, um seinen Abstand zur Geraden zu bestimmen. Da die Differenz auf der rechten Seite von Gleichung (B.12) positiv oder negativ sein kann, erhalten wir freundlicherweise auch noch die Information, ob sich der Punkt  $P$  und der Ursprung  $O$  auf der gleichen Seite der Gerade ( $d_P > 0$ ) oder auf unterschiedlichen Seiten befinden ( $d_P < 0$ ).

### B.2.1 Beispiele

Der in Abbildung B.2 dargestellte Abstandsvektor  $\mathbf{d}$  hat folgende Werte:

$$\mathbf{d} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$$

Seine Länge beträgt also:

$$d = |\mathbf{d}| = \sqrt{4^2 + 3^2} = 5$$

Den Einheitsnormalenvektor  $\mathbf{n}$  erhalten wir dann als normierten Abstandsvektor:

$$\mathbf{n} = \frac{\mathbf{d}}{d} = \frac{\begin{bmatrix} 4 \\ 3 \end{bmatrix}}{5} = \begin{bmatrix} 0.8 \\ 0.6 \end{bmatrix}$$

Die Hessesche Normalform der Geraden lautet daher:

$$\mathbf{o} \cdot \begin{bmatrix} 0.8 \\ 0.6 \end{bmatrix} = 5$$

Im folgenden wollen wir die Abstände dreier Punkt von dieser Geraden berechnen.

Wenn wir beispielsweise den in Abbildung B.2 verwendeten Ortsvektor  $\mathbf{a} = \begin{bmatrix} 7 \\ -1 \end{bmatrix}$  für  $\mathbf{o}_P$  in Gleichung (B.12) einsetzen, erhalten wir einen verschwindenden Abstand:

$$d_P = 5 - \begin{bmatrix} 7 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} 0.8 \\ 0.6 \end{bmatrix} = 5 - (5.6 - 0.6) = 0$$

Der Punkt liegt also (natürlich) auf der Geraden.

Der in Abbildung B.3 eingezeichnete Punkt besitzt den Ortsvektor  $\mathbf{o}_P = \begin{bmatrix} 4 \\ 1 \end{bmatrix}$ . Seinen Abstand berechnen wir also zu:

$$d_P = 5 - \begin{bmatrix} 4 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 0.8 \\ 0.6 \end{bmatrix} = 5 - (3.2 + 0.6) = 5 - 3.8 = 1.2$$

Da der Abstand positiv ist, liegt  $P$  – wie in Abbildung B.3 deutlich sichtbar – auf der gleichen Geradenseite wie der Ursprung.

Wenn wir einen Punkt auf der anderen Seite der Geraden verwenden, beispielsweise  $\mathbf{o}_P = \begin{bmatrix} 8 \\ 2 \end{bmatrix}$ , erhalten wir erwartungsgemäß einen negativen Abstand:

$$d_P = 5 - \begin{bmatrix} 8 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 0.8 \\ 0.6 \end{bmatrix} = 5 - (6.4 + 1.2) = 5 - 7.6 = -2.6$$

## B.3 Kollisionsanalyse

Nachdem wir jetzt die notwendigen Werkzeuge beherrschen, können wir damit die Kollision eines Kreises mit einer Geraden analysieren.

In Abbildung B.4 ist ein blauer Kreis mit dem Radius  $r$  dargestellt, dessen Mittelpunkt  $M$  sich geradlinig mit dem Geschwindigkeitsvektor  $\mathbf{v}$  bewegt. Außerdem sehen wir in Abbildung B.4 eine unbewegliche rote Gerade, die durch ihren (senkrechten) Abstand  $d$  vom Ursprung  $O$  (eingezeichnet im Punkt  $B$ ) und ihren Einheitsnormalenvektor  $\mathbf{n}$ , der senkrecht auf der Geraden steht und vom Ursprung weg zeigt (eingezeichnet im Punkt  $F$ ), definiert ist.

