Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Sicherheitsarchitektur zur Kollisionsvermeidung für autonome Modellautos

Gerrit Hinderland
Matrikelnummer: 4764223
gerrit.hinderland@fu-berlin.de

Eingereicht bei: Prof. Dr. Daniel Göhring

Berlin, 12.10.2018

Zusammenfassung

Sicherheitsarchitekturen im Bereich des autonomen Fahrens dienen zum Schutze des Fahrers und des Fahrzeugs. Doch bevor eine tatsächliche Anwendung im Straßenverkehr stattfinden kann, muss die Software zunächst auf fahrerlosen Modellautos getestet werden. Allerdings sind auch die Modellautos selber gefährdet und könnten Schaden durch Kollisionen nehmen. In dieser Arbeit wird eine Sicherheitsarchitektur aufgebaut, deren konkretes Ziel der Schutz der Modellautos ist.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Dokumentstruktur 1

2 Grundlagen **2**
2.1 Aufbau der Modellautos 2
2.1.1 Hardware 2
2.1.2 Software 3
2.2 Ausgangslage 4

3 Verfahren **7**
3.1 Umgebungskarte erstellen 7
3.1.1 OccupancyGrid Datentyp 7
3.1.2 Nachrichten des Laserscanners 8
3.1.3 Markieren des Feldes 10
3.2 Berechnen des Pfades 12
3.2.1 Kalibrierung der Lenkung 13
3.2.2 Radius 14
3.2.3 Mittelpunkt 15
3.3 Hindernisse finden 16
3.3.1 Bresenhams Midpoint-Kreis-Algorithmus 16
3.3.2 Kreissegmente zeichnen 18
3.4 Lösungsansatz 24

4 Experimente **26**
4.1 Variante der Umgebungskarte 26
4.2 Kreis Algorithmus & Laufzeit 27

5 Fazit und Ausblick **29**

Eidesstattliche Erklärung **30**

Literaturverzeichnis **31**

Kapitel 1

Einleitung

Im Rahmen der Forschung zum autonomen Fahren wird im Fachbereich Informatik an der Freien Universität Berlin mit Modellautos gearbeitet. Sie können entweder über beispielsweise Handy-Apps manuell ferngesteuert werden oder aber durch Programme autonom fahren.

Diese Autos sind jedoch Schäden gegenüber anfällig. Schlimmstenfalls werden durch einen Schaden die Arbeitsabläufe verlangsamt und Ressourcen werden unnötig verschwendet.

Zum Schutz der Fahrzeuge wird auf ihnen bereits ein simples *autostop*-Programm ausgeführt. Dies lässt ein Auto anhalten, sollte in einer festgelegten Entfernung direkt davor (oder dahinter beim Rückwärtsfahren) ein Objekt stehen. Dennoch kann es passieren, dass gerade bei Kurven ein Objekt nicht als gefährlich angesehen wird.

Das Ziel dieser Arbeit ist es, für die Modellautos eine robuste Sicherheitsarchitektur aufzubauen. Durch diese sollen Kollisionen sowohl beim Fernsteuern der Modellautos als auch beim autonomen Fahren vermieden werden.

1.1 Dokumentstruktur

Im folgenden Kapitel werden zunächst die Modellautos, die im Rahmen dieser Arbeit verwendet wurden, und die auf ihnen laufende Software beschrieben. Des Weiteren wird der aktuelle Stand der Sicherheitsarchitektur beschrieben und die darin existierende Problematik hervorgehoben. In Kapitel 3 wird dann eine neue Sicherheitsarchitektur beschrieben, deren Ziel es ist, die Sicherheit der Modellautos zu erhöhen. Zum Schluss werden einzelne Entscheidungen, die im Laufe der Arbeit getroffen werden, durch Vergleiche in der Berechung und Anwendung genauer erklärt.

Kapitel 2

Grundlagen

2.1 Aufbau der Modellautos

In diesem Kapitel wird beschrieben, wie die verwendeten Modellautos aufgebaut sind. Der Aufbau wird sowohl von der Hardwareseite als auch von der Softwareseite betrachtet. Dabei wird ein Fokus auf die im Laufe dieser Arbeit benutzten Technologien gelegt.

Abbildung 2.1: Ein Modellauto
Quelle: AutoModelCarWiki

2.1.1 Hardware

Es werden verschiedene Versionen der Modellautos benutzt, die allerdings ähnlich aufgebaut sind. Wichtig für diese Arbeit sind lediglich zwei Bestandteile der Hardware:

- Der rotierende Laserscanner, durch den in 360° um das Auto herum gemessen wird, in welcher Entfernung sich Objekte befinden.

- Die durch einen Servomotor betriebene Vorderradlenkung. Die Lenkung ist eine sogenannte Ackermann-Lenkung, durch die die beiden Vorderräder mit verschieden großen Winkeln eingelenkt werden. Das führt dazu, dass sich beide Räder auf verschiedenen Kreisbahnen befinden, beide jedoch denselben Mittelpunkt besitzen.

2.1.2 Software

Das auf den Autos laufende Betriebssystem ist Ubuntu Linux[1]. Des Weiteren wird das Framework "Robotic Operating System" (*ROS*)[2] verwendet. ROS liefert mehrere Bibliotheken, die die Entwicklung von Robotiksystemen vereinfachen. Darüber hinaus bringt ROS eine Hardwareabstraktion mit sich, sodass es in verschiedensten Systemen Gebrauch findet.

Programmierung für ROS kann in zwei verschiedenen Programmiersprachen umgesetzt werden: C++[3] und Python[4]. In dieser Arbeit werden die Sprache Python und die Bibliothek *rospy* verwendet.

ROS verwendet ein Graphenkonzept, bestehend aus Knoten und Kanten. Die Knoten entsprechen den ausführbaren Programmen, die auf den Autos laufen und diese steuern. Eine Kante existiert zwischen zwei Knoten, wenn sie untereinander Nachrichten austauschen. Dies kann sowohl in nur einer Richtung als auch in beiden Richtungen geschehen.

Zum Senden und Empfangen solcher Nachrichten werden sogenannte Topics und ein Publisher-Subscriber-Modell verwendet. Topics sind Nachrichtenkanäle, durch die ein Knoten Nachrichten senden (*Publisher*) oder empfangen (*Subscriber*) kann.

Die Programme können sowohl direkt auf den Modellautos laufen als auch von einem Computer gestartet werden, welcher mit dem Auto verbunden ist. Darüber hinaus kann zur sicheren Entwicklung von Programmen das Simulationswerkzeug Gazebo[5] verwendet werden. Indem damit auf dem Computer die Modellautos und ihre Umgebung lediglich simuliert werden, kann Schaden an den echten Autos durch fehlerhaften Code vermieden werden.

[1] https://www.ubuntu.com
[2] http://www.ros.org
[3] https://isocpp.org
[4] https://www.python.org
[5] http://gazebosim.org

2.2 Ausgangslage

Bisher wird ein einfaches Programm verwendet, das die Modellautos daran hindert, gegen Hindernisse zu fahren. Dies wird mittels des Laserscanners (s. Kapitel 3.1.2) und der aktuellen Geschwindigkeit des Autos realisiert.
Die Auszüge aus dem Quellcode[2] dieses *auto_stop*-Programms wurden in C++ geschrieben und zur Leserlichkeit leicht angepasst.

Zunächst wird, falls es die Variable *brake_distance_based_on_speed* verlangt, der Bremsweg in Abhängigkeit von der Geschwindigkeit berechnet:

```
1 float brake_distance_ = brake_distance;
2 if (abs(direction) > 50 && brake_distance_based_on_speed)
3   brake_distance_ = (abs(direction) / 50) * brake_distance;
```

Listing 2.1: Bremsweg & Geschwindigkeit

Dabei ist *brake_distance* standardmäßig 0.45 Meter und *direction* die Geschwindigkeit des Fahrzeugs.

Danach muss zwischen Vorwärts- und Rückwärtsfahren unterschieden werden, da abhängig davon verschiedene Messwerte des Laserscanners betrachtet werden.
Der Laserscanner beginnt in Richtung Heck des Autos zu messen (0°), bewegt sich im Uhrzeigersinn über die Front des Modellautos (180°) und endet wieder am Heck (360°). Mit den Variablen *angle_front* und *angle_back* werden dann für vorne und hinten die Bereiche festgelegt, die nach Objekten getestet werden sollen.

Durch die folgenden Quellcodeausschnitte wird dieses Scanverhalten realisiert. Hierbei ist *scan* die vom Laserscanner gesendete Nachricht und das darin enthaltene Attribut *ranges* ein Array aller Messwerte des Scanners. *pubSpeed_* ist ein Topic, über welches die Geschwindigkeit des Fahrzeugs gesteuert werden kann. *emergencyStop* entspricht dem Wert 0, sodass das Auto komplett zum Stehen kommt.

Beim Vorwärtsfahren wird also mit dem auf 40° festgelegten Wert *angle_front* um 180° herum nach Hindernissen geschaut.

```
1 for(int j = 180 - (angle_front / 2);
          j <= 180 + (angle_front / 2); j++) {
2   if (scan->ranges[j] <= brake_distance_) {
3     pubSpeed_.publish(emergencyStop);
4     return;
5   }
6 }
```

Listing 2.2: Vorwärtsfahren

Da beim Rückwärtsfahren ein Sprung von 360° auf 0° existiert, müssen der linke Scanbereich (0° bis $angle_back/2$) und der rechte Scanbereich (360° bis $360° - angle_back/2$) getrennt betrachtet werden.

```
// Links
for (int i = 0; i <= angle_back / 2; i++) {
  if (scan->ranges[i] <= brake_distance_) {
    pubSpeed_.publish(emergencyStop);
    return;
  }
}

// Rechts
for (int k = 360 - (angle_back / 2); k <= 360; k++) {
  if (scan->ranges[k] <= brake_distance_) {
    pubSpeed_.publish(emergencyStop);
    return;
  }
}
```

Listing 2.3: Rückwärtsfahren

Nachfolgend ist das Scanverhalten zur Veranschaulichung skizziert.

