Dokumentation des Teams *JETI SpaceWars*

Spaceball-Projekt im Modul Informatik von Jens Hofacker und Timo Radcke
26.01.2018

Inhaltsverzeichnis

Teil I

Dokumentation

Kapitel 1

Das Team

1.1 Namensfindung

Wir, Jens Hofacker und Timo Radcke, sind das Team JETI SpaceWars. Wie man sich bestimmt denken kann, kamen wir über die ersten Buchstaben unserer Vornamen zu unserem Teamnamen. So ergab sich mit Star Wars auch sehr schnell ein interessanter und zum Spiel passender Themenbereich, auf dem wir unser Team aufbauen wollten. Diesen Themenhintergrund haben wir dann sowohl in unseren Soundtrack als auch in unsere Website mit einfließen lassen.

1.2 Teammitglieder

Jens Hofacker

Abbildung 1.1: Jens Hofacker

Mein Name ist Jens Hofacker und ich bin 20 Jahre alt. Meine Hauptaufgaben waren Minen und Verteidigung, außerdem war ich zuständig für die Webseite und das Logo.
Motto:
„Wir kämpfen bis zum Schluss. Haben wir mal weniger getankt... dann wird der Verteidigung gedankt!“

Timo Radcke

Abbildung 1.2: Timo Radcke

Mein Name ist Timo Radcke. Ich bin 19 Jahre alt. In unserm Spaceball-Projekt war ich hauptsächlich verantwortlich für die Aufgaben Tanken und Angriff.
Motto:
„Wenn das Tanken und der Angriff gut laufen, braucht man den Rest nicht mehr."

1.3 Logo

Als Logo wollten wir ein sehr auffallendes Logo entwerfen und gestalten. Dazu kam uns anfangs direkt die Idee, eine Explosion im Logo mitaufzunehmen. Dies erschien uns sehr herausstechend. Deshalb haben wir zuerst mit Hilfe von Microsoft PowerPoint eine grobe Skizze unseres Logos angefertigt.

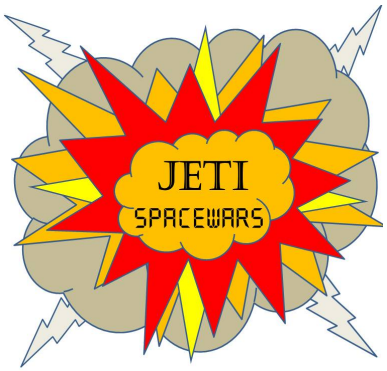

Abbildung 1.3: Altes Logo

Diese Idee haben wir dann noch mit einem professionellen Bildbearbeitungsprogramm[1] neu umgesetzt und schöner dargestellt.
Dazu haben wir zunächst verschiedene Schriftarten ausprobiert und auch verschiedene Farben verwendet. Das Logo soll den Teamnamen JETI SpaceWars verdeutlichen und enthält diesen im Zentrum. Dargestellt ist ein Feuerball mit explosionsartiger Form und die Farben wurden auch entsprechend angepasst. Am Ende kam dann folgendes Logo als Finalfassung heraus.

Abbildung 1.4: Finales Logo

1.4 Soundtrack

Bei unserem Soundtrack haben wir die Idee gehabt, ein rotierendes Lichtschwert nachzustellen. Dazu haben wir das Programm FI Studio 12[2] genutzt.

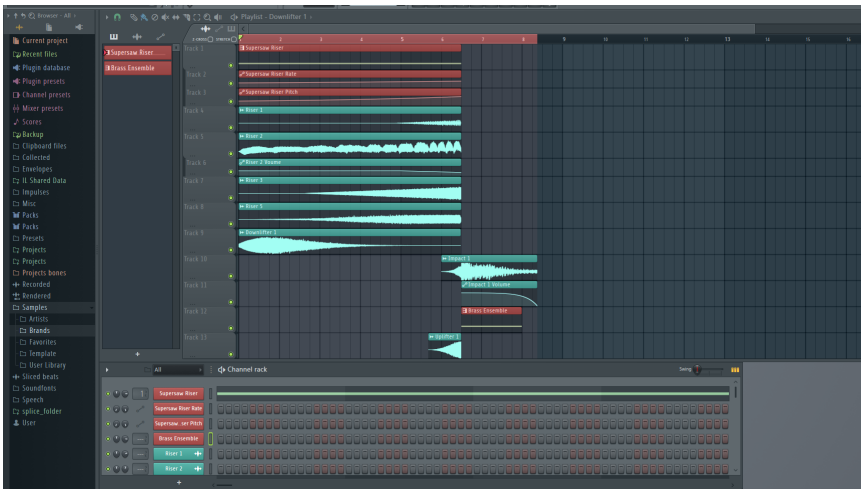

Abbildung 1.5: Soundtrack in FI Studio

Zu Beginn haben wir mit Hilfe des Synthesizer eine Effektmelodie erstellt, die sich ungefähr so anhört, wie ein rotierendes Lichtschwert. Danach haben wir die Wiederholungsgeschwindigkeit und Tonhöhe des Effekts zum Ende weiter erhöht, sodass sich eine gewisse Spannung aufbaut. Nachdem wir die Grundeffektmelodie fertig gestellt hatten, haben wir diese mit Effekten aus Sample-Packs[3] weiter ausgeschmückt und so ausgewählt, dass sich der Soundtrack gut anhört und auch gut endet.

[1] Adobe Illustrator
[2] siehe https://www.image-line.com/flstudio/
[3] Sample-Packs sind Sammlungen von einzelnen kleineren Effekten und Sounds (Samples).

1.5 Website

Für unsere Webseite haben wir uns überlegt, dass sie zu dem Spiel Spaceballs passen soll und, noch viel wichtiger, natürlich auch zu unserem Teamnamen JETI SpaceWars, weshalb wir im Hintergrund[4] dunkle Farben verwendet haben. Die weitere Hauptfarbe der Webseite ist orange, da diese Farbe gut mit unserem Logo einhergeht. Auf der Webseite wollen wir Einblicke in das Spiel und auch Tipps für potentielle Nachfolger geben.
Hier der Link zu unserer Webseite: https://jeti-spacewars.jimdo.com/spaceballs/ . Auf der Home-Seite wird zunächst einmal erklärt, was das Spiel überhaupt ist, wie es grundsätzlich abläuft und welche Regeln gelten. Auf den einzelnen Unterseiten wird dann erklärt, was es mit den einzelnen Bereichen Angriff, Tanken, Minenabwehr und Verteidigung auf sich hat. Dort gibt es auch einen Reiter mit der Teamvorstellung und der jeweiligen Aufgabenverteilung.

Abbildung 1.6: Logo für die Website

Außerdem kann man auf der Website das Turnierergebnis und eine Verlinkung zum Turniervideo finden. Unter Downloads findet man zudem unsere Beschleunigungsdatei sowie diese Dokumentation.

Abbildung 1.7: Website-Startseite

[4] siehe https://jeti-spacewars.jimdo.com/spaceballs/

Kapitel 2

Einleitung

2.1 Spiel

2.1.1 Spielaufbau

Ein Spaceball-Spielfeld beinhaltet zwölf Minen und neun Tankstellen, dessen Anordnung bei jedem Spiel per Zufall neu bestimmt wird. Dabei sind sie immer symmetrisch im Spielfeld angeordnet.