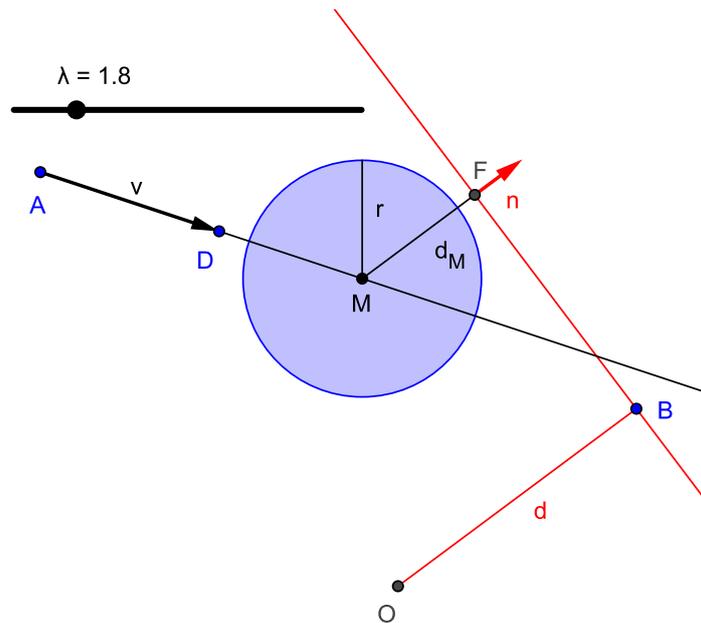


Abbildung B.4: Kollision eines Kreises mit einer Geraden

Wir suchen nun den Zeitpunkt  $\lambda$  und den Ort (des Kreismittelpunktes), wenn der Kreis die Gerade gerade berührt. Dazu beschreiben wir die Bewegung des Kreismittelpunktes  $M$  als Geradengleichung:

$$\mathbf{o}_M = \mathbf{o}_A + \lambda \cdot \mathbf{v} \quad (\text{B.13})$$

Dabei sind

- $\mathbf{o}_M$  der Ortsvektor zum Kreismittelpunkt  $M$
- $\mathbf{o}_A$  der Ortsvektor zum Anfangspunkt  $A$ , bei dem sich der Kreismittelpunkt zum Zeitpunkt  $\lambda = 0$  befindet
- $\lambda$  der normierte Zeitparameter, der während der Bewegung von  $0$  bis  $\infty$  läuft
- $\mathbf{v} = \mathbf{o}_D - \mathbf{o}_A$  der Geschwindigkeitsvektor, mit dem sich der Kreis auf der Geraden bewegt. Zum Zeitpunkt  $\lambda = 1$  befindet sich der Kreismittelpunkt gerade im Punkt  $D$ .

Wenn die Gerade in Hessescher Normalform gegeben ist, können wir den Abstand  $d_m$  des Kreismittelpunktes von der Geraden berechnen, indem wir den Ortsvektor zum Kreismittelpunkt in Gleichung (B.12) einsetzen:

$$d_M = d - \mathbf{o}_M \cdot \mathbf{n} \quad (\text{B.14})$$

Für den Kreismittelpunktsortsvektor setzen wir nun die in Gleichung (B.13) definierte Geradengleichung in Gleichung (B.14) ein

$$\begin{aligned} d_M &= d - (\mathbf{o}_A + \lambda \cdot \mathbf{v}) \cdot \mathbf{n} \\ &= d - \mathbf{o}_A \cdot \mathbf{n} - \lambda \cdot \mathbf{v} \cdot \mathbf{n} \end{aligned} \quad (\text{B.15})$$

und lösen Gleichung (B.15) nach  $\lambda$  auf:

$$\lambda = \frac{d - d_M - \mathbf{o}_A \cdot \mathbf{n}}{\mathbf{v} \cdot \mathbf{n}} \quad (\text{B.16})$$

