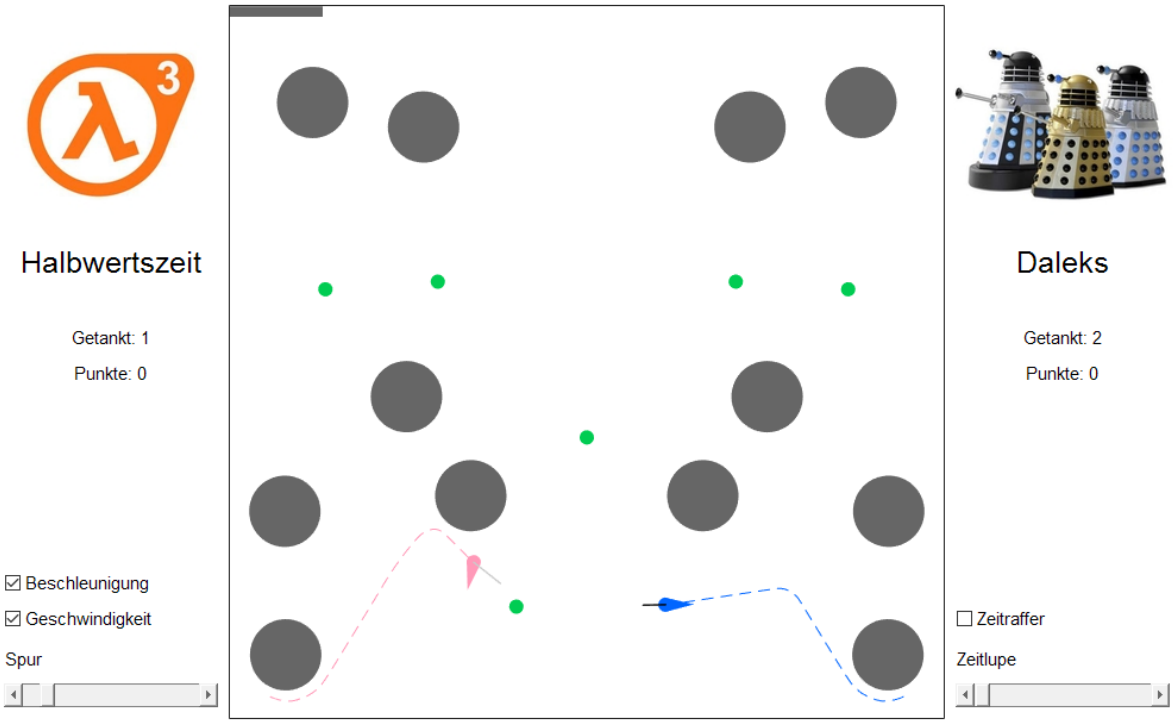


Space Chase 2018

Jörg J. Buchholz

14. Juni 2018

Inhaltsverzeichnis

Teil I

Bedienungsanleitung

1 Einführung

Space Chase ist ein Lehrprojekt im Rahmen des Modulpools [1] der Hochschule Bremen. Die Teilnehmerinnen[1] teilen sich dazu in Projektteams mit bis zu sieben Mitarbeiterinnen 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. Die sich daraus ergebenen Tabelle (Rangliste) fließt zu 50 % in die Note ein; die verbleibenden 50 % ergeben sich aus der Dokumentation, dem Intro und der Webseite (Originalität, Design, Korrektheit, ...).

Im Spiel werden – wie Abbildung 1.1 zu sehen – die beweglichen Spaceballs (rot und blau) genau wie die ortsfesten Tankstellen (grün) und Minen (grau) als ausgefüllte Kreise in der Ebene dargestellt. Das Spielfeld wird durch Banden begrenzt. Das Ziel des Spiels ist es, den Gegner mit mehr Treibstoff zu berühren. Dazu fliegt der Spaceball möglichst schnell möglichst viele Tankstellen an und weicht dabei Minen, Banden und gegebenenfalls dem Gegner aus.

[1] Dieses Dokument verwendet das generische Femininum. Männer sind automatisch mitgemeint.

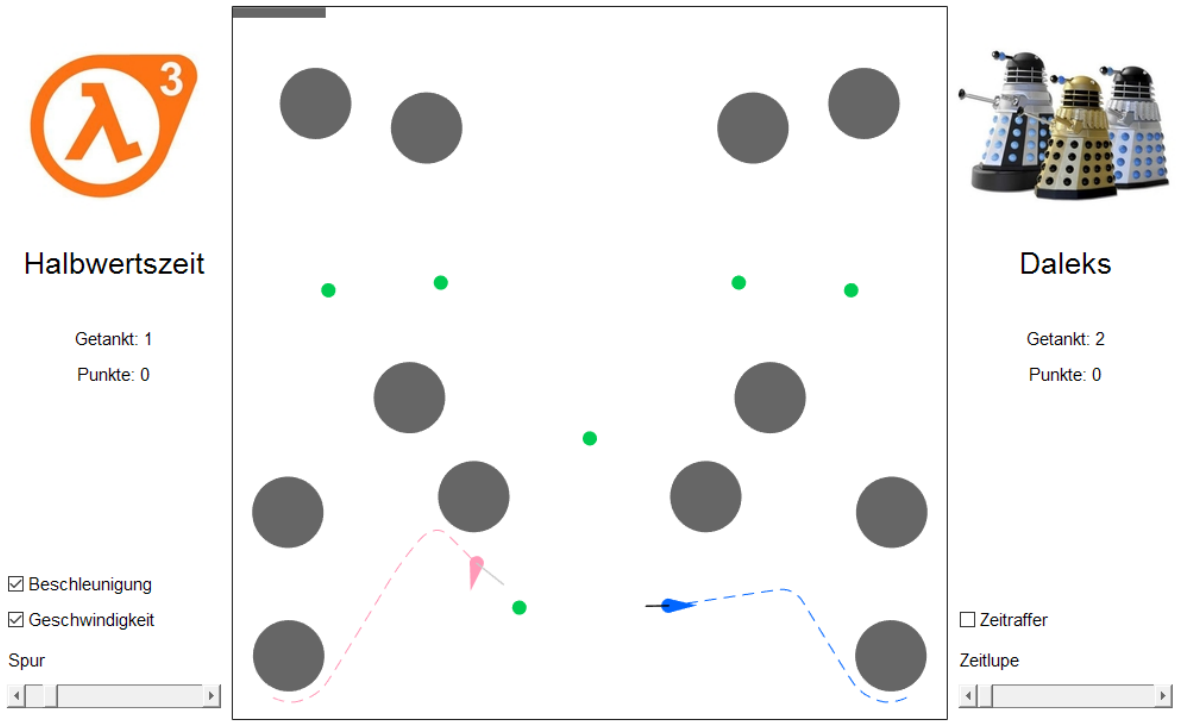


Abbildung 1.1: Space Chase

In Abbildung 1.1 ist beispielsweise zu erkennen, dass beide Spaceballs schon jeweils eine Tankstelle abgefrühstückt haben (die jetzt natürlich nicht mehr zu sehen ist) und der blaue Spaceball, im Gegensatz zum roten, auch schon die zweite Tankstelle besucht hat. Da der rote Spaceball weniger Tankpunkte hat, ist seine Farbe aufgehellt und er scheint sich schon auf der Flucht vor dem blauen zu befinden, da er von der nächstliegendsten Tankstelle weg beschleunigt. Vermutlich hat er nicht genügend Selbstbewusstsein, um zu erkennen, dass er die Tankstelle sicherlich noch vor dem roten erreichen würde.

An der Spur ist zu erkennen, dass der blaue Spaceball offensichtlich bei jeder Tankstelle signifikant abbremst, während der rote Spaceball über das Ziel hinausschießt, was durchaus von Vorteil sein kann, wenn die nächste Tankstelle in gleicher Richtung liegt.

2 Spielablauf

Jeder Spaceball verfügt zur Steuerung über eine Schubdüse, über die ihn seine KI in jede Richtung beschleunigen kann.

Das Spiel endet, sobald eine der drei Bedingungen eintritt:

- Beide Spaceballs treffen mit unterschiedlicher Treibstoffmenge aufeinander.
- Ein Spaceball berührt eine Mine oder eine Bande.
- Das Ende der Spielzeit (60 Sekunden) ist erreicht.

Wenn ein Spaceball mehr Treibstoff als der Gegner besitzt und den Gegner berührt, gewinnt er das Spiel und erhält **einen** Punkt in der Tabelle. Wenn beide Spaceballs einander mit gleicher Treibstoffmenge berühren, geschieht überhaupt nichts und das Spiel geht weiter. Ebenfalls **einen** Punkt erhält ein Spaceball, dessen Gegner eine Mine oder Bande berührt. Nach Ablauf der Spielzeit erhalten beide Teams **keinen** Punkt. Träges Herumlungern und Campen zahlt sich also nicht aus.

Damit sich ein unterlegener Spaceball nicht einfach nur hinter einer Mine verstecken kann, verschwindet, nachdem alle Tankstellen aufgebraucht sind, alle drei Sekunden genau die Mine, die dem Verteidiger am nächsten liegt.

2.1 Tankstellen, Minen und Banden

Sobald ein Spaceball eine Tankstelle berührt, nimmt der Spaceball eine Treibstoffeinheit auf und die Tankstelle verschwindet. Der Tankvorgang selbst benötigt dabei keine Zeit.[1]

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

Wie in Abbildung 1.1 zu sehen ist, werden die jeweils gleichgroßen 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

[1] Der Tankvorgang findet also – im Gegensatz zu Version 2014 – innerhalb eines einzigen Simulationsschrittes statt. Durch das Tanken verändert sich – ebenfalls im Gegensatz zu V 2014 – die Größe (bzw. die simulierte Masse) des Spaceballs nicht. Außerdem verbraucht das Beschleunigen – ebenfalls im Gegensatz zu V 2014 – keinen Treibstoff; es kann also „Bleifuß" geflogen werden. Vielleicht hilft es in V 2018, sich statt des Treibstoffes lieber Mana, Energie, Ammo, Gesundheit, ... vorzustellen, die der Spaceball beim Überfliegen der „Tankstellen" schlagartig aufnimmt und die nicht für den Antrieb benötigt werden.

unteren Ecken (rot links, blau rechts) starten – anfänglich immer die gleichen Chancen. Um sicherzustellen, dass, wenn alle Tankstellen aufgebraucht sind, immer ein Spaceball mehr Treibstoff besitzt als sein Gegner, ist die Anzahl der Tankstellen ungerade. Eine Tankstelle wird dabei genau auf der senkrechten Symmetrieachse zufällig platziert.

2.2 Anzeigen

Neben den Tankstellen, den Minen und den Spaceballs selbst werden auf dem Spielfeld (in der Mitte von Abbildung 1.1) die Spuren der bisher zurückgelegten Wege (rote und blaue gestrichelte Linien), die Geschwindigkeitsvektoren (Länge und Richtung der schwarzen Linien) und die Schub- bzw. Beschleunigungsvektoren[2] (rote und blaue Dreiecke) dargestellt. Diese zusätzlichen Informationen können durch Anklicken der entsprechenden Auswahlfelder links unten neben dem Spielfeld an- bzw. abgeschaltet werden. Mit dem Spur-Schieberegler lässt sich die Länge der Spur zwischen 0 % (linker Anschlag) und 100 % (rechter Anschlag) einstellen.

Links und rechts oben neben dem Spielfeld werden die Logos und die Namen des Teams darstellt. Darunter wird für jeden Spaceball jeweils angezeigt, wie viele Tankeinheiten er schon aufgenommen hat. Zusätzlich wird die Tanksituation dadurch verdeutlicht, dass ein Spaceball mit weniger Treibstoff ungesättigter (blasser, heller) dargestellt wird. So hat in der abgebildeten Situation der blaue Spaceball gerade seine zweite Tankstelle besucht; er wird daher in sattem Blau dargestellt. Der rote Spaceball hat hingegen erst eine Treibstoffration aufgenommen und erscheint deshalb in einem blasseren Rotton.

Rechts unten neben dem Spielfeld kann die Simulation durch den Zeitlupen-Schieberegler verlangsamt oder durch Anklicken des Zeitraffer-Auswahlfeldes beschleunigt werden.[3]

Alle Einstellungen können auch während der Simulation verändert werden.

[2] Die sichtbare Spitze des Dreiecks symbolisiert die Richtung eines Triebwerksstrahls. Der Beschleunigungsvektor hat dann natürlich die entgegengesetzte Orientierung.

[3] Matlab kann mit seiner `timer`-Klasse praktisch in Echtzeit simulieren. Dies hat während der Programmentwicklungsphase allerdings den lästigen Nachteil, dass die vom Zeitgeber aufgerufenen Unterprogramme keine detaillierten Fehlermeldungen ausgeben können.

Aus diesem Grund verzichten wir in dieser Simulationsumgebung auf Timerobjekte und lassen die Simulation stattdessen immer mit maximaler Geschwindigkeit und damit maximaler Prozessorlast laufen. HighTecFrontEndSuperGraphikGamerBoliden können ein 60-Sekundenspiel daher möglicherweise in wenigen Sekunden durchsimulieren und benötigen daher den Zeitlupenregler, um einzelne Taktikdetails überhaupt analysierbar zu machen. Langsame Rechner schaffen im Normalfall keine Echtzeit; der Zeitraffer kann hier bei der Strategieanalyse helfen, indem dann nicht mehr jedes Einzelbild tatsächlich dargestellt wird.

3 Regeln

- Jede Mitarbeiterin stellt an mindestens **zwei** Terminen seinen Projektstand in einer 30-minütigen Minipräsentation[1] dem Dozenten vor. Dazu tragen Sie dann bitte jeweils einen Terminvorschlag unter http://prof.red/contact/calendar.htm ein. Zusätzlich zu den Pflichtterminen können Sie natürlich jederzeit Hilfe- und Diskussionstermine vorschlagen.
- Jedes Team erstellt
  - die Matlab-Datei `beschleunigung`, in der die KI die momentane Beschleunigung(srichtung) definiert
  - die Matlab-Datei `team_daten`, in der der Name des Teams und die Namen und Aufgaben aller Mitarbeiterinnen definiert sind
  - für jede Mitarbeiterin eine JPG-Datei `mitarbeiter.jpg` mit einem Portrait (Kopfbild), auf dem die Mitarbeiterin zu erkennen ist
  - die WAV-Datei `intro.wav` für die Vorstellung des Teams
  - die JPG-Datei `Logo.jpg` als Teamlogo
  - eine ausführliche, in LaTeX (beispielsweise mit [2]) geschriebene Dokumentation des Projektes
  - eine Website, (beispielsweise bei [3][2]), auf der das Projekt in allen Details beschrieben ist
- Bleiben Sie trotz der Wettbewerbssituation bitte fair und hilfsbereit den anderen Teams gegenüber. Was spricht eigentlich gegen ein gelegentliches Freundschaftsspiel[3] ...?
- Abgabetermin der für das Turnier benötigten Dateien (s. o.) ist am 15.1.2019. Das Turnier findet dann am 22.1.2019 statt. Die Dokumentation und die Website[4] müssen bis zum 25.1.2019 fertig sein.
- Die Teamnote setzt sich schließlich folgendermaßen zusammen

[1] Das eigene Team darf dabei natürlich gerne anwesend sein.

[2] Trotz des für den deutschen Markt ziemlich schlampig recherchierten Namens einer der leistungsfähigsten Anbieter kostenloser Websites.

[3] Sie können Ihre `beschleunigung` mit dem Befehl `pcode` für ein Freundschaftsspiel compilieren, so dass Ihre Algorithmen für Ihren Gegner nicht sichtbar sind.

[4] Vielleicht möchten Sie auf der Website ja noch die Turnierspiele Ihrer KI einbinden.

  - 50 % Turnierrangliste
  - 25 % Dokumentation
  - 20 % Website
  - 5 % Portraits, Intro, Logo

- Verwenden Sie bitte nur numerische und keine symbolischen Berechnungen, da das tausendfache Aufrufen der *symbolic engine* in `beschleunigung` die Rechenzeit sehr stark vergrößert. Lösen Sie gegebenenfalls Gleichungen einmalig außerhalb von Space Chase und benutzen Sie den Ergebnisterm in `beschleunigung`. Die Verwendung symbolischer Ausdrücke wie `syms`, `solve`, `subs`, ... wird als kapitaler Fehler gewertet.
- Programmieren Sie bitte effizient und rechenzeitsparend. Ein „Einminutenspiel" darf auf den Rechnern im SI 52 maximal 300 Sekunden dauern. Verwenden Sie bitte **keine** MEX files [4].
- Wenn Ihre KI nicht nur taktisch reagieren, sondern auch strategisch planen soll, können Sie dazu persistente Variablen [5] verwenden, die zwischen Funktionsaufrufen ihren Wert behalten.
- Die Simulationsumgebung ist nicht wasserdicht. Sie könnten sie austricksen. Sie könnten beispielsweise die in der Struktur `spiel` vorhandenen Grafikhandles missbrauchen, um beliebige Objekte auf dem Spielfeld zu verändern (beispielsweise auch die Position eines Spaceballs). Widerstehen Sie bitte der Versuchung. Manipulationen dieser Art werden als Täuschungsversuche gewertet.

  Auch die `pcode` Funktion [6] ist leicht zu knacken:

  „The pcode function obfuscates your code files, it does not encrypt them. While the content in a .p file is difficult to understand, it should not be considered secure. It is not recommended that you P-code files to protect your intellectual property."

  Versuchen Sie es bitte nicht. Gönnen Sie sich Ihre eigenen Erfolgserlebnisse!

Teil II

Unter der Haube

4 Blockschaltbild

Abbildung 4.1 zeigt das komplette Space Chase-Blockschaltbild.

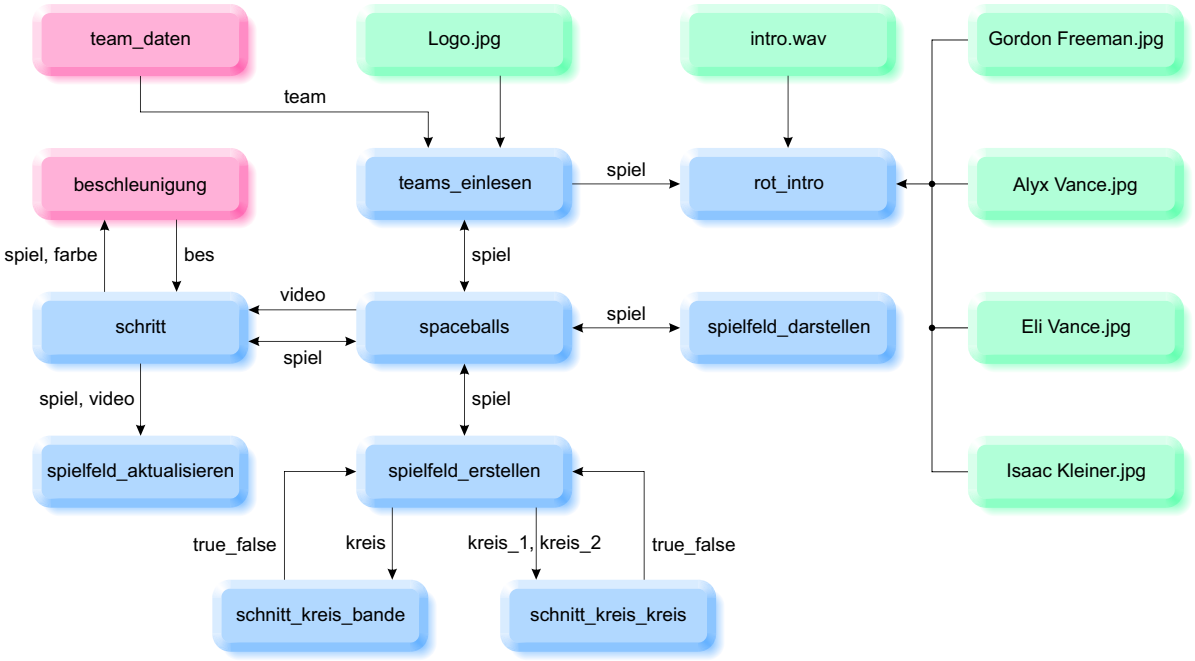


Abbildung 4.1: Blockschaltbild

Darin sind die Matlab-Systemdateien in Blau, die von jedem Team zu erstellenden Matlab-Dateien in Rot und die Mediendateien (Bilder und Sound) in Grün dargestellt.

Das zentral angeordnete `spaceballs`[1] ist das Hauptprogramm und kommuniziert mit den anderen blau dargestellten Unterprogrammen.

Als erstes wird die Struktur `spiel`, in der alle spielrelevanten Daten gespeichert sind, an das Unterprogramm `teams_einlesen` übergeben, das dann das von beiden Teams jeweils erstellte Unterprogramm `team_daten` ausführt. `team_daten` liefert Informationen über die Teamnamen und Mitarbeiterinnen in Form der Struktur `team` zurück, die `teams_einlesen` dann in `spiel` eintütet. Außerdem liest `teams_einlesen` die beiden Logo-Dateien und speichert sie ebenfalls in `spiel` ab. Schließlich übergibt `teams_einlesen` die aktualisierte Spielstruktur wieder zurück an `spaceballs`.

[1] 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, diese Daten in spiel einträgt und die Struktur wieder 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.

Im Unterprogramm spielfeld_darstellen, das von spaceballs als nächstes aufgerufen wird, öffnet Matlab das Darstellungsfenster und zeichnet dort alle Objekte (Texte, Auswahlfelder, Spaceballs, ...) ein.

Schließlich startet spaceballs seine Simulationsschleife, ruft in jedem Simulationsschritt das Unterprogramm schritt auf, in dem die eigentliche Simulation stattfindet und übergibt dabei die zuvor initialisierte Struktur spiel. Außerdem bekommt schritt über die Struktur `video` die Information, ob das aktuelle Spiel als Video aufgezeichnet werden soll. Nach jedem Simulationsschritt gibt schritt die aktualisierte Spielstruktur wieder an spaceballs zurück.

schritt ruft in jedem Simulationsschritt das von beiden Teams jeweils erstellte Unterprogramm beschleunigung auf, übergibt diesem den aktuellen Zustand des Spiels und die Farbe des Spaceballs des jeweiligen Teams. In beschleunigung haben die Teams jeweils ihre KI programmiert, die in jedem Zeitschritt aus dem momentanen Spielzustand den gewünschten Beschleunigungsvektor `bes` des eigenen Spaceballs berechnet und zurück gibt.

Am Ende jedes Simulationsschrittes übergibt schritt die aktualisierten Spieldaten zusammen mit der Videostruktur an spielfeld_aktualisieren, das damit alle grafischen Darstellungen auf dem Spielfeld (Positionen der Spaceballs, ...) aktualisiert. Wenn ein Video aufgezeichnet werden soll, speichert spielfeld_aktualisieren eine Kopie des aktuellen Fensters als Bild in die vorher geöffnete Videodatei.

Neben spaceballs gibt es mit rot_intro ein weiteres unabhängiges Hauptprogramm, das das Unterprogramm teams_einlesen verwendet, um an die Teamdaten und das Logo zu kommen. Außerdem liest rot_intro die Sounddatei intro.wav und die Bilder der Mitarbeiterinnen ein und stellt das Team dann in grafisch ansprechender Form vor.

5 Simulation

5.1 `spaceballs`

`spaceballs` ist das Hauptprogramm, das den Ablauf der Simulation steuert und dazu seine Unterprogramme aufruft. Als erstes löschen wir allerdings alle Variablen[1]

```
clear variables
```

schließen das im letzten Lauf geöffnete Darstellungsfenster

```
close all
```

und löschen das Kommandozeilenfenster:

```
clc
```

Mit

```
rng shuffle
```

initialisieren wir den Zufallsgenerator, so dass in jedem Simulationsdurchlauf ein anderes Spielfeld erzeugt wird.[2]

In den nächsten Zeilen definieren wir die für den geordneten Spielablauf notwendigen Konstanten in der Struktur `spiel`[3]:

```
spiel.dt = 1/100;
spiel.t_end = 60;
spiel.n_t = round (spiel.t_end/spiel.dt);
spiel.i_t = 0;

spiel.n_mine = 12; % Muss gerade sein! (Default: 12)
spiel.n_tanke = 9; % Muss ungerade sein! (Default: 9)
```

[1] Wir verwenden hier den Befehl `clear variables`, da der klassische Kahlschlag `clear all` auch alle Haltepunkte (*breakpoints*) löschen würde, die wir vielleicht zur Fehlersuche benötigen. Wenn es Probleme beim Übersetzen einzelner Unterprogrammen gibt, können wir immer noch manuell im Kommandozeilenfenster ein kleines `clear all` einwerfen.