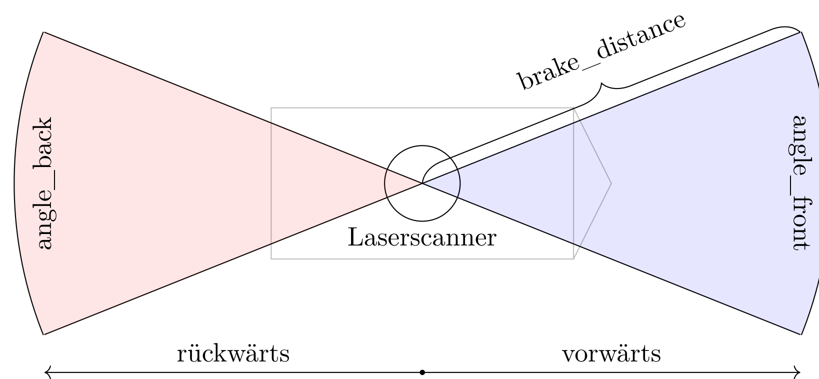


Abbildung 2.2: Visualisierung der aktuellen Methode

Ein Problem allerdings entsteht, wenn sich ein Objekt außerhalb des Sichtfeldes des Modellautos befindet. Fährt das Auto auf dieses in einer Kurve zu, so ist es möglich, dass das Objekt zu spät in den Sichtbereich kommt. Dies könnte schlimmstenfalls zu einer Kollision führen.

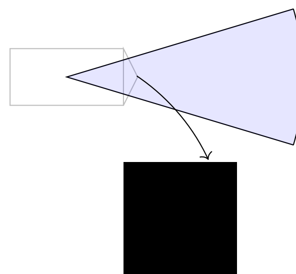


Abbildung 2.3: Problemfall

Um in solchen Fällen Kollisionen zu vermeiden, sollte das Auto immer, wenn sich die Fahrtrichtung ändert, den neuen Pfad nach Hindernissen überprüfen. Dazu muss, anstatt in einem definierten Winkel zu scannen, eine Karte der gesamten Umgebung des Autos erstellt werden. Mit dieser Karte ist der Pfad dann abzugleichen, um zu ermitteln, ob sich darauf Hindernisse befinden. Dieses Verfahren wird im nächsten Kapitel umgesetzt.

Kapitel 3

Verfahren

3.1 Umgebungskarte erstellen

Damit die Modellautos ihre Umgebung ”sehen” und dementsprechend handeln können, wird eine Karte der Umgebung erstellt. Dafür wird der Laserscanner verwendet. Dieser sendet in festgelegten Winkelabständen um das Auto herum Strahlen aus und misst damit, ob und wenn ja in welcher Entfernung Objekte stehen. Diese Informationen müssen zum Übertragen auf eine Karte zunächst verarbeitet werden.

3.1.1 OccupancyGrid Datentyp

```
1 occupancy_grid = OccupancyGrid()
2 occupancy_grid.info.resolution = 0.04
```

Listing 3.1: Initialisiert die Karte

Die Karte wird als ”OccupancyGrid”-Datentyp aus ROS gespeichert. Dieser besteht aus einem Raster quadratischer Felder. Die Größe der Felder ist abhängig von der sogenannten Auflösung des Rasters, die als Seitenlänge der Felder in Meter angegeben wird. Je kleiner also der Wert für die Auflösung ist, desto genauer wird die Karte. Im Umkehrschluss wird dadurch aber auch die Berechnungszeit größer, weshalb ein guter Mittelwert gefunden werden muss. Hier wird ein Wert von 0.04 verwendet, der im fortlaufenden Quellcode als *GRID_RES* bezeichnet wird. Jedem Feld der Karte wird ein ganzzahliger Wert zugewiesen:

-1 Der Scanner kann das Feld nicht sehen, sodass keine Aussage über den Zustand des Feldes getroffen werden kann. Dies ist beispielsweise der Fall, wenn ein Objekt die weitere Sicht des Scanners blockiert oder die Strahlen des Scanners zu große Abstände zueinander haben, sodass mehrere Felder nicht von einem Strahl überstrichen werden. Dieser Wert wird im Folgenden (so wie im Quellcode) als *UNKNOWN* bezeichnet.

0 Das Feld ist frei. Freie Felder können von dem Auto beliebig befahren werden. Dieser Wert wird im Folgenden (so wie im Quellcode) als *FREE* bezeichnet.

100 Das Feld ist von einem Objekt belegt. Würde das Auto ein solches Feld treffen, so fände eine Kollision statt. Dieser Wert wird im Folgenden (so wie im Quellcode) als *OCCUPIED* bezeichnet.

1 - 99 Je näher ein eigentlich freies Feld einem belegten Feld ist, desto höher wird der zugewiesene Wert sein. Dies wird verwendet, um eine "Gefahrenzone" um Objekte herum zu erstellen, die das Auto aus Sicherheitsgründen vermeiden sollte, die aber trotzdem in wenigen konkreten Situationen (z.B. zum Ausweichen anderer Gefahren) befahren werden kann. Im Folgenden werden zwei verschiedene Werte aus diesem Bereich direkt verwendet. Der Wert 50 wird als *DANGEROUS* bezeichnet und dient zur Kennzeichnung von Feldern nahe eines belegten Feldes. *AVOID* entspricht einem Wert von 10 und dient als Vorwarnung für als *DANGEROUS* gekennzeichnete Felder. Dadurch entstehen zwei verschiedene Zonen mit unterschiedlich starken Gefährlichkeitsgraden.

Wenn zum Schluss die Karte vollständig erstellt wurde, muss sie zur Verarbeitung durch andere Knoten veröffentlicht werden. Dazu wird das *scan_grid*-Topic erstellt, über welches die Karte dann verteilt wird.

```
pub_grid = rospy.Publisher("scan_grid", OccupancyGrid)
```

Listing 3.2: Publisher für die Karte

3.1.2 Nachrichten des Laserscanners

```
rospy.Subscriber("scan", LaserScan, scanCallback)
```

Listing 3.3: Subscriber für den Laserscanner

Um Nachrichten des Laserscanners zu erhalten, registriert das Programm einen Subscriber für das *scan*-Topic. Die relevanten Werte in einer Nachricht sind:

angle_min Der Winkel, in dem der erste Strahl zu der Blickrichtung des Modellautos steht.

angle_max Der Winkel, in dem der letzte Strahl zu der Blickrichtung des Modellautos steht.

angle_increment Der Winkelabstand zwischen zwei Messungen.

range_min Die Mindestentfernung in Meter, ab der der Scanner brauchbare Werte misst.

range_max Die Maximalentfernung in Meter, bis zu der der Scanner brauchbare Werte misst.

ranges Ein Array, in dem jedes Element der von einem Strahl gemessenen Entfernung in Meter entspricht. Sollte der Strahl nichts getroffen haben, so ist der zugehörige Werte ”inf” (unendlich). Nur Werte zwischen ”range_min” und ”range_max” sollten verwendet werden.

Bei Eintreffen einer Nachricht wird die Funktion *scanCallback* aufgerufen.

```
def scanCallback(scan_msg):
  resetGrid()  # Erstellt eine leere Karte

  for i in range(0, len(ranges)):
    dist = ranges[i]

    if dist == float("inf") or
       dist > range_max or dist < range_min:
      continue

    alpha = angle_min + i * angle_increment
    if not ((angle_max >= alpha >= angle_min) or
            (angle_max <= alpha <= angle_min)):
      break

    occ_x, occ_y = getGridCoords(dist * math.sin(alpha),
                                 dist * math.cos(alpha))

    # Anzahl der Felder, die der Länge des Autos entsprechen
    l = int(math.ceil(CAR_LENGTH / GRID_RES))
    # Grenzwert zwischen DANGEROUS- und AVOID-Feldern
    boundary = l // 2
    for xi in range(-l, l + 1):
      for yi in range(-l, l + 1):
        if xi == yi == 0:
          setCell(occ_x, occ_y, OCCUPIED)
        elif (abs(xi) <= boundary) and (abs(yi) <= boundary):
          setCell(occ_x + xi, occ_y + yi, DANGEROUS)
        else:
          setCell(occ_x + xi, occ_y + yi, AVOID)

  pub_grid.publish(occupancy_grid)
```

Listing 3.4: scanCallback verarbeitet Nachrichten des Laserscanners

In Zeile 2 wird eine neue Karte erstellt, bei der jedes Feld zunächst auf *UNKNOWN* gesetzt wird. Danach wird mit einer Iterationsvariablen (hier i) nacheinander die von jedem Strahl gemessene Entfernung aus dem *ranges*-Array abgerufen. Hat ein Strahl keine Entfernung messen können (es steht also *inf* an der zugehörigen Stelle in dem Array) oder der Wert ist außerhalb der definierten Grenzen, dann wird durch Erhöhen der Iterationsvariablen der nächste Strahl betrachtet.