Wir suchen jetzt das  $\lambda_1$ , bei dem der Kreis die Gerade erstmalig berührt. In diesem Fall ist der Kreismittelpunktsabstand  $d_M$  gerade gleich dem Kreisradius  $r$ :

$$\lambda_1 = \frac{d - r - \mathbf{o}_A \cdot \mathbf{n}}{\mathbf{v} \cdot \mathbf{n}} \quad (\text{B.17})$$

In B.2 haben wir gezeigt, dass der Abstand für Punkte auf der anderen Seite der Geraden negativ ist. Wir können daher auch den Berührungspunkt auf der anderen Seite der Geraden berechnen, indem wir für  $d_M$  in Gleichung (B.16) nicht  $r$  sondern  $-r$  einsetzen:

$$\lambda_2 = \frac{d + r - \mathbf{o}_A \cdot \mathbf{n}}{\mathbf{v} \cdot \mathbf{n}} \quad (\text{B.18})$$

Die Kreismittelpunkte zu den Berührzeitpunkten erhalten wir schließlich, indem wir  $\lambda_1$  und  $\lambda_2$  in Gleichung (B.13) einsetzen.

### B.3.1 Beispiel

In Abbildung B.4 haben wir folgende Zahlenwerte verwendet:

$$\begin{aligned} d &= 5 \\ \mathbf{n} &= \begin{bmatrix} 0.8 \\ 0.6 \end{bmatrix} \\ r &= 2 \\ \mathbf{o}_A &= \begin{bmatrix} -6 \\ 7 \end{bmatrix} \\ \mathbf{v} &= \begin{bmatrix} 3 \\ -1 \end{bmatrix} \end{aligned}$$

Damit berechnen wir nach Gleichung (B.17)

$$\lambda_1 = \frac{5 - 2 - \begin{bmatrix} -6 \\ 7 \end{bmatrix} \cdot \begin{bmatrix} 0.8 \\ 0.6 \end{bmatrix}}{\begin{bmatrix} 3 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} 0.8 \\ 0.6 \end{bmatrix}} = \frac{3 - (-4.8 + 4.2)}{2.4 - 0.6} = \frac{3 + 0.6}{1.8} = \frac{3.6}{1.8} = 2$$

und nach Gleichung (B.18):

$$\lambda_2 = \frac{5 + 2 + 0.6}{1.8} = \frac{7.6}{1.8} = \frac{38}{9} = 4.\bar{2}$$

Um die Ortsvektoren zu den Kreismittelpunkten zum Zeitpunkt der Berührung zu erhalten, setzen wir  $\lambda_1$  und  $\lambda_2$  in Gleichung (B.13) ein:

$$\begin{aligned} \mathbf{o}_{M1} &= \mathbf{o}_A + \lambda_1 \cdot \mathbf{v} = \begin{bmatrix} -6 \\ 7 \end{bmatrix} + 2 \cdot \begin{bmatrix} 3 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 5 \end{bmatrix} \\ \mathbf{o}_{M2} &= \mathbf{o}_A + \lambda_2 \cdot \mathbf{v} = \begin{bmatrix} -6 \\ 7 \end{bmatrix} + 4.\bar{2} \cdot \begin{bmatrix} 3 \\ -1 \end{bmatrix} = \begin{bmatrix} 6.\bar{6} \\ 2.\bar{7} \end{bmatrix} \end{aligned}$$