[2] Alternativ haben wir die Möglichkeit, mit `rng (42)`, `rng (123456)`, ... zufällige aber reproduzierbare Spielfelder zu erzeugen. Dies ist immer dann sehr praktisch, wenn wir das gleiche Spielfeld mit unterschiedlichen Algorithmen testen wollen.

[3] Erklärungen der einzelnen Größen unter `spiel`.

```
spiel.kreis_radius = 0.075;
spiel.mine_radius = 0.05;
spiel.tanke_radius = 0.01;
spiel.spaceball_radius = 0.01;

spiel.bes = 0.1;

spiel.rot.getankt = 0;
spiel.blau.getankt = 0;

spiel.rot.punkte = 0;
spiel.blau.punkte = 0;

spiel.rot.ereignis = '';
spiel.blau.ereignis = '';

spiel.farbe.rot = hsv2rgb ([0.95 1 1]);
spiel.farbe.blau = hsv2rgb ([0.6 1 1]);
spiel.farbe.gruen = hsv2rgb ([0.4 1 0.8]);
spiel.farbe.hellrot = hsv2rgb ([0.95 0.4 1]);
spiel.farbe.hellblau = hsv2rgb ([0.6 0.4 1]);
spiel.farbe.grau = [0.4 0.4 0.4];
spiel.farbe.hellgrau = [0.8 0.8 0.8];

spiel.spur_anfangswert = 0.1*spiel.n_t;
spiel.zeitlupe_anfangswert = 0;
spiel.ges_checkbox_anfangswert = true;
spiel.bes_checkbox_anfangswert = true;
spiel.zeitraffer_checkbox_anfangswert = false;

spiel.rot.spur(spiel.n_t, 2) = 0;
spiel.blau.spur(spiel.n_t, 2) = 0;
```

In der nächste Zeile kann die Nutzerin bestimmen, ob während der Simulation ein Video[4] aufgezeichnet wird:

```
video.abspeichern = false;
```

Wenn dies der Fall ist

```
if video.abspeichern
```

[4]Die Videoaufzeichnung verballert ziemlich viel Rechenzeit. Wir sollten sie daher wirklich nur bei Bedarf explizit einschalten und nicht vergessen, sie danach auch wieder auszuschalten. Wenn Sie also den Eindruck haben, Ihre Simulation laufe zu langsam, überprüfen Sie bitte als erstes, ob Sie die Videoaufzeichnung auch wirklich ausgeschaltet haben. Die Entscheidung, ein Video aufzuzeichnen, speichern wir in der Struktur `video` und übergeben sie später herunter bis ins Unterprogramm `spielfeld_aktualisieren`, in dem die einzelnen Videobilder gegebenenfalls abgespeichert werden.

definieren wir die Videodatei, die Qualität und Bildfrequenz des Videos und öffnen den Kanal zur Videodatei:

```
  video.writer = VideoWriter ('spaceballs.avi');

  video.writer.Quality = 100;

  video.writer.FrameRate = 25;

  open (video.writer)

end
```

Wie im Blockschaltbild dargestellt, übergeben wir dann die Kontrolle an das Unterprogramm `teams_einlesen`, das die Logos, Namen und Aufgaben der Teams und Mitarbeiterinnen einliest und in der aktualisierten Struktur `spiel` zurückgibt:

```
spiel = teams_einlesen (spiel);
```

Wir übergeben dem Unterprogramm `spielfeld_erstellen` die Struktur `spiel` mit den bislang definierten Daten und lassen es dort die anfänglichen Positionen, Geschwindigkeiten und Beschleunigungen der Spaceballs, Tankstellen und Minen eintragen:

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

Da – im Gegensatz zu Version 2014 – jetzt Matlab selbst die Ad-hoc-Visualisierung der Simulation übernimmt, öffnen wir das Darstellungsfenster und zeichnen im Unterprogramm `spielfeld_darstellen` alle Objekte (Textfelder, Auswahlfelder, Schieberegler, Spaceballs, ...) in das Fenster:

```
spiel = spielfeld_darstellen (spiel);
```

Bevor wir die Simulation starten, initialisieren wir noch das Nutzerdatenfeld des Darstellungsfensters, das wir später verwenden möchten, um durch einen Mausklick in das Fenster die Simulation abzubrechen:

```
set (spiel.fenster_handle, 'UserData', true)
```

Nachdem wir die Zeitmessung angestoßen haben, um zu messen, wie lange die Simulation tatsächlich gerechnet hat

```
tic;
```

steigen wir jetzt in die Simulationsschleife ein. Um die Möglichkeit zu haben, die Simulation jederzeit geordnet[5] beenden zu können, fragen wir im Nutzerdatenfeld des Darstellungsfensters ab, ob irgendein Ereignis dort eine Simulationsabbruchforderung hinterlegt hat:

[5] Die Notbremse Strg-C geht ja auch immer, hinterlässt das System und die Daten aber möglicherweise in einem inkonsistenten Zustand.

```
while get (spiel.fenster_handle, 'UserData')
```

Wenn dies nicht der Fall ist, erhöhen wir den ganzzahligen Schrittzähler

```
  spiel.i_t = spiel.i_t + 1;
```

und rufen das Unterprogramm `schritt` auf, das einen einzelnen Simulationsschritt durchführt:[6]

```
  spiel = schritt (spiel, video);

end
```

Wenn die Simulation abgeschlossen ist (beispielsweise, weil ein Spaceball eine Bande berührt hat), messen wir die vergangene Rechenzeit

```
toc;
```

und räumen bei Bedarf

```
if video.abspeichern
```

das Videoobjekt auf. Dazu schließen wir den Kanal zur Videodatei, damit wir vom Betriebssystem aus auf sie zugreifen können

```
  close (video.writer);
```

und geben den Speicherplatz für das Videoobjekt wieder frei, um dem matlabinternen Speichermüllsammler (*garbage collection*) eine kleine Freude zu bereiten:

```
  delete (video.writer);

end
```

5.2 teams_einlesen

Das Unterprogramm `teams_einlesen` liest die Daten beider Teams ein und gibt sie in der aktualisierten Struktur `spiel` zurück:

```
function spiel = teams_einlesen (spiel)
```

Wir führen die folgenden Schritte nacheinander erst für das rote und dann für das blaue Team durch. Als erstes wechseln wir dazu in das Verzeichnis des roten Teams:

```
cd teams/rot
```

[6] `schritt` braucht neben den momentanen Spieldaten, die es später aktualisiert wieder zurückgibt, auch die Videostruktur, da sein eigenes Unterprogramm `spielfeld_aktualisieren` jeweils ein Bild des Darstellungsfensters abspeichert

und speichern den Funktionshandle [7] der dort[7] vorliegenden roten Beschleunigungsdatei in der Spielstruktur ab:

```
spiel.rot.beschleunigung_handle = @beschleunigung;
```

Als nächstes rufen wir das vom aktuellen Team erstellte Unterprogramm `team_daten` auf, das uns den Namen des Teams und die Namen und Aufgaben der Mitarbeiterinnen liefert:

```
rot_team = team_daten;
```

Wir speichern den Teamnamen

```
spiel.rot.name = rot_team.name;
```

und die Namen und Aufgaben der Mitarbeiterinnen in der Spielstruktur ab

```
spiel.rot.mitarbeiter = rot_team.mitarbeiter;
```

und lesen das im Medienordner liegende Logo des Teams ein:

```
spiel.rot.logo = imread ('media/logo.jpg');
```

Zum Schluss wechseln wir wieder hoch[8] in das Spaceballs-Basisverzeichnis:

```
cd ../..
```

Natürlich müssen wir dann die gleichen Schritte auch noch für das blaue Team durchführen:

```
cd teams/blau

spiel.blau.beschleunigung_handle = @beschleunigung;

blau_team = team_daten;

spiel.blau.name = blau_team.name;

spiel.blau.mitarbeiter = blau_team.mitarbeiter;

spiel.blau.logo = imread ('media/logo.jpg');

cd ../..
```

[7] Leider gibt es – ohne den Verzeichniswechsel – keine Möglichkeit, direkt den Funktionshandle eines in einem Unterverzeichnis liegenden Unterprogrammes zu ermitteln, oder? Die denkbare Alternative, die roten und blauen Unterverzeichnisse mit in den durchsuchten Pfad aufzunehmen, entfällt leider auch, da die Beschleunigungsunterprogramme ja beide den gleichen Namen besitzen.

[8] Zwei Punkte hintereinander (`..`) bezeichnen die noch aus guten alten DOS-Zeiten stammende Abkürzung für das übergeordnete Elternverzeichnis.

5.3 team_daten

Das Unterprogramm `team_daten` wird von jedem Team selbst mit den eigenen Teamdaten gefüllt und gibt die fest vorgegebene Struktur `team` zurück:

```
function team = team_daten
```

Als erstes wird der Name des Teams definiert

```
team.name = 'Halbwertszeit';
```

dann folgen die Namen und Aufgaben der beispielsweise vier[9] Mitarbeiterinnen in Form eines Strukturfeldes:

```
team.mitarbeiter(1).name = 'Gordon Freeman';
team.mitarbeiter(1).aufgabe = 'Angriff';

team.mitarbeiter(2).name = 'Alyx Vance';
team.mitarbeiter(2).aufgabe = 'Verteidigung';

team.mitarbeiter(3).name = 'Eli Vance';
team.mitarbeiter(3).aufgabe = 'Tanken';

team.mitarbeiter(4).name = 'Isaac Kleiner';
team.mitarbeiter(4).aufgabe = 'Minen';
```

Die fünfte (letzte) „Mitarbeiterin" ist das Logo, das wir hier nur deshalb als Mitarbeiterin auflisten, um uns bei der tabellarischen Vorstellung des Teams in `rot_intro` das Leben etwas zu erleichtern:

```
team.mitarbeiter(5).name = 'Logo';
team.mitarbeiter(5).aufgabe = '';
```

5.4 spielfeld_erstellen

Im Unterprogramm `spielfeld_erstellen` definieren wir die Größen und Anfangspositionen der kreisförmigen Spaceballs, Tankstellen und Minen. Dazu bekommt das Unterprogramm über seine Parameterliste die Struktur `spiel` übergeben, in der es Informationen über die Anzahl und Größen der Spielobjekte findet. Am Ende gibt es die Spielstruktur wieder zurück, in der dann die Spaceballs, Tankstellen und Minen beschrieben sind:

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

[9] Obwohl das Team hier aus vier Mitarbeiterinnen besteht, lässt die formale Syntax natürlich auch mehr oder weniger als vier Mitarbeiterinnen zu.

Die Gesamtzahl aller zu definierenden Kreise ergibt sich aus der Summe beider Spaceballs und aller Minen und Tankstellen:

```
n_kreis = 2 + 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 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 untere Ecke des Spielfelds mit einem kleinen Sicherheitsabstand[10] von den Banden:

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

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

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

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

Als nächstes erzeugen wir die erste Tankstelle, die genau auf der senkrechten Symmetrieachse des Spielfeldes liegen soll; ihre y-Koordinate soll aber natürlich zufällig sein:

```
kreis(3).pos = [ ...
  0.5, ...
  spiel.kreis_radius + rand*(1 - 2*spiel.kreis_radius)];

kreis(3).radius = spiel.kreis_radius;
```

Die y-Koordinate bestimmen wir dabei mit einer von der Zufallszahl `rand` abhängigen linearen Funktion, die dafür sorgt, dass die Tankstelle einen ausreichenden Abstand sowohl von der unteren als auch von der oberen Bande hat. Wenn `rand` nämlich den Wert null besitzt, hat die y-Koordinate ja den Wert

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

`spiel.kreis_radius + 0*(1 - 2*spiel.kreis_radius)`

und damit den unteren Grenzwert `spiel.kreis_radius`

Wenn `rand` hingegen den Wert eins besitzt, ergibt sich für die y-Koordinate

`spiel.kreis_radius + 1*(1 - 2*spiel.kreis_radius)`

was genau dem oberen Grenzwert `1 - spiel.kreis_radius` entspricht.

Als nächstes initialisieren wir den Kreisindex auf den Wert vier, da die ersten drei Indizes ja für die beiden Spaceballs und die erste Tankstelle reserviert sind und die nächste Tankstelle demnach den Index vier hat:

```
i_kreis = 4;
```

In einer Schleife über alle restlichen Kreise

```
while i_kreis < n_kreis
```

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

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

und einen vordefinierten Radius:

```
  kreis(i_kreis).radius = spiel.kreis_radius;
```

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[11] einen schon vorhandenen Kreis schneidet, um einander überlappende Minen oder Tankstellen zu verhindern. Dazu setzen wir einen binären Statusanzeiger zurück, der im Überlappungsfall 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))
```

[11] Den gespiegelten Kreis müssen wir nicht extra untersuchen, da sein Überlappungsverhalten in einer symmetrischen Welt natürlich dem seines Symmetriepartners entspricht.

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

```
      kreise_schneiden = true;

      break

    end

  end
```

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

akzeptieren wir den Kandidaten und seinen Symmetriepartner und erhöhen 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

```
spiel.rot.radius = kreis(1).radius;
spiel.rot.pos = kreis(1).pos;
```

und initialisieren seine Geschwindigkeit und Beschleunigung jeweils auf null:

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

Der zweite Kreis ist der blaue Spaceball:

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

Die nächsten Kreise sind die Tankstellen

```
spiel.tanke(1 : spiel.n_tanke) = kreis(3 : spiel.n_tanke + 2);
```

die alle einen festen vordefinierten Radius erhalten:

```
for i_tanke = 1 : spiel.n_tanke

  spiel.tanke(i_tanke).radius = spiel.tanke_radius;

end
```

Die restlichen Kreise sind die Minen, ebenfalls mit festen Radien:

```
spiel.mine(1 : spiel.n_mine) = kreis(spiel.n_tanke + 3 : n_kreis);

for i_mine = 1 : spiel.n_mine

  spiel.mine(i_mine).radius = spiel.mine_radius;

end
```

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

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 || ... % Untere Bande
  kreis.pos(1) + kreis.radius >= 1 || ... % Rechte Bande
  kreis.pos(2) + kreis.radius >= 1        % Obere 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 Rückgabevariable 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.

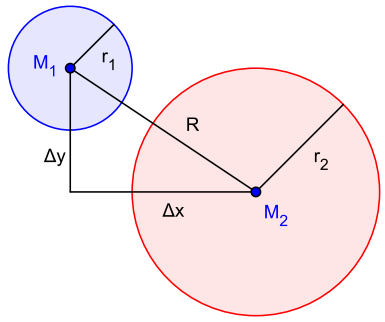


Abbildung 5.1: Schnitt zweier Kreise

Wenn dies der Fall ist

$$r_1 + r_2 \geqq R \tag{5.1}$$

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:

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

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, da alle Längen sowieso positiv sind und erhalten mit Gleichung (5.2):

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

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

  true_false = true;

else

  true_false = false;

end
```

5.7 spielfeld_darstellen

Im Unterprogramm `spielfeld_darstellen` schreiben und zeichnen wir alle sichtbaren Objekte (Logos, Texte, Auswahlfelder, Schieberegler, Spaceballs, Tanken, Minen, Geschwindigkeits-, Beschleunigungs- und Spurelemente, ...) in das Darstellungsfenster:

```
function spiel = spielfeld_darstellen (spiel)
```

Darstellungsfenster Als erstes definieren und öffnen wir das Darstellungsfenster:

```
spiel.fenster_handle = figure ( ...
  'Position', [100 100 1000 600], ...
  'Menu', 'none', ...
  'NumberTitle', 'off', ...
  'Name', 'Spaceballs', ...
  'Color', 'white' ...
  );
```

Dessen Größe ist mit 1000×600 Pixeln ein Kompromiss zwischen halbwegs erträglicher Simulationsgeschwindigkeit[12] und der Erkennbarkeit einzelner Objekte. Beim Öffnen des

[12] Die Darstellung hoch aufgelöster Grafikobjekte verbraucht signifikant mehr Rechenzeit. Es kann also durchaus sinnvoll sein, das Darstellungsfenster radikal zu verkleinern, wenn die Simulation zu langsam läuft. Und wer es etwas größer mag: Alle Objekte werden bei einer Vergrößerung des

Fensters schalten wir das matlabspezifische Menü und die Anzeige der Fensternummerierung ab und geben dem Fenster einen eigenen Titelnamen. Außerdem ersetzen wir das matlabspezifische Grau des Fensterhintergrunds durch Weiß. Den Fensterhandle speichern wir in der Spielstruktur ab, um später leicht auf das Fenster zugreifen zu können.

Logos Zur Darstellung des roten Logos erzeugen wir ein Achsensystem in der linken oberen Ecke des Fensters:

```
spiel.rot.logo_handle = axes ( ...
  'Position', [0/1000 400/600 200/1000 200/600] ...
  );
```

Die Position eines Achsensystems wird standardmäßig relativ zum Elternfenster in normalisierter Einheit (von 0.0 bis 1.0) angegeben. Die linke untere Ecke des Achsensystems hat hier also die Koordinaten $\left[\frac{0}{1000} \quad \frac{400}{600}\right]$, was bei einer Fenstergröße von 1000×600 Pixeln der absoluten Position von $\left[\frac{0}{1000} \cdot 1000 \quad \frac{400}{600} \cdot 600\right] = \left[0 \quad 400\right]$ Pixeln entspricht. Die Breite des Achsensystems beträgt $\frac{200}{1000}$ der Fensterbreite, was $\frac{200}{1000} \cdot 1000 = 200$ Pixeln entspricht. Die Höhe des Achsensystems beträgt $\frac{200}{600}$ der Fensterhöhe, was $\frac{200}{600} \cdot 600 = 200$ Pixeln entspricht. Grundsätzlich können wir also auch im Folgenden bei einer Standardfenstergröße von 1000×600 Pixeln im Zähler der Positionskomponenten jeweils die absoluten Größe in der Einheit Pixel ablesen.

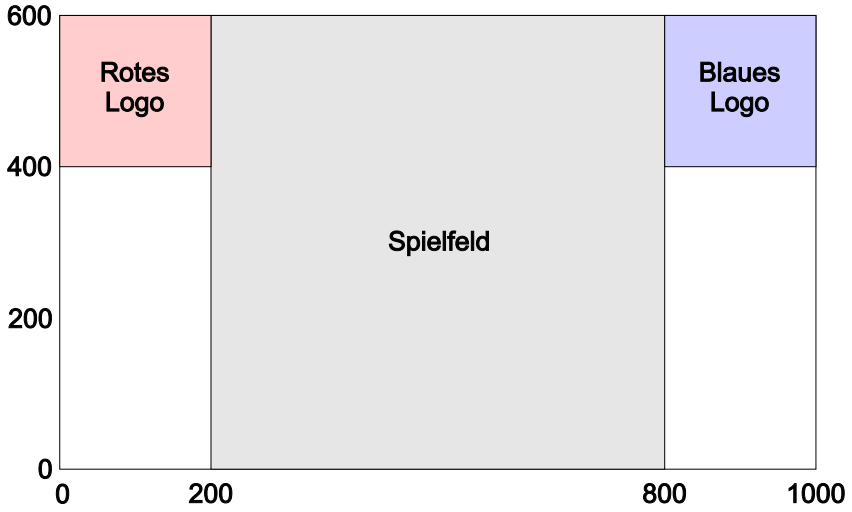


Abbildung 5.2: Position der Achsensysteme bei einer Fenstergröße von 1000×600

Fensters automatisch verlustfrei hoch skaliert. Nur die Logos werden als Rastergrafik (Pixelgrafik) natürlich nur bei einer Fenstergröße von 1000×600 in ihrer nativen Größe (200×200) dargestellt (Abbildung 5.2).

Auch das Auslagern des Darstellungsfensters auf einen zweiten Monitor ist möglich. So stellt beispielsweise `'Position', [-1100 700 1000 600]` das Fenster auf einem links neben dem Hauptmonitor stehenden Zweitmonitor dar.

Im roten Achsensystem stellen wir jetzt das rote Logo dar, das wir vorher in teams_einlesen eingelesen haben:

```
image (spiel.rot.logo, ...
  'Parent', spiel.rot.logo_handle ...
  )
```

Da der Befehl image dummerweise die Achsenskalierung sichtbar macht, schalten wir sie nachträglich wieder aus:

```
set (spiel.rot.logo_handle, ...
  'Visible', 'off' ...
  );
```

Die gleichen Schritte wiederholen wir für das blaue Logo

```
spiel.blau.logo_handle = axes ( ...
  'Position', [800/1000 400/600 200/1000 200/600] ...
  );

image (spiel.blau.logo, ...
  'Parent', spiel.blau.logo_handle ...
  )

set (spiel.blau.logo_handle, ...
  'Visible', 'off' ...
  );
```

dessen linke untere Ecke jetzt aber gemäß Abbildung 5.2 bei einer x-Koordinate von 800 Pixeln liegt.

Spielfeld Als nächstes zeichnen wir das Spielfeld:

```
spiel.spielfeld_handle = axes ( ...
  'ButtonDownFcn', ...
  'set (spiel.fenster_handle, ''UserData'', false)', ...
  'box', 'on', ...
  'XTick', [], ...
  'YTick', [], ...
  'Position', [200/1000 0/600 600/1000 600/600], ...
  'XLim', [0 1], ...
  'YLim', [0 1] ...
  );
```

In x-Richtung beginnt das quadratische Spielfeld bei Standardfenstergröße gemäß Abbildung 5.2 bei 200 Pixeln und hat eine Breite von 600 Pixeln. Seine Höhe entspricht der gesamten Fensterhöhe von 600 Pixeln.

Mit der box-Eigenschaft schalten wir die Umrandung des Spielfeldes ein, löschen aber mit XTick und YTick die Achsenskalierung und definieren feste normalisierte Achsengrenzen (XLim, YLim).

Außerdem definieren wir mit 'set (spiel.fenster_handle, "UserData", false)' eine Abbruchbedingung, die ausgelöst wird, wenn die Nutzerin (auf eine weiße Fläche) im Spielfeld klickt (ButtonDownFcn). In diesem Fall setzen wir das Nutzerdatenfeld[13] des Spielfeldes auf false, was in der Simulationsschleife von spaceballs dazu führt, dass die Simulation geordnet abgebrochen wird.

Zeitbalken Am oberen Rand des Spielfeldes zeigt, wie in Abbildung 1.1 zu sehen, ein grauer, rechteckförmiger Zeitbalken die vergangene Simulationszeit an:

```
spiel.zeit_handle = rectangle ( ...
  'Parent', spiel.spielfeld_handle, ...
  'EdgeColor', 'none', ...
  'FaceColor', spiel.farbe.grau ...
  );
```

Seine tatsächlichen Positionsdaten aktualisieren wir später in jedem Simulationsschritt im Unterprogramm spielfeld_aktualisieren.

Schieberegler Wir möchten der Nutzerin zwei Schieberegler anbieten, mit denen er die Länge der Spuren und die Geschwindigkeit der Zeitlupe anpassen kann.

Der Spurschieberegler hat rechts, links und unten einen Randabstand von 10 Pixeln und eine Höhe von 20 Pixeln.