Ist aber eine valide Entfernung gegeben, wird in der Karte das zugehörige Feld markiert. Da aber weder in der Nachricht des Laserscanners noch in dem Array

angegeben wird, in welchem Winkel zur Blickrichtung des Autos ein Objekt gemessen wurde, muss dieser aus den gegebenen Informationen berechnet werden:

$$\alpha = angle_min + i \times angle_increment \tag{3.1}$$

Durch die Iterationsvariable ist bekannt, der wievielte Strahl betrachtet wird. Außerdem sind in der Scannernachricht der Anfangswinkel und der Winkelabstand zwischen zwei Strahlen gegeben. Wird der Winkelabstand mit dem Index des aktuellen Strahls multipliziert und das Produkt zu dem Anfangswinkel addiert, ergibt sich der gesuchte Winkel. In Zeile 11 wird dann überprüft, ob dieser auch in den gegebenen Grenzen liegt.

3.1.3 Markieren des Feldes

Mit dem nun bekannten Winkel und der vom Scanner gemessenen Entfernung (hier *dist*) lassen sich die Koordinaten des zu markierenden Feldes berechnen:

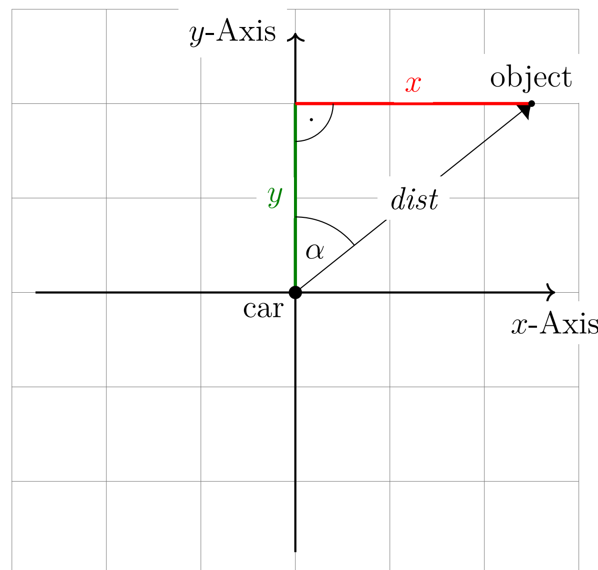


Abbildung 3.1: Berechnung der Koordinaten eines belegten Feldes

Wie in Abbildung 3.1 zu sehen ist, sind der Abstand zu dem Objekt und die noch zu ermittelnden Koordinaten des Objekts als rechtwinkliges Dreieck darstellbar. Das Auto steht hierbei im Ursprung des Koordinatensystems und ist in y-Richtung orientiert. Demnach liegt der berechnete Winkel zwischen der y-Achse und der direkten Linie vom Auto zu dem Objekt.

Aus den Definitionen für Sinus und Kosinus ergeben sich die folgenden beiden Formeln für die Koordinaten:

$$x = dist \times \sin \alpha \tag{3.2}$$

$$y = dist \times \cos \alpha \tag{3.3}$$

Die Koordinaten geben Entfernungen in Metern an. Die Felder der Karte haben aber je nach Auflösung eine Länge, die ungleich einem Meter ist. Um also ein Objekt bei den errechneten Koordinaten einzuzeichnen, müssen diese zunächst durch die Funktion *getGridCoords* in Zeile 14 in Koordinaten auf der Karte umgewandelt werden.

Das Feld an den errechneten Koordinaten erhält den Wert *OCCUPIED*. Die angrenzenden Felder werden als Gefahrenzone definiert. Die Größe dieser Zone entspricht der Länge des Modellautos. Felder, die sich in der dem belegten Feld näher liegenden Hälfte der Zone befinden, werden dann als *DANGEROUS* markiert. Ebenso werden die Felder der anderen Hälfte durch *AVOID* gekennzeichnet. Dadurch entsteht um ein belegtes Feld eine Gefahrenzone, die das Auto vermeiden sollte, aber, falls unbedingt notwendig, noch befahren kann. Außerdem werden dadurch Messfehler, die zu Lücken in Hindernissen führen, korrigiert.

```
1  def setCell(x_grid, y_grid, status):
2    if x_grid >= occupancy_grid.info.width  or x_grid < 0 or
         y_grid >= occupancy_grid.info.height or y_grid < 0:
3      return

5    p = x_grid + occupancy_grid.info.width * y_grid
6    occupancy_grid.data[p] += (100 - occupancy_grid.data[p]) *
                               status / 100
```

Listing 3.5: setCell markiert ein Feld

Beim Markieren eines Feldes wird es passieren, dass ein Wert (hier v) eingetragen werden soll, obwohl das Feld bereits einen Wert zugewiesen bekommen hat (hier o). Würde aber im schlimmsten Fall *OCCUPIED* durch einen geringeren Wert überschrieben werden, so könnte dies fatale Folgen haben. Um diese Überlappung zweier Werte sinnvoll zu bearbeiten, wird, anstatt o zu überschreiben, aus beiden Werten ein neuer errechnet (hier n).
Damit also der Gefährlichkeitsgrad eines Feldes nicht verringert wird, muss die Berechnung garantieren, dass n mindestens genauso groß ist wie sowohl o als auch v. Wie im Folgenden gezeigt, werden durch die in Zeile 6 verwendete Formel

$$n = o + (100 - o) \times (\frac{v}{100}) \tag{3.4}$$

diese Bedingungen erfüllt.
Darüber hinaus haben durch diese Berechnung Felder größere Werte, wenn sie sich näher an einer Reihe oder Gruppe belegter Felder befinden. Die Gefahrenzone hat dadurch nicht mehr eine klare Grenze zwischen *DANGEROUS* und *AVOID*, sondern einen fließenden Übergang.

Weiter oben wurde bereits beschrieben, dass für die Felder ausschließlich ganzzahlige Werte im Bereich von -1 bis 100 gültig sind, wobei bei -1 keine Aussage darüber getroffen werden kann, ob ein Feld belegt ist oder nicht. Da ein Wert von -1 aber zu ungewollten Ergebnissen führen würde, wird bei der Berechnung für unbekannte Felder angenommen, sie wären frei. Der verwendete Wert ist dann

also nicht -1, sondern 0.
Daraus ergibt sich für o und v ein Wertebereich von [0, 100].

Zunächst wird bewiesen, dass $n \geq o$ gilt:

$$\begin{array}{rrcll} & n = o + (100 - o) \times (\frac{v}{100}) & \geq & o & \mid -o \\ \Leftrightarrow & (100 - o) \times (\frac{v}{100}) & \geq & 0 & \mid \times 100 \\ \Leftrightarrow & (100 - o) \times v & \geq & 0 & \end{array}$$

Für $v = 0$:

$$\begin{array}{rrcl} & (100 - o) \times 0 & \geq & 0 \\ \Leftrightarrow & 0 & \geq & 0 \end{array}$$

Für $v > 0$:

$$\begin{array}{rrcll} & (100 - o) \times v & \geq & 0 & \mid : v \\ \Leftrightarrow & 100 - o & \geq & 0 & \mid + o \\ \Leftrightarrow & 100 & \geq & o & \end{array}$$

Nun muss noch $n \geq v$ gezeigt werden:

$$\begin{array}{rrcll} & n = o + (100 - o) \times (\frac{v}{100}) & \geq & v & \mid \times 100 \\ \Leftrightarrow & 100o + (100 - o) \times v & \geq & 100v & \\ \Leftrightarrow & 100o + 100v - vo & \geq & 100v & \mid -100v \\ \Leftrightarrow & 100o - vo & \geq & 0 & \end{array}$$

Für $o = 0$:

$$\begin{array}{rrcl} & 100 \times 0 - v \times 0 & \geq & 0 \\ \Leftrightarrow & 0 & \geq & 0 \end{array}$$

Für $o > 0$:

$$\begin{array}{rrcll} & 100o - vo & \geq & 0 & \mid : o \\ \Leftrightarrow & 100 - v & \geq & 0 & \mid + v \\ \Leftrightarrow & 100 & \geq & v & \end{array}$$

Somit ist bewiesen, dass die Formel 3.4 den geforderten Bedingungen entspricht. Der neu berechnete Wert n überschreibt dann den vorherigen Wert des Feldes.

3.2 Berechnen des Pfades

Während des Fahrens müssen die Modellautos den Pfad, auf dem sie sich bewegen, nach Hindernissen überprüfen. Doch durch das Lenken der Fahrzeuge kann sich dieser Pfad recht abrupt ändern. Daher muss zusätzlich bei jeder Änderung der Radstellung der neue Pfad bestimmt werden.

In Kapitel 3.3 wird der hier ermittelte Pfad verwendet, um Hindernisse auf dem Pfad zu finden. Außer wenn die Räder genau parallel zu der Längsachse des Autos stehen, ist der Pfad eine Kreisbahn. Für die Kreisbahn müssen sowohl der Radius als auch der Mittelpunkt, um den das Auto fährt, bestimmt werden.

3.2.1 Kalibrierung der Lenkung

Die Modellautos werden über die Vorderräder gelenkt. Beim Lenken in eine Richtung schlagen beide Vorderräder ein. Dadurch entsteht ein Spurkreis. Doch damit das Auto auf einer gleichmäßigen Kreisbahn fährt, ist der Winkel beider Räder unterschiedlich groß. Das dem Spurkreismittelpunkt näher liegende Rad bewegt sich also auf einem kleineren Kreis als das äußere Rad. Der Mittelpunkt beider Kreise ist jedoch derselbe.