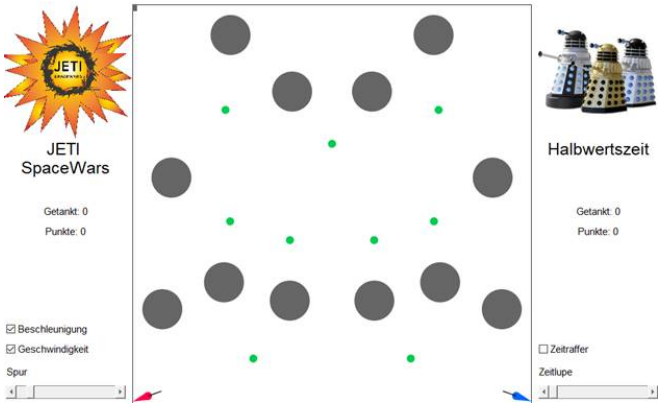

Abbildung 2.1: Zufälliges Spielfeld

2.1.2 Regeln

Ziel des Spiels ist es, mehr Tankstellen einzusammeln als der Gegner. Die Spieldauer ist auf eine Minute begrenzt, sobald aber ein Spaceball eine Bande oder Mine trifft, ist das Spiel sofort beendet und das gegnerische Team erhält einen Punkt. Sobald alle Tankstellen verschwunden sind, verschwindet nach einem bestimmten zeitlichen Abstand auch eine Mine nach der anderen.

Nun gibt es verschiedene Szenarien zur Verteilung der Punkte:

- Wenn A mehr tankt als B und B fängt, gewinnt A und erhält dafür einen Punkt.
- Wenn A mehr tankt als B und B in den 60 Sekunden nicht fängt, ist das Spiel unentschieden und niemand bekommt einen Punkt.
- Wenn A in eine Mine oder Bande fährt, gewinnt automatisch B und erhält dafür einen Punkt.

Umgekehrt gilt dasselbe.

2.1.3 Spielfeld und Spielelemente

Das Spielfeld kann man sich wie ein normales kartesisches Koordinatensystem im ersten Quadranten vorstellen mit den maximalen Abmessungen in x und y-Richtung von 1.0.
Positionen im Spielfeld werden somit durch zwei Koordinaten beschrieben, genauso wie daraus resultierende Vektoren. Spaceballs, Tanken oder Minen haben dabei bestimmte Radien, welche um ihre Positionen aufgespannt sind.
Somit sind Größe und Form der Spielelemente genau beschrieben.

2.2 Programmcode

2.2.1 Aufbau der Datei

Die Künstliche Intelligenz ist eine Funktion mit dem Dateinamen *beschleunigung* und sie berechnet Schritt für Schritt die Beschleunigung (=*bes*) des Spaceballs. Zu Beginn liest das Spiel die Daten der Teams aus und es werden verschiedene Eingabeparameter benötigt, um der richtigen Farbe die richtige *bes*-Datei zuzuordnen. Danach wird von jeder der zwei Beschleunigungsdateien für jeden Simulationsschritt ein Beschleunigungsvektor an das Spiel übergeben. Dabei gelten *ich.* für den eigenen Spaceball und *gegner.* für den Gegner. Diese Zuordnung wird im Kapitel 3.1.1 erklärt und gezeigt.

2.2.2 Steuern des Spaceball

Im Spiel lässt sich der Spaceball nur durch einen Beschleunigungsvektor mit der festen Länge 0.1 steuern. Das bedeutet, dass Beschleunigen, Abbremsen und Lenken durch einen einzigen Vektor umgesetzt werden muss.
Die Aufgabe der Datei ist somit nur, für die einzelnen Spielsituationsschritte einen passenden Beschleunigungsvektor auszugeben. Da der Beschleunigungswert konstant ist, kann man so gut zeitlich abhängige Prozesse berechnen und abschätzen.
Der von der Datei ausgegebene Vektor, zeigt die Richtung an, in die der Spaceball beschleunigen soll.

2.2.3 Grundlegende Berechnungen

Zu grundlegenden Berechnungen gehören:

- Geschwindigkeitsbetrag: Der Betrag von Geschwindigkeiten kann mit Hilfe der Hypotenusenformel für rechtwinklige Dreiecke $|v| = \sqrt{v_x^2 + v_y^2}$ bestimmt werden, da der Geschwindigkeitsvektor aus zwei Teilgeschwindigkeiten in zueinander orthogonalen Richtungen besteht. In MATLAB kann dies durch den norm-Befehl geschehen.

```
1 ich_ges = norm(ich.ges);
```

- Bremsweg: Der Bremsweg berechnet sich gemäß der physikalischen Bremswegformel $s = \frac{v^2}{|2a|}$ auch während des Spiels. Er wird häufig benutzt, um vor Hindernissen rechtzeitig zu bremsen oder Risikobewertungen von Hindernissen zu machen.

```
1 bremsweg=(ich_ges)^2/(2*spiel.bes);
```

- Abstandsmessung: Man kann den Abstand zweier Positionen errechnen, indem man einen Vektor zwischen beiden Positionen erstellt und dessen Betrag $|x| = \sqrt{x_x^2 + v_y^2}$ errechnet. Auch hier kann man diese Berechnung mit dem norm-Befehl durchführen.

```
1 betrag_x=norm(x);
```

2.2.4 Konstanten

Konstanten bieten sich an, um mehrere Werte gleichzeitig zu ändern (zum Beispiel: Sicherheitsfaktoren). Des Weiteren kann man Konstanten dazu nutzen, Funktionen, die in der Datei folgen, zu aktivieren oder deaktivieren. Außerdem kann man ganz normale Zuordnungen treffen, welche man im Verlaufe der Datei während eines Simulationsschrittes weiterverwenden kann.

2.2.5 Persistente Variablen

Persistente Variablen treten immer wieder auf. Sie dienen dazu, dass der einer Variablen zugewiesene Wert über verschiedene Abfragen von *beschleunigung* erhalten bleiben soll. Es ist notwendig, am Anfang des Codes die persistenten Variablen zu deklarieren und es wird zunächst ein Anfangswert zugewiesen (meist *true*, *false* oder 0 und 1). Dieser Wert kann dann im Verlaufe der Funktion beliebig verändert werden. Wie schon gesagt, bleibt dieser Wert über folgende Simulationsschritte konstant, solange ihm kein neuer Wert zugewiesen wird. So kann man über mehrere Simulationsschritte hinaus mit der eigenen Beschleunigungsdatei kommunizieren.

Kapitel 3

Steuerung des Spaceballs

3.1 Hilfsfunktionen

3.1.1 Vorberechnungen und Konstanten

Bevor letztendlich damit begonnen werden kann den Code zu schreiben, muss in der Funktion erst einmal klar festgelegt werden, wie in dem Spiel der rote und blaue Spaceball zugeordnet werden. Dazu wird entweder das rote Spiel dem eigenen Spaceball und das blaue dem Gegner zugeordnet oder genau umgekehrt.

```
1 if strcmp( farbe, 'rot')
2     ich=spiel.rot;
3     gegner=spiel.blau;
4 else
5     ich=spiel.blau;
6     gegner=spiel.rot;
7 end
```

Des Weiteren haben wir danach zunächst einmal unsere Beschleunigung gleich null gesetzt, sodass es, wenn in den nachfolgenden Rechnungen und Funktionen der Beschleunigung keine genauen Werte zugewiesen werden sollten, trotzdem einen zugewiesenen Wert für die Beschleunigung gibt und kein Fehler auftaucht. Dasselbe geschieht auch für die verschiedenen Teilbeschleunigungen der einzelnen Modus-Funktionen.