```
spiel.spur_slider_handle = uicontrol ( ...
  'Style', 'slider', ...
  'Min', 0, ...
  'Max', spiel.n_t, ...
  'Value', spiel.spur_anfangswert, ...
  'Units', 'normalized', ...
  'Position', [10/1000 10/600 180/1000 20/600] ...
  );
```

Seine Extremwerte wählen wir so, dass die Nutzerin die Spur sowohl komplett ausschalten ('Min', 0) als auch den gesamten zurückgelegten Weg anzeigen lassen kann

[13] Die Kommunikation einer *callback function* mit einem Hauptprogramm ist nicht trivial, da beide in unterschiedlichen Speicherbereichen arbeiten. Am einfachsten findet der Datenaustausch daher über ein Objekt statt, auf das beide zugreifen können; dazu bietet sich das Nutzerdatenfeld eines grafischen Objekts an, das global allen Unterprogrammen zur Verfügung steht, die seinen Handle kennen. Wir schreiben, da wir nur diese eine Information übertragen wollen, direkt quick-and-dirty in das Nutzerdatenfeld; professionellerweise würden wir erst noch eine Struktur mit selbstsprechendem Namen erzeugen ...

('Max', spiel.n_t). Als Anfangseinstellung nutzen wir den in spaceballs definierten Anfangswert (spiel.spur_anfangswert).

Direkt über dem Schieberegler geben wir seine Bezeichnung als linksbündigen Text aus:

```
spiel.spur_text_handle = uicontrol ( ...
  'Style', 'text', ...
  'String', 'Spur', ...
  'Units', 'normalized', ...
  'BackgroundColor', 'white', ...
  'FontUnits', 'normalized ', ...
  'Fontsize', 0.5, ...
  'HorizontalAlignment', 'left', ...
  'Position', [10/1000 30/600 180/1000 30/600] ...
  );
```

Auch das Textfeld[14] hat links und rechts einen Randabstand von 10 Pixeln. Seine Höhe beträgt 30 Pixel und seine Unterkante beginnt genau über dem Schieberegler bei 30 Pixeln. Durch die Verwendung von normalisierten Fonteinheiten skaliert die Schrift bei einer Größenveränderung des Fensters stufenlos mit.

Den Zeitlupen-Schieberegler und seine Bezeichnung zeichnen wir auf ganz ähnliche Weise:

```
spiel.zeitlupe_slider_handle = uicontrol ( ...
  'Style', 'slider', ...
  'Min', 0, ...
  'Max', 1, ...
  'Value', spiel.zeitlupe_anfangswert, ...
  'Units', 'normalized', ...
  'Position', [810/1000 10/600 180/1000 20/600] ...
  );

spiel.zeitlupe_text_handle = uicontrol ( ...
  'Style', 'text', ...
  'String', 'Zeitlupe', ...
  'Units', 'normalized', ...
  'BackgroundColor', 'white', ...
  'FontUnits', 'normalized ', ...
  'Fontsize', 0.5, ...
  'HorizontalAlignment', 'left', ...
  'Position', [810/1000 30/600 180/1000 30/600] ...
  );
```

[14] Wir könnten Text auch mit dem Befehl text ausgeben. Dann müssten wir als Elternobjekt aber ein Achsensystem angeben und damit wäre die Berechnung der relativen Koordinaten des Textes innerhalb der relativen Koordinaten des Achsensystems sehr lästig. Durch die Verwendung eines uicontrol können wir hingegen die Textkoordinaten direkt relativ zum Darstellungsfenster angeben.

Sein Maximalwert von 1 bedeutet, dass die maximale Pause zwischen zwei Abtastschritten eine Sekunde beträgt. Ausgewertet wird die Stellung der Schieberegler in `spielfeld_aktualisieren`.

Auswahlfelder Mit drei Auswahlfeldern geben wir der Nutzerin die Möglichkeit, die Geschwindigkeits- und Beschleunigungsanzeige aus- und einen Zeitraffer einzuschalten. Das Geschwindigkeitsauswahlfeld zeichnen wir 10 Pixel oberhalb des Spur-Schiebereglertextes. Seine Unterkante liegt also bei 70 Pixeln und seine Höhe beträgt 30 Pixel:

```
spiel.ges_checkbox_handle = uicontrol ( ...
  'Style', 'checkbox', ...
  'Value', spiel.ges_checkbox_anfangswert, ...
  'Units', 'normalized', ...
  'BackgroundColor', 'white', ...
  'String', 'Geschwindigkeit', ...
  'FontUnits', 'normalized ', ...
  'Fontsize', 0.5, ...
  'Position', [10/1000 70/600 180/1000 30/600] ...
  );
```

Auch hier übernehmen wir den Startwert wieder aus `spaceballs`.

Das Beschleunigungsauswahlfeld platzieren wir direkt oberhalb des Geschwindigkeitsauswahlfelds:

```
spiel.bes_checkbox_handle = uicontrol ( ...
  'Style', 'checkbox', ...
  'Value', spiel.bes_checkbox_anfangswert, ...
  'Units', 'normalized', ...
  'BackgroundColor', 'white', ...
  'String', 'Beschleunigung', ...
  'FontUnits', 'normalized ', ...
  'Fontsize', 0.5, ...
  'Position', [10/1000 100/600 180/1000 30/600] ...
  );
```

Das Auswahlfeld für den Zeitraffer kommt schließlich nach rechts (bei 810 Pixeln) über den Zeitlupenschieberegler:

```
spiel.zeitraffer_checkbox_handle = uicontrol ( ...
  'Style', 'checkbox', ...
  'Value', spiel.zeitraffer_checkbox_anfangswert, ...
  'Units', 'normalized', ...
  'BackgroundColor', 'white', ...
  'String', 'Zeitraffer', ...
  'FontUnits', 'normalized ', ...
  'Fontsize', 0.5, ...
  'Position', [810/1000 70/600 180/1000 30/600] ...
  );
```

Texte Unterhalb jedes Logos geben wir vier Texte aus:

- Teamname
- Anzahl der besuchten Tankstellen
- Anzahl der Punkte
- Ereignis

Den Teamnamen setzen wir direkt unter[15] das Logo:

```
uicontrol ( ...
  'Style', 'text', ...
  'String', spiel.rot.name, ...
  'Units', 'normalized', ...
  'BackgroundColor', 'white', ...
  'FontUnits', 'normalized ', ...
  'Fontsize', 0.25, ...
  'Position', [10/1000 300/600 180/1000 100/600] ...
  );
```

Seine Höhe beträgt 100 Pixel, damit er nicht nur eine Zeile lang sein darf. Da er statisch ist, also später nicht mehr geändert wird, brauchen wir seinen Handle nicht abzuspeichern.

Darunter kommen dann bündig die jeweils 30 Pixel hohen restlichen roten Textfelder, die später mit der Anzahl der besuchten Tankstellen, der Anzahl der Punkte und den Ereignissen gefüllt werden:

```
spiel.rot.getankt_handle = uicontrol ( ...
  'Style', 'text', ...
  'Units', 'normalized', ...
  'BackgroundColor', 'white', ...
  'FontUnits', 'normalized ', ...
  'Fontsize', 0.5, ...
  'Position', [10/1000 270/600 180/1000 30/600] ...
  );

spiel.rot.punkte_handle = uicontrol ( ...
  'Style', 'text', ...
  'Units', 'normalized', ...
  'BackgroundColor', 'white', ...
  'FontUnits', 'normalized ', ...
  'Fontsize', 0.5, ...
  'Position', [10/1000 240/600 180/1000 30/600] ...
  );
```

[15] Auch den Texten geben wir 10 Pixel Sicherheitsabstand rechts und links. Andernfalls könnten sie beispielsweise die direkt angrenzende Spielfeldumrandung überdecken.

```
spiel.rot.ereignis_handle = uicontrol ( ...
  'Style', 'text', ...
  'Units', 'normalized', ...
  'BackgroundColor', 'white', ...
  'FontUnits', 'normalized ', ...
  'Fontsize', 0.5, ...
  'Position', [10/1000 210/600 180/1000 30/600] ...
  );
```

Die blauen Textfelder sehen praktisch identisch aus, nur dass wir als x-Koordinate der Texte jetzt 810 wählen.

Minen und Tankstellen Die kreisförmigen, grauen Minen erzeugen wir mit Matlabs `rectangle`-Befehl:

```
for i_mine = 1 : spiel.n_mine

  spiel.mine(i_mine).graphics_handle = rectangle ( ...
    'Parent', spiel.spielfeld_handle, ...
    'Position', [...
    spiel.mine(i_mine).pos - spiel.mine(i_mine).radius, ...
    2*spiel.mine(i_mine).radius, ...
    2*spiel.mine(i_mine).radius], ...
    'Curvature', [1 1], ...
    'FaceColor', spiel.farbe.grau, ...
    'EdgeColor', 'none' ...
    );

end
```

Um aus einem Rechteck einen Kreis zu machen, setzen wir den horizontalen und vertikalen `curvature`-Parameter (also die Eckenabrundung) jeweils auf sein Maximum von 1. Für den Vektor zur linken unteren Ecke (des Rechtecks) ziehen wir von den Kreismittelpunktskoordinaten den Kreisradius ab; für die Breite und die Höhe verwenden wir den Durchmesser, also den doppelten Radius.

Auch die grünen Tankstellen zeichnen wir auf die gleiche Weise:

```
for i_tanke = 1 : spiel.n_tanke

  spiel.tanke(i_tanke).graphics_handle = rectangle ( ...
    'Parent', spiel.spielfeld_handle, ...
    'Position', [...
    spiel.tanke(i_tanke).pos - spiel.tanke(i_tanke).radius, ...
    2*spiel.tanke(i_tanke).radius, ...
    2*spiel.tanke(i_tanke).radius], ...
    'Curvature', [1 1], ...
```

```
    'FaceColor', spiel.farbe.gruen, ...
    'EdgeColor', 'none' ...
    );

end
```

Spur Jede Spur ist ein einziger Polygonzug, bestehend aus bis zu 6000 Punkten:

```
spiel.rot.spur_handle = line ( ...
  'Parent', spiel.spielfeld_handle, ...
  'LineStyle', '--' ...
  );

spiel.blau.spur_handle = line ( ...
  'Parent', spiel.spielfeld_handle, ...
  'LineStyle', '--' ...
  );
```

Hier initialisieren wir nur gestrichelt-Eigenschaften der Kurve. Die Farbe und die Punkte der Spur setzen wir dann später in jedem Simulationsschritt in `spielfeld_aktualisieren`.

Spaceballs Auch die Spaceballs selbst sind, wie die Minen und Tankstellen, nur Kreise, deren Farben und Positionen in `spielfeld_aktualisieren` gesetzt werden:

```
spiel.rot.spaceball_handle = rectangle ( ...
  'Parent', spiel.spielfeld_handle, ...
  'Curvature', [1 1], ...
  'EdgeColor', 'none' ...
  );

spiel.blau.spaceball_handle = rectangle ( ...
  'Parent', spiel.spielfeld_handle, ...
  'Curvature', [1 1], ...
  'EdgeColor', 'none' ...
  );
```

Beschleunigung Die Beschleunigungsanzeiger sind, wie in `spielfeld_aktualisieren` beschrieben, gefüllte Dreiecke, die wir in Matlab mit dem Befehl `patch` zeichnen können:

```
spiel.rot.bes_handle = patch ( ...
  'Parent', spiel.spielfeld_handle, ...
  'EdgeColor', 'none' ...
  );
```

```
spiel.blau.bes_handle = patch ( ...
  'Parent', spiel.spielfeld_handle, ...
  'EdgeColor', 'none' ...
  );
```

Geschwindigkeit Die Geschwindigkeitsanzeiger sind einfache, durchgezogene Geraden:

```
spiel.rot.ges_handle = line ( ...
  'Parent', spiel.spielfeld_handle, ...
  'LineWidth', 1 ...
  );

spiel.blau.ges_handle = line ( ...
  'Parent', spiel.spielfeld_handle, ...
  'LineWidth', 1 ...
  );
```

5.8 schritt

Im Unterprogramm `schritt` führen wir die eigentliche Simulation durch:

```
function spiel = schritt (spiel, video)
```

Zeiten Als erstes berechnen wir die aktuelle Zeit[16] aus dem ganzzahligen Schrittzähler und der Schrittweite:

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

Wenn die Simulation den letzten Simulationsschritt erreicht hat

```
if spiel.i_t == spiel.n_t
```

puffern wir die entsprechende Information in einer Variable, die wir im Unterprogramm `spielfeld_aktualisieren` in das Ereignistextfeld eintragen.

```
  spiel.rot.ereignis = 'Zeit ist abgelaufen.';
  spiel.blau.ereignis = 'Zeit ist abgelaufen.';
```

und setzen den Statusanzeiger, der der Simulationsschleife in `spaceballs` signalisiert, dass die Simulation abgebrochen werden soll:

```
  set (spiel.fenster_handle, 'UserData', false);

end
```

[16] Für die Simulation selbst benötigen wir die kontinuierliche Zeit eigentlich nicht. Aber vielleicht können die KIen in `beschleunigung` sie ja gebrauchen.

Minen zerstören Um es Verteidigern schwieriger zu machen, sich einfach nur hinter einer Mine zu verstecken, löschen wir, nachdem alle Tanken verbraucht sind, alle 300 Simulationsschritte (was bei einer Standardschrittweite von 10 ms ja drei Sekunden entspricht) die Mine vom Spielfeld, die dem Verteidiger am nächsten liegt. Wenn also noch Minen da sind

```
if ...
  spiel.n_mine > 0 && ...
```

die Tanken aber schon aufgebraucht sind

```
  spiel.n_tanke == 0 && ...
```

und beim Teilen des aktuellen Schrittzählers durch 300 kein Rest übrig bleibt

```
  mod (spiel.i_t, 300) == 0
```

dann untersuchen wir, ob der rote Spaceball mehr Tanken gesammelt hat:

```
    if spiel.rot.getankt > spiel.blau.getankt
```

Wenn dies der Fall ist, ist der Verteidiger der blaue Spaceball und bekommt dessen Position:

```
      verteidiger_pos = spiel.blau.pos;
```

Wenn hingegen der blaue Spaceball mehr getankt hat[17]

```
    else
```

ist der rote Spaceball der Verteidiger:

```
      verteidiger_pos = spiel.rot.pos;

    end
```

Als nächstes suchen wir die Mine, die zur aktuellen Position des Verteidigers den geringsten Abstand hat. Dazu initialisieren wir den anfänglichen Minimalabstand auf unendlich, damit beim ersten Schritt auf jeden Fall ein neues Minimum gefunden wird

```
    abstand_minimum = inf;
```

und beginnen eine Schleife über alle Minen:

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

In der Schleife berechnen wir den Abstand des Verteidigers zur gerade untersuchten Mine

```
      abstand_looser_mine = ...
        norm (spiel.mine(i_mine).pos - verteidiger_pos);
```

[17] Wenn der sehr unwahrscheinliche Fall eintritt, dass beide Spaceballs gleich viel getankt haben, gibt es ja gar keinen Verteidiger und es ist deshalb ziemlich egal, wer als Verteidiger definiert ist und welche Mine als nächstes zerplatzt.

Wenn dieser Abstand kleiner ist als der bisher kleinste Abstand

```
      if abstand_looser_mine < abstand_minimum
```

haben wir eine neue „Nächster Nachbar"-Mine gefunden. Wir merken uns dann den neuen aktuelle Minimalabstand

```
        abstand_minimum = abstand_looser_mine;
```

und den Index der gefundenen Mine

```
        index_minimum = i_mine;

      end

    end
```

Nachdem wir alle Minen auf ihren Abstand zum Verteidiger untersucht haben, löschen wir die Mine mit dem kleinsten Abstand zum Verteidiger auf dem Spielfeld:

```
  delete (spiel.mine(index_minimum).graphics_handle);
```

Zusätzlich müssen wir die Mine auch noch aus der Liste aller Minen entfernen

```
  spiel.mine(index_minimum) = [];
```

und die Anzahl der Minen aktualisieren:

```
  spiel.n_mine = length (spiel.mine);

end
```

Beschleunigung In `teams_einlesen` haben wir uns die Funktionshandle zu den Beschleunigungsdateien besorgt, in denen die Teams ihre KIen programmiert haben. Diese verwenden wir jetzt, um die jeweilige Beschleunigungsdatei aufzurufen:

```
spiel.rot.bes = spiel.rot.beschleunigung_handle (spiel, 'rot');
```

Dabei übergeben wir der KI neben der gesamten Struktur `spiel` auch ihre eigene Farbe, damit sie weiß, wo sie ihre Daten in der Struktur finden kann. Die KI liefert uns dann in jedem Abtastschritt den von ihr gewünschten Beschleunigungsvektor zurück.

Da der (maximale) Betrag der Beschleunigung beider Spaceballs in `spaceballs` vorgegeben und damit konstant ist, interessiert uns eigentlich nur die Richtung des angeforderten Beschleunigungsvektors. Wir berechnen daher den Vektorbetrag der Beschleunigung:

```
rot_bes_norm = norm (spiel.rot.bes);
```

Wenn dieser nicht[18] verschwindet (wenn also die Nutzerin nicht gerade einen Nullvektor anfordert)

[18] Die Abfrage auf exakt null ist aufgrund von Rundungsfehlern numerisch immer etwas kritisch; wir verwenden daher als Grenze lieber eine sehr kleine Zahl ($1 \cdot 10^{-9}$).

```
if rot_bes_norm > 1e-9
```

berechnen wir den Einheitsbeschleunigungsvektor `spiel.rot.bes / rot_bes_norm` und multiplizieren ihn mit der spielspezifischen Beschleunigung `spiel.bes`:

```
  spiel.rot.bes = spiel.bes * spiel.rot.bes / rot_bes_norm;

end
```

Blau ... entsprechend ...

Eulerschritt Nachdem wir jetzt die Beschleunigung dieses Zeitschrittes kennen, können wir die neue Geschwindigkeit mit einem einfachen Eulerschritt [8] erhalten:

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

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

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

und wiederholen das Ganze für den blauen Spaceball:

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

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

Tankstellen In den folgenden Programmabschnitten untersuchen wir, ob die Spaceballs Tankstellen, Minen, Banden oder einander berührt[19] haben.

Bevor wir damit beginnen, initialisieren wir noch eine Variable, in der wir anschließend die Indizes der zu löschenden Tankstellen sammeln:

```
zu_loeschende_tanken = [];
```

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 (rote) Spaceball die aktuelle Tankstelle schneidet:

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

[19] Grundsätzlich ist es bei der Kollisionsanalyse in diskreten Simulationen immer möglich, dass sich ein Objekt in einem Simulationsschritt kurz vor einem anderen Objekt befindet und im nächsten Simulationsschritt schon dahinter. Wir dürften dann nicht einfach nur überprüfen, ob sich die beiden Objekte überlappen (was sie ja nie tun), sondern müssten in jedem Schritt eine echte Kollisionsanalyse gemäß Kapitel 10 durchführen. Bei den hier vorliegenden sehr kleinen Schrittweiten und durch die maximale Beschleunigung und die Banden begrenzten Maximalgeschwindigkeiten können wir aber mit ruhigem Gewissen darauf verzichten.

Wenn dies der Fall ist, erhöhen wir den Zähler der (von rot) besuchten Tankstellen

```
    spiel.rot.getankt = spiel.rot.getankt + 1;
```

und hängen den Index der aktuellen Tankstelle an die Löschliste:

```
    zu_loeschende_tanken = [zu_loeschende_tanken, i_tanke];

  end
```

Die gleiche Untersuchung führen wir auch für den blauen[20] Spaceball durch:

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

    spiel.blau.getankt = spiel.blau.getankt + 1;

    zu_loeschende_tanken = [zu_loeschende_tanken, i_tanke];

  end

end
```

Wenn es jetzt Tankstellen gibt, die in diesem Simulationsschritt besucht wurden und daher gelöscht werden müssen

```
if ~isempty (zu_loeschende_tanken)
```

definieren wir einen Vektor[21] der Grafikhandles der zu löschenden Tankstellen und löschen mit ihm die Visualisierung(en) der Tankstelle(n) im Spielfeld:

```
  delete ([spiel.tanke(zu_loeschende_tanken).graphics_handle]);
```

Natürlich müssen wir die verbrauchten Tankstellen auch aus der Liste der (noch aktiven) Tankstellen entfernen

```
  spiel.tanke(zu_loeschende_tanken) = [];
```

und die Anzahl der verbleibenden Tankstellen aktualisieren:

```
  spiel.n_tanke = length (spiel.tanke);

end
```

[20] Es kann „theoretisch" passieren, dass beide Spaceballs dieselbe Tankstelle im gleichen Simulationsschritt berühren. In diesem Fall bekommen gerechterweise beide Spaceballs die Tankstelle zugeschrieben. Dadurch kann es „theoretisch" zu einem Tankstellengleichstand kommen; und dadurch wird das Endspiel dann nicht so richtig spannend. Punkte gibt es dann konsequenterweise für keinen ... „theoretisch" ...

[21] Es wäre ja auch möglich, dass rot und blau im gleichen Schritt verschiedene Tankstellen berühren. In diesem Fall müssen die Handles dem `delete`-Befehl syntaxgerecht in Form eines Vektors übergeben werden. Der Fall, dass ein Spaceball gleichzeitig mehrere Tankstellen berührt, ist beim vordefinierten Sicherheitskreisradius um jede Tankstelle in dieser Version nicht möglich.

Minen Im folgenden Programmabschnitt wollen wir – analog zur vorherigen Tankstellenberührung – ermitteln, ob ein Spaceball eine Mine berührt hat.