# Literaturverzeichnis

- [1] Hochschule Bremen. (2014) Internationaler Studiengang Luftfahrtssystemtechnik und -management B.Eng. [Online]. Available: <http://www.hs-bremen.de/internet/de/studium/stg/ilst/>
- [2] Wikipedia. (2014) Gaußsche Summenformel. [Online]. Available: [http://de.wikipedia.org/wiki/Gau%C3%9Fsche\\_Summenformel](http://de.wikipedia.org/wiki/Gau%C3%9Fsche_Summenformel)
- [3] J. J. Buchholz. (2014) Spaceballs Projektstagebücher. [Online]. Available: <http://www.fbm.hs-bremen.de/projektstagebuch/>
- [4] The LyX Team. (2014) LyX - The Document Processor. [Online]. Available: <http://www.lyx.org/>
- [5] Wix.com. (2014) Erstellen Sie eine eigene kostenlose Homepage. [Online]. Available: [wix.com](http://wix.com)
- [6] Wikipedia. (2014) Document Object Model. [Online]. Available: [http://de.wikipedia.org/wiki/Document\\_Object\\_Model](http://de.wikipedia.org/wiki/Document_Object_Model)
- [7] ——. (2014) Extensible Markup Language. [Online]. Available: [http://de.wikipedia.org/wiki/Extensible\\_Markup\\_Language](http://de.wikipedia.org/wiki/Extensible_Markup_Language)
- [8] The Mathworks. (2014) function\_handle (@). [Online]. Available: [http://www.mathworks.de/de/help/matlab/ref/function\\_handle.html](http://www.mathworks.de/de/help/matlab/ref/function_handle.html)
- [9] Wikipedia. (2014) Explizites Euler-Verfahren. [Online]. Available: [http://de.wikipedia.org/wiki/Explizites\\_Euler-Verfahren](http://de.wikipedia.org/wiki/Explizites_Euler-Verfahren)
- [10] ——. (2014) JavaScript Object Notation. [Online]. Available: [http://de.wikipedia.org/wiki/JavaScript\\_Object\\_Notation](http://de.wikipedia.org/wiki/JavaScript_Object_Notation)
- [11] Q. Fang. (2014) JSONlab: a toolbox to encode/decode JSON files in MATLAB/Octave. MatlabCentral File Exchange. [Online]. Available: <http://www.mathworks.com/matlabcentral/fileexchange/33381-jsonlab--a-toolbox-to-encode-decode-json-files-in-matlab-octave>
- [12] D. Hull. (2008, April) Array of structures vs Structures of arrays. The Mathworks. [Online]. Available: <http://blogs.mathworks.com/pick/2008/04/22/matlab-basics-array-of-structures-vs-structures-of-arrays/>
- [13] Wikipedia. (2014) HTML5. [Online]. Available: <http://de.wikipedia.org/wiki/HTML5>

- 
- [14] ——. (2014) Cascading Style Sheets. [Online]. Available: [http://de.wikipedia.org/wiki/Cascading\\_Style\\_Sheets](http://de.wikipedia.org/wiki/Cascading_Style_Sheets)
- [15] ——. (2014) JavaScript. [Online]. Available: <http://de.wikipedia.org/wiki/JavaScript>
- [16] ——. (2014) UTF-8. [Online]. Available: <http://de.wikipedia.org/wiki/UTF-8>
- [17] J. Seidelin. (2009) HTML5 Canvas Cheat Sheet. [Online]. Available: <http://blog.nihilogic.dk/2009/02/html5-canvas-cheat-sheet.html>
- [18] J. Jenkov. (2014) HTML5 Canvas: Animation. [Online]. Available: <http://tutorials.jenkov.com/html5-canvas/animation.html>
- [19] Autodesk. (2014) Autodesk. [Online]. Available: <http://www.autodesk.de/>
- [20] J. J. Buchholz. (2014) Kollision Kreis Kreis. [Online]. Available: <http://tube.geogebra.org/student/m153548>
- [21] Wikipedia. (2014) Quadratische Gleichung. [Online]. Available: [http://de.wikipedia.org/w/index.php?title=Quadratische\\_Gleichung](http://de.wikipedia.org/w/index.php?title=Quadratische_Gleichung)
- [22] Google. (2014) Google Fonts. [Online]. Available: <https://www.google.com/fonts>
- [23] J. J. Buchholz. (2014) Kollision Kreis Gerade. [Online]. Available: <http://www.geogebra.org/student/m154094>