Das */manual_control/steering*-Topic gibt allerdings nur einen einzelnen Radwinkel im Bereich von 0 bis 180 an. Dieser bezieht sich nicht auf eines der beiden Räder. Stattdessen wird zur Vereinfachung ein Einspurmodell verwendet:

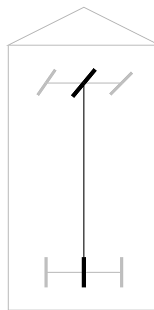


Abbildung 3.2: Einspurmodell

In diesem Modell befindet sich auf der Längsachse des Autos jeweils vorne und hinten ein einzelnes Rad. In Abbildung 3.2 sind diese durch deutlich schwarze Linien gekennzeichnet, wohingegen die eigentlichen Räder und Achsen in einem Hellgrau abgebildet sind.

Anstatt dass also zwei verschiedene Radwinkel benötigt werden, reicht ein alleiniger aus. Der Radwinkel in diesem Einspurmodell liegt vom Wert genau zwischen den eigentlichen Winkeln beider Räder. Die durch das Topic gegebene Ausrichtung ist also der Winkel, der zur Berechnung in dem Einspurmodell verwendet werden kann. Dieser Wert wird von den Modellautos in einen Echtweltwinkel (die Radstellung) umgesetzt, wobei 0 und 180 jeweils die Maximalwerte für eine Ausrichtung nach links bzw. rechts sind.

Das Problem hierbei ist, dass zwei verschiedene Modellautos bei denselben Eingabewerten verschiedene Echtweltwinkel einstellen. Des Weiteren ist es möglich, dass bei manchen Modellautos die Steuerung achsenverkehrt zu anderen Modellautos funktioniert.

Daher müssen die Modellautos zunächst kalibriert werden. Realisiert wird dies durch das *fub_steering_calibration*-Paket[3]. Um ein Fahrzeug zu kalibrieren, muss das `calibration.sh` Skript ausgeführt werden. Dieses führt für jeden der Werte 0, 30, 60, 90, 120, 150 und 180 folgenden Algorithmus aus:

- Der Nutzer wird aufgefordert, das Auto einen Meter entfernt mit Blickrichtung zu einer Wand zu positionieren.
- Das Auto ermittelt durch den Laserscanner die Position und den Abstand der Wand.
- Die Radstellung wird auf den aktuellen Wert eingestellt.
- Das Modellauto fährt ungefähr fünf Sekunden lang von der Wand weg.
- Erneut werden die Position und der Abstand der Wand bestimmt.
- Aus den gemessenen Werten können der Echtweltwinkel und der Radius des Kreises, auf dessen Bahn sich das Auto bewegt, berechnet werden. Diese Ergebnisse werden zusammen mit der eingestellten Radstellung in einer Datei gespeichert.

Besonders die ermittelten Radien werden im nächsten Abschnitt Nutzen finden.

3.2.2 Radius

Die Berechnung des Kreisradius hängt davon ab, ob der Pfad während des Fahrens ohne Richtungsänderung oder bei der Lenkung des Autos bestimmt wird.

Wird aufgrund der Lenkung ein neuer Pfad ermittelt, so können die Daten der Kalibrierung aus Kapitel 3.2.1 verwendet werden. Aus diesen Daten kann eine Funktion hergeleitet werden, welche die Radstellung auf den Radius der Kreisbahn abbildet. Dazu werden die Werte der Radstellung als Eingabewerte der Funktion betrachtet, die Radien als Ausgabewerte. Mittels polynomieller Interpolation kann aus diesem Datensatz dann die gesuchte Funktion bestimmt werden. Sobald über das */manual_control/steering*-Topic eine neue Radstellung gesendet wird, kann also der dazugehörige Radius berechnet werden.

Zum Bestimmen des Radius während des Fahrens muss der Verlauf der Positionen des Modellautos betrachtet werden. Dafür wird das */odom*-Topic betrachtet, über welches die Odometrie des Fahrzeugs bekannt gegeben wird. Die Odometrie gibt Auskunft sowohl über die Position als auch über die Orientierung des Autos.

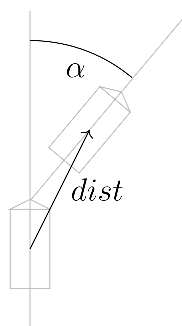


Abbildung 3.3: Bewegung des Autos

Aus zwei aufeinanderfolgenden Odometrie-Nachrichten kann dann die Strecke $dist$ ermittelt werden, die das Modellauto zurückgelegt hat, indem der euklidische Abstand beider Positionen berechnet wird. Aus der Differenz der beiden Orientierungen des Autos ergibt sich der Winkel α, um den das Fahrzeug sich bei der Bewegung gedreht hat. Durch die Dreiecksberechnung zwischen den beiden Positionen und dem Kreismittelpunkt kann der Radius r der Kreisbahn berechnet werden.

$$r = \frac{dist}{2 \sin \frac{\alpha}{2}} \tag{3.5}$$

3.2.3 Mittelpunkt

Die Umgebungskarte aus Kapitel 3.1 hat ihren Ursprung im Laserscanner des Autos. Der Mittelpunkt M_p der Kreisbahn liegt versetzt dazu auf Höhe der Hinterachse.

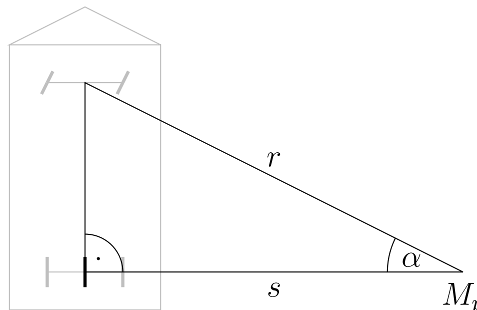


Abbildung 3.4: Berechnung des Kreisursprungs

Der Abstand zur Längsachse des Modellautos entspricht dabei der Strecke s. Aus der Definition des Kosinus lässt sich s berechnen:

$$\begin{aligned} \cos\alpha &= \frac{s}{r} && \mid \times r \\ \Leftrightarrow \qquad s &= r \times \cos\alpha \end{aligned} \tag{3.6}$$

Dabei entspricht s der x-Koordinate des Mittelpunktes.

Die y-Koordinate entspricht dem Abstand zwischen der Hinterachse des Fahrzeuges und dem Laserscanner. Diese muss für jedes Modellauto manuell gemessen und in dem Auto gespeichert werden. Dazu wird bei Start des Programms getestet, ob der Wert bereits existiert. Ist dies nicht der Fall, so wird der Nutzer aufgefordert, den Abstand einzugeben.

3.3 Hindernisse finden

Die in Kapitel 3.1 erstellte Umgebungskarte und der in Kapitel 3.2 berechnete Pfad werden nun verwendet, um zu ermitteln, ob sich Hindernisse auf der Fahrbahn des Autos befinden. Um den Pfad, welcher letztendlich ein Kreisbogen ist, auf der Karte abzubilden, müssen die Felder berechnet werden, die der Bahn entsprechen. Ein Ansatz dazu wäre, die x- und y-Koordinaten durch die Gleichungen

$$x = r \times \cos \alpha_i \tag{3.7}$$

$$y = r \times \sin \alpha_i \tag{3.8}$$

zu ermitteln. Dabei wäre r der Radius der Kreisbahn. Der Winkel α des Kreisbogens würde dazu in viele kleine α_i unterteilt werden, für die jeweils die Koordinaten zu berechnen wären. Praktisch ist dieser Ansatz jedoch nicht. Zunächst müsste die Unterteilung des Winkels fein genug sein, um wirklich alle Felder der Karte zu treffen, die unter dem Kreisbogen liegen. Doch dabei ist es nicht unwahrscheinlich, dass mehrmals das selbe Feld berechnet werden würde. Bei einer zu groben Unterteilung entstünden allerdings Lücken in dem Kreisbogen. Darüber hinaus müsste die Unterteilung für jeden Kreis in Abhängigkeit des Radius berechnet werden.

Als Alternative wird anlehnend an Bresenhams Midpoint-Kreis-Algorithmus[1, S. 83-85] ein Algorithmus verwendet, durch welchen die Felder des Kreisbogens inkrementell berechnet werden. Die Felder werden dann, ausgehend vom Auto, auf ihren Wert getestet. Liegt auf der Kreisbahn ein Feld, das belegt ist, so muss der in Kapitel 3.4 beschriebene Lösungsweg angewandt werden.

Des Weiteren ist es sinnvoll, eine Karte zu erstellen, auf der die Fahrbahn des Autos eingezeichnet ist. Diese Karte hat keinen Nutzen für das Programm selbst, sondern dient lediglich zur Visualisierung für den Nutzer. Durch die Karte kann dieser nachvollziehen, warum der Algorithmus bestimmte Entscheidungen trifft.

3.3.1 Bresenhams Midpoint-Kreis-Algorithmus

Zunächst wird Bresenhams Midpoint-Kreis-Algorithmus beschrieben, aus dem dann in Kapitel 3.3.2 ein eigener Algorithmus hergeleietet wird. Bresenhams Algorithmus eignet sich aufgrund der ausschließlichen Verwendung von Ganzzahlen sehr gut für das Zeichnen von Kreisen auf einem Raster, da dessen Koordinaten ebenfalls ganzzahlig sind.

In dem Algorithmus wird lediglich der zweite Oktant betrachtet. Die Punkte für die restlichen Oktanten werden durch die Symmetrieeigenschaften von Kreisen ermittelt. Der Startpunkt des Kreises in diesem Oktanten liegt bei $(0, r)$, wobei r der Radius ist. Der Endpunkt befindet sich mit einem Winkel von 45° zwischen den Achsen des Koordinatensystems.