Dazu initialisieren wir zwei Anzeiger, die wir anschließend auswerten werden

```
rot_trifft_mine = false;

blau_trifft_mine = false;
```

und starten eine Schleife über alle Minen

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

Innerhalb der Schleife untersuchen wir, ob sich die Kreise von aktueller Mine und Spaceball überschneiden:

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

Wenn dies der Fall ist, setzen wir den entsprechenden Anzeiger

```
    rot_trifft_mine = true;
```

und das passende Ereignis:

```
    spiel.rot.ereignis = 'Rot trifft Mine.';

  end
```

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

```
  if schnitt_kreis_kreis (spiel.mine(i_mine), spiel.blau)

    blau_trifft_mine = true;

    spiel.blau.ereignis = 'Blau trifft Mine.';

  end

end
```

Banden Auch für die Bandenberührungsanalyse initialisieren wir wieder zwei Anzeiger

```
rot_trifft_bande = false;
blau_trifft_bande = false;
```

und müssen dann noch nicht einmal eine Schleife starten, da alle vier Banden schon im Unterprogramm `schnitt_kreis_bande` abgearbeitet werden:

```
if schnitt_kreis_bande (spiel.rot)

  rot_trifft_bande = true;
```

```
  spiel.rot.ereignis = 'Rot trifft Bande.';

end

if schnitt_kreis_bande (spiel.blau)

  blau_trifft_bande = true;

  spiel.blau.ereignis = 'Blau trifft Bande.';

end
```

Auswertung Die Auswertung und Punkteverteilung ist nicht ganz trivial, da es ja beispielsweise passieren kann, dass der eine Spaceball gerade in eine Mine und der andere im gleichen Simulationsschritt in eine Bande rauscht. In diesem Fall bekommt gerechterweise keiner von beiden einen Tabellenpunkt. Wir analysieren daher als erstes, ob (mindestens) ein Spaceball eine Mine oder eine Bande berührt:

```
if ...
  rot_trifft_mine || ...
  blau_trifft_mine || ...
  rot_trifft_bande || ...
  blau_trifft_bande
```

In jedem dieser Fälle ist das Spiel beendet:

```
  set (spiel.fenster_handle, 'UserData', false);

end
```

Jetzt geht es um die Punkteverteilung. Wenn Rot eine Mine oder Bande berührt und gleichzeitig Blau weder eine Mine noch eine Bande berührt

```
if ...
  (rot_trifft_mine || ...
  rot_trifft_bande) && ...
  ~blau_trifft_mine && ...
  ~blau_trifft_bande
```

dann hat Blau gewonnen und bekommt einen Tabellenpunkt:

```
  spiel.blau.ereignis = 'Blau gewinnt.';

  spiel.blau.punkte = 1;
```

Wenn hingegen Blau eine Mine oder Bande berührt und gleichzeitig Rot weder eine Mine noch eine Bande berührt

```
elseif ...
  (blau_trifft_mine || ...
  blau_trifft_bande) && ...
  ~rot_trifft_mine && ...
  ~rot_trifft_bande
```

dann geht der Sieg an Rot:

```
  spiel.rot.ereignis = 'Rot gewinnt.';

  spiel.rot.punkte = 1;

end
```

Kontakt Als letztes wollen wir entscheiden, was geschehen soll, wenn beide Spaceballs einander berühren:

```
if schnitt_kreis_kreis (spiel.rot, spiel.blau)
```

Wenn beide Spaceballs die gleiche Anzahl Tankstellen besucht haben, geschieht überhaupt nichts; die Spaceballs fliegen dann praktisch durcheinander durch. Wenn aber der rote Spaceball beim Tankstellenpunktesammeln erfolgreicher war

```
  if spiel.rot.getankt > spiel.blau.getankt
```

dann wird der Sieg des roten Spaceballs für beide Ereignisfeldern gesetzt

```
    spiel.rot.ereignis = 'Rot trifft Blau.';
    spiel.blau.ereignis = 'Rot trifft Blau.';
```

Rot bekommt seinen verdienten Punkt

```
    spiel.rot.punkte = 1;
```

und das Spiel ist beendet:

```
    set (spiel.fenster_handle, 'UserData', false);
```

Das Entsprechende geschieht, wenn Blau beim Zusammentreffen mehr Tankstellenpunkt besitzt als Rot:

```
  elseif spiel.rot.getankt < spiel.blau.getankt

    spiel.rot.ereignis = 'Blau trifft Rot.';

    spiel.blau.ereignis = 'Blau trifft Rot.';

    spiel.blau.punkte = 1;

    set (spiel.fenster_handle, 'UserData', false);
```

```
  end

end
```

Grafik aktualisieren Bevor wir die Grafikobjekte des Darstellungsfensters aktualisieren, speichern[22] wir die aktuelle Position beider Spaceballs im jeweiligen Spurfeld ab:

```
spiel.rot.spur(spiel.i_t, :) = spiel.rot.pos;

spiel.blau.spur(spiel.i_t, :) = spiel.blau.pos;
```

Alsdann lassen wir `spielfeld_aktualisieren` das Spielfeld neu zeichnen:

```
spielfeld_aktualisieren (spiel, video);
```

5.9 spielfeld_aktualisieren

Das Unterprogramm `spielfeld_aktualisieren` wird einmal pro Abtastschritt von `schritt` aufgerufen, zeichnet alle Grafikobjekte des Darstellungsfensters neu und schreibt gegebenenfalls den aktuellen Fensterinhalt in die Videodatei:

```
function spielfeld_aktualisieren (spiel, video)
```

Anzeigen Als erstes aktualisieren[23] wir den Zeitbalken:

```
set (spiel.zeit_handle, ...
  'Position', [0, 590/600, spiel.i_t/spiel.n_t, 10/690] ...
  );
```

Dabei geben wir ihm eine Höhe von 10 Pixeln und kleben ihn ganz oben an das Spielfeld (untere Kante des Zeitbalkens bei 590). Seine Breite berechnen wir linear aus der aktuellen Zeit: `spiel.i_t/spiel.n_t`. Wenn das Spiel beginnt, ist `spiel.i_t` gleich null, so dass auch der Balken eine verschwindende Breite besitzt; wenn das Spiel nach 60 Sekunden ergebnislos endet, ist `spiel.i_t` gleich `spiel.n_t`, so dass der Balken über die gesamte Breite des Spielfeld verläuft.

Als nächstes schreiben wir die Anzahl der besuchten Tankstellen in die ebenfalls in `spielfeld_darstellen` erzeugten Tankanzeigefelder

[22] Wir könnten das Füllen des Spurfeldes auch innerhalb von `spielfeld_aktualisieren` durchführen, dann müssten wir aber die geänderte Spielstruktur von `spielfeld_aktualisieren` auch wieder zurückgeben lassen. Durch die gewählte Vorgehensweise ist `spielfeld_aktualisieren` wirklich nur eine rein grafische Ausgaberoutine, die tatsächlich nur das Spielfeld beackert und selbst keine Spielinhalte verändert.

[23] Da wir die Grafikobjekte ja schon in `spielfeld_darstellen` erzeugt haben, reicht es jetzt während des Spiels, direkt nur die veränderten Eigenschaften zu aktualisieren.

```
set (spiel.rot.getankt_handle, ...
  'String', ['Getankt: ', num2str(spiel.rot.getankt)] ...
  );

set (spiel.blau.getankt_handle, ...
  'String', ['Getankt: ', num2str(spiel.blau.getankt)] ...
  );
```

die Punkte in die Punkteanzeigefelder

```
set (spiel.rot.punkte_handle, ...
  'String', ['Punkte: ', num2str(spiel.rot.punkte)] ...
  );

set (spiel.blau.punkte_handle, ...
  'String', ['Punkte: ', num2str(spiel.blau.punkte)] ...
  );
```

und die Ereignisse in die Ereignisanzeigefelder

```
set (spiel.rot.ereignis_handle, ...
  'String', spiel.rot.ereignis ...
  );

set (spiel.blau.ereignis_handle, ...
  'String', spiel.blau.ereignis ...
  );
```

Spur Für die Spur lesen wir den Wert, den die Nutzerin aktuell im Spurschieberegler eingestellt hat

```
n_spur = round (get (spiel.spur_slider_handle, 'Value'));
```

und verwenden ihn dazu, die letzten in `spiel.rot.spur` abgespeicherten Punkte als Daten des Spur-Polygonzugs zu definieren:

```
set (spiel.rot.spur_handle, ...
  'XData', spiel.rot.spur ...
  (max (spiel.i_t - n_spur, 1) : spiel.i_t, 1), ...
  'YData', spiel.rot.spur ...
  (max (spiel.i_t - n_spur, 1) : spiel.i_t, 2) ...
  );
```

Dabei müssen wir den Anfangsindex mit Hilfe des `max`-Befehls nach unten auf den Wert 1 begrenzen, da `spiel.i_t - n_spur` am Anfang der Simulation durchaus negativ sein könnte und damit dann natürlich nicht als Index in ein Feld verwendbar wäre.

Blau ... entsprechend ...

Spaceballs Bei den Spaceballs müssen wir für die linke untere Ecke des umgebenden Quadrates von der Mittelpunktsposition jeweils einen Radius abziehen:

```
set (spiel.rot.spaceball_handle, ...
  'Position', ...
  [...
  spiel.rot.pos(1) - spiel.rot.radius, ...
  spiel.rot.pos(2) - spiel.rot.radius, ...
  2*spiel.rot.radius, ...
  2*spiel.rot.radius])
```

Breite und Höhe des Quadrat entsprechen dem Durchmesser und damit dem doppelten Radius des Kreises.

Blau ... entsprechend ...

Geschwindigkeit Wenn die Nutzerin den Geschwindigkeitsanzeiger sehen möchte:

```
if get (spiel.ges_checkbox_handle, 'Value')
```

setzen wir den Anfangspunkt der Geschwindigkeitslinie auf die aktuelle Position selbst und den Endpunkt in Richtung des aktuellen Geschwindigkeitsvektors:

```
  set (spiel.rot.ges_handle, ...
    'XData', [ ...
    spiel.rot.pos(1), ...
    spiel.rot.pos(1) + 0.5*spiel.rot.ges(1)], ...
    'YData', [ ...
    spiel.rot.pos(2), ...
    spiel.rot.pos(2) + 0.5*spiel.rot.ges(2)] ...
    );
```

Die Länge der Geschwindigkeitslinie wählen wir mit dem Faktor `0.5` ziemlich willkürlich als Kompromiss zwischen ausreichender Sichtbarkeit und störender Dominanz. Wenn die Nutzerin die Geschwindigkeit sehen möchte, müssen wir sie natürlich auch anzeigen:[24]

```
  set (spiel.rot.ges_handle, 'Visible', 'on');
```

Blau ... entsprechend ...

Wenn die Nutzerin hingegen auf die Geschwindigkeitslinie verzichten möchte

```
else
```

machen wir sie einfach unsichtbar:

[24] Beim ersten Mal müssten wir die Geschwindigkeitslinie natürlich noch nicht sichtbar machen; aber wenn die Nutzerin sie erst ausgeblendet hat und dann wieder einblendet ...

```
  set (spiel.rot.ges_handle, 'Visible', 'off');

  set (spiel.blau.ges_handle, 'Visible', 'off');

end
```

Beschleunigung Die Beschleunigung symbolisieren wir, wie in Abbildung 5.3 dargestellt, als gleichschenkliges Dreieck, dessen Mittelsenkrechte $\overline{M3} = 4r$ vier Radienlängen ($\overline{M1} = \overline{M2} = r$) beträgt.

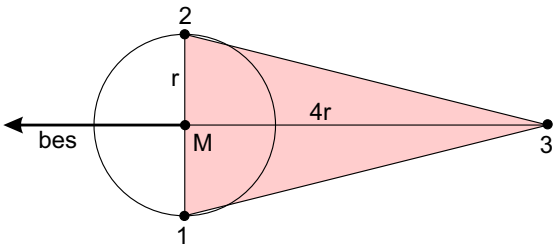


Abbildung 5.3: Beschleunigung als dreieckiger Triebwerksstrahl

Der Punkt 3 zeigt dabei in Richtung des Triebwerkstrahls. In Abbildung 5.3 hat der Beschleunigungsvektor also nur eine negative x-Komponente (nach links).

Auch bei der Anzeige der Beschleunigung fragen wir als erstes ab, ob die Nutzerin sie überhaupt sehen möchte:

```
if get (spiel.bes_checkbox_handle, 'Value')
```

Wenn dies der Fall ist, definieren wir eine Rotationsmatrix

```
  dreh_trafo = [0 -1; 1 0];
```

die, wenn sie von links an einen Vektor heran multipliziert wird, diesen um 90 Grad mathematisch positiv links herum dreht (Abbildung 5.4).

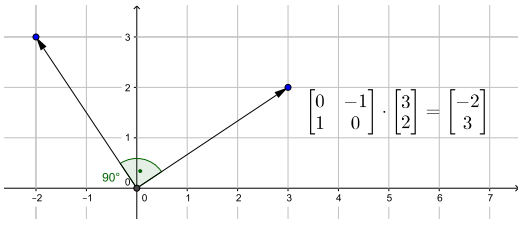


Abbildung 5.4: Beispiel einer Vektordrehung mittels einer Rotationsmatrix

Als nächstes müssen wir untersuchen, ob die KI überhaupt einen (vom Nullvektor verschiedenen) Beschleunigungsvektor angefordert hat. Dazu bilden wir den Betrag des Beschleunigungsvektors

```
rot_bes_norm = norm (spiel.rot.bes);
```

und fragen ab, ob dieser größer als „null" ist:

```
if rot_bes_norm > 1e-9
```

Wenn die KI also beschleunigen möchte, können wir den Beschleunigungseinheitsvektor berechnen

```
  rot_bes_einheit = spiel.rot.bes/norm (spiel.rot.bes);
```

und ihn mit Hilfe der Rotationsmatrix[25] um 90 Grad nach links drehen:

```
  rot_bes_einheit_gedreht = dreh_trafo*rot_bes_einheit';
```

Mit diesem Normaleneinheitsvektor[26] können wir jetzt Ortsvektoren zur linken Ecke (Punkt 1 in Abbildung 5.3)

```
  rot_linke_ecke = ...
    spiel.rot.pos + ...
    spiel.rot.radius*rot_bes_einheit_gedreht';
```

und zur rechten Ecke (Punkt 2 in Abbildung 5.3) berechnen:

```
  rot_rechte_ecke = ...
    spiel.rot.pos - ...
    spiel.rot.radius*rot_bes_einheit_gedreht';
```

Den Ortsvektor zum hinteren Punkt 3 erhalten wir, indem wir erst zum Mittelpunkt und dann gemäß Abbildung 5.3 um vier Radien in negative Beschleunigungsrichtung gehen:

```
  rot_hintere_ecke = ...
    spiel.rot.pos - ...
    4*spiel.rot.radius*rot_bes_einheit;
```

Mit diesen drei Eckpunkten können wir jetzt die Daten des Dreieckes aktualisieren

```
  set (spiel.rot.bes_handle, ...
  'XData', [ ...
  rot_linke_ecke(1), ...
  rot_rechte_ecke(1), ...
  rot_hintere_ecke(1)], ...
```

[25] Dabei müssen wir den Einheitsvektor von einem Zeilenvektor in einen Spaltenvektor transponieren, da alle Vektoren in diesem Programm Zeilenvektoren sind, die Rotation aber üblicherweise einen Spaltenvektor erwartet.

[26] Vor der Verwendung des gedrehten Einheitsvektors müssen wir diesen dann natürlich wieder zurück in einen Zeilenvektor transponieren.

```
    'YData', [ ...
    rot_linke_ecke(2), ...
    rot_rechte_ecke(2), ...
    rot_hintere_ecke(2)] ...
    );
```

und es sichtbar machen:

```
    set (spiel.rot.bes_handle, 'Visible', 'on');
```

Wenn die KI gerade nicht beschleunigen möchte

```
  else
```

verstecken wir das Beschleunigungsdreieck:

```
    set (spiel.rot.bes_handle, 'Visible', 'off');

  end
```

Blau ... entsprechend ...

Und wenn die Nutzerin die Beschleunigung überhaupt nicht sehen will

```
else
```

verstecken wir natürlich auch beide Dreiecke:

```
  set (spiel.rot.bes_handle, 'Visible', 'off');

  set (spiel.blau.bes_handle, 'Visible', 'off');

end
```

Farbe Die Anzahl der bislang besuchten Tankstellen wird ja für beide Spaceballs unterhalb des Logos angezeigt. Allerdings ist es während eines spannenden Spiels sehr ablenkend, immer wieder zu den Zahlen blicken zu müssen. Wir möchten daher über die Farbe eines Spaceballs deutlich machen, ob er gerade weniger Tankpunkte als sein Gegner hat und damit ein potenzielles Opfer darstellt.

Wenn also der rote Spaceball mehr getankt hat als der blaue Spaceball

```
if spiel.rot.getankt > spiel.blau.getankt
```

dann hellen wir den blauen Spaceball auf, indem wir seine blaue Farbe (Körper, Beschleunigungsdreieck und Spur) durch hellblau ersetzen und seinen Geschwindigkeitsanzeiger von schwarz nach hellgrau umfärben:

```
  set (spiel.blau.spaceball_handle, ...
    'Facecolor', spiel.farbe.hellblau)
```

```
  set (spiel.blau.bes_handle, ...
    'Facecolor', spiel.farbe.hellblau)

  set (spiel.blau.ges_handle, ...
    'Color', spiel.farbe.hellgrau)

  set (spiel.blau.spur_handle, ...
    'Color', spiel.farbe.hellblau)
```

Wenn hingegen Blau mehr Punkte gesammelt hat

```
elseif spiel.rot.getankt < spiel.blau.getankt
```

dämpfen wir den roten Spaceball:

```
  set (spiel.rot.spaceball_handle, ...
    'Facecolor', spiel.farbe.hellrot)

  set (spiel.rot.bes_handle, ...
    'Facecolor', spiel.farbe.hellrot)

  set (spiel.rot.ges_handle, ...
    'Color', spiel.farbe.hellgrau)

  set (spiel.rot.spur_handle, ...
    'Color', spiel.farbe.hellrot)
```

Und bei Punktegleichstand[27]

```
else
```

bekommen alle Objekte ihre Originalfarbe zurück:

```
  set (spiel.rot.spaceball_handle, ...
    'Facecolor', spiel.farbe.rot)

  set (spiel.blau.spaceball_handle, ...
    'Facecolor', spiel.farbe.blau)

  set (spiel.rot.bes_handle, ...
    'Facecolor', spiel.farbe.rot)

  set (spiel.blau.bes_handle, ...
    'Facecolor', spiel.farbe.blau)

  set (spiel.rot.ges_handle, ...
```

[27] Da die Tankstellen so weit voneinander entfernt liegen, dass es unmöglich für einen Spaceball ist, in einem Simulationsschritt zwei Tankstellen zu leeren, muss zwischen dem Wechsel der Opferrolle immer kurz ein Punktegleichstand stattfinden, in dem die Farbe wieder zurückgesetzt wird.

```
    'Color', [0 0 0])

  set (spiel.blau.ges_handle, ...
    'Color', [0 0 0])

  set (spiel.rot.spur_handle, ...
    'Color', spiel.farbe.rot)

  set (spiel.blau.spur_handle, ...
    'Color', spiel.farbe.blau)

end
```

Video, Zeitraffer und Zeitlupe Wenn die Nutzerin in `spaceballs` ausgewählt hat, dass er während der Simulation ein Video schreiben möchte

```
if video.abspeichern
```

nehmen wir bei jedem vierten[28] Simulationsschritt

```
  if mod (spiel.i_t, 4) == 0
```

ein Bild des aktuellen Fensters auf

```
    frame = getframe(spiel.fenster_handle);
```

und speichern es in die schon in `spaceballs` geöffnete Videodatei ab:

```
    writeVideo (video.writer, frame)

  end
```

Wenn die Nutzerin kein Video aufnehmen möchte, aber das Zeitrafferauswahlfeld angeklickt hat

```
elseif get (spiel.zeitraffer_checkbox_handle, 'Value')
```

verwenden wir den Zeichenbefehl mit dem Parameter `limitrate`

```
  drawnow limitrate
```

der bewirkt, dass Matlab die Echtzeitsimulationsrate auf 20 Hz beschränkt, was insbesondere auf Rechner, die grafisch etwas schwach ausgestattet sind, zu deutlich schnelleren Simulationen führen kann.

Wenn auch kein Zeitraffer gewünscht ist, aber die Nutzerin den Zeitlupenschieberegler aus seiner Nullposition herausgeschoben hat

[28] Das Video wird bei einer Standardsimulationsschrittweite von $T = \frac{1}{100}\,\mathrm{s} = 10\,\mathrm{ms}$ ($f = \frac{1}{T} = 100\,\mathrm{Hz}$) also nur mit 25 Hz aufgezeichnet, was sicherlich keine cineastischen Beifallsstürme hervorruft, aber die Simulationsgeschwindigkeit und die Dateigröße in erträglichem Rahmen hält.

```
elseif get (spiel.zeitlupe_slider_handle, 'Value') > 0
```

pausieren wir einfach die Simulation um den gewünschten Wert:

```
  pause (get (spiel.zeitlupe_slider_handle, 'Value'))
```

Im „Normalfall", wenn also kein Video geschrieben werden soll und weder Zeitraffer noch Zeitlupe eingeschaltet sind[29]

```
else
```

zeichnen wir einfach alles in das aktuelle Fenster:

```
  drawnow

end
```

5.10 beschleunigung

Im Unterprogramm `beschleunigung` programmiert jedes Team ihre KI. Diese bekommt über die Parameterliste den aktuellen Zustand des Spiels in Form der Struktur `spiel` und die Farbe des eigenen Spaceballs geliefert und berechnet daraus die gewünschte Beschleunigung(srichtung) `bes`:

```
function bes = beschleunigung (spiel, 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 Spaceball
  ich = spiel.rot;

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

% Wenn meine Farbe nicht rot ist,
else
```

[29] Durch die sequenzielle Bedingungsstruktur (`... elseif ...`) sind Zeitraffer und Zeitlupe unwirksam, wenn ein Video aufgezeichnet wird und die Zeitlupe hat keine Funktion, wenn der Zeitraffer eingeschaltet ist. Das kann durchaus praktisch sein; wir können die Zeitlupe einschalten, um Details genau analysieren zu können und durch kurzzeitiges Einschalten des Zeitraffers schnell an einen interessanten Punkt „vorspulen".

```
  % dann bin ich der blaue Spaceball
  ich = spiel.blau;

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

end
```

Alsdann können wir über beispielsweise über `ich.pos` auf den eigenen Positionsvektor oder über `gegner.ges(1)` auf die x-Koordinate des gegnerischen Geschwindigkeitsvektors zugreifen.

Das `beschleunigung` aufrufende Unterprogramm `schritt` erwartet für die Beschleunigung `bes` immer einen gültigen Inhalt zurück geliefert zu bekommen. Um dies sicherzustellen, empfiehlt es sich, ganz oben im Unterprogramm etwas wie

```
bes = [0, 0];
```

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

5.11 spiel

Die Struktur `spiel` beinhaltet die Daten der gesamten Simulation. Sie wird, wie in Abbildung 4.1 dargestellt, von den Unterprogrammen `teams_einlesen`, `spielfeld_erstellen`, `spielfeld_darstellen`, `schritt` und `spielfeld_aktualisieren` gefüllt und steht auch am Ende der Simulation noch für weitere Untersuchungen zur Verfügung.

Die Struktur besitzt laut Abbildung 5.5 neben skalaren Daten (`dt`, ...) auch Strukturen (`rot`, ...), Boolesche Variablen (`ges_checkbox_anfangswert`, ...) und Handles (`fenster_handle`, ...).

Abbildung 5.5: Spieldaten in der Stuktur `spiel`

Im Einzelnen haben die Felder folgende Bedeutung:

`dt` Simulationsschrittweite (in Sekunden). Standardmäßig wird mit 100 Hz (100 Simulationsschritte pro Sekunde) abgetastet. Es ergibt sich daher eine Simulationsschrittweite von $\Delta t = \frac{1}{100} = 0.01$ Sekunden.

`t_end` Maximale Simulationszeit (in Sekunden). Jedes Spiel dauert standardmäßig höchstens eine Minute. Manche Spiele enden vorzeitig nach wenigen Sekunden.

`n_t` Maximale Anzahl von Simulationsschritten. Es könnten also maximal $n_t = \frac{t_{end}}{\Delta t} = \frac{60}{\frac{1}{100}} = 60 \cdot 100 = 6000$ Schritte simuliert werden.

`i_t` Aktueller Schrittzähler. Die Zahl $i_t = 788$ bedeutet, dass das Spiel nach $t = i_t \cdot \Delta t = \frac{788}{100} = 7.88$ Sekunden abgebrochen wurde.

`n_mine` Gerade Anzahl der auf dem Spielfeld zufällig aber symmetrisch verteilten Minen (Standardwert: 12)

`n_tanke` Ungerade Anzahl der auf dem Spielfeld zufällig aber symmetrisch verteilten Tankstellen (Standardwert: 9)

`kreis_radius` Radius des Sicherheitskreises um Tankstellen und Minen. Der Sicherheitsradius sollte so groß sein, dass sich ein Spaceball gerade noch sicher zwischen Mine und Bande durchquetschen kann. (Standardwert: 0.075)

`mine_radius` Konstanter Radius einer Mine (Standardwert: 0.05)

`tanke_radius` Konstanter Radius einer Tankstelle (Standardwert: 0.01)

`spaceball_radius` Konstanter Radius eines Spaceballs (Standardwert: 0.01)

`bes` Konstanter Betrag des Beschleunigungsvektors (Standardwert: 0.1)

`rot` Struktur (`spiel.rot`) der Daten des roten Teams

`blau` Struktur der Daten des blauen Teams

`farbe` Struktur (`spiel.farbe`) der im Spiel verwendeten Farben

`spur_anfangswert` Anfängliche Anzahl der Punkt der Spur (Standardwert: `0.1*spiel.n_t` $\sim 10\,\%$ der Gesamtlänge)

`zeitlupe_anfangswert` Verzögerung in jedem Simulationsschritt in Sekunden (Standardwert: 0)

`ges_checkbox_anfangswert` Soll die Geschwindigkeit angezeigt werden? (Standardwert: ja)

`bes_checkbox_anfangswert` Soll die Beschleunigung angezeigt werden? (Standardwert: ja)

`zeitraffer_checkbox_anfangswert` Ist der Zeitraffer eingeschaltet? (Standardwert: nein)

`tanke` Struktur (`spiel.tanke`) mit allen Tankstellen

`mine` Struktur (`spiel.mine`) mit allen Minen

`fenster_handle` Handle zum Darstellungsfenster

`spielfeld_handle` Handle zum quadratischen Spielfeld-Achsensystem in der Mitte des Darstellungsfensters

`zeit_handle` Handle zum Rechteck zur Darstellung der verstrichenen Zeit am oberen Spielfeldrand

`spur_slider_handle` Handle zum Schieberegler zur Einstellung der Länge der Spur

`spur_text_handle` Handle zur Überschrift des Spurlängenschiebereglers

`zeitlupe_slider_handle` Handle zum Schieberegler zur Einstellung der Zeitlupe

`zeitlupe_text_handle` Handle zur Überschrift des Zeitlupenschiebereglers

`ges_checkbox_handle` Handle zum Geschwindigkeitsauswahlfeld

`bes_checkbox_handle` Handle zum Beschleunigungsauswahlfeld

`zeitraffer_checkbox_handle` Handle zum Zeitrafferauswahlfeld

`t` Analoge Zeit in Sekunden

5.11.1 `spiel.rot`

Das in Abbildung 5.6 dargestellte Feld `spiel.rot` beinhaltet den Daten des roten[30] Teams.