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

Nach dieser Basiszuweisung werden bei uns alle Konstanten, welche wir im Verlaufe der Funktion gebrauchen könnten, mit dem jeweiligen Wert belegt.
Dann folgen bei unserem Code die Zuweisungen der persistenten Variablen, die wir während unserer Funktionen brauchen.
Nach den persistenten Variablen legen wir bestimmte Eckpunkte für alle vier Ecken fest. Diese Eckpunkte beinhalten auch bereits einen Sicherheitsabstand zu den Banden. Wir verwenden die Punkte in unseren Verteidigung.

```
1 eol=[0.006,0.994];
2 eor=[0.994,0.994];
3 eur=[0.994,0.006];
4 eul=[0.006,0.006];
```

Eine weitere Berechnung, welche wir dann durchführen, ist die Abstandsberechnung zu allen vier Banden.

```
1 abs_rb=1-ich.pos(1);
2 abs_ob=1-ich.pos(2);
3 abs_lb=ich.pos(1);
4 abs_ub=ich.pos(2);
```

Dann führen wir die Betragsrechnung für unseren Geschwindigkeitsvektor durch.
Anschließend rechnen wir noch den Abstand zum Gegner, Bremswege in alle möglichen Richtungen und mögliche Bandenkollisionen aus.

3.1.2 Kollisionsanalyse

Zur Kollisionsanalyse wurde mit dem mathematischen Werkzeug aus der Dokumentation[1] von Prof. J.J. Buchholz gearbeitet. Im Abschnitt 10 „Kollision zweier Kreise“ findet man die Erklärung, wie man die Berührpunkte beim Schnitt von zwei Kreisen bestimmt. Daraus haben wir die Funktion *szk* (Schnitt zweier Kreise) geschrieben, die sehr wichtig für alle Bereiche unserer Datei ist. In der Abbildung ist zu sehen, dass die Funktion *kollision* verschiedene Parameter benötigt (z.B. *ich.pos*, *ich.ges* und Kreisradien). Dann werden uns mit *lambda* die Berührzeitpunkte ausgegeben, die wir dann anschließend weiterverwenden können.

```
1  function kollision = szk(m1, ges1, r1, m2, ges2, r2)
2          radien = r1 + r2; %Radien addieren
3          e=m2 - m1;          %Entfernung zu Minen
4          f=ges2-ges1;       %Vektor ich.pos in Richtung der Mine
5          alpha=dot(f, f);
6          beta=dot(e, f);
7          gamma=dot(e,e)-radien^2;
8          delta=beta^2 - alpha*gamma;
9          if delta>0
10           lambda=[(-beta+sqrt(delta))/alpha;(-beta-sqrt(delta))/alpha];
11                 if lambda(1)<0
12                         lambda(1)=inf;
13                 end
14                 if lambda(2)<0
15                         lambda(2)=inf;
16                 end
17         else
18                 lambda=[inf,inf];
19         end
20         kollision = sortrows(lambda);
21 end
```

Genauer gesagt berechnet die Funktion immer zwei Berührzeitpunkte. Zum einen den ersten Kontakt der Objekte, dann folgt eine Überschneidungsphase, und zum anderen den letzten Berührpunkt. Deshalb muss man die Ergebnisse dieser Funktion nach der Ausgabe zunächst noch auf Brauchbarkeit überprüfen und nur die Daten herausfiltern, die man in der Situation auch wirklich braucht und verwerten kann. Eine erste Vorselektion findet bei uns schon innerhalb der *kollision*-Funktion statt. Eine weitere Selektion führen wir kurz vor Verwenden der Daten in den verschiedenen Modi und Teilfunktionen durch.

[1] http://buchholz.hs-bremen.de/info/info.htm

3.1.3 Vektorprojektion

Die Vektorprojektion-Funktionen sind sehr hilfreich in mehreren Bereichen. Man kann sie als eine Art Winkelberechnung benutzen oder als Kurskorrekturvektor.
Sie basiert ebenfalls auf dem mathematischen Werkzeug, welches durch Prof. J. J. Buchholz zu Verfügung gestellt wurde.

Die Vektorprojektion *c* berechnet den Projektionsvektor eines Ausgangsvektors, welcher rechtwinklig auf einen anderen Vektor zeigt. Mit diesem errechneten Vektor kann man zum Beispiel den eigenen Geschwindigkeitsvektor des Spaceballs in eine bestimmte Richtung ändern. So kann man der Geschwindigkeit entsprechend eine rasche Richtungsänderung durchführen, ohne merklich viel an Geschwindigkeit zu verlieren.
Wir haben die Vektorprojektion *c* auch beim Minen-Ausweichmanöver verwendet. Genaueres dazu steht in dem Unterkapitel *tangver*-Funktion[2].

```
1 %% Vektorprojektion c
2 function vecpro=vp(z,pos,posges)
3         zw=z-pos;
4         ba=(dot(zw,posges)/(dot(zw,zw)))*zw;
5         vecpro=ba-posges;
6 end
```

Die Vektorprojektion *ba* berechnet den projizierten Vektor des Ausgangsvektors. Die Projektion des Ausgangsvektors ist im Betrag immer kleiner als der Ausgangsvektor. Man kann mit dieser Funktion winkelähnliche Aussagen ausrechnen, um festzustellen, wie groß notwendige Richtungsänderungen sein müssen. Ein weiteres Anwendungsgebiet dieser Funktion ist die Bestimmung des Anteils einer Geschwindigkeit in eine bestimmte andere Richtung. Wir haben die Anteilsgeschwindigkeit dazu verwendet, mit Hilfe der Kollisionsanalyse theoretische Berührzeitpunkte unseres Spaceballs mit andern Objekten zu errechnen, obwohl unsere Geschwindigkeit gar nicht in Richtung des Objektes gerichtet war.

```
1 %% Vektorprojektion ba
2 function ba=vpba(z,pos,posges)
3         zw=z-pos;
4         ba=(dot(zw,posges)/(dot(zw,zw)))*zw;
5 end
```

3.1.4 Orthogonaler Vektor

Der Orthogonaler Vektor ist eine einfache Vertauschung und Veränderung der Vektorbestandteile, um einen Vektor zu erzeugen, welcher senkrecht zum Ausgangsvektor liegt. Dazu wird der erste, obere Wert des Ausgangsvektor mit -1 multipliziert und an die zweite, untere Stelle des Ausgabevektors gestellt, während die zweite, untere Stelle des Ausgangsvektors unverändert an die erste, obere Stelle des Ausgabevektors wandert.

```
1 function ortho=og(a)
2         ortho=[a(2),-a(1)];
3 end
```

[2] siehe Kapitel 3.1.7

3.1.5 sumabs-Funktion

Die *sumabs*-Funktion haben wir uns einfallen lassen, um festzustellen, wo es sich während des Tankprozesses eher lohnt hinzufahren, da dort mehr Tanken näher beieinander liegen.
Dazu haben wir uns gedacht, wir bilden einen theoretischen Tankstellen-Schwerpunkt im Spielfeld, indem wir die Koordinaten aller noch existierenden Tanken addieren und durch deren Anzahl teilen. Nachdem wir so unseren Tankenschwerpunkt berechnet haben, berechnen wir dann für einzelne Tanken den Abstand zu diesem Schwerpunkt. Je kleiner dieser Abstand ist, desto wahrscheinlicher ist es, dass in der Nähe dieser Tanke auch noch weitere Tanken liegen. Somit würden wir, wenn wir diese Tankstelle ansteuern, in Richtung einer höheren Tankendichte steuern. Dieses Kriterium kann man als Kriterium in die Tankwegbestimmung mit einfließen lassen.

```
function summeabs=sumabs(xb)
        if spiel.n_tanke>1
            tankdenmatrix=zeros(spiel.n_tanke,1);
          for x=1:spiel.n_tanke
            tankdenmatrix(x,1)=x;
            tankdenmatrix(x,2)=spiel.tanke(x).pos(1);
            tankdenmatrix(x,3)=spiel.tanke(x).pos(2);
          end
            abssum=sum(tankdenmatrix);
            abssumx=(abssum(1,2))/spiel.n_tanke;
            abssumy=(abssum(1,3))/spiel.n_tanke;
        summeabs=norm(spiel.tanke(xb).pos-[abssumx,abssumy]);
        else
            summeabs=100;
        end
end
```