Beginnend vom Startpunkt werden immer zwei mögliche Punkte betrachtet. Der Punkt, der näher an der Kreisbahn liegt, wird dann gewählt und gezeichnet. Dazu wird berechnet, ob der Mittelpunkt zwischen beiden Punkten (daher auch Midpoint-Algorithmus) innerhalb, außerhalb oder auf dem Kreis liegt. Bei der Berechnung der Punkte wird die x-Koordinate mit jedem Schritt um 1 erhöht. Sie wird daher als Primärrichtung bezeichnet. Die Sekundärrichtung ist die y-Koordinate und wird nur um -1 verringert, falls es nötig ist.

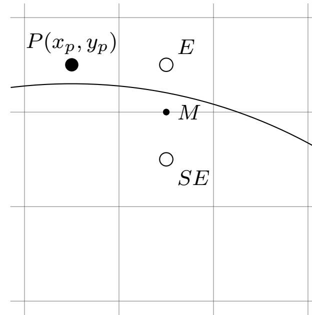


Abbildung 3.5: Wahl des nächsten Punktes

Wie in Abbildung 3.5 zu sehen ist, kann von einem Punkt $P(x_p, y_p)$ aus entweder $E(x_p+1, y_p)$ (der östliche Punkt) oder $SE(x_p+1, y_p-1)$ (der südöstliche Punkt) gewählt werden. Der Mittelpunkt M liegt genau zwischen E und SE. Mittels der Kreisgleichung

$$d = F(x, y) = x^2 + y^2 - r^2 \tag{3.9}$$

kann für M getestet werden, wo der Punkt in Relation zu dem Kreis liegt. Gilt für die Entscheidungsvariable $d < 0$, befindet sich der Punkt innerhalb des Kreises, bei $d > 0$ außerhalb und sonst genau auf dem Kreis.

Um jedoch einen inkrementellen Algorithmus zu erhalten, darf die Variable d nicht immer wieder neu berechnet werden. Stattdessen müssen ein Initialwert und der Wert, um den sich d in jeder Iteration verändert, bestimmt werden. Der Wert, um den d sich verändert, ist allerdings abhängig davon, welcher Punkt gewählt wurde. Bei Wahl des Punktes E erhöht d sich um Δ_E, bei Wahl von SE um Δ_{SE}.

In Abbildung 3.5 liegt der nächste Mittelpunkt bei $M(x_p+1, y_p-\frac{1}{2})$. In diesem Fall gilt für die Entscheidungsvariable

$$d_{old} = F(M) = (x_p+1)^2 + (y_p - \frac{1}{2})^2 - r^2 \tag{3.10}$$

Bei $d_{old} < 0$ liegt dann der Punkt E näher an der Kreisbahn und wird als nächster Punkt gewählt. Im nächsten Durchlauf wird wieder zwischen den Punkten östlich und südöstlich von E entschieden. Durch den neuen Mittelpunkt gilt für d_{new}:

$$\begin{aligned} d_{new} &= F(x_p+2, y_p-\tfrac{1}{2}) \\ &= (x_p+2)^2 + (y_p-\tfrac{1}{2})^2 - r^2 \end{aligned} \tag{3.11}$$

Die Differenz der beiden Entscheidungsvariablen entspricht dann Δ_E:

$$\begin{aligned} \Delta_E &= d_{new} - d_{old} \\ &= 2x_p + 3 \end{aligned} \tag{3.12}$$

Ist aber in Gleichung 3.10 $d_{old} \geq 0$, so wird SE gewählt. Der darauf folgende Mittelpunkt liegt bei $(x_p + 2, y_p - \frac{3}{2})$.

$$\begin{aligned} d_{new} &= F(x_p + 2, y_p - \tfrac{3}{2}) \\ &= (x_p + 2)^2 + (y_p - \tfrac{3}{2})^2 - r^2 \end{aligned} \tag{3.13}$$

Daraus ergibt sich für Δ_{SE}

$$\begin{aligned} \Delta_{SE} &= d_{new} - d_{old} \\ &= 2x_p - 2y_p + 5 \end{aligned} \tag{3.14}$$

Damit d in jeder Iteration überhaupt verändert werden kann, wird ein Initialwert benötigt. Dieser berechnet sich aus dem ersten Mittelpunkt des Oktanten. Wie bereits erwähnt, beginnt der zweite Oktant im Punkt $(0, r)$. Der erste Mittelpunkt liegt folglich bei $(1, r - \frac{1}{2})$. Der Initialwert von d ist daher

$$\begin{aligned} d &= F(1, r - \tfrac{1}{2}) \\ &= 1 + (r - \tfrac{1}{2})^2 - r^2 \\ &= \tfrac{5}{4} - r \end{aligned} \tag{3.15}$$

Um aber statt mit Gleitkommazahlen mit Ganzzahlen zu rechnen, kann d um $-\frac{1}{4}$ korrigiert werden. Dabei muss allerdings auch der Vergleich von $d < 0$ zu $d < -\frac{1}{4}$ geändert werden. Da jedoch d eine Ganzzahl ist, kann es nahe $-\frac{1}{4}$ lediglich -1 oder 0 sein. Es kann also weiterhin $d < 0$ verglichen werden.

Foley beschreibt eine weitere Optimierung des Algorithmus, indem auch Δ_E und Δ_{SE} inkrementell berechnet werden[1, S. 85-87]. Doch in eigenen Tests (s. 4.2) war die bessere Performance nur so gering, dass es sich der Aufwand der Implentierung nicht lohnen würde.

3.3.2 Kreissegmente zeichnen

In dieser Form lässt sich der Algorithmus von Bresenham jedoch noch nicht vernünftig zum Finden von Hindernissen auf einer Kreisbahn nutzen. Zum einen zeichnet der Algorithmus komplette Kreise. Es soll aber lediglich auf einem Teil des Kreises getestet werden, ob dort Hindernisse stehen, sonst würden die Modellautos aufgrund von Hindernissen anhalten, die weit entfernt stünden. Zum anderen nutzt Bresenhams Algorithmus die Symmetrie von Kreisen aus, um mehrere

Punkte gleichzeitig zu markieren. Da aber beim Finden der Hindernisse wichtig ist, den Pfad beginnend beim Auto bis zum ersten Hindernis oder Endpunkt zu untersuchen, muss jeder Punkt einzeln bestimmt werden. Darüber hinaus spielt es eine Rolle, mit welcher Umlaufrichtung (also im oder gegen den Uhrzeigersinn) das Fahrzeug auf dem Kreisbogen fährt.

Daher muss der Algorithmus so angepasst werden, dass statt des gesamten Kreises nur Kreissegmente gezeichnet werden. Dafür benötigt der Algorithmus sowohl einen Startwert als auch einen Endwert, angegeben im Bogenmaß. Anhand dieser Werte werden dann der Start- und Endpunkt berechnet.

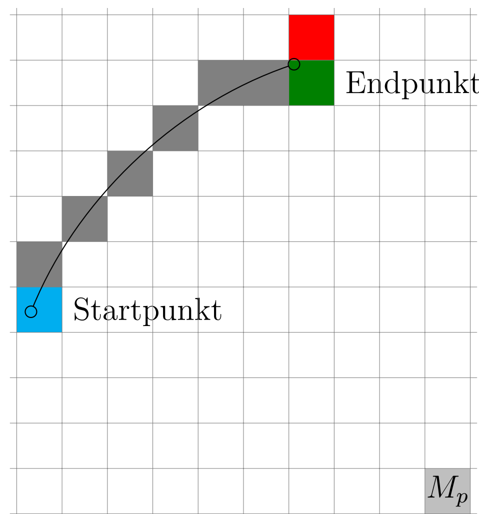


Abbildung 3.6: Start- & Endpunkt eines Kreissegments

In Abbildung 3.6 ist jedoch zu sehen, dass dabei ein neues Problem auftritt. Da bei Bresenhams Algorithmus diagonale Schritte zulässig sind, werden manche Felder unter dem echten Kreis übersprungen. Das Endfeld wird allerdings anhand des echten Kreises exakt berechnet. Dabei kann es passieren, dass der berechnete Endpunkt (grün markiert) einer der übersprungenen Punkte ist. Statt also diesen zu treffen, bewegt sich Bresenhams Algorithmus diagonal zu einem Feld, das über den Endpunkt hinaus liegt (rot markiert).

Aus diesem Grund muss der neue Algorithmus in der Lage sein, sich nicht nur in die Primärrichtung und diagonal zu bewegen, sondern auch in die Sekundärrichtung. Außerdem kann auch nicht mehr nur ein einzelner Oktant betrachtet werden. Stattdessen muss der Algorithmus die nächsten Punkte in Abhängigkeit des Oktanten wählen.

Anpassen des Algorithmus

Vom Prinzip her funktioniert der Algorithmus zum Zeichnen von Kreissegmenten wie der von Bresenham. Zunächst wird erneut nur der zweite Oktant betrachtet und komplett gezeichnet. Ausgehend vom Punkt P werden hier jedoch drei

Punkte betrachet. Wie zuvor erhält man E durch Erhöhen der x-Koordinate (Primärrichtung) und SE durch die Änderung beider Achsen. Neu hinzu kommt der Punkt S, der durch Verringern der y-Koordinate (Sekundärrichtung) entsteht. Der Mittelpunkt dieser Punkte liegt dann bei $M(x_p + \frac{1}{2}, y_p - \frac{1}{2})$.