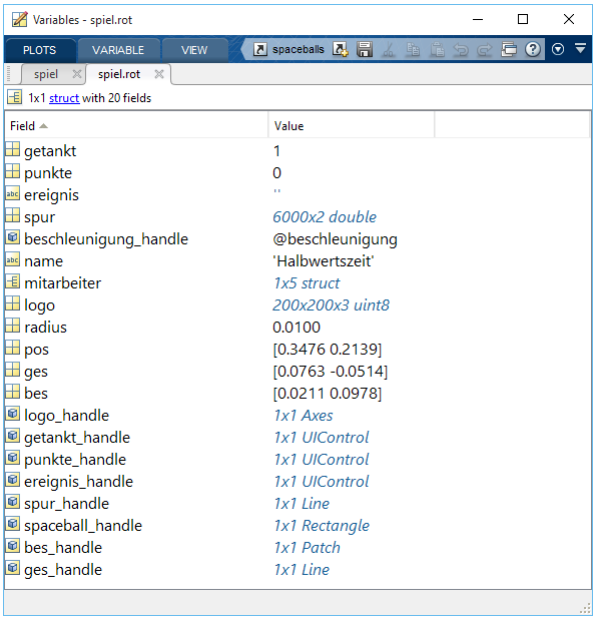


Abbildung 5.6: Daten des roten Spaceballs in der Struktur `spiel.rot`

Im Einzelnen haben die Felder folgende Bedeutung:

`getankt` Anzahl der bisher besuchten Tankstellen

`punkte` Anzahl der Tabellenpunkte am Ende des Spiels

`ereignis` Zeichenkette zum Zwischenspeichern eines Spielereignisses

`spur` Feld `spiel.rot.spur` mit den bisherigen Positionen. Kann am Ende des Spiels für weitere Analysen verwendet werden.

`beschleunigung_handle` Handle zur Beschleunigungsfunktion (`beschleunigung`)

`name` Teamname

`mitarbeiter` Struktur (`spiel.rot.mitarbeiter`) mit den Namen und Aufgaben der Teammitarbeiterinnen

`logo` Teamlogo der Größe 200×200 Pixel

`radius` Radius des Spaceballs

`pos` Aktuelle Position des Spaceballs

[30] Natürlich gibt es auch die entsprechenden Felder des blauen Teams, die wir hier aber aus Redundanzgründen nicht aufführen.

`ges` Aktuelle Geschwindigkeit des Spaceballs

`bes` Aktuelle Beschleunigung des Spaceballs

`logo_handle` Handle zum Teamlogo

`getankt_handle` Handle zur Anzeige der Anzahl der bisher besuchten Tankstellen

`punkte_handle` Handle zur Anzeige der Anzahl der Tabellenpunkte am Ende des Spiels

`ereignis_handle` Handle zur Anzeige eines Spielereignisses

`spur_handle` Handle zur Spur

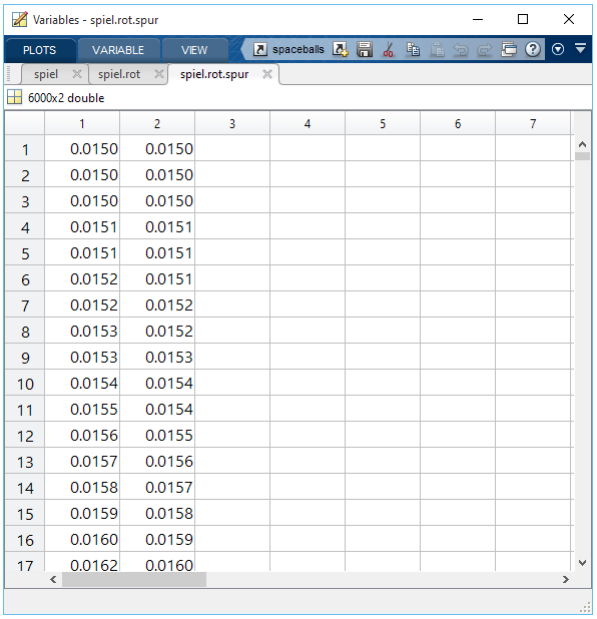
`spaceball_handle` Handle zum Spaceball

`bes_handle` Handle zur Beschleunigungsanzeige

`ges_handle` Handle zur Geschwindigkeitsanzeige

5.11.2 `spiel.rot.spur`

Während und auch Ende des Spiels steht für jeden Spaceball der gesamte (bisherige) Positionsverlauf als Feld zur Verfügung (Abbildung 5.7). Wir können diesen beispielsweise für nachträgliche Ausschnittsvergrößerungen verwenden, wenn wir uns ein bestimmtes Spieldetail ganz genau ansehen wollen.



	1	2	3	4	5	6	7
1	0.0150	0.0150					
2	0.0150	0.0150					
3	0.0150	0.0150					
4	0.0151	0.0151					
5	0.0151	0.0151					
6	0.0152	0.0151					
7	0.0152	0.0152					
8	0.0153	0.0152					
9	0.0153	0.0153					
10	0.0154	0.0154					
11	0.0155	0.0154					
12	0.0156	0.0155					
13	0.0157	0.0156					
14	0.0158	0.0157					
15	0.0159	0.0158					
16	0.0160	0.0159					
17	0.0162	0.0160					

Abbildung 5.7: Positionsverlauf im Feld `spiel.rot.spur`

5.11.3 `spiel.rot.mitarbeiter`

Das Unterprogramm `teams_einlesen` liest die Namen und die Aufgaben der Teammitglieder aus der Datei `team_daten` und speichert sie in dem in Abbildung 5.8 dargestellten Feld von Strukturen.[31]

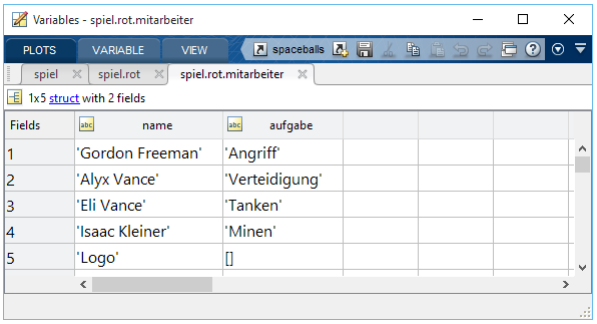


Abbildung 5.8: Namen und Aufgaben der Mitarbeiterinnen im Feld `spiel.rot.mitarbeiter`

5.11.4 `spiel.farbe`

Die im Spiel verwendeten Farben definieren wir in `spaceballs` im HSV-Farbraum [10] und lassen sie von Matlab mit dem Befehl `hsv2rgb` in ihre RGB-Werte umrechnen (Abbildung 5.9). HSV steht für Farbton (englisch: **H**ue), Farbsättigung (englisch: **S**aturation) und Hellwert (englisch: **V**alue). Die Verwendung des HSV-Farbraums hat den großen Vorteil, dass wir sehr leicht die zu den Vollfarben (gesättigten Farben) passenden etwas blasseren Farben finden können. So definieren wir beispielsweise „unser" Rot[32] mit den HSV-Werten `[0.95 1 1]`. Den entsprechenden blasseren Farbton erhalten wir dann sehr einfach, indem wir den Farbsättigungswert auf `0.4` verkleinern: `[0.95 0.4 1]`.

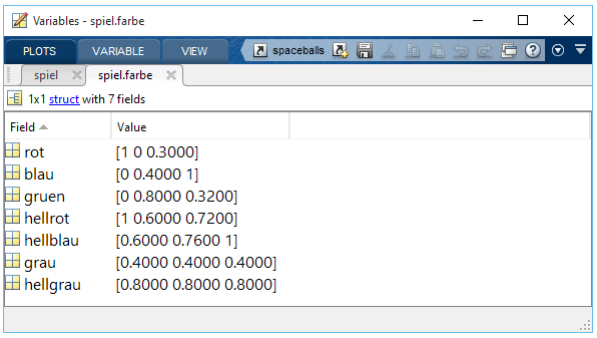


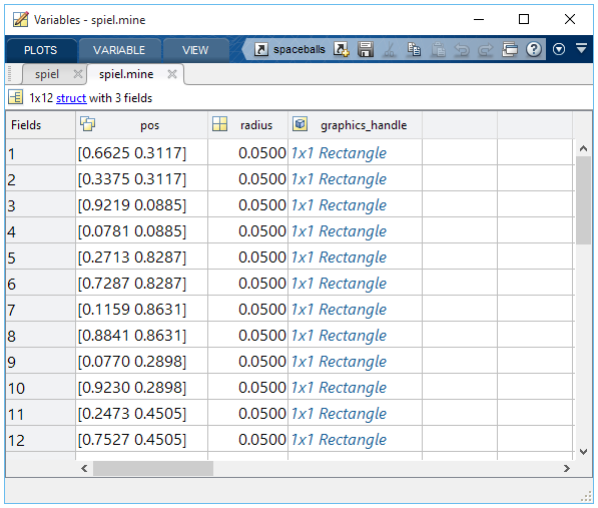
Abbildung 5.9: RGB-Werte der verwendeten Farben in der Struktur `spiel.farbe`

[31] Zum Unterschied zwischen einer Feldstruktur („Structure of arrays") und Strukturfeldern („Array of structures") hat sich Doug Hull von den Mathworks in [9] ein paar Gedanken gemacht. Auch die Kommentare auf der Seite sind teilweise recht hilfreich.

[32] Reines Rot hätte die HSV-Werte `[1 1 1]`. Unser Rot hat mit seinem Farbton von `0.95` daher noch einen leichten Blauanteil, was nach Abbildung 5.9 zu RGB-Werten von `[1 0 0.3]` führt.

5.11.5 `spiel.mine`

Abbildung 5.10 zeigt die Positionen und Radien der 12 Minen im Strukturfeld `mine`.

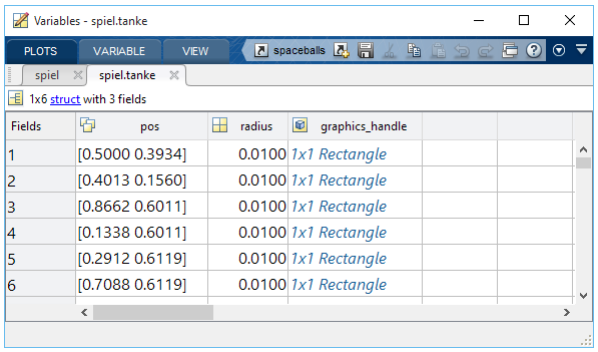


Fields	pos	radius	graphics_handle
1	[0.6625 0.3117]	0.0500	1x1 Rectangle
2	[0.3375 0.3117]	0.0500	1x1 Rectangle
3	[0.9219 0.0885]	0.0500	1x1 Rectangle
4	[0.0781 0.0885]	0.0500	1x1 Rectangle
5	[0.2713 0.8287]	0.0500	1x1 Rectangle
6	[0.7287 0.8287]	0.0500	1x1 Rectangle
7	[0.1159 0.8631]	0.0500	1x1 Rectangle
8	[0.8841 0.8631]	0.0500	1x1 Rectangle
9	[0.0770 0.2898]	0.0500	1x1 Rectangle
10	[0.9230 0.2898]	0.0500	1x1 Rectangle
11	[0.2473 0.4505]	0.0500	1x1 Rectangle
12	[0.7527 0.4505]	0.0500	1x1 Rectangle

Abbildung 5.10: Positionen und Radien der Minen im Feld `spiel.mine`

5.11.6 `spiel.tanke`

Die sechs Tankstellen des Strukturfelds `tanke` sind mit ihren Positionen und Radien in Abbildung 5.11 dargestellt.



Fields	pos	radius	graphics_handle
1	[0.5000 0.3934]	0.0100	1x1 Rectangle
2	[0.4013 0.1560]	0.0100	1x1 Rectangle
3	[0.8662 0.6011]	0.0100	1x1 Rectangle
4	[0.1338 0.6011]	0.0100	1x1 Rectangle
5	[0.2912 0.6119]	0.0100	1x1 Rectangle
6	[0.7088 0.6119]	0.0100	1x1 Rectangle

Abbildung 5.11: Positionen und Radien der Tankstellen im Feld `spiel.tanke`

6 rot_intro

Das von der Simulation (spaceballs) unabhängige Programm rot_intro stellt die Mitarbeiterinnen des (roten) Teams, ihre Aufgaben und ihr Teamlogo vor (Abbildung 6.1).

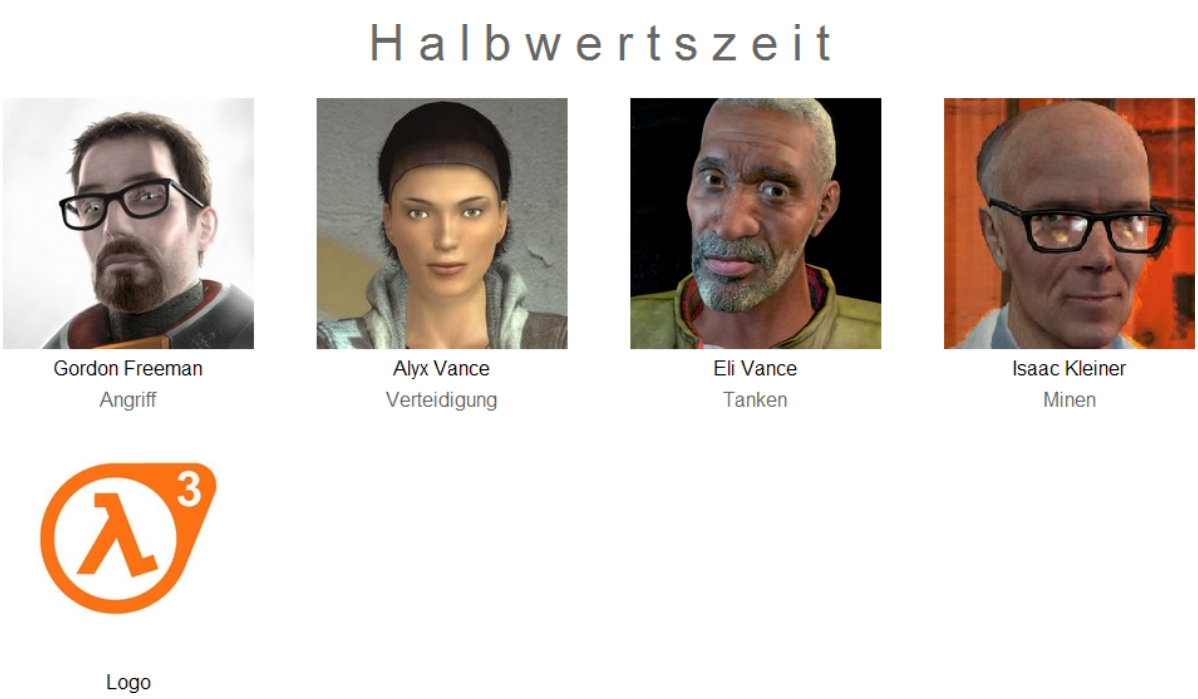


Abbildung 6.1: Vorstellung des Teams

6.1 Bilder und Sound

Jedes Team erstellt Bilder ihrer Mitarbeiterinnen, ein Logo und eine Sounddatei. Während des Spiels befinden sich die Medien des roten Teams im Ordner spaceballs/teams/rot/media.

Achtung Zur Abgabe des Projektes müssen alle Mediendateien einfach in einem Unterordner media liegen, der wiederum neben beschleunigung und team_daten liegt.

6.1.1 mitarbeiter.jpg

Jedes Team erstellt für jede ihrer Mitarbeiterinnen ein Kopfbild, auf dem die Mitarbeiterin eindeutig zu erkennen ist. Die Bilder müssen vom Typ JPG sein und eine einheitliche Größe von 200 × 200 Pixeln besitzen. Die Dateinamen ergeben sich direkt aus den in team_daten definierten Mitarbeiterinnennamen.

Beispiel: Gordon Freeman.jpg

6.1.2 Logo.jpg

Jedes Team erstellt ein Logo, das in rot_intro und während des Spiels über dem eigenen Teamnamen dargestellt wird. Das Logo muss eine JPG-Datei mit dem Namen Logo.JPG sein und eine Größe von 200 × 200 Pixeln besitzen.

6.1.3 intro.wav

Jedes Team erstellt eine maximal 10 Sekunden lange Sounddatei, die beim Anzeigen von rot_intro abgespielt wird. Die Datei muss intro.wav heißen und sich (während der Vorstellung des roten Teams) im Ordner spaceballs/teams/rot/media befinden.

Bitte erstellen Sie das Logo und die Sounddatei – im Gegensatz zum Dozenten, dem das natürlich auch sehr peinlich und unangenehm ist – selbst.

6.2 Code

Die folgenden Programmcodezeilen stellen die Bilder, Namen und Aufgaben der Mitarbeiterinnen in Form einer vierspaltigen Tabelle vor.

Als erstes löschen wir – wie üblich – alle vorhandenen Variablen, schließen alle Matlab-Fenster und leeren das Kommandofenster:

```
clear all
close all
clc
```

Um an die Daten des Teams zu kommen, verwenden wir teams_einlesen, das seinerseits, wie in Abbildung 4.1 dargestellt, das vom Team geschriebene Unterprogramm team_daten aufruft:

```
spiel = teams_einlesen;
```

Zur Darstellung öffnen wir ein Fenster in der Standardauflösung von 1000 × 600 Pixeln mit weißem Hintergrund und ersetzen den Fenstertitel durch unseren eigenen:

```
fenster_handle = figure ( ...
  'Position', [100 100 1000 600], ...
  'Menu', 'none', ...
  'NumberTitle', 'off', ...
  'Name', 'Intro', ...
  'color', 'white' ...
  );
```

Da wir den Teamnamen mit jeweils einem Leerzeichen zwischen zwei Buchstaben strecken wollen, ermitteln wir als erstes die Anzahl seiner Buchstaben:

```
n_name = length (spiel.rot.name);
```

Alsdann erzeugen wir eine Zeichenkette, in der doppelt[1] so viele Leerzeichen enthalten sind:

```
name_text = repmat (' ', 1, 2*n_name - 1);
```

Schließlich ersetzen wir jedes zweite Leerzeichen durch den entsprechenden Buchstaben des Teamnamens:

```
name_text(1 : 2 : 2*n_name) = spiel.rot.name;
```

Den hübsch formatierten Teamnamen stellen wir dann bei einer y-Koordinate von 520 Pixeln mit einer Höhe von 50 Pixeln in Grau[2] dar:

```
uicontrol ( ...
  'Style', 'text', ...
  'String', name_text, ...
  'FontUnits', 'normalized ', ...
  'FontSize', 0.8, ...
  'BackgroundColor', 'white', ...
  'ForegroundColor', [0.4 0.4 0.4], ...
  'Units', 'normalized', ...
  'Position', [0 520/600 1 50/600] ...
  )
```

Da wir davon ausgehen müssen, dass nicht alle Teams exakt vier[3] Mitarbeiterinnen haben, ermitteln wir deren tatsächliche Anzahl (einschließlich des Logos):

```
n_mitarbeiter = size (spiel.rot.mitarbeiter, 2);
```

[1] Als Länge der Zeichenkette verwenden wir hier `... - 1`, so dass das letzte Zeichen des Teamnamens auch tatsächlich an der letzten Position der Zeichenkette endet. In der Praxis würde ein zusätzliches Leerzeichen am Ende beim Zentrieren vermutlich niemandem auffallen – aber man hat ja auch seinen Stolz ...

[2] Da rot_intro ja von `spaceballs` unabhängig ist, können wir hier nicht auf die dort definierten Farben zurückgreifen und definieren die graue Farbe hier (nochmals) explizit.

[3] Dies könnte ja nur dann der Fall sein, wenn die Gesamtzahl der Studierenden zufälligerweise ein Vielfaches von vier wäre. Mehr als sieben Mitarbeiterinnen (plus Logo) können wir auf diese Art natürlich nicht darstellen.

Jetzt stellen wir in einer Schleife alle Mitarbeiterinnen in Form einer vierspaltigen Tabelle vor:

```
for i_mitarbeiter = 1 : n_mitarbeiter
```

Dazu verwenden wir als (nullbasierten) Zeilenindex den ganzzahligen Teiler des Mitarbeiterinnenindexes durch vier

```
  i_zeile = fix ((i_mitarbeiter - 1)/4);
```

und als (nullbasierten) Spaltenindex den bei der Ganzzahldivision durch vier auftretenden Rest:

```
  i_spalte = mod (i_mitarbeiter - 1, 4);
```

Mit diesen beiden Indizes können wir jetzt für das aktuelle Mitarbeiterinnenbild ein 200×200 Pixel großes Achsensystem erzeugen:

```
  mitarbeiter_handle = axes ( ...
    'Position', [ ...
    i_spalte*250/1000 + 25/1000 ...
    (1 - i_zeile)*250/600 + 50/600 ...
    200/1000 ...
    200/600] ...
    );
```

So beginnt das erste Achsensystem (`i_zeile = i_spalte = 0`) beispielsweise bei einer x-Koordinate von 25 Pixeln und einer y-Koordinate von 300 Pixeln. Das zweite Achsensystem hat dann mit `i_spalte = 1` eine x-Koordinate von 275 Pixeln, so dass zwischen den ersten beiden Koordinatensystemen ein horizontaler Abstand von 50 Pixeln liegt. Das erste Koordinatensystem der zweiten Zeile (`i_zeile = 1`, `i_spalte = 0`) besitzt entsprechend eine y-Koordinate von 50 Pixeln. Ergo beträgt auch der vertikale Abstand zwischen den Koordinatensystemen 50 Pixel.

Zur Anzeige lesen wir jetzt das Bild der aktuellen Mitarbeiterin ein

```
  bild = imread ( ...
    ['teams/rot/media/', ...
    spiel.rot.mitarbeiter(i_mitarbeiter).name, ...
    '.jpg']);
```

und stellen es im aktuellen Achsensystem dar:

```
  image (bild);
```

Leider müssen wir – wie schon in `spielfeld_darstellen` beschrieben – die Ränder und Beschriftungen des Achsensystems (wieder) unsichtbar machen, da der Befehl `image` sie ärgerlicherweise einblendet:

```
  set (mitarbeiter_handle, 'Visible', 'off')
```

Als nächstes schreiben wir den Namen der aktuellen Mitarbeiterin

```
  uicontrol ( ...
    'Style', 'text', ...
    'String', spiel.rot.mitarbeiter(i_mitarbeiter).name, ...
    'FontUnits', 'normalized ', ...
    'FontSize', 0.8, ...
    'BackgroundColor', 'white', ...
    'Units', 'normalized', ...
    'Position', [ ...
    i_spalte*250/1000 + 25/1000 ...
    (1 - i_zeile)*250/600 + 25/600 ...
    200/1000 ...
    20/600] ...
    )
```

und seine Aufgabe (in Grau) mittig unter sein Bild:

```
  uicontrol ( ...
    'Style', 'text', ...
    'String', spiel.rot.mitarbeiter(i_mitarbeiter).aufgabe, ...
    'FontUnits', 'normalized ', ...
    'FontSize', 0.8, ...
    'BackgroundColor', 'white', ...
    'ForegroundColor', [0.4 0.4 0.4], ...
    'Units', 'normalized', ...
    'Position', [ ...
    i_spalte*250/1000 + 25/1000 ...
    (1 - i_zeile)*250/600 ...
    200/1000 ...
    20/600] ...
    )

end
```

Schließlich laden wir die vom Team bereitgestellte Sounddatei

```
[daten, rate] = audioread ('teams/rot/media/intro.wav');
```

und spielen sie ab:

```
sound (daten, rate);
```

Teil III

Mathematisches Werkzeug

7 Vektorprojektion

Bevor wir die Kollision von Kreisen und Geraden analysieren, wollen wir als mathematisches Werkzeug die Projektion eines Vektors auf einen anderen Vektor und die Darstellung einer Gerade in Hessescher Normalform (Kapitel 8) verstehen. Die eigentlichen Kollisionsanalysen findet dann in Kapitel 9 und 10 statt.