3.1.6 wegmin-Funktion

Die *wegmin*-Funktion ist aus dem Grund entstanden, dass wir eine Analyse durchführen wollten, die überprüft, ob auf dem Weg zu einem Zielpunkt Minen im Weg sind oder gefährlich werden könnten. Dazu erstellen wir ebenfalls wieder eine Matrix mit allen Minen auf dem Spielfeld und sortieren diese nach Abstand zu uns und nach Berührzeitpunkt, welcher sich ergeben würde, wenn wir direkt in Richtung unseres Zielpunktes fliegen würden. Diese Kriterien ergeben wiederum einen Gesamtgütegrad. Nach diesem Gesamtgütegrad sortieren wir die Minen und erhalten so die für uns gefährlichste Mine. Wir nutzen diese Analyse ebenfalls während des Tankprozesses, indem wir dann Tanken, bei denen Minen auf dem Weg liegen, mit Strafwerten beaufschlagen, sodass eventuell andere Tanken unter diesem Kriterium bevorzugt werden und eventuell einfacher zu erreichen sind.

```
function wegminex=wegmin(zp)
    if spiel.n_mine>0
       wegminematrix=zeros(spiel.n_mine,1);
      for x=1:spiel.n_mine
       wegminematrix(x,1)=x;
       wegminematrix(x,2)=norm(spiel.mine(x).pos-ich.pos);
       wegminematrix(x,3)=spiel.mine(x).pos(1);
       wegminematrix(x,4)=spiel.mine(x).pos(2);
       wegminematrix(x,5)=szklx(ich.pos,zp-ich.pos,spiel.spaceball_radius,
           spiel.mine(x).pos,[0,0],spiel.mine_radius);
       wegminematrix(x,6)=wegminematrix(x,2)*kmw1+wegminematrix(x,5)*kmw2;
      end
       wegminematrix1=sortrows(wegminematrix,6);
       wegminepos=[wegminematrix1(1,3),wegminematrix1(1,4)];
       wegminex=wegminematrix1(1,1);
    else
       wegminex=1;
    end
end
```

3.1.7 tangver-Funktion

Die *tangver*-Funktion ist unsere Tangenten-Ausweichfunktion, wenn wir auf Minen zusteuern. Bevor diese Funktion überhaupt aktiviert wird, muss unsere Minenabwehr erst einmal feststellen, ob eine Mine vor uns im Weg steht. Dies wird im Kapitel Minenabwehr[3] genauer beschrieben Dort ist es, grob gesagt, definiert, wenn unser Teilbremsweg in Richtung der gefährlichen Mine kleiner wird, als das Dreifache des Abstandes zu dieser Mine, so wird das Tangentenverfahren eingeleitet. Dann wird ledigdlich nur noch der Vektorprojektionsvektor vom Vektor unserer Position zum Minenmittelpunkt zu dem Vektor unserer Position in Richtung unseres Zielpunktes gebildet. So wird ein günstiger Ausweichprozess auf dem kürzesten Weg um die Mine herum eingeleitet, bis die Mine nicht mehr im Weg liegt. Falls wir während dieses Prozesses der Mine doch gefährlich nah kommen sollten, greift eine Minennotabwehrfunktion.

```
1 function tng=tangver(zielp,nmp)
2         g1m=nmp-ich.pos;
3         g2z=zielp-ich.pos;
4         tng=(vp(zielp,ich.pos,g1m)/(norm(vp(zielp,ich.pos,g1m))))*0.07;
5 end
```

[3] siehe Kapitel 3.5 Minenabwehr

3.2 Tanken

3.2.1 Bedingungen zur Aktivierung des Tankmodus

Unser Tankmodus wird grundsätzlich aktiviert, wenn überhaupt noch Tanken auf dem Spielfeld sind, wenn unser Spaceball weniger als zehn Tanken gesammelt hat, da so alle Tanken eingesammelt werden, und wenn der Gegner weniger als fünf Tanken getankt hat, da unser Spaceball sonst in die Verteidigung wechseln würde.
Sind alle diese Bedingungen erfüllt, wird unsere Tankfunktion aufgerufen.

```
1 if ich.getankt<10 && gegner.getankt<5 && spiel.n_tanke>0
2    bestanken=tanken;
3 end
```

3.2.2 Zusammenführen von Daten über einzelnen Tanken

Wenn überhaupt noch Tanken auf dem Spielfeld sind, wird eine Matrix mit so vielen Zeilen erstellt, wie noch Tankstellen auf dem Spielfeld sind.
Die Matrix sammelt verschiedene Daten über die einzelnen Tanken.

Es wurden folgende Daten gesammelt:

- die vom Spiel gegebene Tankennummer: Die Tankennummer wird der spieleigenen Tankenmatrix entnommen.
- die Tankstellen-Koordinaten: Die Koordinaten der Tanken werden ebenfalls aus der spieleigenen Tankenmatrix entnommen.
- der Abstand der jeweiligen Tanke zu unserem Spaceball: Der Abstand wird über die Koordinaten mit normaler Vektorabstandsmessung[4] ermittelt.
- ein theoretischer Berührzeitpunkt unseres Spaceballs mit der jeweiligen Tankstelle: Der theoretische Berührzeitpunkt wird mit Hilfe der Kollisionsanalyse und unsere Geschwindigkeitsanteils[5] in Richtung der jeweiligen Tanken ermittelt.
- der individuelle Abstand der Tanken zum Tanken-Schwerpunkt im Spielfeld: Der individuelle Abstand einer Tanke zum Tankstellen-Schwerpunkt kann mit Hilfe der *sumabs*-Funktion ermittelt werden.
- Kollisionsanalyse-Ergebnisse mit Minen: Hier wird ebenfalls mit Hilfe der Kollisionsanalyse überprüft, ob eine Mine auf dem Weg zur jeweiligen Tankstelle gefährlich werden könnte.

Diese Daten werden dann für jede Tanke zu einem Gesamtgütegrad mit verschiedenen Faktoren zusammengefasst. Prinzip:

$$x \cdot a + y \cdot b + z \cdot c = Gesamtgütegrad$$

Nach diesem Gesamtgütegrad werden die Tanken dann nach dem kleinsten Wert sortiert.

```
1 nexttankematrix1=sortrows(nexttankematrix,10);
```

Zu Gütegradbestimmung unserer Turnier-Version haben wir den Abstand von uns zur Tanke, den theoretischen Berührzeitpunkt mit der Tanke und die Minenkollisionsanalyse mit einfließen lassen. Dabei haben wir den Abstand im Vergleich zu den anderen Kriterien sehr stark bewertet. Die Tanke, die nach dem Sortiervorgang oben in der Matrix steht, ist unsere nächstbeste Tanke, welche wir anfahren wollen.