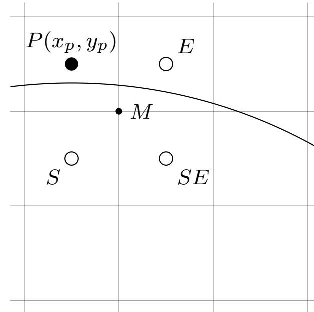


Abbildung 3.7: Wahl des nächsten Punktes

Die Kreisformel 3.9 bleibt unverändert. Der Wert d_{old} muss aber für den Punkt M erneut berechnet werden:

$$d_{old} = (x_p + \frac{1}{2})^2 + (y_p - \frac{1}{2})^2 - r^2 \tag{3.16}$$

Bei $d_{old} < 0$ wird wieder der Punkt E gewählt, sodass der nächste Mittelpunkt um 1 in x-Richtung verschoben liegt.

$$d_{new} = (x_p + \frac{3}{2})^2 + (y_p - \frac{1}{2})^2 - r^2 \tag{3.17}$$

Aus 3.16 und 3.17 ergibt sich $\Delta_E = 2x_p + 2$.

Ist aber $d_{old} > 0$ (*Wichtig*: größer als, nicht größer gleich), so wird der Punkt S gewählt. Der nächste Mittelpunkt liegt dann bei $(x_p + \frac{1}{2}, y_p - \frac{3}{2})$.

$$d_{new} = (x_p + \frac{1}{2})^2 + (y_p - \frac{3}{2})^2 - r^2 \tag{3.18}$$

Demnach ist $\Delta_S = -2y_p + 2$.

Anders als bei Bresenhams Algorithmus wird hier $d_{old} = 0$ extra behandelt. In diesem Fall wird der Punkt SE als nächster gewählt. Aber um das in Abbildung 3.6 gezeigte Problem zu beheben, werden zusätzlich zu SE sowohl E als auch S gezeichnet. Der darauf folgende Mittelpunkt hat die Koordinaten $(x_p + \frac{3}{2}, y_p - \frac{3}{2})$.

$$d_{new} = (x_p + \frac{3}{2})^2 + (y_p - \frac{3}{2})^2 - r^2 \tag{3.19}$$

Daher ist $\Delta_{SE} = 2x_p - 2y_p + 4$.

Auch hier muss wieder der Initialwert von d berechnet werden. Aber mit dem Hintergedanken, dass der Startpunkt letztendlich frei wählbar sein wird, ergibt

sich kein konkreter Wert, sondern eine Formel abhängig vom Startpunkt. Der Startpunkt wird demnach als (x, y) betrachtet, sodass der Mittelpunkt bei $(x + \frac{1}{2}, y - \frac{1}{2})$ liegt. Daraus ergibt sich:

$$\begin{aligned} d &= (x + \tfrac{1}{2})^2 + (y - \tfrac{1}{2})^2 - r^2 \\ &= x^2 + x + \tfrac{1}{4} + y^2 - y + \tfrac{1}{4} - r^2 \\ &= x^2 + x + y^2 - y - r^2 + \tfrac{1}{2} \end{aligned} \tag{3.20}$$

Wie auch schon bei Bresenhams Algorithmus kann der Initialwert auf eine Ganzzahl korrigiert werden, indem $\frac{1}{2}$ von d abgezogen wird. Der Vergleich würde ebenfalls zu $d < -\frac{1}{2}$ werden, was aber vernachlässigbar ist, da d in jedem Fall eine Ganzzahl ist. Daher ist der Initialwert $d = x^2 + x + y^2 - y - r^2$.

Übertragen auf alle Oktanten

Doch diese Werte gelten alle nur für ein Kreissegment im zweiten Oktanten. Wie bereits in Kapitel 3.3.1 beschrieben, gibt es eine Primärrichtung (*pri*) und eine Sekundärrichtung (*sec*). Ob die x-Richung oder y-Richtung die Primärrichtung ist, ist abhängig von den Beträgen der Koordinaten des aktuellen Punktes.

$$\begin{aligned} &|x| < |y| \Rightarrow x \text{ ist } pri,\ y \text{ ist } sec \\ &|x| > |y| \Rightarrow y \text{ ist } pri,\ x \text{ ist } sec \end{aligned} \tag{3.21}$$

Für den Fall $|x| = |y|$, bei dem sich der Punkt genau auf einer der Diagonalen befindet, muss die Umlaufrichtung des Fahrzeugs betrachtet werden. Die Umlaufrichtung *rot_dir* ist 1, wenn das Auto gegen den Uhrzeigersinn fährt, und sonst -1. Darüber hinaus ist es wichtig, auf welcher Diagonalen sich das Auto befindet, also welche Vorzeichen die Koordinaten haben. Die Vorzeichenfunktion *sgn* ist 1 für positive Zahlen, -1 für negative und 0 für die Zahl 0.

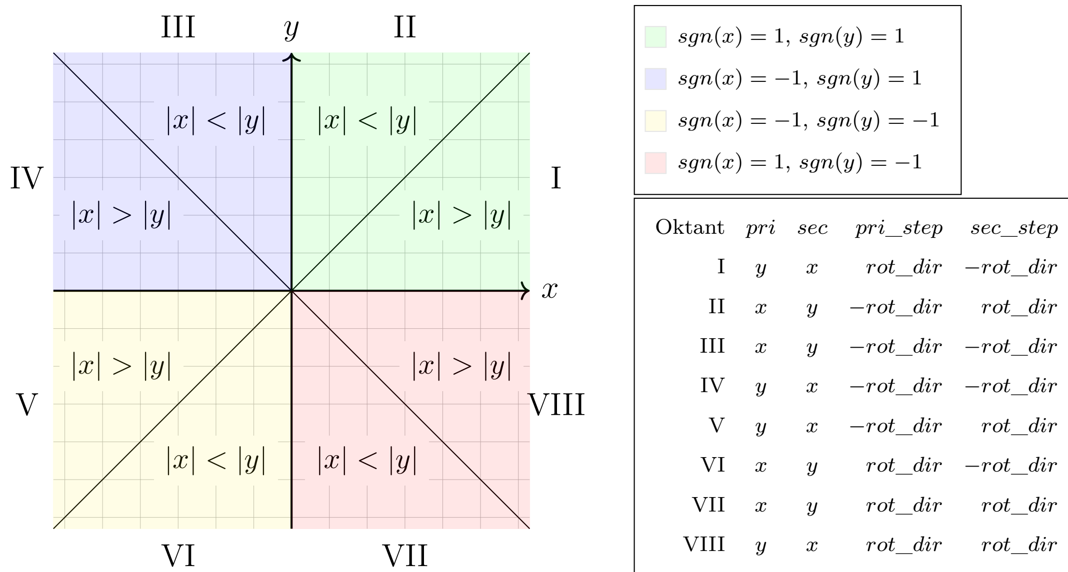
$$|x| = |y| \Rightarrow \begin{cases} x \text{ ist } pri, & \text{falls } \operatorname{sgn}(x \times y) \times rot_dir = 1 \\ y \text{ ist } pri, & \text{sonst} \end{cases} \tag{3.22}$$

Im zweiten Oktanten wurde x als Primärrichtung in jeder Iteration erhöht, wohingegen y - falls nötig - verringert wurde. Diese Änderung der Werte wird als Primärschritt (*pri_step*) bzw. Sekundärschritt (*sec_step*) bezeichnet. Die Schrittwerte können entweder 1 oder -1 annehmen. Abhängig ist dies wiederum von den Vorzeichen der Koordinaten des aktuellen Punktes, der Umlaufrichtung und ob sich der Punkt auf einer der Achsen oder Diagonalen befindet.

$$pri_step = \begin{cases} \operatorname{sgn}(-x), & \text{falls } x = y \\ \operatorname{sgn}(x - y) \times rot_dir, & \text{sonst} \end{cases} \tag{3.23}$$

$$sec_step = \begin{cases} \operatorname{sgn}(x), & \text{falls } x = y \\ \operatorname{sgn}(-x), & \text{falls } y = 0 \\ \operatorname{sgn}(-y), & \text{falls } x = 0 \\ \operatorname{sgn}(x - y) \times \operatorname{sgn}(x \times y) \times (-rot_dir), & \text{sonst} \end{cases} \tag{3.24}$$

Zur Veranschaulichung sind die Eigenschaften jedes Oktanten in Abbildung 3.8 gezeigt.



Oktant	*pri*	*sec*	*pri_step*	*sec_step*
I	y	x	rot_dir	$-rot_dir$
II	x	y	$-rot_dir$	rot_dir
III	x	y	$-rot_dir$	$-rot_dir$
IV	y	x	$-rot_dir$	$-rot_dir$
V	y	x	$-rot_dir$	rot_dir
VI	x	y	rot_dir	$-rot_dir$
VII	x	y	rot_dir	rot_dir
VIII	y	x	rot_dir	rot_dir

Abbildung 3.8: Die Eigenschaften der Koordinaten in jedem Oktanten

Damit der Algorithmus für alle Oktanten funktioniert, müssen die Initialwerte von d, Δ_E, Δ_S und Δ_{SE} in Abhängigkeit zum Oktanten gebracht werden.