Um den in Abbildung 7.1 dargestellten Projektionsvektor[1] $\mathbf{b_a}$ von $\mathbf{b}$ auf $\mathbf{a}$ zu berechnen, ermitteln wir zuerst seine Länge $b_a = |\mathbf{b_a}|$.

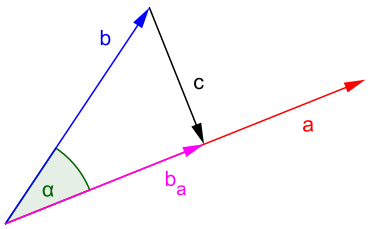


Abbildung 7.1: Projektionsvektor $\mathbf{b_a}$ des Vektors $\mathbf{b}$ auf den Vektor $\mathbf{a}$

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

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \cdot |\mathbf{b}| \cdot \cos\alpha = a \cdot b \cdot \cos\alpha \tag{7.1}$$

Solange die beiden Vektoren einen spitzen Winkel ($-90° < \alpha < 90°$) bilden, ist der Kosinus andererseits der Quotient von Ankathete und Hypotenuse im rechtwinkligen Dreieck in Abbildung 7.1:

$$\cos\alpha = \frac{b_a}{b} \tag{7.2}$$

Wenn wir Gleichung (7.2) in Gleichung (7.1) einsetzen, erhalten wir eine weitere Veranschaulichung des Skalarproduktes als Produkt der Länge b_a des Projektionsvektors mit der Länge a des Vektors, auf den projiziert wird:

$$\mathbf{a} \cdot \mathbf{b} = a \cdot b \cdot \frac{b_a}{b} = a \cdot b_a \tag{7.3}$$

[1] Vektoren schreiben wir hier und im Folgenden **fett** und nicht *kursiv*. Der Skalar a ist also der Betrag des Vektors $\mathbf{a}$. Projektionsvektoren definieren wir, indem wir den Vektor, auf den projiziert wird, als Index verwenden.

Gleichung (7.3) lösen wir nach der gesuchten Länge des Projektionsvektors auf:

$$b_a = \frac{\mathbf{a} \cdot \mathbf{b}}{a} \tag{7.4}$$

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:

$$\mathbf{x} = x \cdot \mathbf{x^0} \tag{7.5}$$

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 (7.5) erhalten, indem wir $\mathbf{a}$ durch seinen Betrag a teilen:

$$\mathbf{b_a} = b_a \cdot \mathbf{b_a^0} = b_a \cdot \mathbf{a^0} = b_a \frac{\mathbf{a}}{a} \tag{7.6}$$

Wenn wir jetzt noch die Länge des Projektionsvektors aus Gleichung (7.4) in Gleichung (7.6) einsetzen, erhalten wir:

$$\mathbf{b_a} = \frac{\mathbf{a} \cdot \mathbf{b}}{a} \cdot \frac{\mathbf{a}}{a} = \frac{\mathbf{a} \cdot \mathbf{b}}{a \cdot a} \cdot \mathbf{a} = \frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{a} \cdot \mathbf{a}} \cdot \mathbf{a} \tag{7.7}$$

In der letzten Umwandlung in Gleichung (7.7) haben wir die in Gleichung (10.8) beschriebene Tatsache verwendet, dass wir statt des Betragsquadrates auch das Skalarprodukt eines Vektors mit sich selbst verwenden können. Die letzte Form von Gleichung (7.7) 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.[2]

Den auf $\mathbf{a}$ und damit natürlich auch auf $\mathbf{b_a}$ senkrecht stehenden Verbindungsvektor $\mathbf{c}$ berechnen wir dann einfach als:

$$\mathbf{c} = \mathbf{b_a} - \mathbf{b}$$

[2] Natürlich können wir in Gleichung (7.7) 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.

8 Hessesche Normalform

In der Hesseschen Normalform können wir die in Abbildung 8.1 dargestellte grüne Gerade über ihren Abstand d vom Ursprung O und ihren vom Ursprung weg zeigenden Einheitsnormalenvektor $\mathbf{n}$ beschreiben.

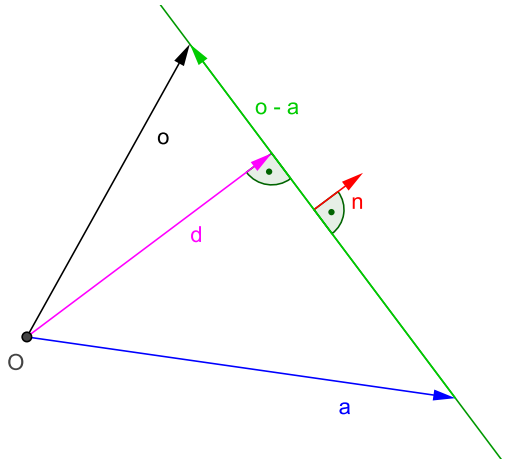


Abbildung 8.1: Hessesche Normalform

Dazu definieren wir den schwarzen, allgemeinen, variablen Ortsvektor $\mathbf{o}$ zur Gerade und einen blauen Ortsvektor $\mathbf{a}$ zu einem beliebigen aber festen Punkt auf der Gerade. Der grüne Differenzvektor $\mathbf{o} - \mathbf{a}$ ist dann ein Richtungsvektor der Gerade. Da der Einheitsnormalenvektor $\mathbf{n}$ senkrecht auf der Gerade 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:

$$\mathbf{o} \cdot \mathbf{n} = \mathbf{a} \cdot \mathbf{n} \tag{8.1}$$

In Gleichung (7.3) 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 (8.1) umschreiben können

$$\mathbf{o} \cdot \mathbf{n} = a_n \cdot n \tag{8.2}$$

wobei a_n der Betrag des Projektionsvektors $\mathbf{a_n}$ ist, der sich ergibt, wenn wir $\mathbf{a}$ auf $\mathbf{n}$ projizieren. In Abbildung 8.1 wird deutlich, dass die Projektion von $\mathbf{a}$ auf $\mathbf{n}$ dem Abstandsvektor $\mathbf{d}$ entspricht, der ja auch senkrecht auf der Gerade steht und damit kollinear zu $\mathbf{n}$ ist:[1]

$$\mathbf{a_n} = \mathbf{d}$$

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 (8.2) zu:

$$\mathbf{o} \cdot \mathbf{n} = d \tag{8.3}$$

In der Hesseschen Normalform können wir den Abstand eines Punktes zur Gerade 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 8.2 grün gestrichelt dargestellte Gerade, die natürlich noch den gleichen Normalenvektor, aber einen anderen Abstand vom Ursprung hat.

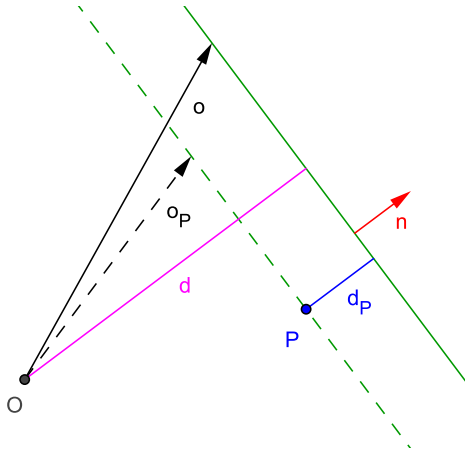


Abbildung 8.2: Abstand des Punktes P von der Gerade $\mathbf{o} \cdot \mathbf{n} = d$

Da wir die Gerade ja genau um den gesuchten Abstand d_P des Punktes von der Gerade verschoben haben, hat die verschobene Gerade den Abstand $d - d_p$ vom Ursprung und besitzt damit gemäß Gleichung (8.3) die Gleichung:

$$\mathbf{o_P} \cdot \mathbf{n} = d - d_P \tag{8.4}$$

Gleichung (8.4) können wir nach dem gesuchten Abstand auflösen und erhalten:

$$d_P = d - \mathbf{o_P} \cdot \mathbf{n} \tag{8.5}$$

[1] Der Abstandsvektor $\mathbf{d}$ ist auch ein Normalenvektor der Gerade. Der Einheitsnormalenvektor $\mathbf{n}$ ist damit der normierte Abstandsvektor: $\mathbf{n} = \frac{\mathbf{d}}{d}$

Da $\mathbf{o_P}$ dabei der allgemeine Ortsvektor zu einen beliebigen Punkt auf der verschobenen Gerade ist, können wir jetzt den Ortsvektor zu jedem Punkt der Gerade (also auch zum Punkt P selbst) für $\mathbf{o_P}$ in Gleichung (8.5) einsetzen, um seinen Abstand zur Gerade zu bestimmen. Da die Differenz auf der rechten Seite von Gleichung (8.5) 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 ($d_P < 0$).

8.1 Beispiele

Der in Abbildung 8.1 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 Gerade lautet daher:

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

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

Wenn wir beispielsweise den in Abbildung 8.1 verwendeten Ortsvektor $\mathbf{a} = \begin{bmatrix} 7 \\ -1 \end{bmatrix}$ für $\mathbf{o_P}$ in Gleichung (8.5) 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 Gerade.

Der in Abbildung 8.2 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 8.2 deutlich sichtbar – auf der gleichen Geradenseite wie der Ursprung.

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

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

9 Kollision eines Kreises mit einer Gerade

Nachdem wir jetzt die notwendigen Werkzeuge beherrschen (Kapitel 8), können wir damit die Kollision eines Kreises mit einer Gerade analysieren.

In Abbildung 9.1 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 9.1 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 Gerade steht und vom Ursprung weg zeigt (eingezeichnet im Punkt F), definiert ist.

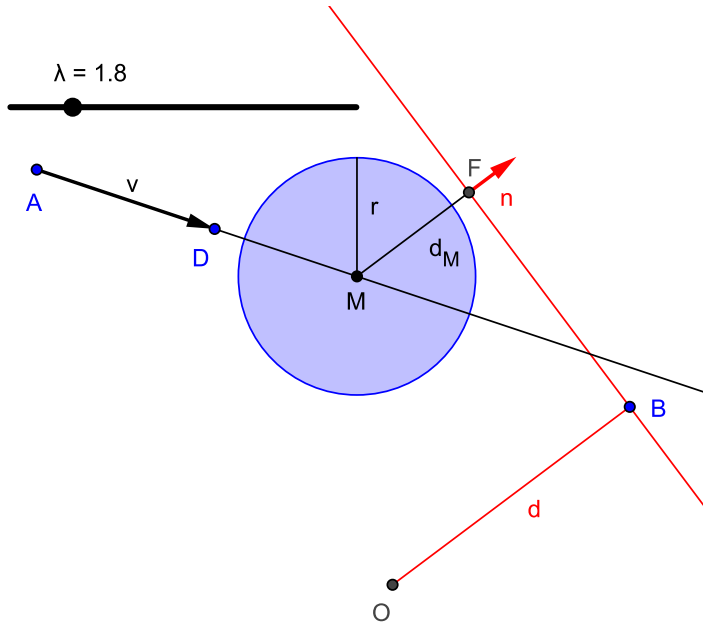


Abbildung 9.1: Kollision eines Kreises mit einer Gerade

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

$$\mathbf{o_M} = \mathbf{o_A} + \lambda \cdot \mathbf{v} \tag{9.1}$$

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
- λ der normierte Zeitparameter, der während der Bewegung von 0 bis ∞ läuft
- $\mathbf{v} = \mathbf{o_D} - \mathbf{o_A}$ der Geschwindigkeitsvektor, mit dem sich der Kreis auf der Gerade 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 Gerade berechnen, indem wir den Ortsvektor zum Kreismittelpunkt in Gleichung (8.5) einsetzen:

$$d_M = d - \mathbf{o_M} \cdot \mathbf{n} \tag{9.2}$$

Für den Kreismittelpunktsortsvektor setzen wir nun die in Gleichung (9.1) definierte Geradengleichung in Gleichung (9.2) ein

$$\begin{aligned} d_M &= d - (\mathbf{o_A} + \lambda \cdot \mathbf{v}) \cdot \mathbf{n} \\ &= d - \mathbf{o_A} \cdot \mathbf{n} - \lambda \cdot \mathbf{v} \cdot \mathbf{n} \end{aligned} \tag{9.3}$$

und lösen Gleichung (9.3) nach λ auf:

$$\lambda = \frac{d - d_M - \mathbf{o_A} \cdot \mathbf{n}}{\mathbf{v} \cdot \mathbf{n}} \tag{9.4}$$

Wir suchen jetzt das λ_1, bei dem der Kreis die Gerade erstmalig berührt. In diesem Fall ist der Kreismittelpunktsabstand d_M gerade gleich dem Kreisradius r:

$$\lambda_1 = \frac{d - r - \mathbf{o_A} \cdot \mathbf{n}}{\mathbf{v} \cdot \mathbf{n}} \tag{9.5}$$

In 8 haben wir gezeigt, dass der Abstand für Punkte auf der anderen Seite der Gerade negativ ist. Wir können daher auch den Berührpunkt auf der anderen Seite der Gerade berechnen, indem wir für d_M in Gleichung (9.4) nicht r sondern $-r$ einsetzen:

$$\lambda_2 = \frac{d + r - \mathbf{o_A} \cdot \mathbf{n}}{\mathbf{v} \cdot \mathbf{n}} \tag{9.6}$$

Die Kreismittelpunkte zu den Berührzeitpunkten erhalten wir schließlich, indem wir λ_1 und λ_2 in Gleichung (9.1) einsetzen.

9.1 Beispiel

In Abbildung 9.1 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 (9.5)

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

und nach Gleichung (9.6):

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

Um die Ortsvektoren zu den Kreismittelpunkten zum Zeitpunkt der Berührung zu erhalten, setzen wir λ_1 und λ_2 in Gleichung (9.1) ein:

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

10 Kollision zweier Kreise

In diesem Kapitel 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 10.1 bewegt sich der rote Kreis mit dem Radius r_1 und dem Mittelpunkt M_1 auf der Gerade, 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 Gerade, 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 10.1 sehen wir die Momentaufnahme der Bewegung zum Zeitpunkt $\lambda = 0.54$.

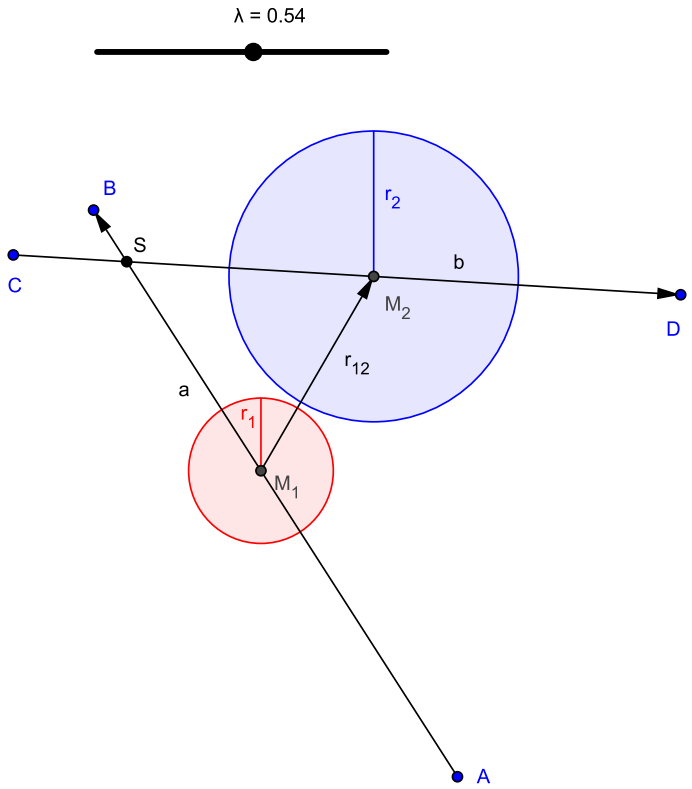


Abbildung 10.1: Keine Kollision, obwohl sich die Geraden im Punkt S schneiden

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 10.1 dargestellten Zeitpunkt kommen sich die Kreise niemals.

In der in Abbildung 10.2 und Abbildung 10.3 dargestellten Anordnung der Anfangs- und Endpunkte hingegen berühren sich die Kreise zweimal. Zum ersten Mal treffen sie zum Zeitpunkt λ=0.56 (Abbildung 10.2), aufeinander und dann verlassen sie einander zum Zeitpunkt λ=0.94 (Abbildung 10.3) wieder [11].

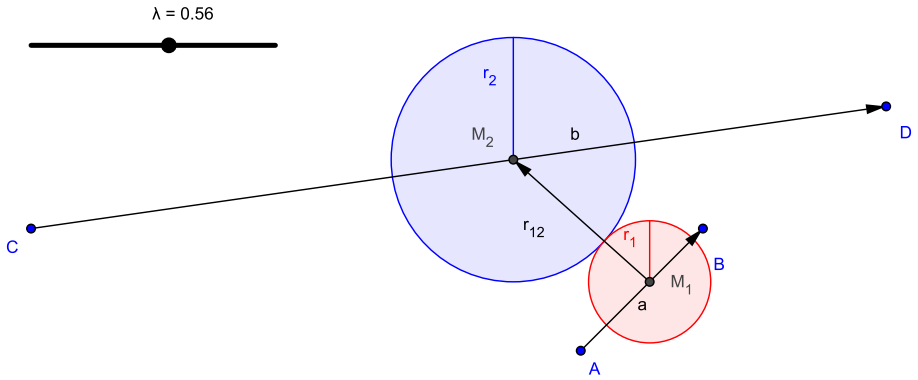


Abbildung 10.2: Erster Berührpunkt zum Zeitpunkt λ=0.56

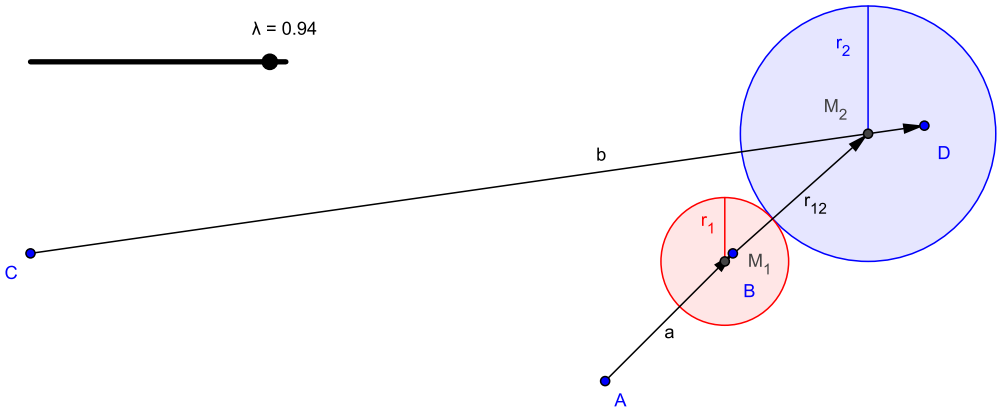


Abbildung 10.3: Zweiter Berührpunkt zum Zeitpunkt λ=0.94

Zwischen den beiden Berührzeitpunkten überlappen sich die Kreise. Unser Ziel ist es nun, die Zeitpunkte λ_1 und λ_2 der beiden Berührungen und die Orte der Mittelpunkte beider Kreise zu den Berührzeitpunkten zu berechnen. Dazu definieren wir die Gleichung der ersten Gerade in Parameterform:

$$\mathbf{o_1} = \mathbf{o_A} + \lambda \cdot \mathbf{a} \tag{10.1}$$

Dabei sind

- $\mathbf{o_1}$ der Ortsvektor zum Mittelpunkt M_1 des ersten Kreises auf der ersten Gerade
- $\mathbf{o_A}$ der Ortsvektor zum Anfangspunkt A der ersten Gerade
- $\mathbf{a} = \mathbf{o_B} - \mathbf{o_A}$ der Richtungsvektor der ersten Gerade vom Anfangspunkt A zum Endpunkt B
- λ der (Zeit-)Parameter der ersten (und zweiten) Gerade

Die zweite Gerade definieren wir entsprechend:

$$\mathbf{o_2} = \mathbf{o_C} + \lambda \cdot \mathbf{b} \tag{10.2}$$

Dabei sind

- $\mathbf{o_2}$ der Ortsvektor zum Mittelpunkt M_2 des zweiten Kreises auf der zweiten Gerade
- $\mathbf{o_C}$ der Ortsvektor zum Anfangspunkt C der zweiten Gerade
- $\mathbf{b} = \mathbf{o_D} - \mathbf{o_C}$ der Richtungsvektor der zweiten Gerade vom Anfangspunkt C zum Endpunkt D
- λ der (Zeit-)Parameter der (ersten und) zweiten Gerade

Wenn wir λ 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 λ, an dem sich beide Kreise berühren. Dazu berechnen wir den Vektor $\mathbf{r_{12}}$, der die beiden Kreismittelpunkte miteinander verbindet, als Differenz der Mittelpunktsortsvektoren:

$$\mathbf{r_{12}} = \mathbf{o_2} - \mathbf{o_1} \tag{10.3}$$

Wenn wir jetzt die beiden Geradengleichungen Gleichung (10.1) und Gleichung (10.2) in Gleichung (10.3) einsetzen, erhalten wir:

$$\mathbf{r_{12}} = \mathbf{o_C} + \lambda\mathbf{b} - (\mathbf{o_A} + \lambda\mathbf{a}) = \mathbf{o_C} - \mathbf{o_A} + \lambda\,(\mathbf{b} - \mathbf{a}) \tag{10.4}$$

Zur Vereinfachung definieren wir die beiden Vektordifferenzen in Gleichung (10.4) als neue Differenzvektoren $\mathbf{e}$ und $\mathbf{f}$:

$$\begin{aligned} \mathbf{e} &= \mathbf{o_C} - \mathbf{o_A} \\ \mathbf{f} &= \mathbf{b} - \mathbf{a} \end{aligned} \tag{10.5}$$

Der Mittelpunktabstandsvektor vereinfacht sich dann zu:

$$\mathbf{r_{12}} = \mathbf{e} + \lambda\mathbf{f} \tag{10.6}$$

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 10.2):

$$|\mathbf{r_{12}}| = r_{12} = r_1 + r_2 \tag{10.7}$$

Um nun das λ 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:
>
> $$|\mathbf{x}|^2 = x^2 = \left(\sqrt{x_1^2 + x_2^2 + \ldots}\right)^2 = x_1^2 + x_2^2 + \cdots = \mathbf{x} \cdot \mathbf{x} \tag{10.8}$$

Wir wenden Gleichung (10.8) auf Gleichung (10.7) an, setzen Gleichung (10.6) ein und erhalten:

$$|\mathbf{r_{12}}|^2 = (\mathbf{e} + \lambda\mathbf{f}) \cdot (\mathbf{e} + \lambda\mathbf{f}) = r_{12}^2 \tag{10.9}$$

Das Skalarprodukt können wir ausmultiplizieren und sortieren:

$$\lambda^2\mathbf{f} \cdot \mathbf{f} + 2\lambda\mathbf{e} \cdot \mathbf{f} + \mathbf{e} \cdot \mathbf{e} = r_{12}^2 \tag{10.10}$$

Da die einzelnen Skalarprodukte in Gleichung (10.10) natürlich Skalare sind, können wir sie weiter vereinfachen, indem wir folgende Ersetzungen durchführen:

$$\begin{aligned} \alpha &= \mathbf{f} \cdot \mathbf{f} \\ \beta &= \mathbf{e} \cdot \mathbf{f} \\ \gamma &= \mathbf{e} \cdot \mathbf{e} - r_{12}^2 \end{aligned} \tag{10.11}$$

Gleichung (10.10) wird dann zu einer einfachen quadratischen Gleichung in λ

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

die nach der vereinfachten Mitternachtsformel [12] genau dann reelle Lösungen hat, wenn die Diskriminante δ positiv ist:

$$\delta = \beta^2 - \alpha\gamma > 0 \tag{10.12}$$

Die beiden Lösungen lauten dann:

$$\lambda_{1,2} = \frac{-\beta \pm \sqrt{\delta}}{\alpha} \tag{10.13}$$

Die beiden Zeitpunkte λ_1 und λ_2 können wir schließlich in Gleichung (10.1) und Gleichung (10.2) einsetzen, um die Orte der Kreismittelpunkte bei Berührung zu berechnen.