[4] siehe Kapitel 2.2.3 Grundlegende Berechnungen
[5] siehe Kapitel 3.1.3 Vektorprojektion

In unserer Turnier-Beschleunigungsdatei haben wir bewusst nicht alle gesammelten Daten genutzt, da weitere Auswahlkriterien den Auswahlprozess zwar genauer und besser machen können, diesen aber auf jeden Fall komplexer machen. Wir haben deshalb auf weniger Auswahlkriterien zurückgegriffen, da sich so der Auswahlprozess über die nächstbeste Tanke klarer, intuitiver und einfacher gestaltet.
Diese Berechnungen stehen allerdings noch in unserer Datei, damit diese Gedanken auch nach diesem Projekt noch weiter verfolgt und fortgeführt werden können.

3.2.3 Sonderfälle während des Tankens

Nach dieser Matrix-Berechnung wird für das Tangentenverfahren der neue Zielpunkt (*zp*) definiert, damit das Tangentenprogramm möglichst günstig um die Minen herumsteuern kann.
Danach wird gesondert berechnet, ob der Gegner oder wir schneller an unserer nächsten Tanke ankommen. Dazu nutzen wir eine Berührzeitpunkt-Analyse. Wenn der Gegner schneller an unserer nächstbesten Tanke ist, steuern wir dann auf unsere zweitbeste Tanke zu. Daraufhin wird diese Tanke dann zu unserem nächsten Zielpunkt.

```
1  if ich.getankt<5 && lambdatankeich>lambdatankegegner && spiel.n_tanke>1 &&
       geradetanke==0
2     bestanken1=nexttankepos2-ich.pos-ich.ges;
3       ....
4  end
```

Des Weiteren wird überprüft, ob zwei oder drei Tankstellen mit unserem Spaceball-Geschwindigkeits-Vektor auf einer Geraden liegen, sodass unser Spaceball dann direkt durchbeschleunigt bis zur letzten Tanke auf der Geraden.

```
1  if   spiel.n_tanke>1 && geradetankem(1,6)<100 && geradetankem(2,6)<100
              && wegmin3>szklx2(ich.pos,ich.ges,spiel.spaceball_radius,
              [geradetankem(2,3),geradetankem(2,4)],[0,0],spiel.tanke_radius)
                && norm([geradetankem(2,3),geradetankem(2,4)]-ich.pos)<0.5
2      ....
3     gerade=1;
4  end
```

Eine weitere Sonderberechnung in der Tankfunktion ist für den Fall der letzten Tanke. Steht es vier zu vier und ist der Gegner schneller an der letzten Tanke, wird daraufhin der Verteidigungsmodus aktiviert.

3.2.4 Anfahren der Tanken

Nachdem die Tanke, die wir als nächste anfahren wollen, ausgewählt ist, beginnt die Auswahl des Anfahrverfahrens.

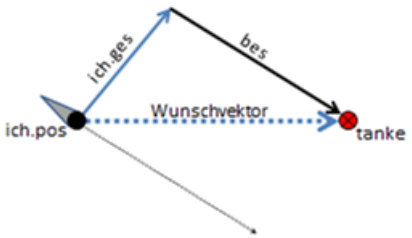

Abbildung 3.1: Anfahren einer Tanke

Grundsätzlich bewegen wir uns mit Hilfe einer Vektordifferenz auf die Tanke zu. Das hat den Vorteil, dass wir gleichzeitig auf den Zielpunkt zu beschleunigen, während wir auch die Richtung auf den Zielpunkt ändern.
Wenn wir nur noch einen sehr kleinen Abstand bis zur nächsten Tanke haben, beschleunigen wir nur noch zu Tanke hin.

3.2.5 Spontanausführungen

Für den Fall, dass wir dem Gegner sehr nahe kommen und wir zudem noch mehr getankt haben, schaltet sich im Tankmodus ein Spontanangriff ein, der schnell versucht, den Gegner zu erwischen.

```
1 % Spontanangriff
2  if gegner.getankt<ich.getankt && abs_geg<0.06 && norm(ich.ges)<0.1
3     bestanken1=gegner.pos-ich.pos;
4 end
```

Für den genau umgekehrten Fall, wenn wir dem Gegner nahe kommen und er mehr getankt hat als wir, schaltet sich ein Spontanverteidigungsmodus ein, welcher in dem Moment nur versucht, vom Gegner weg zu beschleunigen.

```
1 % Spontanverteidigung
2  if gegner.getankt>ich.getankt && abs_geg<0.05
3     bestanken1=vp(ich.pos,gegner.pos,gegner.ges);
4 end
```

3.3 Angriff

3.3.1 Bedingungen zur Aktivierung des Angriffs

Wenn unser Spaceball fünf Tanken oder mehr getankt hat und keine Tanke mehr auf dem Spielfeld ist, aktiviert sich unser Angriffsmodus.
In einem früheren Stadium unser Datei haben wir bereits den Angriff begonnen, wenn wir genau fünf Tanken getankt haben. Dies hat sich aber mit der Zeit als ungünstig erwiesen, da es dem Gegner die Möglichkeit gab, sich hinter den Minen zu verstecken. Deswegen aktiviert sich unser Angriff auch erst, wenn die letzte Tankstelle getankt wurde.

```
1 if ich.getankt>=5 && spiel.n_tanke==0
2     bestanken=[0,0];
3     besangriff=ang;
4 end
```

3.3.2 Aufbau der Angriffsfunktion

Zu Beginn der Angriffsfunktion wird eine eigene Kollisionsanalyse mit Minen durchgeführt, welche besonders wichtig für unseren Spezialangriff ist. Dazu haben wir ebenfalls eine Matrix mit allen noch vorhandenen Minen auf dem Spielfeld erstellt. Diese Matrix haben wird dann nach einem Gütegrad aus Abstand und möglichem Berührzeitpunkt sortiert, sodass wir so die gefährlichste Mine für unser Angriffsverfahren herausfinden können.

```
1 agerademinem=sortrows(agerademinematrix,5);
```

Wenn wir dann in der Kollisionsanalyse mit einbezogener Sicherheit mit der gefährlichsten Mine einen Berührzeitpunkt haben, so wird unsere Konstante *nomine* zu 1 gesetzt, sodass bestimmte Angriffe daraufhin zunächst einmal gar nicht mehr möglich sind, solange eine Mine im Weg ist.
Daraufhin folgen die einzelnen Fälle mit den zugehörigen Angriffs-Anfahrverfahren.