Je nachdem welche Koordinate die Primärrichtung ist und welchen Wert die entsprechenden Schrittwerte haben, ist der erste Mittelpunkt M zur Berechnung des Initialwertes von d bei $(x \pm \frac{1}{2}, y \pm \frac{1}{2})$. Wird dieser Punkt in die Kreisgleichung eingesetzt, ergibt sich ähnlich wie bei Gleichung 3.20

$$\begin{aligned} x \text{ ist } pri \Rightarrow \quad d &= x^2 + (x \times pri_step) + y^2 + (y \times sec_step) - r^2 \\ y \text{ ist } pri \Rightarrow \quad d &= x^2 + (x \times sec_step) + y^2 + (y \times pri_step) - r^2 \end{aligned} \tag{3.25}$$

Δ_E, Δ_S und Δ_{SE} sind Bezeichnungen konkret für den zweiten Oktanten. Daher werden sie zunächst umbenannt in Δ_P (bei Bewegen in Primärrichtung), Δ_S (bei Bewegen in Sekundärrichtung) und Δ_B (bei diagonalem Bewegen in beide Richtungen). Analog zu dem zweiten Oktanten können dann die Gleichungen für die anderen Oktanten berechnet werden. Aus allen kann man dann die folgenden verallgemeinerten Gleichungen herleiten, die für jeden Oktanten gleichermaßen gelten:

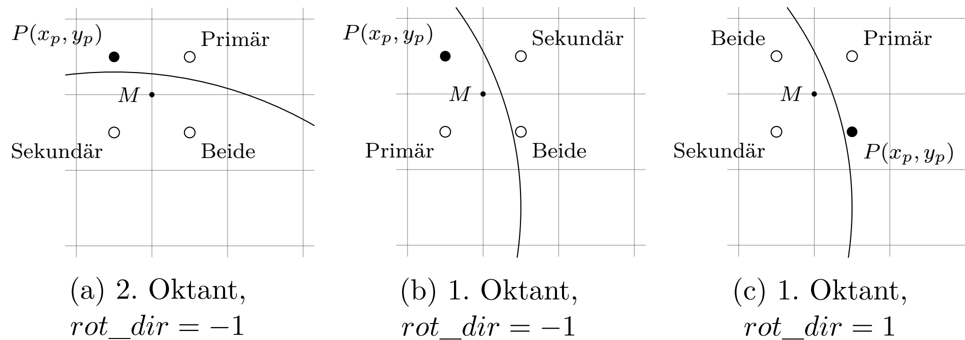
$$\Delta_P = 2pri \times pri_step + 2 \tag{3.26}$$

$$\Delta_S = 2sec \times sec_step + 2 \tag{3.27}$$

$$\Delta_B = 2pri \times pri_step + 2sec \times sec_step + 4 \tag{3.28}$$

Vom aktuellen Oktanten und der Umlaufrichtung ist zusätzlich abhängig, welcher Punkt bei $d > 0$ bzw. $d < 0$ gewählt wird. In Abbildung 3.9a und Abbildung 3.9c

wird beispielsweise bei $d < 0$ der Punkt in Primärrichtung gewählt, in Abbildung 3.9b jedoch der Punkt in Sekundärrichtung.



(a) 2. Oktant, $rot_dir = -1$

(b) 1. Oktant, $rot_dir = -1$

(c) 1. Oktant, $rot_dir = 1$

Abbildung 3.9: Wahl des nächsten Punktes in verschiedenen Oktanten

Um den Vergleich zu vereinheitlichen, muss das Vorzeichen von d durch die Umlaufrichtung und die beiden Schrittwerte angepasst werden:

$$d_{comp} = d \times rot_dir \times pri_step \times sec_step \tag{3.29}$$

Abschließend muss der Algorithmus überprüfen, in welchen Oktanten der aktuelle Punkt liegt. Hat sich der Oktant im Vergleich zum vorherigen Punkt geändert, so müssen anhand der Formeln 3.25, 3.26, 3.27, 3.28 die Initialwerte erneut berechnet werden.

Der Quellcode zur Berechnung eines Punktes ist wie folgt:

```
# Bestimme den aktuellen Oktanten.
new_octant = get_octant(x, y, rot_dir)
# Falls sich der Oktant geändert hat,
  berechne neue Initialwerte
if prev_octant != new_octant:
 prev_octant = new_octant
 # Bestimme pri, sec und die jeweiligen Schrittwerte
 is_x_pri, pri_step, sec_step = get_steps(x, y, rot_dir)
 if is_x_pri:
  d = x * x + pri_step * x + y * y + sec_step * y - r * r
 else:
  d = x * x + sec_step * x + y * y + pri_step * y - r * r

# Normalisiere das Vorzeichen von d
d_comp = d * pri_step * sec_step * rot_dir
if is_x_pri:
 if d_comp > 0: # Primärrichtung wird gewählt
  d += 2 * x * pri_step + 2
  x += pri_step
 elif d_comp < 0: # Sekundärrichtung wird gewählt
  d += 2 * y * sec_step + 2
  y += sec_step
 else: # Beide werden gewählt
```

```
23     d += 2 * x * pri_step + 2 * y * sec_step + 4
24     checkCell(x + pri_step, y)
25     y += sec_step
26     checkCell(x, y)
27     x += pri_step
28   else:
29    if d_comp < 0: # Primärrichtung wird gewählt
30     d += 2 * y * pri_step + 2
31     y += pri_step
32    elif d_comp > 0: # Sekundärrichtung wird gewählt
33     d += 2 * x * sec_step + 2
34     x += sec_step
35    else: # Beide werden gewählt
36     d += 2 * x * sec_step + 2 * y * pri_step + 4
37     checkCell(x + sec_step, y)
38     y += pri_step
39     checkCell(x, y)
40     x += sec_step

42  # Überprüfe das Feld
43  checkCell(x, y)
```

Listing 3.6: Berechnung eines Punktes

3.4 Lösungsansatz

Je schneller das Modellauto fährt, desto geringer ist die Reaktionszeit, um auszuweichen oder stehen zu bleiben. Auch der Bremsweg nimmt bei höherer Geschwindigkeit zu. Dementsprechend muss das Kreissegment in Abhängigkeit von der Geschwindigkeit betrachtet werden. Der Startpunkt liegt bei der Spitze des Autos. Für den Endpunkt kann, wie auch in dem bereits verwendeten *auto_stop*-Programm, zunächst eine Strecke festgelegt werden, die nach Hindernissen überprüft werden soll. Je nach Geschwindigkeit wird diese dann größer oder kleiner. Aus der Strecke und dem Winkel des Startpunktes lässt sich der Winkel des Endpunktes auf der Kreisbahn berechnen.

$$\alpha_{end} = \alpha_{start} + \frac{brake_distance}{r} \tag{3.30}$$

Dabei ist zu beachten, dass bei $\frac{brake_distance}{r} \geq 2\pi$ der Endpunkt auf den Startpunkt gesetzt werden muss, sodass der gesamte Kreis bestimmt wird. Mehr als den ganzen Kreis zu berechnen, wäre redundant.

Ist der untersuchte Pfad des Fahrzeugs komplett frei, so kann das Auto ungestört weiter fahren. Wird allerdings ein gefährliches oder sogar belegtes Feld gefunden, muss das Auto dementsprechend handeln. Dazu wird zwischen verschiedenen Fällen unterschieden:

1. Auf dem Pfad liegt wenigstens ein Feld mit einem Gefahrenwert größer oder

gleich 90. In diesem Fall ist es für das Modellauto zu riskant, weiter zu fahren. Es hält daher sofort an.

2. Es befinden sich nur Felder mit einem Wert kleiner als 90 auf dem Pfad. Der Gefährlichkeitsgrad wird aber im Laufe des Pfades immer größer. Bei diesem Fall kann angenommen werden, dass sich das Auto auf ein Objekt zu bewegt, aber noch ausreichend weit entfernt ist. Zur Sicherheit sollte hier das Fahrzeug nach und nach langsamer werden. Zu einem gewissen Zeitpunkt wird dann entweder der vorherige oder der nächste Fall eintreffen.

3. Es befinden sich nur Felder mit einem Wert kleiner als 90 auf dem Pfad, der Gefährlichkeitsgrad wird aber ab einem bestimmten Punkt immer kleiner. Da der Gefahrenwert der Felder sinkt und kein Feld 90 überschritten hat, kann daraus gefolgert werden, dass sich das Auto von einem Objekt entfernt. Es kann daher normal weiter fahren.

4. Der Gefährlichkeitsgrad der Felder bleibt unverändert. Dies ist der Fall, wenn sich das Modellauto parallel zu einem Objekt bewegt. Auch hier geht keine Gefahr von dem Objekt aus und das Auto kann normal weiter fahren.

Kapitel 4

Experimente

4.1 Variante der Umgebungskarte

Ursprünglich wurden in der in Kapitel 3.1 erstellten Umgebungskarte zusätzlich zu den gefährlichen und belegten Feldern auch freie Felder eingezeichnet. Mittels der Daten des Laserscanners wurden alle unbekannten Felder unter einem Laserstrahl bis hin zu dem belegten Feld als *FREE* gekennzeichnet. Jedoch wurde dieses Vorgehen aus drei Gründen im Rahmen dieser Arbeit wieder verworfen:

1. Durch eine hohe Auflösung des Rasters und einen dafür zu großen Abstand zwischen den Strahlen des Scanners würden viele Felder unbekannt bleiben. Dies wirft die Frage auf, ob und falls ja wie man unbekannte Felder von freien Feldern unterscheiden sollte. Im nächsten Punkt wird allerdings erklärt, warum die Frage irrelevant ist.