10.1 Beispiel

Die Zahlenwerte für die in Abbildung 10.2 und Abbildung 10.3 dargestellte Situation [11] 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 (10.5) 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 (10.7) die Länge des Mittelpunktabstandsvektors:

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

Jetzt können wir nach Gleichung (10.11) 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 λ lautet also:

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

Um zu überprüfen, ob es überhaupt reelle Berührpunkte gibt, untersuchen wir die Diskriminante nach Gleichung (10.12):

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

Da die Diskriminante positiv ist, gibt es zwei Berührzeitpunkte:

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

Die Mittelpunkte der Kreise zu den Berührzeitpunkten finden wir durch Einsetzen der Berührzeitpunkte in die Geradengleichungen (Gleichung (10.1) und Gleichung (10.2)):

$\lambda = \lambda_1$:

$$\mathbf{o_{11}} = \mathbf{o_A} + \lambda_1 \cdot \mathbf{a} = \begin{bmatrix}10\\1\end{bmatrix} + 0.5637 \cdot \begin{bmatrix}2\\2\end{bmatrix} = \begin{bmatrix}11.1273\\2.1273\end{bmatrix}$$
$$\mathbf{o_{21}} = \mathbf{o_C} + \lambda_1 \cdot \mathbf{b} = \begin{bmatrix}1\\3\end{bmatrix} + 0.5637 \cdot \begin{bmatrix}14\\2\end{bmatrix} = \begin{bmatrix}8.8913\\4.1273\end{bmatrix}$$

$\lambda = \lambda_2$:

$$\mathbf{o_{12}} = \mathbf{o_A} + \lambda_2 \cdot \mathbf{a} = \begin{bmatrix}10\\1\end{bmatrix} + 0.9363 \cdot \begin{bmatrix}2\\2\end{bmatrix} = \begin{bmatrix}11.8727\\2.8727\end{bmatrix}$$
$$\mathbf{o_{22}} = \mathbf{o_C} + \lambda_2 \cdot \mathbf{b} = \begin{bmatrix}1\\3\end{bmatrix} + 0.9363 \cdot \begin{bmatrix}14\\2\end{bmatrix} = \begin{bmatrix}14.1087\\4.8727\end{bmatrix}$$

Diese beiden Berührpunkte können wir sehr schön veranschaulichen, wenn wir in [11] die beiden berechneten Berührzeitpunkte einstellen.

11 Kreis durch drei Punkte

Wir wollen in diesem Kapitel den Mittelpunkt M und den Radius r des in Abbildung 11.1 dargestellten Kreises berechnen, der durch die drei Punkte P_1, P_2 und P_3 definiert ist.

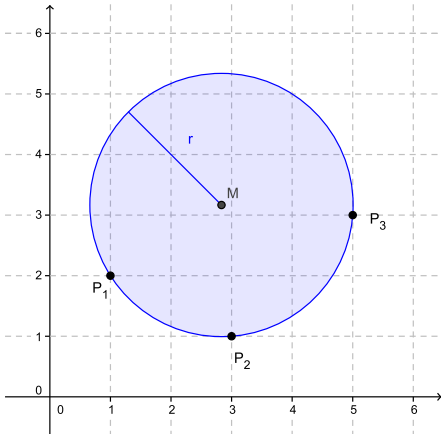


Abbildung 11.1: Kreis durch drei Punkte

Dazu verwenden wir die bekannte Kreisgleichung

$$(x - x_M)^2 + (y - y_M)^2 = r^2 \tag{11.1}$$

wobei x_M und y_M natürlich die Komponenten des Ortsvektors $\mathbf{o_M}$ zum Kreismittelpunkt M sind:

$$\mathbf{o_M} = \begin{bmatrix} x_M \\ y_M \end{bmatrix}$$

Wir quadrieren Gleichung (11.1) aus

$$x^2 - 2xx_M + x_M^2 + y^2 - 2yy_M + y_M^2 = r^2$$

und sortieren ein wenig um:

$$2xx_M + 2yy_M + r^2 - x_M^2 - y_M^2 = x^2 + y^2$$

Als nächstes führen wir drei Abkürzungen ein

$$\begin{aligned} \alpha &= 2x_M \\ \beta &= 2y_M \\ \gamma &= r^2 - x_M^2 - y_M^2 \end{aligned} \tag{11.2}$$

und erhalten so eine einfache lineare Gleichung für die drei Unbekannten α, β und γ:

$$\alpha x + \beta y + \gamma = x^2 + y^2 \tag{11.3}$$

Für drei Unbekannte brauchen wir jetzt drei Gleichungen. Dazu setzen wir die Komponenten der Ortsvektoren zu den drei Punkten

$$\mathbf{o_{P_1}} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \qquad \mathbf{o_{P_2}} = \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} \qquad \mathbf{o_{P_3}} = \begin{bmatrix} x_3 \\ y_3 \end{bmatrix}$$

jeweils in Gleichung (11.3) ein

$$\begin{aligned} \alpha x_1 + \beta y_1 + \gamma &= x_1^2 + y_1^2 \\ \alpha x_2 + \beta y_2 + \gamma &= x_2^2 + y_2^2 \\ \alpha x_3 + \beta y_3 + \gamma &= x_3^2 + y_3^2 \end{aligned}$$

und erhalten so ein lineares Gleichungssystem für die drei Unbekannten α, β und γ.

Das lineare Gleichungssystem können wir natürlich auch in Matrizenschreibweise darstellen:

$$\begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ x_3^2 + y_3^2 \end{bmatrix}$$

Wenn wir jetzt noch die Matrix und die Vektoren abkürzen

$$\mathbf{A} = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix} \qquad \boldsymbol{\xi} = \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ x_3^2 + y_3^2 \end{bmatrix} \tag{11.4}$$

können wir das Gleichungssystem sehr kompakt darstellen:

$$\mathbf{A}\boldsymbol{\xi} = \mathbf{b}$$

In Matlab[1] können wir lineare Gleichungssysteme mit dem Backslash-Operator sehr effizient numerisch lösen. Die kurze Quelltextzeile

[1] Matlab ist ja die Abkürzung von Matrix Laboratory und kann damit traditionell besonders gut mit Problemen der linearen Algebra umgehen. Schon die allererste, fast 40 Jahre alte Matlabversion [13] konnte eine LU-Zerlegung von Matrizen durchführen und damit lineare Gleichungssysteme lösen.

```
xi = A\b
```

liefert uns in einem Schritt den Vektor $\boldsymbol{\xi}$ mit den drei gesuchten Komponenten α, β und γ. Im letzten Schritt müssen wir dann nur noch die in Gleichung (11.2) definierten Abkürzungen umkehren, um die gesuchten Mittelpunktskoordinaten und den Radius zu finden:

$$\begin{aligned} x_M &= \frac{\alpha}{2} \\ y_M &= \frac{\beta}{2} \\ r &= \sqrt{\gamma + x_M^2 + y_M^2} \end{aligned} \tag{11.5}$$

11.1 Beispiel

Die drei den Kreis definierenden Punkte in Abbildung 11.1 besitzen folgende Ortsvektoren:

$$\begin{aligned} \mathbf{o_{P_1}} &= \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \\ \mathbf{o_{P_2}} &= \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \end{bmatrix} \\ \mathbf{o_{P_3}} &= \begin{bmatrix} x_3 \\ y_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \end{bmatrix} \end{aligned}$$

Wenn wir die Komponenten in Gleichung (11.4) einsetzen, erhalten wir die Koeffizientenmatrix $\mathbf{A}$ und den Vektor der rechten Seite $\mathbf{b}$ des linearen Gleichungssystems:

$$\mathbf{A} = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \\ 3 & 1 & 1 \\ 5 & 3 & 1 \end{bmatrix} \tag{11.6}$$

$$\mathbf{b} = \begin{bmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ x_3^2 + y_3^2 \end{bmatrix} = \begin{bmatrix} 1^2 + 2^2 \\ 3^2 + 1^2 \\ 5^2 + 3^2 \end{bmatrix} = \begin{bmatrix} 5 \\ 10 \\ 34 \end{bmatrix} \tag{11.7}$$

Lassen wir jetzt Matlab das Gleichungssystem mit Hilfe des Backslash-Operators numerisch lösen

```
xi = A\b
```

so erhalten wir den folgenden Lösungsvektor

$$\boldsymbol{\xi} = \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} 5.\bar{6} \\ 6.\bar{3} \\ -13.\bar{3} \end{bmatrix}$$

aus dem wir nach Gleichung (11.5) die Mittelpunktskoordinaten und den Radius des Kreises ausrechnen können:

$$\begin{aligned} x_M &= \frac{\alpha}{2} = \frac{5.\bar{6}}{2} = 2.8\bar{3} \\ y_M &= \frac{\beta}{2} = \frac{6.\bar{3}}{2} = 3.1\bar{6} \\ r &= \sqrt{\gamma + x_M^2 + y_M^2} = \sqrt{-13.\bar{3} + 2.8\bar{3}^2 + 3.1\bar{6}^2} = \sqrt{4.7\bar{2}} = 2.17 \end{aligned}$$

11.2 Alternativer Weg

Alternativ könnten wir versuchen, die drei Punkte symbolisch in die Kreisgleichung einzusetzen und das daraus folgende nichtlineare Gleichungssystem mit Hilfe der Symbolic-Toolbox von Matlab analytisch lösen zu lassen:

```
syms x y
syms x_1 x_2 x_3
syms y_1 y_2 y_3
syms r positive
syms xm ym

kreis = (x - xm)^2 + (y - ym)^2 == r^2

k1 = subs (kreis, {x, y}, {x_1, y_1})
k2 = subs (kreis, {x, y}, {x_2, y_2})
k3 = subs (kreis, {x, y}, {x_3, y_3})

[r, xm, ym] = solve (k1, k2, k3, r, xm, ym)

r = simplify (r)
xm = simplify (xm)
ym = simplify (ym)
```

Und tatsächlich findet Matlab allgemeine Lösungen für den Radius[2] und die Mittelpunktskoordinaten:

```
r = -((x_1*y_2 - x_2*y_1 - x_1*y_3 + x_3*y_1 + x_2*y_3 - x_3*y_2
   )^2*(x_1^2 - 2*x_1*x_2 + x_2^2 + y_1^2 - 2*y_1*y_2 + y_2^2)*(
   x_1^2 - 2*x_1*x_3 + x_3^2 + y_1^2 - 2*y_1*y_3 + y_3^2)*(x_2^2
   - 2*x_2*x_3 + x_3^2 + y_2^2 - 2*y_2*y_3 + y_3^2))^(1/2)/(2*(
   x_1*y_2 - x_2*y_1 - x_1*y_3 + x_3*y_1 + x_2*y_3 - x_3*y_2)^2)
```

[2]Beim (natürlich positiven) Radius müssen wir – in Abhängigkeit von den Punktekoordinaten – eventuell das Vorzeichen korrigieren.

```
xm = (x_1^2*y_2 - x_1^2*y_3 - x_2^2*y_1 + x_2^2*y_3 + x_3^2*y_1
   - x_3^2*y_2 + y_1^2*y_2 - y_1^2*y_3 - y_1*y_2^2 + y_1*y_3^2 +
   y_2^2*y_3 - y_2*y_3^2)/(2*(x_1*y_2 - x_2*y_1 - x_1*y_3 + x_3*
   y_1 + x_2*y_3 - x_3*y_2))
```

```
ym = (- x_1^2*x_2 + x_1^2*x_3 + x_1*x_2^2 - x_1*x_3^2 + x_1*y_2
   ^2 - x_1*y_3^2 - x_2^2*x_3 + x_2*x_3^2 - x_2*y_1^2 + x_2*y_3^2
    + x_3*y_1^2 - x_3*y_2^2)/(2*(x_1*y_2 - x_2*y_1 - x_1*y_3 +
   x_3*y_1 + x_2*y_3 - x_3*y_2))
```

Wir könnten diese Formeln jetzt direkt per Kopieren-und-Einfügen in eine neue Matlab-Funktion einbauen und mit ihr ohne Backslash-Operator die gesuchten Kreisparameter berechnen lassen. Ohne dies näher quantifiziert zu haben, ist der Autor aber der Meinung, dass die vorher vorgestellte Variante mit dem Backslash-Operator die elegantere, effizientere und weitaus weniger fehleranfälligere Methode darstellt.

12 Tangenten an zwei Kreise

Unser Ziel ist es in diesem Kapitel, die vier gemeinsamen Tangenten zweier Kreise zu bestimmen. Wenn wir dem einen Kreis einen verschwindenden Radius geben, erhalten wir – darin als Sonderfall enthalten – die beiden Tangenten von einem Punkt an einen Kreis.

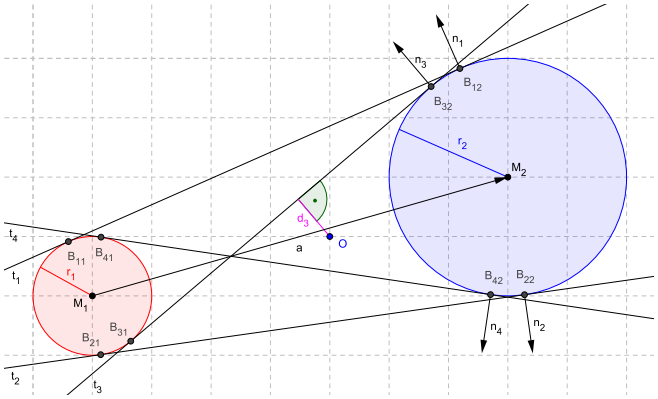


Abbildung 12.1: Tangenten an zwei Kreise

In Abbildung 12.1 sind die beiden Kreise mit den Mittelpunkten M_1 und M_2 und den Radien r_1 und r_2 dargestellt. Wir suchen dann die vier Tangenten $t_1 \dots t_4$, die jeweils beide Kreise berühren. Außerdem möchten wir die acht Berührpunkte $B_{11} \dots B_{42}$ bestimmen. Dabei verweist der erste Index auf die Tangente und der zweite Index auf den Kreis. Der Punkt B_{32} ist also der Berührpunkt der dritten Tangente mit dem zweiten Kreis. Die dritte Tangente t_3 ist dann durch die beiden Berührpunkte B_{31} und B_{32} eindeutig festgelegt.

Des Weiteren haben wir in Abbildung 12.1 die Einheitsnormalenvektoren $\mathbf{n_1} \dots \mathbf{n_4}$ eingezeichnet, die senkrecht auf den Tangenten stehen und eine Länge von eins besitzen. Ihre Orientierung haben wir gemäß Kapitel 8 so gewählt, dass alle Normalenvektoren vom Ursprung O weg zeigen.

Zur Berechnung einer Tangente drücken wir diese in der Hesseschen Normalform (Kapitel 8) aus. Die Gleichung der dritten Tangente t_3 lautet dann beispielsweise entsprechend Gleichung (8.3):

$$\mathbf{o_3} \cdot \mathbf{n_3} = d_3$$

wobei $\mathbf{n_3}$ der Einheitsnormalenvektor auf der Tangente t_3, d_3 der (beispielhaft eingezeichnete) senkrechte Abstand der Tangente vom Ursprung O und $\mathbf{o_3}$ der allgemeine Ortsvektor zu einem beliebigen Punkt der Tangente ist.

12.1 Tangenten t_1 und t_2

Gemäß Gleichung (8.5) können wir in der Hesseschen Normalform den Abstand eines Punktes von einer Gerade besonders einfach berechnen, indem wir den Ursprungsabstand der in den Punkt verschobenen Gerade vom Ursprungsabstand der Gerade selbst abziehen. Auf diese Weise drücken wir beispielsweise den (bekannten) Abstand r_1 des ersten Kreismittelpunktes M_1 von der ersten Tangente t_1 aus:

$$r_1 = d_1 - \mathbf{o_{M_1}} \cdot \mathbf{n_1} \tag{12.1}$$

Dabei sind

- r_1 der bekannte Radius des ersten Kreises
- d_1 der (aus Übersichtlichkeitsgründen in Abbildung 12.1 nicht eingezeichnete) gesuchte senkrechte Abstand der ersten Tangente t_1 vom Ursprung
- $\mathbf{o_{M_1}}$ der (ebenfalls nicht eingezeichnete) bekannte Ortsvektor vom Ursprung zum Mittelpunkt des ersten Kreises
- $\mathbf{n_1}$ der gesuchte Einheitsnormalenvektor auf der Tangente t_1

Da wir in Gleichung (12.1) jetzt die drei Parameter der Tangente t_1 suchen, nämlich ihren Abstand d_1 und die beiden Komponenten ihres Normalenvektors $\mathbf{n_1}$, brauchen wir zu deren Bestimmung noch zwei weitere Gleichungen. Dazu berechnen wir auch den Abstand des zweiten Kreismittelpunktes von der ersten Tangente:

$$r_2 = d_1 - \mathbf{o_{M_2}} \cdot \mathbf{n_1} \tag{12.2}$$

Als dritte Gleichung verwenden wir die Tatsache, dass alle Tangenteneinheitsnormalenvektoren einen Betrag von eins besitzen:

$$n_x^2 + n_y^2 = 1 \tag{12.3}$$

Durch geschicktes Kombinieren der drei Gleichungen 12.1, 12.2 und 12.3 können wir jetzt die gesuchten Parameter der ersten Tangente berechnen. Dazu subtrahieren wir Gleichung (12.2) von Gleichung (12.1) und eliminieren auf diese Weise den unbekannten Tangentenursprungsabstand d_1:

$$\begin{aligned} r_1 - r_2 &= d_1 - \mathbf{o_{M_1}} \cdot \mathbf{n_1} - (d_1 - \mathbf{o_{M_2}} \cdot \mathbf{n_1}) \\ &= (\mathbf{o_{M_2}} - \mathbf{o_{M_1}}) \cdot \mathbf{n_1} \end{aligned} \tag{12.4}$$

In Gleichung (12.4) kürzen wir jetzt die Radiendifferenz mit

$$r = r_1 - r_2 \tag{12.5}$$

und den in Abbildung 12.1 eingezeichneten Mittelpunkteabstandsvektor mit

$$\mathbf{a} = \mathbf{o_{M_2}} - \mathbf{o_{M_1}} \tag{12.6}$$

ab und erhalten so eine sehr kompakte Gleichung

$$r = \mathbf{a} \cdot \mathbf{n_1} \tag{12.7}$$

in der nur noch die beiden Komponenten des Normalenvektors $\mathbf{n_1}$ unbekannt sind. Um Gleichung (12.7) zusammen mit der skalaren Gleichung (12.3) verwenden zu können, multiplizieren wir das Skalarprodukt aus und erhalten:

$$\begin{aligned} r &= \mathbf{a} \cdot \mathbf{n_1} \\ &= \begin{bmatrix} a_x \\ a_y \end{bmatrix} \cdot \begin{bmatrix} n_x \\ n_y \end{bmatrix} \\ &= a_x n_x + a_y n_y \end{aligned} \tag{12.8}$$

Dabei verzichten wir aus Übersichtlichkeitsgründen bei den Komponenten des Normalenvektors auf den weiteren Index 1, der uns ja nur daran erinnert, dass wir gerade die erste Tangente ermitteln. Gleichung (12.8) lösen wir nach der zweiten Normalenvektorkomponente n_y auf[1]

$$n_y = \frac{r - a_x n_x}{a_y} \tag{12.9}$$

und setzen Gleichung (12.9) in die Betragsbedingung Gleichung (12.3) ein:

$$n_x^2 + \left(\frac{r - a_x n_x}{a_y} \right)^2 = 1 \tag{12.10}$$

Nach Multiplikation von Gleichung (12.10) mit a_y^2 und Ausquadrieren erhalten wir:

$$a_y^2 n_x^2 + r^2 - 2 r a_x n_x + a_x^2 n_x^2 = a_y^2$$

Durch Umsortieren und Zusammenfassen erkennen wir, dass wir, wie schon in Kapitel 10, eine quadratische Gleichung in der einzigen verbleibenden Unbekannten n_x vorliegen haben:

$$\left(a_y^2 + a_x^2 \right) n_x^2 - 2 r a_x n_x + r^2 - a_y^2 = 0 \tag{12.11}$$

[1] Wir müssen daher später unbedingt dafür sorgen, dass a_y nicht null wird.

Auch hier führen wir, analog zu Gleichung (10.11), weitere Abkürzungen ein

$$\begin{aligned} \alpha &= a_x^2 + a_y^2 \\ \beta &= -ra_x \\ \gamma &= r^2 - a_y^2 \end{aligned} \tag{12.12}$$

sodass wir Gleichung (12.11) in der üblichen Mitternachtsform [12] einer quadratischen Gleichung darstellen können

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

die ja bekanntlich genau dann reelle Lösungen hat, wenn die Diskriminante δ positiv ist:

$$\delta = \beta^2 - \alpha\gamma > 0 \tag{12.13}$$

Die beiden Lösungen lauten dann:

$$n_{x_{1,2}} = \frac{-\beta \pm \sqrt{\delta}}{\alpha} \tag{12.14}$$

Da alle bislang gemachten Annahmen sowohl für die Tangente t_1 als auch für die Tangente t_2 gelten, liefern uns die beiden n_x freundlicherweise auch gleich die Parameter beider Tangenten[2] t_1 und t_2.

Die gefundenen n_x setzen wir in Gleichung (12.9) ein, um die zugehörigen n_y zu erhalten. Zusammen bilden die jeweiligen n_x- und n_y-Paare dann die gesuchten Einheitsnormalenvektoren:

$$\mathbf{n_1} = \begin{bmatrix} n_{x_1} \\ n_{y_1} \end{bmatrix} \qquad \mathbf{n_2} = \begin{bmatrix} n_{x_2} \\ n_{y_2} \end{bmatrix} \tag{12.15}$$

Die Ursprungsabstände der Tangenten erhalten wir, indem wir Gleichung (12.1) (oder Gleichung (12.2)) nach dem gesuchten Abstand auflösen und die Normalenvektoren einsetzen:

$$d_{1,2} = r_1 + \mathbf{o_{M_1}} \cdot \mathbf{n_{1,2}} \tag{12.16}$$

Die Tangenten t_1 und t_2 sind damit eindeutig bestimmt. Um nun ihre jeweiligen Berührpunkte auszurechnen, nutzen wir die Tatsache, dass die Normalenvektoren senkrecht auf den Tangenten und damit auch auf den Kreisen stehen und sie daher auch Radiusvektoren der Kreise darstellen. Den Radiusvektor $\mathbf{r_{11}}$ vom Mittelpunkt M_1 des ersten Kreises zum Berührpunkt B_{11} der ersten Tangente erhalten wir daher beispielsweise als Produkt des ersten Einheitsnormalenvektors $\mathbf{n_1}$ mit der Länge des ersten Radius r_1:

$$\mathbf{r_{11}} = r_1 \mathbf{n_1}$$

[2] Die beiden Tangenten t_3 und t_4 berechnen wir dann im Abschnitt 12.2.