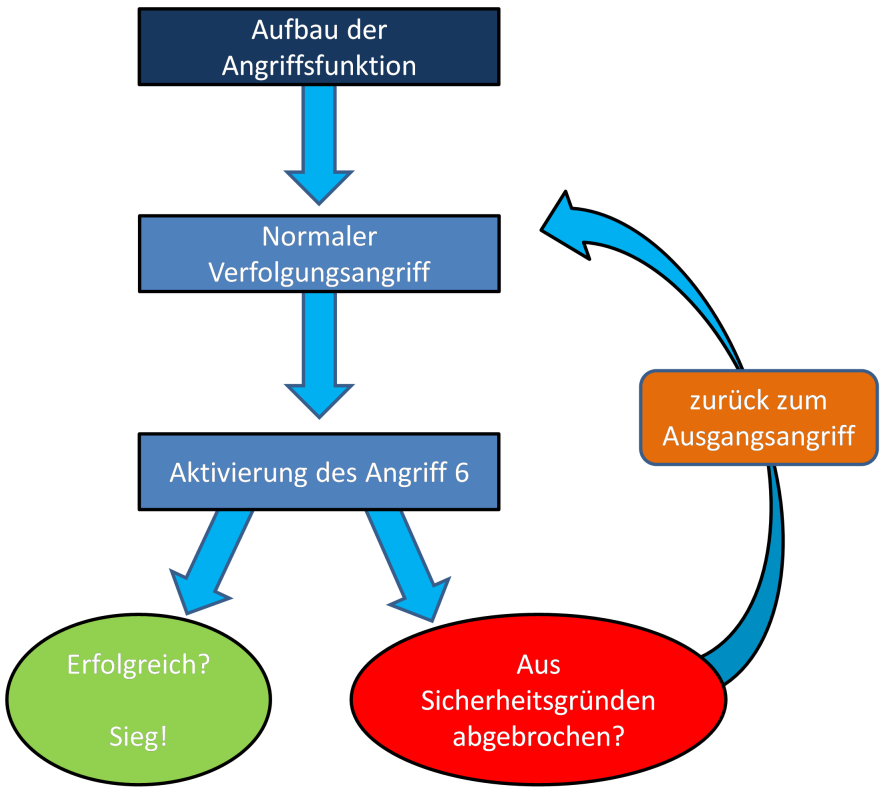

Abbildung 3.2: Ablauf der Angriffsfunktion

3.3.3 Allgemeine Angriffsarten

Auch im Angriff bewegen wir uns grundsätzlich mit der Vektordifferenz fort. Da der Gegner höchstwahrscheinlich versucht, uns zu entkommen, hielten wir es für sinnvoll, grundsätzlich einen Vorhaltewinkel zum Gegner anzusteuern. Diesen Vorhaltewinkel generieren wir, indem wir nicht die Position des Gegners selber anfliegen, sondern einen fiktiven Punkt ansteuern, der entsteht, wenn wir zur Position des Gegners dessen Geschwindigkeitsvektor addieren oder sogar ein Vielfaches davon. So schafft man es, während des Anfahrprozesses immer einen etwas kürzeren Weg zu fahren als der Gegner, was dem eigenen Spaceball Zeit spart und den Abstand zum Gegner immer weiter verkleinert.
Bei größeren Geschwindigkeiten unseres eigenen Spaceballs nutzen wir dann auch die Vektorprojektion, da der Spaceball sonst zu schnell wird und seine Agilität verliert. Deshalb soll unser Spaceball ab einer Geschwindigkeit größer als 0.25 nur noch Richtungsänderungen über die Vektorprojektion durchführen, da so die Geschwindigkeit nicht weiter erhöht wird und unser Spaceball so besser auf den Gegner zielen kann.

```
1 besangriff1=(gegner.pos+1*gegner.ges)-ich.pos-ich.ges;
2    dang=1;
3 if norm(ich.ges)>0.25
4    besangriff1=vp((gegner.pos+2*gegner.ges),ich.pos,ich.ges);
5    ....
6 end
```

3.3.4 Spezialangriff

Der Spezialangriff soll an sich ein unbesiegbarer Angriff sein. Solange nicht eine Mine auf dem Weg unseres Spaceballs auftaucht, hat der Gegner eigentlich keine Chance mehr, unserem Angriff auszuweichen.
Damit dieser Angriff eingeleitet wird, muss eine Reihe an Bedingungen erfüllt sein.

1. Eine Kollisionsanalyse wird zwischen unserem Spaceball und dem Gegner durchgeführt. Diese sollte einen bestimmten zeitlichen Wert unterschreiten, damit der Spaceball nah genug am Gegner ist, darf aber gleichzeitig auch nicht kleiner als Null werden, da der Gegner sonst hinter dem eigenen Spaceball ist.

2. Der Betrag der Geschwindigkeit unseres Spaceballs sollte zudem großer als 0.1 sein, damit er sich grundsätzlich nicht zu langsam fortbewegt.

3. Wie viele Minen noch auf dem Spielfeld sind ist egal, da der Spezialangriff dann abgebrochen und neu aufgenommen wird.

4. Die Geschwindigkeit unseres Spaceballs soll um das 1.5-fache größer sein als die Geschwindigkeit des Gegners, damit man auf jeden Fall eine Geschwindigkeit in Richtung des Gegners hat.

5. Die letzte Bedingung, die gelten muss, ist, dass keine Mine auf dem Weg liegt, welches durch die Minenkollisionsanalyse am Anfang der Datei überprüft wird.

Somit schafft man es quasi automatisch, auf dem Kollisionskurs mit dem Gegner zu bleiben, solange der Angriff nicht durch eine Mine unterbrochen wird.

Sind all diese Bedingungen erfüllt, wird der *Angriff 6* aktiviert und eingeleitet. Unser Spaceball macht dann einfach dieselbe Beschleunigung wie der Gegner. Somit kann der Gegner auch nicht mehr ausweichen.
Wenn alle Bedingungen zutreffen, wird unsere persistente Variable *Angriff 6* gleich *true* gesetzt und unser Beschleunigungsvektor macht ab diesem Zeitpunkt dann nur noch das, was der Beschleunigungsvektor des Gegners macht. Wenn die persistente Variable *Angriff 6* gleich *true* ist, so wird gleichzeitig auch die Bandenabwehr deaktiviert, da diese sonst den *Angriff 6* negativ beeinflussen und bremsen würde. Dies tun wir, da wir nach Aktivierung des Angriffs wissen, dass wir ohne Hindernisse den Gegner treffen. Lediglich die Minenabwehr bleibt aktiviert, da sich noch Minen auf dem Spielfeld befinden, die eventuell im Weg stehen. Wird während des Angriffs festgestellt, dass eine Mine im Weg ist, wird der *Angriff 6* unterbrochen, indem *Angriff 6* wieder gleich *false* gesetzt wird. Dann sorgen Minenabwehr und Bandenabwehr zunächst wieder dafür, dass unser Spaceball nicht in Minen oder Banden fährt. Ist dies sichergestellt, so wird der Angriff neu aufgenommen. Je kleiner die Anzahl der noch übrigen Minen ist, desto größer ist die Chance, dass, während des unbesiegbaren *Angriff 6* kein Hindernis im Weg liegt. Wenn man mehr Tankstellen getankt hat als der Gegner, ist mit diesem Angriff die Wahrscheinlichkeit ziemlich hoch, dass man das Spiel letztendlich auch gewinnt.

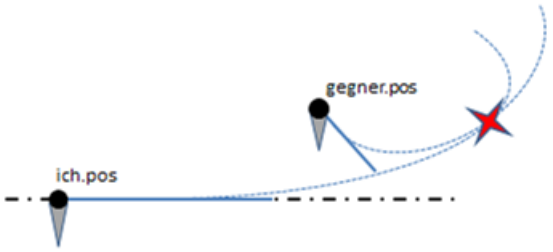

Abbildung 3.3: Spezialangriff

3.4 Verteidigung

3.4.1 Idee

Mit der Verteidigung haben wir uns eher gegen Ende des Projekts beschäftigt. Wir hatten zu Beginn lediglich eine ganz simple Verteidigung programmiert, um dem Gegner ein bisschen auszuweichen.
Später wurde die Verteidigung immer wichtiger, weil es doch einige Spiele gab, in denen der Gegner mehr getankt hatte als wir. Deshalb haben wir uns überlegt, dass es schlau wäre, die Verteidigung so zu programmieren, dass wir möglicherweise doch noch ein Unentschieden beziehungsweise doch sogar noch einen Sieg erreichen können. Dadurch kam uns die Idee der Eckenverteidigung. Ziel hierbei ist es, die nächstgelegene Ecke auszuwählen, wenn der Gegner mehr tankt als wir, und sich dann dort hin zu begeben. Sobald der genannte Fall eintritt, sucht sich das Programm aus der erstellten Eckenmatrix die nächste Ecke und fährt dort hin: *besdef=nexteckepos-ich.pos*. Dafür muss für jede Ecke eine bestimmte *if*-Bedingung aufgestellt werden.