2. Beim Verwenden der Karte zum Finden von Hindernissen wurde deutlich, dass die Information, dass ein Feld frei ist, keinerlei Anwendung findet. Lediglich gefährliche oder belegte Felder wurden verwendet, um Entscheidungen zu treffen. Natürlich heißt dies nicht, dass ein unbekanntes Feld automatisch frei ist. Unbekannte Felder hinter bereits gefundenen Objekten könnten selber durchaus in Wirklichkeit belegt sein. Aber diese Felder werden im Laufe des Fahrens sichtbar und bleiben daher nicht permanent unbekannt.

3. Zusätzlich zum Berechnen der Koordinaten des belegten Feldes wurde jedes Feld bestimmt, das unter dem Strahl liegt. Dafür gibt es durchaus schnelle Algorithmen. Da aber für die Zielsetzung dieser Arbeit kein Nutzen für die Unterscheidung zwischen unbekannten und freien Feldern existiert, kann die Rechenzeit dazu eingespart werden.

Folglich wurden nur die gefährlichen und belegten Felder markiert. Zum Erstellen der Karte nach Kapitel 3.1 wurde das Modellauto in einem Raum platziert. Ins nähere Umfeld des Autos wurden ein Karton und ein Ball als Hindernisse gestellt.

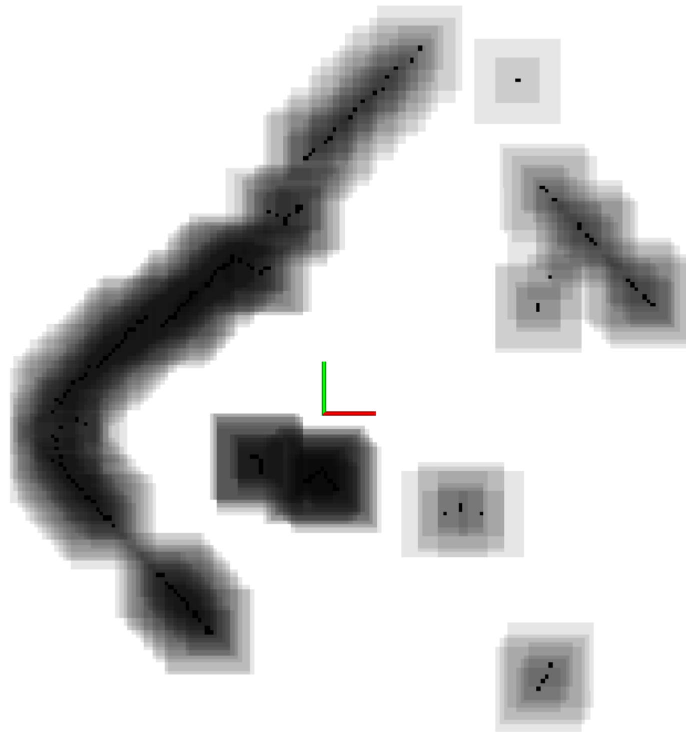
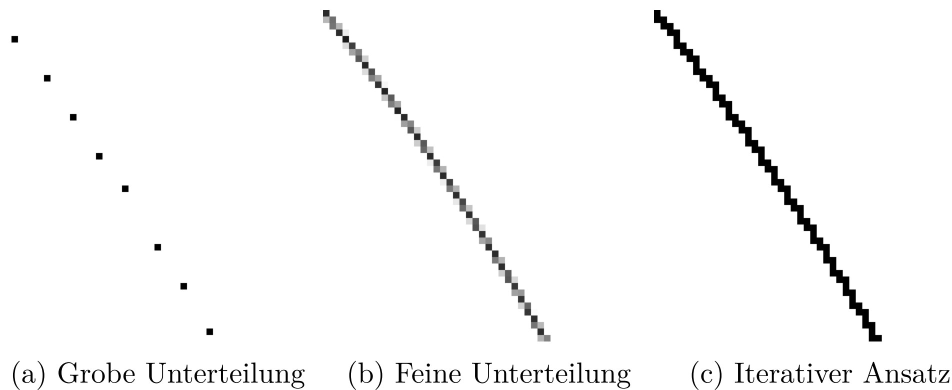


Abbildung 4.1: Fertige Umgebungskarte

Hierbei ist ein Feld umso gefährlicher, je dunkler es ist. Belegte Felder sind dementsprechend schwarz markiert. Das Modellauto befindet sich im Ursprung der eingezeichneten Achsen (rot $\hat{=}$ x-Achse, grün $\hat{=}$ y-Achse). Direkt südlich vom Fahrzeug liegt der Karton, westlich daneben der Ball. Die längere Linie, die sich vom Norden der Karte bis zum Westen streckt und dann in Richtung Süden abknickt, ist die Wand des Raumes. Dazu waren noch einige andere Objekte im Raum, die nicht für den Versuch platziert wurden.

4.2 Kreis Algorithmus & Laufzeit

Der erste Ansatz zum Zeichnen von Kreisen war es, die Punkte durch die Formeln 3.7 und 3.8 zu berechnen. Doch dafür muss der Kreis in viele kleine Abschnitte unterteilt werden. Werden zu große Abschnitte gewählt, so entstehen zwischen den Punkten des Kreises Lücken (s. Abbildung 4.2a). Sind die Abschnitte allerdings klein genug, um Lücken zu vermeiden, so werden manche Felder mehrfach berechnet. Dies ist in Abbildung 4.2b veranschaulicht. Je dunkler dort ein Feld ist, desto öfter wurde es als auf der Kreisbahn liegend markiert.



(a) Grobe Unterteilung (b) Feine Unterteilung (c) Iterativer Ansatz

Abbildung 4.2: Kreisausschnitte der verschiedenen Algorithmen

Ein durch eine feine Unterteilung erstellter Kreis unterscheidet sich so gut wie nicht von einem Kreis, welcher durch den in Kapitel 3.3 beschriebenen Algorithmus kreiert wurde. Allerdings wird bei Letzterem jeder Punkt exakt nur ein einziges Mal bestimmt, sodass die redundante Mehrfachberechnung vermieden wird. Dies hilft, eine bessere Laufzeit zu erreichen.

Um zu bestimmen, wie viel schneller der verwendete Algorithmus ist, wurde ein Programm geschrieben, welches beide Algorithmen eine Million Mal ausführt und die Laufzeiten misst. Die zu testenden Algorithmen wurden in der Programmiersprache C implementiert. Der Laufzeittest wurde auf einem Computer mit einem $2,6$ GHz Intel Core i5 Prozessor und 8 GB RAM ausgeführt. Das Betriebssystem ist macOS 10.14.1 Beta. Für die Berechnung des Kreis mittels der Winkelfunktionen wurde eine möglichst große Unterteilung verwendet, bei der trotzdem keine Felder übersprungen werden. Dieser Algorithmus lief dabei für $564,633$ Sekunden. Der iterative Ansatz zur Kreisberechnung brauchte hingegen $281,689$ Sekunden. Damit ist letzterer etwa doppelt so schnell wie die Berechnung über die Winkelfunktionen.

In Kapitel 3.3.1 wurde von Foley eine weitere Optimierung für Bresenhams Algorithmus erwähnt. Setzt man diese für den iterativen Algorithmus um und misst die Laufzeit des resultierenden Algorithmus, ergibt sich eine Laufzeit von $279,3$ Sekunden. Dies ist knappe 2 Sekunden schneller, benötigt aber beim Programmieren mehr Aufwand. Aufgrund dessen wird diese marginale Optimierung nicht verwendet.

Kapitel 5

Fazit und Ausblick

Zunächst wurden die bereits vorhandene Sicherheitsarchitektur analysiert und ihre Schwachpunkte ermittelt. Um eine Verbesserung zu erreichen, wurde eine Umgebungserkennung entwickelt, die die gesamte Umgebung des Autos abbildet. Zusätzlich dazu wurde die Berechnung des Bewegungspfades eingeführt. Dadurch entstand eine Sicherheitsarchitektur, die auf mehr Gefahren reagiert und besser auf diese eingeht.

In dieser Arbeit wurde der Quellcode zwar überwiegend in Python geschrieben, aber sollte an einer Stelle die Laufzeit zu langsam sein, ist es empfehlenswert, das Programm in C++ zu portieren.

Der hier gewählte Lösungsansatz ist noch recht simpel und könnte durchaus noch erweitert werden. Eine Möglichkeit wäre es, anstatt das Auto zu stoppen, anhand der Umgebungskarte einen Pfad zu berechnen, der sowohl um das Hindernis herum führt als auch sich dem eigentlich gewählten Pfad letztendlich wieder annähert. Ein anderer Lösungsansatz könnte verhindern, dass das Auto überhaupt erst in Richtung eines Hindernisses einlenken kann. Auch die Erkennung der Hindernisse kann insofern verbessert werden, dass bewegte Hindernisse erkannt und ihr Pfad bestimmt werden.

Mit der Weiterentwicklung solcher Sicherheitsarchitekturen und der künftigen Übertragung vom Modell auf das reale Auto kann ein wichtiger Beitrag zur Sicherheit im modernen Straßenverkehr geleistet werden.

Eidesstattliche Erklärung

Ich versichere, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer Personen entnommen sind, habe ich als entnommen gekennzeichnet. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Unterschrift: *Ort, Datum:*

Literaturverzeichnis

[1] James D. Foley. *Computer Graphics: Principles and Practice (2. ed.)*. Addison-Wesley, 1990.

[2] AutoModelCar GitHub. auto_stop.cpp. https://github.com/AutoModelCar/model_car/blob/version-4.0/catkin_ws/src/auto_stop/src/auto_stop.cpp, 2018.

[3] AutoModelCar GitHub. fub_steering_calibration. https://github.com/AutoModelCar/model_car/tree/version-4.0/catkin_ws/src/fub_steering_calibration, 2018.