Die Ortsvektoren zu den ersten vier Berührpunkten bestimmen wir dann als Vektorsummen, indem wir jeweils vom Ursprung zu den Kreismittelpunkten und weiter von den Mittelpunkten über die Radiusvektoren zu den Berührpunkten gehen:

$$\begin{aligned} \mathbf{o_{B_{11}}} &= \mathbf{o_{M_1}} + r_1\mathbf{n_1} \\ \mathbf{o_{B_{12}}} &= \mathbf{o_{M_2}} + r_2\mathbf{n_1} \\ \mathbf{o_{B_{21}}} &= \mathbf{o_{M_1}} + r_1\mathbf{n_2} \\ \mathbf{o_{B_{22}}} &= \mathbf{o_{M_2}} + r_2\mathbf{n_2} \end{aligned} \tag{12.17}$$

12.2 Tangenten t_3 und t_4

Zur Berechnung der anderen beiden Tangenten t_3 und t_4 verwenden wir praktisch die gleichen Formeln – allerdings mit einem entscheidenden Unterschied gleich zu Beginn: Der Mittelpunkt des ersten Kreises und der Ursprung liegen jeweils auf unterschiedlichen Seiten von t_3 bzw. t_4. In Kapitel 8 haben wir gelernt, dass der mit Gleichung (8.5) berechnete Abstand eines Punktes von einer Gerade dann gerade negativ ist, wenn Punkt und Ursprung auf unterschiedlichen Seiten der Gerade liegen. Wir müssen also in der Gleichung (12.1) entsprechenden Gleichung für t_3 (und t_4) den Abstand des ersten Kreismittelpunktes von der dritten Tangente negativ ansetzen:

$$-r_1 = d_3 - \mathbf{o_{M_1}} \cdot \mathbf{n_3} \tag{12.18}$$

Der Mittelpunkt des zweiten Kreises liegt hingegen auch bei t_3 (und t_4) auf der gleichen Seite wie der Ursprung:

$$r_2 = d_3 - \mathbf{o_{M_2}} \cdot \mathbf{n_3} \tag{12.19}$$

Wenn wir jetzt wieder Gleichung (12.19) von Gleichung (12.18) abziehen, erhalten wir auf der linken Seite natürlich ein negatives Vorzeichen vor r_1:

$$-r_1 - r_2 = (\mathbf{o_{M_2}} - \mathbf{o_{M_1}}) \cdot \mathbf{n_1}$$

so dass wir entsprechend Gleichung (12.5) die folgende Abkürzung einführen:

$$r = -r_1 - r_2 \tag{12.20}$$

Die restliche Herleitung mittels der Gleichungen (12.6) bis (12.15) sieht auch bei t_3 und t_4 genauso aus wie bei t_1 und t_2, so dass wir schließlich die anderen beiden gesuchten Normalenvektoren erhalten:

$$\ldots \quad \mathbf{n_3} = \begin{bmatrix} n_{x_3} \\ n_{y_3} \end{bmatrix} \qquad \mathbf{n_4} = \begin{bmatrix} n_{x_4} \\ n_{y_4} \end{bmatrix} \tag{12.21}$$

Lediglich bei der Gleichung (12.16) entsprechenden Abstandsberechnung müssen wir natürlich auch den negativen Radius verwenden:

$$d_{3,4} = -r_1 + \mathbf{o_{M_1}} \cdot \mathbf{n_{3,4}} \tag{12.22}$$

Auch bei den Berührpunkten müssen wir die Tatsache berücksichtigen, dass die Normalenvektoren auf t_3 und t_4 im ersten Kreis nicht vom Mittelpunkt zum Berührpunkt sondern umgekehrt vom Berührpunkt zum Mittelpunkt zeigen. Wir müssen daher bei den Berührpunkten des ersten Kreises das Vorzeichen der Radiusvektoren umdrehen:

$$\begin{aligned}
\mathbf{o_{B_{31}}} &= \mathbf{o_{M_1}} - r_1\mathbf{n_3} \\
\mathbf{o_{B_{32}}} &= \mathbf{o_{M_2}} + r_2\mathbf{n_3} \\
\mathbf{o_{B_{41}}} &= \mathbf{o_{M_1}} - r_1\mathbf{n_4} \\
\mathbf{o_{B_{42}}} &= \mathbf{o_{M_2}} + r_2\mathbf{n_4}
\end{aligned} \tag{12.23}$$

12.3 Diskriminante

In Gleichung (12.13) haben wir die Diskriminante δ berechnet, die als Radikand bei reellen Lösungen positiv sein muss. Diese Diskriminante wollen wir in diesem Kapitel noch etwas genauer untersuchen.

Wenn wir die in Gleichung (12.12) definierten Abkürzungen in Gleichung (12.13) einsetzen, erhalten wir:

$$\begin{aligned}
\delta &= \beta^2 - \alpha\gamma \\
&= (-ra_x)^2 - \left(a_x^2 + a_y^2\right)\left(r^2 - a_y^2\right)
\end{aligned}$$

Ausmultiplizieren liefert uns:

$$\begin{aligned}
\delta &= r^2a_x^2 - a_x^2r^2 + a_x^2a_y^2 - a_y^2r^2 + a_y^4 \\
&= a_x^2a_y^2 - a_y^2r^2 + a_y^4 \\
&= a_y^2\left(a_x^2 + a_y^2 - r^2\right)
\end{aligned} \tag{12.24}$$

Aus Gleichung (12.24) können wir jetzt das Vorzeichen der Diskriminante leicht bestimmen: Da der Vorfaktor a_y^2 sicher positiv ist, ist die Diskriminante immer dann positiv, wenn die Klammer positiv ist, wenn also

$$\begin{aligned}
a_x^2 + a_y^2 - r^2 &> 0 \\
a_x^2 + a_y^2 &> r^2
\end{aligned} \tag{12.25}$$

Auf der linken Seite der Ungleichung (12.25) finden wir genau das Quadrat des Abstands der beiden Kreismittelpunkte

$$a^2 = |\mathbf{a}|^2 = \mathbf{a}^2 = \mathbf{a} \cdot \mathbf{a} = \begin{bmatrix} a_x \\ a_y \end{bmatrix} \cdot \begin{bmatrix} a_x \\ a_y \end{bmatrix} = a_x^2 + a_y^2$$

und auf der rechten Seite im Falle von t_1 und t_2 gemäß Gleichung (12.5) das Quadrat der Radiendifferenz

$$t_1,\, t_2: \quad r^2 = (r_1 - r_2)^2$$

und im Falle von t_3 und t_4 gemäß Gleichung (12.20) das Quadrat der Radiensumme

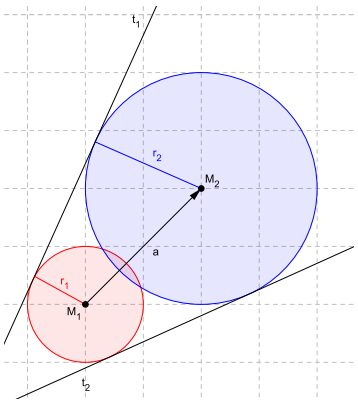
$$\begin{aligned} t_3,\, t_4: \quad r^2 &= (-r_1 - r_2)^2 \\ &= (r_1 + r_2)^2 \end{aligned}$$

Die Bedingungen für die Existenz der Tangenten t_1 und t_2 lautet also:

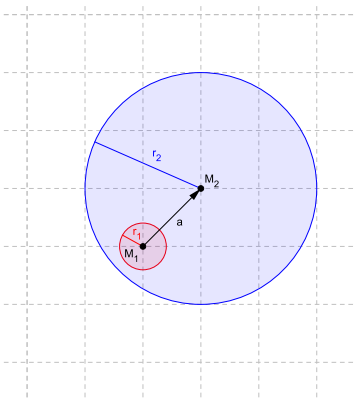
$$\begin{aligned} t_1,\, t_2: \quad a^2 &> (r_1 - r_2)^2 \\ a &> |r_1 - r_2| \end{aligned}$$

Entsprechend existieren t_3 und t_4 nur dann, wenn:

$$\begin{aligned} t_3,\, t_4: \quad a^2 &> (r_1 + r_2)^2 \\ a &> r_1 + r_2 \end{aligned}$$



(a) Nur t_1 und t_2: $a < r_1 + r_2$



(b) Gar keine Tangenten: $a < |r_1 - r_2|$

Abbildung 12.2: Sonderfälle, wenn der Kreisabstand zu gering ist

Dies bedeutet, dass die Tangenten t_3 und t_4 verschwinden, wenn sich – wie in Abbildung 12.2a dargestellt – die Kreise überlappen, wenn also der Kreismittelpunktsabstand a kleiner als die Summe der Radien wird. Wenn sogar der eine Kreis komplett innerhalb des anderen Kreises liegt (Abbildung 12.2b), gibt es überhaupt keine (reellen) Tangenten. In diesem Fall ist sogar die (positive) Differenz der beiden Radien größer als der Abstand der Mittelpunkte.

12.4 x und y vertauschen

Wie schon in der Fußnote zu Gleichung (12.9) erwähnt, gibt es ein Problem, wenn a_y verschwindet, da dann in Gleichung (12.9) durch null geteilt wird. Dies ist immer dann der Fall, wenn beide Kreismittelpunkte „auf gleicher Höhe liegen", wenn sie also eine identische y-Komponente besitzen. Da dieser in Abbildung 12.3 dargestellte Fall durchaus in der Praxis auftreten kann, müssen wir das Problem lösen.

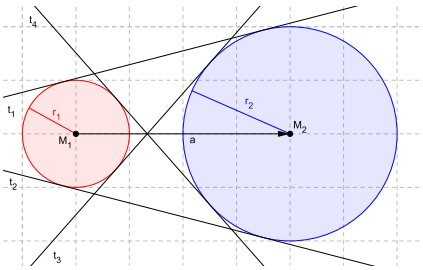


Abbildung 12.3: Verschwindendes a_y

Wir könnten jetzt versuchen, genau diesen Fall des verschwindenden a_y abzufangen; das Abfragen von Fließkommazahlen auf exakt null ist aber immer problematisch. Die numerischen Probleme mit sehr kleinen bzw. sehr großen Zahlen treten ja auch schon dann auf, wenn a_y nur hinreichend klein wird. Wir gehen daher einen anderen Weg, der uns einen großen Sicherheitsabstand vom kritischen Fall verspricht:

- Solange $|a_y| \geqq |a_x|$ ist, verwenden wir die hergeleiteten Berechnungsvorschriften (12.20) bis (12.23).
- Wenn aber $|a_y| < |a_x|$ ist, vertauschen wir in allen Gleichungen x und y, so dass im Nenner von Gleichung (12.9) nicht mehr durch das kleinere a_y sondern durch das größere a_x geteilt wird. Wir haben dadurch praktisch den in Abbildung 12.3 dargestellten kritischen Fall durch Spiegelung an der Winkelhalbierenden ($y = x$) auf den in Abbildung 12.4 dargestellten unkritischen Fall zurückgeführt.

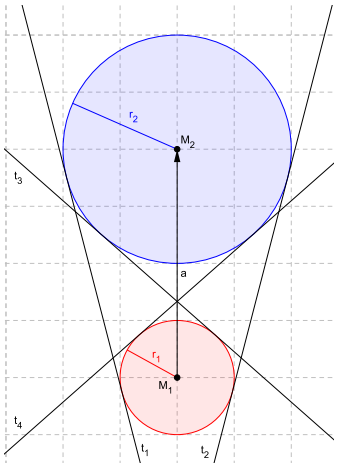


Abbildung 12.4: Verschwindendes a_x

Natürlich müssen wir später bei allen Vektoren ($\mathbf{n_1} \ldots \mathbf{n_4}$) und Punkten ($B_{11} \ldots B_{42}$) die x- und y-Komponenten wieder zurück tauschen. Unter Matlab geschieht dies komfortabel mit dem Befehl `fliplr`.

12.5 Beispiel

12.5.1 Ohne Vertauschung von x und y

Wir betrachten den in Abbildung 12.1 dargestellten Fall. Die beiden Kreise haben dort folgende Daten:

$$\mathbf{o_{M_1}} = \begin{bmatrix} -4 \\ -1 \end{bmatrix}$$
$$\mathbf{o_{M_2}} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$
$$r_1 = 1$$
$$r_2 = 2$$

Für die Tangenten t_1 und t_2 berechnen wir gemäß Gleichung (12.5) die Radiendifferenz

$$r \quad = \quad r_1 - r_2 = 1 - 2 = -1$$

und nach Gleichung (12.6) den Mittelpunkteabstandsvektor:

$$\mathbf{a} = \mathbf{o_{M_2}} - \mathbf{o_{M_1}} = \begin{bmatrix}3\\1\end{bmatrix} - \begin{bmatrix}-4\\-1\end{bmatrix} = \begin{bmatrix}7\\2\end{bmatrix}$$

Als Abkürzungen erhalten wir mittels Gleichung (12.12):

$$\begin{aligned}\alpha &= a_x^2 + a_y^2 = 7^2 + 2^2 = 53\\ \beta &= -ra_x = -(-1)\cdot 7 = 7\\ \gamma &= r^2 - a_y^2 = (-1)^2 - 2^2 = -3\end{aligned}$$

Die Diskriminante (Gleichung (12.13)) berechnen wir dann zu

$$\delta = \beta^2 - \alpha\gamma = 7^2 - 53\cdot(-3) = 208$$

und stellen fest, dass diese positiv ist und die Tangenten t_1 und t_2 damit existieren. Die x-Werte ihrer Normalenvektoren erhalten wir aus Gleichung (12.14)

$$\begin{aligned}n_{x_{1,2}} &= \frac{-\beta \pm \sqrt{\delta}}{\alpha} = \frac{-7 \pm \sqrt{208}}{53}\\ n_{x_1} &= -0.4042\\ n_{x_2} &= 0.1400\end{aligned}$$

und die zugehörigen y-Werte aus Gleichung (12.9):

$$\begin{aligned}n_{y_1} &= \frac{r - a_x n_{x_1}}{a_y} = \frac{-1 - 7\cdot(-0.4042)}{2} = 0.9147\\ n_{y_2} &= \frac{r - a_x n_{x_2}}{a_y} = \frac{-1 - 7\cdot 0.1400}{2} = -0.9901\end{aligned}$$

Die Normalenvektoren der ersten beiden Tangenten lauten dann also:

$$\begin{aligned}\mathbf{n_1} &= \begin{bmatrix}-0.4042\\0.9147\end{bmatrix}\\ \mathbf{n_2} &= \begin{bmatrix}0.1400\\-0.9901\end{bmatrix}\end{aligned} \tag{12.26}$$

Nach Gleichung (12.17) folgen daraus die gesuchten Berührpunkte:

$$\begin{aligned}\mathbf{o_{B_{11}}} &= \mathbf{o_{M_1}} + r_1\mathbf{n_1} = \begin{bmatrix}-4\\-1\end{bmatrix} + 1\cdot\begin{bmatrix}-0.4042\\0.9147\end{bmatrix} = \begin{bmatrix}-4.4042\\-0.0853\end{bmatrix}\\ \mathbf{o_{B_{12}}} &= \mathbf{o_{M_2}} + r_2\mathbf{n_1} = \begin{bmatrix}3\\1\end{bmatrix} + 2\cdot\begin{bmatrix}-0.4042\\0.9147\end{bmatrix} = \begin{bmatrix}2.1916\\2.8293\end{bmatrix}\\ \mathbf{o_{B_{21}}} &= \mathbf{o_{M_1}} + r_1\mathbf{n_2} = \begin{bmatrix}-4\\-1\end{bmatrix} + 1\cdot\begin{bmatrix}0.1400\\-0.9901\end{bmatrix} = \begin{bmatrix}-3.8600\\-1.9901\end{bmatrix}\\ \mathbf{o_{B_{22}}} &= \mathbf{o_{M_2}} + r_2\mathbf{n_2} = \begin{bmatrix}3\\1\end{bmatrix} + 2\cdot\begin{bmatrix}0.1400\\-0.9901\end{bmatrix} = \begin{bmatrix}3.2801\\-0.9803\end{bmatrix}\end{aligned}$$

Für die beiden anderen Tangenten t_3 und t_4 müssen wir laut Gleichung (12.20) die negative Summe der Kreisradien verwenden:

$$r = -r_1 - r_2 = -1 - 2 = -3$$

Die weiteren Rechenschritte sind die gleichen wie bei den Tangenten t_1 und t_2:

$$\begin{aligned} \alpha &= a_x^2 + a_y^2 = 7^2 + 2^2 = 53 \\ \beta &= -ra_x = -(-3) \cdot 7 = 21 \\ \gamma &= r^2 - a_y^2 = (-3)^2 - 2^2 = 5 \end{aligned}$$

$$\delta = \beta^2 - \alpha\gamma = 21^2 - 53 \cdot 5 = 176$$

$$\begin{aligned} n_{x_{3,4}} &= \frac{-\beta \pm \sqrt{\delta}}{\alpha} = \frac{-21 \pm \sqrt{176}}{53} \\ n_{x_3} &= -0.6465 \\ n_{x_4} &= -0.1459 \end{aligned}$$

$$\begin{aligned} n_{y_3} &= \frac{r - a_x n_{x_3}}{a_y} = \frac{-3 - 7 \cdot (-0.6465)}{2} = 0.7629 \\ n_{y_4} &= \frac{r - a_x n_{x_4}}{a_y} = \frac{-3 - 7 \cdot (-0.1459)}{2} = -0.9893 \end{aligned}$$

$$\begin{aligned} \mathbf{n_3} &= \begin{bmatrix} -0.6465 \\ 0.7629 \end{bmatrix} \\ \mathbf{n_4} &= \begin{bmatrix} -0.1459 \\ -0.9893 \end{bmatrix} \end{aligned}$$

Lediglich die Berührpunkte der Tangenten t_1 und t_2 erfordern gemäß Gleichung (12.23) negative Vorzeichen der Normalenvektoren beim ersten Kreis:

$$\begin{aligned} \mathbf{o_{B31}} &= \mathbf{o_{M_1}} - r_1\mathbf{n_3} = \begin{bmatrix} -4 \\ -1 \end{bmatrix} - 1 \cdot \begin{bmatrix} -0.6465 \\ 0.7629 \end{bmatrix} = \begin{bmatrix} -3.3535 \\ -1.7629 \end{bmatrix} \\ \mathbf{o_{B32}} &= \mathbf{o_{M_2}} + r_2\mathbf{n_3} = \begin{bmatrix} 3 \\ 1 \end{bmatrix} + 2 \cdot \begin{bmatrix} -0.6465 \\ 0.7629 \end{bmatrix} = \begin{bmatrix} 1.7069 \\ 2.5258 \end{bmatrix} \\ \mathbf{o_{B41}} &= \mathbf{o_{M_1}} - r_1\mathbf{n_4} = \begin{bmatrix} -4 \\ -1 \end{bmatrix} - 1 \cdot \begin{bmatrix} -0.1459 \\ -0.9893 \end{bmatrix} = \begin{bmatrix} -3.8541 \\ -0.0107 \end{bmatrix} \\ \mathbf{o_{B42}} &= \mathbf{o_{M_2}} + r_2\mathbf{n_4} = \begin{bmatrix} 3 \\ 1 \end{bmatrix} + 2 \cdot \begin{bmatrix} -0.1459 \\ -0.9893 \end{bmatrix} = \begin{bmatrix} 2.7082 \\ -0.9786 \end{bmatrix} \end{aligned}$$

12.5.2 Mit Vertauschung von x und y

Nach Abschnitt 12.4 sollten wir im in Abbildung 12.1 dargestellten Fall x und y vertauschen, da

$$\mathbf{a} = \begin{bmatrix} 7 \\ 2 \end{bmatrix}$$

und damit

$$a_y = 2 < a_x = 7$$

ist. Wir werden daher in diesem Abschnitte alle Vektoren, deren x- und y-Komponenten vertauscht wurden, mit einer Tilde ($\sim$) kennzeichnen und zeigen, dass nach der Rücktauschung natürlich das gleiche Endergebnis für die Normalenvektoren wie im Fall ohne Hin- und Rücktauschung heraus kommt.

Für t_1 und t_2 ergeben sich dann folgende Zahlenwerte:

$$\tilde{\mathbf{a}} = \begin{bmatrix} 2 \\ 7 \end{bmatrix}$$

$$\begin{aligned} \tilde{\alpha} &= \tilde{a}_x^2 + \tilde{a}_y^2 = 2^2 + 7^2 = 53 \\ \tilde{\beta} &= -r\tilde{a}_x = -(-1) \cdot 2 = 2 \\ \tilde{\gamma} &= r^2 - \tilde{a}_y^2 = (-1)^2 - 7^2 = -48 \end{aligned}$$

$$\tilde{\delta} = \tilde{\beta}^2 - \tilde{\alpha}\tilde{\gamma} = 2^2 - 53 \cdot (-48) = 2548$$

$$\begin{aligned} \tilde{n}_{x_{1,2}} &= \frac{-\tilde{\beta} \pm \sqrt{\tilde{\delta}}}{\tilde{\alpha}} = \frac{-2 \pm \sqrt{2548}}{53} \\ \tilde{n}_{x_1} &= 0.9147 \\ \tilde{n}_{x_2} &= -0.9901 \end{aligned}$$

$$\begin{aligned} \tilde{n}_{y_1} &= \frac{r - \tilde{a}_x \tilde{n}_{x_1}}{\tilde{a}_y} = \frac{-1 - 2 \cdot 0.9147}{7} = -0.4042 \\ \tilde{n}_{y_2} &= \frac{r - \tilde{a}_x \tilde{n}_{x_2}}{\tilde{a}_y} = \frac{-1 - 2 \cdot (-0.9901)}{7} = 0.1400 \end{aligned}$$

$$\begin{aligned} \tilde{\mathbf{n}}_\mathbf{1} &= \begin{bmatrix} 0.9147 \\ -0.4042 \end{bmatrix} \\ \tilde{\mathbf{n}}_\mathbf{2} &= \begin{bmatrix} -0.9901 \\ 0.1400 \end{bmatrix} \end{aligned}$$

Das Rücktauschen der Normalenvektorkomponenten liefert uns dann natürlich die gleichen Ergebnisse wie in Gleichung (12.26):

$$\begin{aligned} \mathbf{n_1} &= \begin{bmatrix} -0.4042 \\ 0.9147 \end{bmatrix} \\ \mathbf{n_2} &= \begin{bmatrix} 0.1400 \\ -0.9901 \end{bmatrix} \end{aligned}$$

Das das Verfahren auch bei den Tangenten t_3 und t_4 funktioniert, glauben wir jetzt einfach mal ...

Literaturverzeichnis

[1] Hochschule Bremen. (2018) Spaceballs im Modulpool der Hochschule Bremen. [Online]. Available: http://www.hs-bremen.de/internet/de/weiterbildung/koowb/Modulpool/fachuebergreifendeWahlmodule/

[2] The LyX Team. (2018) LyX - The Document Processor. [Online]. Available: http://www.lyx.org/

[3] Wix.com. (2018) Erstellen Sie eine eigene kostenlose Homepage. [Online]. Available: https://de.wix.com/

[4] The Mathworks. (2018) Introducing mex files. [Online]. Available: https://de.mathworks.com/help/matlab/matlab_external/introducing-mex-files.html

[5] ——. (2018) persistent. [Online]. Available: https://de.mathworks.com/help/matlab/ref/persistent.html

[6] ——. (2018) pcode. [Online]. Available: https://de.mathworks.com/help/matlab/ref/pcode.html

[7] ——. (2018) function_handle (@). [Online]. Available: https://de.mathworks.com/help/matlab/function-handles.html

[8] Wikipedia. (2018) Explizites Euler-Verfahren. [Online]. Available: https://de.wikipedia.org/wiki/Explizites_Euler-Verfahren

[9] 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/

[10] Wikipedia. (2018) HSV-Farbraum. [Online]. Available: https://de.wikipedia.org/wiki/HSV-Farbraum

[11] J. J. Buchholz. (2014) Kollision Kreis Kreis. [Online]. Available: https://www.geogebra.org/m/hJWKwjSe

[12] Wikipedia. (2018) Quadratische Gleichung. [Online]. Available: https://de.wikipedia.org/w/index.php?title=Quadratische_Gleichung

[13] C. Moler. (2004) The origins of matlab. [Online]. Available: https://de.mathworks.com/company/newsletters/articles/the-origins-of-matlab.html