Nachdem wir uns dann in einer Ecke befinden und der Gegner sich auf uns zubewegt, beschleunigen wir im richtigen Moment und fahren an der Bande entlang zur nächsten Ecke und dann wieder zur nächsten Ecke und so weiter. Je nachdem, wie der Gegner seinen Angriff programmiert hat, trifft dieser mit etwas Pech die Bande und wir würden noch einen Punkt einfahren. Wir versuchen also mit unserem Verteidigungsmodus, das Ecken- & Bandenvermeidungsprogramm des Gegners auszureizen und auf die Probe zu stellen. Auf diese Weise kann ein Spiel, trotz Rückstand, noch in einen Sieg umgewandelt werden. Das Hauptziel ist es, aber auf jeden Fall, nicht gefangen zu werden und mit der Idee der Eckenverteidigung erschien uns dies als solide Verteidigungsstrategie, die wir folglich umgesetzt haben.

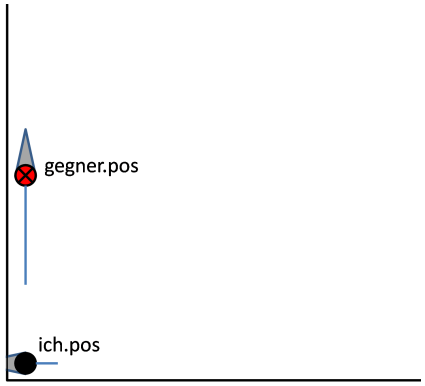

Abbildung 3.4: Eckenverteidigung

3.4.2 Umsetzung der Idee

Zuerst wurden am Anfang der Datei die Eckpunkte der Datei definiert.

```
%% Ecken
eol=[0.006,0.994];
eor=[0.994,0.994];
eur=[0.994,0.006];
eul=[0.006,0.006];
```

Danach haben wir eine Matrix erstellt, welche die vier Ecken nach Abstand zu unserem Spaceball sortiert. Wir definieren die Ecke mit dem kleinsten Abstand zu uns als nächste Ecke, die wir anfliegen wollen. Dies tun wir deshalb, da sich ab diesem Zeitpunkt unsere Eckenverteidigung aktiviert.

```
if norm(nexteckepos-ich.pos)<0.069
    nexteckepkt=1;
end
```

Haben wir den Eckpunkt erreicht, so wird die persistente Variable *nexteckepkt* gleich eins gesetzt und unsere Eckenverteidigung ist somit aktiviert.
Wenn dies der Fall ist, überprüfen wir in Fallunterscheidungen in welcher Ecke wir uns befinden und von wo der Gegner kommt. Mit dieser Fallunterscheidung wird festgelegt, in welche Ecke wir als nächstes fahren.
Dann wird zusätzlich berechnet, welchen Berührzeitpunkt der Gegner zu unserem Spaceball hat.

```
if szklx2(gegner.pos,gegner.ges,spiel.spaceball_radius,ich.pos,ich.ges,
    spiel.spaceball_radius)<abdeft
    t_ausw=true;
else
    t_ausw=false;
end
```

Wird dieser Abstand kleiner als fünf, so wird das Anfahren in die nächste Ecke, welche vorher selektiert wurde, gestartet.
Die Beschleunigungen werden über eine persistente Variable zwischen den Entscheidungsbereichen in der Nähe der Ecken konstant gehalten.
Gleichzeitig wird während der Fallunterscheidung der Sicherheitsfaktor der Bandenabwehr auf 1.1 verkleinert, da theoretisch immer nur noch eine Bande gefährlich werden kann. Zusätzlich schalten wir durch eine weitere persistente Variable die Minenabwehr aus.
Somit fliegt der Spaceball nun selbstentscheidend von Ecke zu Ecke und versucht so dem Gegner auszuweichen. Dabei werden die Entscheidungen, wohin unser Spaceball als nächstes fliegt, jeweils nur in der Nähe der Ecken getroffen.

3.5 Minenabwehr

3.5.1 Idee

Die Minenvermeidung spielt bei Spaceballs eine große Rolle und es ist sehr wichtig, dass diese gut funktioniert. Minen stellen zu jedem Zeitpunkt des Spiels eine Gefahr da, egal ob man sich im Angriff, der Verteidigung oder noch im Tankmodus befindet. Solange also noch Minen auf dem Spielfeld existieren, besteht die Gefahr, einen Punkt abzugeben, falls man eine Mine trifft.
Beim Programmieren der Minenverteidigung stellt man sich der Herausforderung, die Minen so zu umfahren, dass andere Prozesse (z.B. das Tanken) nicht in großem Maße behindert werden. Die Zielpunkte um die Mine herum sollten also mit einer möglichst kleinen, aber ausreichenden Sicherheit gewählt werden, sodass man so wenig Zeit wie möglich beim Tankprozess verliert. Zu Beginn des Vermeidungsprogramms wird die Bedingung abgefragt, ob Minen auf dem Spielfeld vorhanden sind. Schließlich wird die Minenvermeidung nur dann benötigt, wenn auch noch Minen vorhanden sind. Nachdem alle Tanken eingesammelt wurden, verschwindet nämlich alle drei Sekunden eine Mine bis keine mehr vorhanden ist.

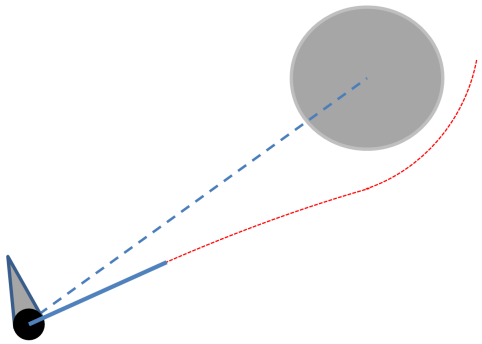

Abbildung 3.5: Minen-Ausweichmanöver

3.5.2 Programmierung

Wir haben zunächst eine Minenmatrix erstellt, um dann die Position der nächsten Mine herauszufinden. Auch hier haben wir mit der Funktion Schnitt zweier Kreise (*szk*) gearbeitet.

Tangentenverfahren

```
1 if norm(nextminepos-ich.pos)-0.0601< bwmineba*3 && zeit_nextminetang(1)<
      zeit_zpt
2     ....
3         bestanken=[0,0];
4         besangriff=[0,0];
5         besdefence=[0,0];
6         minendef1=tangver(zp,nextminepos);
7 end
```

Grundsätzlich wollen wir die Minen aber möglichst effizient umfahren, um beim Tankprozess nicht behindert zu werden und damit wir keine Geschwindigkeit dadurch verlieren, dass wir unnötigerweise abbremsen.

Deshalb haben wir ein Tangentenverfahren ins Leben gerufen, um uns geschickt um die Minen zu bewegen. Wir rechnen uns durch die Matrix zunächst die gefährlichste Mine aus. Wenn dann der Bremsweg in Richtung dieser Mine plus dem Minenradius größer wird als der Abstand selbst, dann greift unser Tangentenprogramm[6]. Hier haben wir den Faktor größer gewählt, als beim Notprogramm (hier: Faktor 3), weil das Verfahren dann früher greift als bei der Notverteidigung. Wenn man dies als Ausweichkriterium nimmt und die Bedingungen stimmen, dann arbeiten wir in dem Tangentenmanöver mit dem Vektorprodukt. Das Vektorprodukt ergibt sich aus der Geraden zu dem Zielpunkt, den wir immer definiert haben, und der Geraden von uns zum Minenmittelpunkt. Der Projektionsvektor, der sich wiederum dadurch ergibt, gilt nun als Ausweichvektor um die Mine herum. Dieser gilt dann solange, bis wir nicht mehr auf Kollisionskurs sind und unser Spaceball fliegt wieder geradeaus.

Minennotabwehr

Das Minennotfallprogramm hat die Aufgabe dann einzugreifen, wenn das normale Tangentenverfahren nicht aufgerufen wird.
Wenn der Fall eintritt, dass das Notfallprogramm greift, soll in jedem Fall die Kollision mit einer Mine verhindert werden, indem vor der Mine abgebremst wird. Wie der Name bereits sagt, soll dieses Programm nur im Notfall greifen, da ansonsten unser Tangentenverfahren um die Mine herum angewendet wird.
Für die Minennotabwehr berechnen wir den Bremsweg bis zur Mine und vergleichen diesen mit der Strecke von uns zu dieser gefährlichen Mine. Wenn der Bremsweg mit einer eingebauten Sicherheit, in diesem Fall mit dem Faktor 1.1 größer als der Bremsweg wird, beschleunigen wir entgegengesetzt und weichen somit der Mine aus. Dieses Programm soll wirklich nur im Notfall greifen, wenn es nicht anders möglich ist. Hier sieht man die Minennotabwehr, die wir auch als erstes programmiert haben.

```
1 if norm(nextminepos-ich.pos)-0.0601< bwmineba*1.1 && zeit_nextmine<100
2     ....
3      bestanken=[0,0];
4      besangriff=[0,0];
5      besdefence=[0,0];
6      minendef1=ich.pos-nextminepos;
7 end
```

[6]siehe Kapitel 3.1.7 *tangver*-Funktion

3.6 Bandenabwehr

Am Anfang der Beschleunigungsdatei werden in jedem Schritt die Abstände zu allen Banden und die Bremswege in x-Richtung und in y-Richtung berechnet.
Dann wird für jede Bande berechnet, ob der Spaceball eine positive Geschwindigkeit in Richtung der Bande hat und ob der Abstand zur Bande kleiner oder gleich dem Bremsweg in Bandenrichtung multipliziert mit einem Sicherheitsfaktor ist. Ist eine der vier Bandenbedingungen erfüllt, so werden zunächst die Teilbeschleunigungen des Tankens, des Angriffs und der Verteidigung gleich null gesetzt. Danach wird eine Beschleunigung ausgegeben, die entgegen der gefährlichen Bande gerichtet ist, sodass die Geschwindigkeit in Richtung dieser Bande komplett abgebaut wird.

```
1 % Bandenprotektion rechte Bande
2 if abs_rb<=bwx*bv && ich.ges(1)>0
3     bestanken=[0,0];
4     besangriff=[0,0];
5     besdefence=[0,0];
6     bes1=[-1,0];
7 end
```

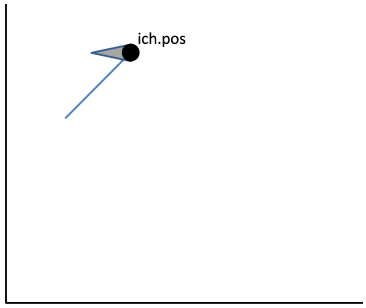

Abbildung 3.6: Ein-Bandenabwehr

Sollte der Fall eintreten, dass zwei Bandenbedingungen gleichzeitig erfüllt sind, so werden die beiden Beschleunigungsvektoren entgegen der gefährlichen Banden einfach addiert zu einem Abbremsvektor. Dadurch, dass wir in die Erkennung der Banden einen Sicherheitsfaktor miteinbezogen haben, reicht der somit leicht längere Bremsweg[7] in Richtung der gefährlichen Banden trotzdem aus, um vor den Banden zum Stehen zu kommen.

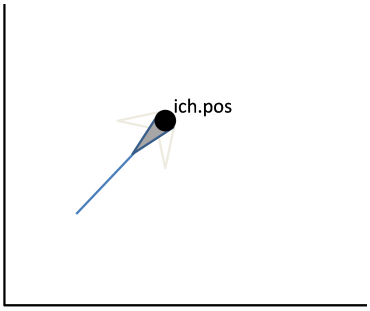

Abbildung 3.7: Zwei-Bandenabwehr

[7] im Vergleich zum Direktbremsweg in einer Bandenrichtung

3.7 Zusammenführung aller Teilbeschleunigungen

Am Ende unserer Beschleunigungsdatei kommt unsere finale Beschleunigungsvektorbestimmung. Diese unterteilt sich in zwei Teile.
Der erste Teil ist eine Notbremsfunktion für den Fall, dass bei hohen Geschwindigkeiten des Spaceballs Minenabwehr und Bandenabwehr gleichzeitig aktiv werden. Wenn zum Beispiel unser Spaceball mit hoher Geschwindigkeit einer Bande ausweicht und aus dem Grund gezwungener Maßen eine Mine in der Flugbahn auftaucht, muss eine Notbremsung durchgeführt werden, um die Geschwindigkeit so schnell es geht zu verringern. Die Geschwindigkeit muss so weit verringert werden, bis die Beschleunigungsfunktion wieder nur auf eine der beiden Gefahren reagieren muss.
Der zweite Teil ist die normale Beschleunigungsbestimmung durch Zusammenführung der Teilbeschleunigungen von den verschiedenen Modi. Dabei werden einfach alle Teilbeschleunigungen zusammen addiert. Dabei werden während den verschiedenen Modus-Funktionen bestimmte Teilbeschleunigungen gleich null gesetzt, sodass diese bei der Zusammenführung nicht gelten. So ist gewährleistet, dass immer nur der Beschleunigungswert durch die Zusammenführung kommt, welcher auch ausgegeben werden soll.

```
1 if  sonderfall_bande==1  && minendang==1
2     bes=-ich.ges;
3 else
4     bes=bes1+bes2+bes3+bes4+bestanken+besangriff+besdefence+besmine;
5 end
```

Kapitel 4

Rückblick

4.1 Turnierergebnis

Nach einigen Wochen harter Arbeit und großem Zeitaufwand blicken wir auf ein zufriedenstellendes Ergebnis zurück. Unser Ziel war es, eine wettbewerbsfähige „Künstliche Intelligenz“ ins Rennen zu schicken. Dazu haben wir die Hauptaufgaben zunächst untereinander aufgeteilt, aber mit der Zeit haben wir uns gegenseitig bei verschiedenen Problemen geholfen. Die Programmierung der „Künstlichen Intelligenz“ an sich war eine sehr umfangreiche Aufgabe, die von uns gut bewältigt werden konnte. Mit dem zweiten Platz beim Spaceballs Turnier sind wir nun sehr zufrieden. Wir haben eine von Grund auf solide „Künstliche Intelligenz“ programmiert und die Platzierung kann sich durchaus sehen lassen. Das Projekt könnte in vielen Bereichen durchaus noch fortgesetzt und verbessert werden, aber alles in allem hat sich die Arbeit gelohnt und die Zeit, die wir investiert haben wurde mit dem zweiten Platz belohnt.

Platz	Team	Punkte
1	The Dark Side	15
2	JETI SpaceWars	13
3	Asyl Air	12
4	Space Sweeper	10
5	SpaceHansa	8
6	Halbwertszeit	6
7	Snitch Snatchers	6
8	bremAIRs	2

Abbildung 4.1: Turnierergebnis